Vyper Documentation

Vyper Team (originally created by Vitalik Buterin)

Jun 16, 2026

CONTENTS

CHAPTER

ONE

OVERVIEW

Vyper is a Pythonic smart contract language that compiles to Ethereum Virtual Machine (EVM) bytecode. It prioritises **security**, **auditability**, and **simplicity**.

1.1 Principles

- **Security**: Building secure smart contracts should be natural, not an uphill battle.
- **Simplicity**: Both the language and compiler should be easy to understand.
- **Auditability**: Code should be maximally human-readable. Simplicity for the reader matters more than convenience for the writer.

1.2 Key Features

Safety by default

- Bounds and overflow checking on array accesses and arithmetic
- Reentrancy protection via the `@nonreentrant` decorator (see *Control Structures*)
- Strong typing with explicit *type conversions*

Predictable execution

- Decidable gas consumption: every function call has a calculable upper bound
- Bounded loops only (compile-time maximum iterations)
- No recursion: execution flow is structurally decreasing

Clean code reuse

- *Module imports* instead of class inheritance
- Explicit `extcall` and `staticcall` keywords for external contract interactions
- Support for *pure functions* that cannot modify state

1.3 Compiler-Enforced Security

Vyper eliminates entire vulnerability classes by excluding features that enable dangerous patterns:

Excluded Feature	Why It Matters
Inline assembly	Preserves type safety, overflow protection, and searchability of variable usage
Class inheritance	Removes ambiguity about which code executes and simplifies auditing
Modifiers	All checks are inline and visible, no hidden pre/post conditions
Function overloading	Function calls are unambiguous; `foo(x)` always means the same thing
Operator overloading	Arithmetic operators do exactly what they appear to do
Infinite loops	Gas costs are always bounded and predictable
Recursive calls	Call graphs are simple and gas limits are enforceable

These constraints mean developers cannot accidentally introduce dangerous patterns, even under time pressure or with limited blockchain experience.

1.4 Deep Verification

Vyper's design makes **deep verification** practical on a production smart-contract language.

See *Deep Verification* for the full discussion of verification depth, verification gap, and the current state of Vyper's formal semantics and compiler verification work.

1.5 Decimal Fixed Point

Vyper uses decimal (not binary) fixed point numbers. This ensures that literals like `0.1` have exact representations, avoiding the subtle precision errors common in binary floating-point arithmetic.

CHAPTER

TWO

INSTALLING VYPER

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

Tip: New to Vyper? Start with the *Quickstart* guide to get a project running quickly.

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

- Try Vyper!: maintained by the Vyper team, requires GitHub login
- Remix: maintained by the Ethereum Foundation, activate the vyper-remix plugin in the Plugin manager

2.1 Standalone

The Vyper CLI can be installed with `uv tool` or `pipx`. If you do not have these installed, first visit their installation pages:

- https://github.com/astral-sh/uv
- https://github.com/pypa/pipx

Then install Vyper:

```
uv tool install vyper
```

or:

```
pipx install vyper
```

2.2 Binaries

Alternatively, prebuilt Vyper binaries for Windows, Mac and Linux are available for download from the GitHub releases page: https://github.com/vyperlang/vyper/releases.

2.3 PIP

2.3.1 Installing Python

Vyper can only be built using Python 3.11 and higher. If you need to know how to install the correct version of python, follow the instructions from the official Python website.

2.3.2 Creating a virtual environment

Because pip installations are not isolated by default, this method of installation is meant for more experienced Python developers who are using Vyper as a library, or want to use it within a Python project with other pip dependencies.

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.

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

2.3.3 Installing Vyper

Each tagged version of vyper is uploaded to pypi, and can be installed inside a virtual environment:

```
pip install vyper
```

or:

```
uv pip install vyper
```

To install a specific version use:

```
pip install vyper==0.4.0
```

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

```
vyper --version
```

2.4 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>
[{"stateMutability": "nonpayable", "type": "function", "name": "test1", "inputs": [{"name
↪": "a", "type": "uint256"}, {"name": "b", "type": "bytes"}], "outputs": []}, {
↪"stateMutability": "nonpayable", "type": "function", "name": "test2", "inputs": [{"name
↪": "a", "type": "uint256"}], "outputs": []}]
```

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

2.5 nix

View the versions supported through nix at nix package search

Note: The derivation for Vyper is located at nixpkgs

2.5.1 Installing Vyper

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

CHAPTER

THREE

QUICKSTART

This guide gets you from zero to a compiled Vyper contract in 5 minutes.

3.1 Which Framework Should I Use?

Short answer: Use Moccasin for new projects.

Moccasin is a Vyper-first development framework built on Titanoboa (Vyper's native testing tool). It provides:

- Project scaffolding
- Compilation
- Testing with pytest (see *Testing a Contract*)
- Deployment scripts
- Network management

If you're coming from Solidity/Foundry, Moccasin is the closest equivalent for Vyper.

Note: What about Foundry, Hardhat, or Ape?

- **Foundry:** Primarily for Solidity. Vyper support exists but requires workarounds.
- **Hardhat:** JavaScript-based, has a Vyper plugin, but not the recommended path.
- **Ape:** Good if you need multi-language support, but adds complexity.
- **Brownie:** Deprecated. Do not use for new projects.

3.2 Prerequisites

- Python 3.11 or higher
- uv (recommended) or pip

Note: If you're new to Python or having environment issues, see *Troubleshooting* at the bottom of this page.

3.3 Installing Moccasin

We recommend installing Moccasin using uv:

```
# Install uv (if you don't have it)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Install Moccasin
uv tool install moccasin
```

Verify it works:

```
mox --version
```

You should see something like `Moccasin CLI v0.4.3`.

Note: Using uv vs pip

We recommend `uv tool install` because it handles Python environment isolation automatically. If you prefer pip, you can use `pip install moccasin`, but you may need to manage virtual environments yourself.

3.4 Creating a Project

```
mox init my_project
cd my_project
```

This creates a ready-to-use project structure:

```
my_project/
├── src/              # Your Vyper contracts
├── tests/            # Your tests
├── script/           # Deployment scripts
└── moccasin.toml     # Configuration
```

Moccasin generates a sample contract and test to get you started.

3.5 Compiling

```
mox compile
```

This compiles all `.vy` files in the `src/` folder.

3.6 Running Tests

```
mox test
```

You should see output like:

```
============================= test session starts ==============================
collected 1 item
tests/test_counter.py .                                                  [100%]
============================== 1 passed in 0.03s ===============================
```

3.7 Exploring the Sample Contract

Open `src/Counter.vy` to see a minimal Vyper contract:

```
#pragma version ^0.4.1

number: public(uint256)

@external
def set_number(new_number: uint256):
    self.number = new_number

@external
def increment():
    self.number += 1
```

This demonstrates:

- **Version pragma:** `#pragma version ^0.4.1` specifies the compiler version
- **State variables:** `number: public(uint256)` creates storage with an automatic getter
- **Functions:** `@external` marks functions callable from outside the contract

3.8 Next Steps

- Browse *Vyper by Example* for more complex contracts
- Read about *Types* and *Control Structures*
- Learn about *Modules* for code reuse
- See the Moccasin documentation for deployment and advanced features

3.9 Troubleshooting

"command not found: mox"

If you installed with `uv tool install`, restart your terminal or run:

```
source ~/.bashrc  # or ~/.zshrc on macOS
```

"pip install moccasin" fails

Use uv instead:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
uv tool install moccasin
```

Python version issues

Moccasin requires Python 3.11+. Check your version:

```
python3 --version
```

CHAPTER

FOUR

DIFFERENCES FROM SOLIDITY

This page covers the key differences between Solidity and Vyper, the reasoning behind them, and their consequences.

4.1 Quick Reference

Solidity	Vyper	Rationale
`modifier`	Inline checks	Control flow is explicit in the function body
`class inheritance`	`import + exports + @abstract / @override`	Explicit dependencies, compile-time resolution
`assembly { }`	Not supported	No direct EVM opcode access; use specific builtins (`raw_call`, `create_minimal_proxy_to`, etc.)
`while (true)`	`for i in range(n)`	Bounded gas costs
`mapping`	`HashMap`	Same semantics
`emit Event()`	`log Event()`	Same semantics
`require()`	`assert / raise`	Different semantics; explicit error paths
`contract.call()`	`extcall / staticcall`	Explicit external calls

4.2 Philosophy

Vyper prioritizes three properties: security, simplicity, and auditability.

To achieve these properties, Vyper excludes features that obscure control flow or make code difficult to reason about. Each omission is a deliberate tradeoff: less flexibility in exchange for explicit behavior. See *Principles* for the full rationale.

4.3 No Modifiers

Solidity modifiers wrap function execution:

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}

function withdraw() public onlyOwner {
    // ...
}
```

While `onlyOwner` appears simple, modifiers can execute code before and after the function body, modify state, and obscure the actual logic. Understanding a function requires reading the modifier definitions elsewhere in the codebase. For this reason, Vyper does not have modifiers.

In Vyper, checks are written inline:

```
@external
def withdraw():
    assert msg.sender == self.owner, "Not owner"
    # ...
```

Inline checks keep the control flow visible from top to bottom.

4.4 Modules Instead of Inheritance

Solidity supports multiple inheritance, which introduces the diamond problem and C3 linearization complexity. Vyper replaces inheritance with a module system that achieves the same goals of code reuse and behavioral customization through different, more explicit mechanisms.

In Vyper 0.4.0, a module system was introduced for powerful code reuse:

```
import ownable

initializes: ownable
exports: ownable.transfer_ownership

@deploy
def __init__():
    ownable.__init__()
```

Three declarations manage module relationships: `initializes` (this contract manages the module's storage), `uses` (this contract reads module state without initializing), and `exports` (expose module functions in the ABI). See *Modules* for details.

Where Solidity uses `abstract contract` with `virtual` / `override` for polymorphism, Vyper uses `@abstract` and `@override` decorators on internal module functions. An abstract module declares methods with no implementation; the module that `initializes` it supplies the concrete `@override`. Each abstract module has exactly one initializer, so there is no ambiguity about which override is chosen. All overrides are resolved at compile time — there is no virtual dispatch and no method-resolution order. See *Abstract Modules* for details.

A contract can be understood by reading one file and its direct imports; dependencies and what is exposed in the external function table are explicit.

4.5 No Inline Assembly

Vyper excludes inline assembly. For low-level operations, use the *built-in functions*: `raw_call`, `raw_create`, `create_minimal_proxy_to`, `create_from_blueprint`.

Assembly bypasses compiler safety checks: type verification, overflow protection, memory safety, and requires reviewers to reason about raw opcodes. Vyper's built-in functions provide low-level access through explicit, auditable function calls.

4.6 No Function Overloading

Solidity permits multiple functions with the same name and different parameters:

```
function transfer(address to, uint256 amount) public { }
function transfer(address to, uint256 amount, bytes data) public { }
```

Vyper requires unique function names, keeping the ABI and call sites explicit during review.

4.7 No Operator Overloading

`a + b` always performs arithmetic addition. Operators cannot be redefined for custom types, so operator behavior is consistent across the codebase.

4.8 No Infinite Loops

Vyper requires all loops to have a compile-time upper bound:

```
for i: uint256 in range(100):
    # Loop body

# Variable count, but capped at compile time
for i: uint256 in range(count, bound=100):
    # Loop body
```

Unbounded storage iteration can exceed the block gas limit, making contracts unusable. Bounded loops prevent this class of issue.

Note: Vyper's bounded loops and lack of recursion make gas costs statically analyzable—every function call has a calculable upper bound (see *Principles*).

4.9 No Recursion

Functions cannot call themselves, directly or through intermediate functions. Recursive logic must be converted to bounded iteration.

This constraint keeps the call graph acyclic and analyzable at compile time.

4.10 Bounded Dynamic Arrays

Storage arrays require a maximum size at compile time:

```
balances: DynArray[uint256, 100]
```

This keeps gas costs predictable and can prevent denial-of-service attacks. For unbounded collections, use *HashMap*.

4.11 Explicit Type Conversions

Vyper requires explicit type conversions:

```
x: uint256 = 100
y: int256 = convert(x, int256)

addr: address = 0x1234...
num: uint160 = convert(addr, uint160)
```

Vyper allows safe automatic widening (e.g., `uint8` to `uint256`) but requires explicit `convert()` for potentially lossy or semantically significant conversions, such as signed/unsigned, addresses to integers, or narrowing types. See *Types* for the complete type reference.

4.12 Decimal Type

Native base-10 fixed-point arithmetic with 10 fractional digits:

```
a: decimal = 0.1
b: decimal = 0.2
total: decimal = a + b  # exactly 0.3
```

Values like `0.1` and `0.2` cannot be represented exactly in binary floating point, but Vyper's base-10 decimal type handles them precisely.

Solidity lacks a native fixed-point type, requiring manual integer scaling.

4.13 Bounds Checking

Array accesses and arithmetic are bounds-checked at runtime. Out-of-bounds access reverts. Integer overflow reverts.

Solidity 0.8+ provides similar overflow protection, which is disabled in `unchecked` blocks. In Vyper, there is no way to disable the checks. For cases where wrapping behavior is needed, there are explicit *unsafe_* builtins*.

4.14 Reentrancy Protection

Built-in `@nonreentrant` decorator:

```
@external
@nonreentrant
def withdraw():
    # Cannot be re-entered
```

The compiler generates the mutex. No manual reentrancy guard implementation required.

Note: The 2016 DAO hack exploited reentrancy to drain ~$60M in ETH. This led to the Ethereum hard fork that created Ethereum Classic.

The `extcall` keyword makes external call sites explicit and easy to spot during code review. Note that `@nonreentrant` is opt-in and uses a global lock that protects against same-contract reentrancy: if any `@nonreentrant` function is executing, no other `@nonreentrant` function in the same contract can be entered.

Alternatively, `#pragma nonreentrancy` enables reentrancy protection by default for all functions in the contract, so `@nonreentrant` is only needed when not using the pragma. It does not prevent cross-contract reentrancy (i.e., contract A calling contract B which calls back into contract A). See *Control Structures* for details on the lock behavior.

4.15 Syntax Differences

Practical syntax translations for common patterns.

Note: Every Vyper file must start with a version pragma: `#pragma version ^0.4.0`. This is similar to Solidity's `pragma solidity ^0.8.0;` but uses a comment syntax. Vyper files use the `.vy` extension.

4.15.1 State Variables

Solidity:

```
uint256 public counter;
address private owner;
```

Vyper:

```
counter: public(uint256)
owner: address
```

Variables are private by default. Use `public()` to generate a getter.

4.15.2 Functions

Solidity:

```
function deposit() external payable returns (uint256) {
    return msg.value;
}
```

Vyper:

```
@external
@payable
def deposit() -> uint256:
    return msg.value
```

Decorators specify visibility (`@external`, `@internal`) and mutability (`@payable`, `@view`, `@pure`).

4.15.3 Constructor

Solidity:

```
constructor(address _owner) {
    owner = _owner;
}
```

Vyper:

```
@deploy
def __init__(owner: address):
    self.owner = owner
```

The `@deploy` decorator marks the constructor.

4.15.4 Events

Solidity:

```
event Transfer(address indexed from, address indexed to, uint256 value);

function _transfer(address to, uint256 amount) internal {
    emit Transfer(msg.sender, to, amount);
}
```

Vyper:

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

@internal
def _transfer(to: address, amount: uint256):
    log Transfer(msg.sender, to, amount)
```

`log` instead of `emit`.

4.15.5 Mappings

Solidity:

```
mapping(address => uint256) public balances;
mapping(address => mapping(address => uint256)) public allowances;
```

Vyper:

```
balances: public(HashMap[address, uint256])
allowances: public(HashMap[address, HashMap[address, uint256]])
```

4.15.6 Interfaces

Solidity:

```
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
}
```

Vyper (inline declaration):

```
interface IERC20:
    def transfer(to: address, amount: uint256) -> bool: nonpayable
```

Interfaces can also be defined in separate `.vyi` files (see *Interfaces*). Vyper ships with built-in interfaces for ERC20, ERC721, etc. via `from ethereum.ercs import IERC20`.

4.15.7 Error Handling

Solidity:

```
require(amount > 0, "Amount must be positive");
revert("Operation failed");
```

Vyper:

```
assert amount > 0, "Amount must be positive"
raise "Operation failed"
```

`assert` for conditions, `raise` to revert.

4.15.8 Self Reference

State variables require `self.` prefix:

```
self.counter = self.counter + 1
```

Storage access is always explicit. Since storage operations cost more gas than memory operations, this distinction surfaces gas-intensive operations during review.

4.15.9 External Calls

Solidity:

```
IERC20(token).transfer(to, amount);
uint256 balance = IERC20(token).balanceOf(address(this));
```

Vyper:

```
extcall IERC20(token).transfer(to, amount)
balance: uint256 = staticcall IERC20(token).balanceOf(self)
```

`extcall` for state-changing calls, `staticcall` for view/pure functions. The keywords make external calls and potential reentrancy points visible in code.

Note: The `extcall` keyword is required for all state-changing external calls. There is no implicit external call syntax in Vyper: every external call is syntactically marked.

4.15.10 Structs

Solidity:

```
struct Person {
    string name;
    uint256 age;
}

Person public owner;
```

Vyper:

```
struct Person:
    name: String[64]
    age: uint256

owner: public(Person)
```

Note that Vyper strings require explicit maximum length (`String[64]`).

4.15.11 Constants and Immutables

Solidity:

```
uint256 constant FEE = 100;
address immutable owner;

constructor() {
    owner = msg.sender;
}
```

Vyper:

```
FEE: constant(uint256) = 100
owner: immutable(address)

@deploy
def __init__():
    owner = msg.sender
```

`constant` values are inlined at compile time. `immutable` values are set once during deployment and cannot be changed.

4.15.12 Default Function

Solidity:

```
fallback() external payable { }
receive() external payable { }
```

Vyper:

```
@external
@payable
def __default__():
    pass
```

Vyper uses a single `__default__` function for both fallback and receive. It executes when no other function matches or when receiving plain ETH.

4.16 Why Vyper?

Use Vyper if:

- **You have Python experience.** The syntax is familiar.
- **You want compiler-enforced constraints.** The compiler rejects unbounded loops, implicit conversions, and recursive calls.
- **You prefer explicit code.** One way to do most things. No modifiers, no inheritance, no operator overloading.
- **You want no global opt-out for safety checks.** Overflow and bounds checks can only be bypassed per-operation via `unsafe_*` builtins.

CHAPTER

FIVE

VYPER BY EXAMPLE

This section contains practical examples demonstrating Vyper's features and common smart contract patterns. Each example includes the full contract code and a detailed walkthrough of the implementation.

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.

5.1 Simple Open Auction

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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 note that Vyper uses Python-like syntax, making it familiar to Python developers, but it is a distinct language with its own type system, decorators (like `@deploy`, `@external`), and keywords.

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.

```
#pragma version >0.3.10

# 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)

```

(continues on next page)

(continued from previous page)

```
# 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`.
@deploy
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.
# 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)
```

(continues on next page)

(continued from previous page)

```
    # 3. interacting with other contracts
    # If these phases are mixed up, the other contract could call
    # back into the current contract and modify the state or cause
    # effects (Ether payout) to be performed multiple times.
    # If functions called internally include interaction with external
    # contracts, they also have to be considered interaction with
    # external contracts.

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

    # 2. Effects
    self.ended = True

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

As you can see, this example only has a constructor, three 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!

```
# 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])
```

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` to manage the highest bid amount in wei. The variable `ended` is a boolean to determine whether the auction is officially over. The variable `pendingReturns` is a `HashMap` which enables the use of key-value pairs to keep proper track of the auction's 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.

```
# `_bidding_time` seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
@deploy
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 store the beneficiary and auction start time, then compute `self.auctionEnd` by adding `_bidding_time` to `self.auctionStart`. 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, let's look at how our users can make bids.

```
# Bid on the auction with the value sent
# together with this transaction.
# The value will only be refunded if the
# auction is not won.
@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 all three `assert` statements pass, we can safely continue to the next lines; otherwise, the `bid()` method will throw an error and revert the transaction. We then record the previous highest bid in the `pendingReturns` mapping (following the withdrawal pattern for security), and update `highestBid` and `highestBidder` to reflect the new winning bid.

```
# 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)
```

The `withdraw()` method allows previously outbid participants to withdraw their funds. Rather than sending refunds directly during `bid()` (which would allow a malicious contract to block new bids), we use the withdrawal pattern: each bidder pulls their own refund. The method reads the pending amount, zeroes it out (to prevent re-entrancy), and sends the funds.

```
# 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

    # 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.

5.2 Blind Auction

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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

```
#pragma version >0.3.10

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

struct Bid:
  blindedBid: bytes32
  deposit: uint256

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

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

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

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

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

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

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


# Create a blinded auction with `_biddingTime` seconds bidding time and
# `_revealTime` seconds reveal time on behalf of the beneficiary address
# `_beneficiary`.
@deploy
```

(continues on next page)

(continued from previous page)

```
def __init__(_beneficiary: address, _biddingTime: uint256, _revealTime: uint256):
    self.beneficiary = _beneficiary
    self.biddingEnd = block.timestamp + _biddingTime
    self.revealEnd = self.biddingEnd + _revealTime


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

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

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


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

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

    # Place bid successfully and update auction state
    self.highestBid = _value
    self.highestBidder = bidder

```

(continues on next page)

(continued from previous page)

```
    return True


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

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

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

    # Calculate refund for sender
    refund: uint256 = 0
    for i: int128 in range(MAX_BIDS):
        # Note that loop may break sooner than 128 iterations if i >= _numBids
        if (i >= _numBids):
            break

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

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

        # Bid was not actually revealed
        # Do not refund deposit
        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
```

(continues on next page)

(continued from previous page)

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


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

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


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

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

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

```
# 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
```

One difference is that in this example, we use a fixed-size array, limiting 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.

5.3 Safe Remote Purchases

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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 other 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!

```
#pragma version >0.3.10

# 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)
finalized: public(bool)

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

(continues on next page)

(continued from previous page)

```
    self.unlocked = True

@external
def abort():
    assert not self.finalized
    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.
    self.finalized = True
    assert self.balance > 0 and self.balance == 2 * self.value
    send(self.seller, self.balance)

@external
@payable
def purchase():
    assert not self.finalized
    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.finalized
    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
    self.finalized = True

    # 3. Interaction
    send(self.buyer, self.value) # Return the buyer's deposit (=value) to the buyer.
    assert self.balance == 3 * self.value
    send(self.seller, self.balance) # 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)
ended: public(bool)
finalized: 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*.

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

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

```
@external
def abort():
    assert not self.finalized
    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.
    self.finalized = True
    assert self.balance > 0 and self.balance == 2 * self.value
    send(self.seller, self.balance)
```

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 sends the balance back to the seller, effectively canceling the sale.

```
@external
@payable
def purchase():
    assert not self.finalized
    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.finalized
    assert not self.unlocked # Is the item already purchased and pending
```

(continues on next page)

(continued from previous page)

```
                                # confirmation from the buyer?
    assert msg.sender == self.buyer
    assert not self.ended

    # 2. Effects
    self.ended = True
    self.finalized = True

    # 3. Interaction
    send(self.buyer, self.value) # Return the buyer's deposit (=value) to the buyer.
    assert self.balance == 3 * self.value
    send(self.seller, self.balance) # 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, completing the transaction.

5.4 Crowdfund

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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.

```
#pragma version >0.3.10

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

# example of a crowd funding contract

funders: HashMap[address, uint256]
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
timelimit: public(uint256)
finalized: bool

# Setup global variables
@deploy
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
```

(continues on next page)

(continued from previous page)

```
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    assert _goal > 0, "Goal must be non-zero"
    self.goal = _goal

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

    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 not expired yet"
    assert self.balance >= self.goal, "goal has not been reached"
    assert self.balance > 0
    self.finalized = True

    send(self.beneficiary, self.balance)

# Let participants withdraw their fund
@external
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal
    assert not self.finalized
    assert self.funders[msg.sender] > 0

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

    send(msg.sender, value)
```

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

```
funders: HashMap[address, uint256]
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
timelimit: public(uint256)
finalized: bool
```

Like other examples, we begin by initiating our variables. Some variables like `deadline`, `goal` and `timelimit` are declared with the `public` function, making them readable by external callers. Variables without `public` 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.

```
# Setup global variables
@deploy
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    assert _goal > 0, "Goal must be non-zero"
    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 let's 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 has expired"
    assert not self.finalized

    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`. The contribution is added to the `funders` HashMap, which maps each participant's address to their total contribution amount.

```
# Enough money was raised! Send funds to the beneficiary
@external
def finalize():
    assert block.timestamp >= self.deadline, "deadline has not expired yet"
    assert self.balance >= self.goal, "goal has not been reached"
    assert self.balance > 0
    self.finalized = True

    send(self.beneficiary, self.balance)
```

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 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 not self.finalized
    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.

5.5 Voting

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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

```
#pragma version >0.3.10

# 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
```

(continues on next page)

(continued from previous page)

```
22
23  voters: public(HashMap[address, Voter])
24  proposals: public(HashMap[int128, Proposal])
25  voterCount: public(int128)
26  chairperson: public(address)
27  int128Proposals: public(int128)
28
29
30  @view
31  @internal
32  def _delegated(addr: address) -> bool:
33      return self.voters[addr].delegate != empty(address)
34
35
36  @view
37  @external
38  def delegated(addr: address) -> bool:
39      return self._delegated(addr)
40
41
42  @view
43  @internal
44  def _directlyVoted(addr: address) -> bool:
45      return self.voters[addr].voted and (self.voters[addr].delegate == empty(address))
46
47
48  @view
49  @external
50  def directlyVoted(addr: address) -> bool:
51      return self._directlyVoted(addr)
52
53
54  # Setup global variables
55  @deploy
56  def __init__(_proposalNames: bytes32[2]):
57      self.chairperson = msg.sender
58      self.voterCount = 0
59      for i: int128 in range(2):
60          self.proposals[i] = Proposal(
61              name=_proposalNames[i],
62              voteCount=0
63          )
64          self.int128Proposals += 1
65
66  # Give a `voter` the right to vote on this ballot.
67  # This may only be called by the `chairperson`.
68  @external
69  def giveRightToVote(voter: address):
70      # Throws if the sender is not the chairperson.
71      assert msg.sender == self.chairperson
72      # Throws if the voter has already voted.
73      assert not self.voters[voter].voted
```

(continues on next page)

(continued from previous page)

```
    # 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: int128 in range(4):
        if self._delegated(target):
            target = self.voters[target].delegate
            # The following effectively detects cycles of length <= 5,
            # in which the delegation is given back to the delegator.
            # This could be done for any int128ber of loops,
            # or even infinitely with a while loop.
            # However, cycles aren't actually problematic for correctness;
            # they just result in spoiled votes.
            # So, in the production version, this should instead be
            # the responsibility of the contract's client, and this
            # check should be removed.
            assert target != delegate_with_weight_to_forward
        else:
            # Weight will be moved to someone who directly voted or
            # hasn't voted.
            break

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

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

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

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

# 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
```

(continues on next page)

(continued from previous page)

```
    # 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
    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: int128 in range(2):
        if self.proposals[i].voteCount > winning_vote_count:
            winning_vote_count = self.proposals[i].voteCount
            winning_proposal = i
    return winning_proposal

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


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

(continues on next page)

(continued from previous page)

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

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

```
# 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)
```

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 `voteCount`. 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.

```
# Setup global variables
@deploy
def __init__(_proposalNames: bytes32[2]):
    self.chairperson = msg.sender
    self.voterCount = 0
    for i: int128 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.voterCount` 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, let's take a look at the functionality.

```
# 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
```

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 setting the voter's `weight` property. We sequentially check for 3 conditions using `assert`. The `assert not` statement 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`.

```
# 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)
```

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 mark the `msg.sender` as having voted and record the delegate address. Finally, we call `_forwardWeight()` which handles following the chain of delegation and transferring voting weight appropriately.

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

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

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

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. The `@view` decorator indicates that these functions only read contract state and do not modify it. When called externally (not as part of a transaction), view functions do not cost gas.

```
# 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: int128 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()`.

```
# 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
```

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?

5.6 Company Stock

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

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.

```
#pragma version >0.3.10

# Financial events the contract logs

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

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

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

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


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

(continues on next page)

(continued from previous page)

```
totalShares: public(uint256)
price: public(uint256)

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

# Set up the company.
@deploy
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
    self.holdings[msg.sender] += buy_order

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

(continues on next page)

(continued from previous page)

```
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(seller=msg.sender, sell_order=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(sender=msg.sender, receiver=receiver, value=transfer_order)

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

    # Pay the bill!
    send(vendor, amount)

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


```

(continues on next page)

(continued from previous page)

```
# 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.

```
# Financial events the contract logs

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

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

(continues on next page)

(continued from previous page)

```
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.
@deploy
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:
    return self._stockAvailable()
```

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

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. When called externally (not

as part of a transaction), view functions do not cost gas. 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 its holdings. Because `_stockAvailable()` is an internal method, we also include the `stockAvailable()` method to allow external access.

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

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

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

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

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

```
# Find out how much stock any address (that's owned by someone) has.
@view
@internal
def _getHolding(_stockholder: address) -> uint256:
    return self.holdings[_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):
    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(seller=msg.sender, sell_order=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(sender=msg.sender, receiver=receiver, value=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!
```

(continues on next page)

(continued from previous page)

```
    send(vendor, amount)

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

```
# 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
```

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.

5.7 Storage Patterns

> **Warning:** This is example code for learning purposes. Do not use in production without thorough review and testing.

Let's start with a minimal contract that demonstrates state storage. This contract stores a single integer that can be set by anyone.

5.7.1 Simple Storage

```
#pragma version >0.3.10

storedData: public(int128)

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

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

This example shows:

- A public state variable `storedData` with an auto-generated getter
- A constructor (`__init__`) that sets the initial value
- An external function `set()` that modifies state

The `public` modifier on `storedData` automatically creates a getter function, so external contracts can read the value by calling `contract.storedData()`.

5.7.2 Advanced Storage

Building on the simple storage example, this contract adds input validation, events, and a reset function.

```
#pragma version >0.3.10

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

storedData: public(int128)

@deploy
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(setter=msg.sender, value=_x)

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

New concepts introduced:

- **Events**: The `DataChange` event logs who changed the value and what they changed it to. The `indexed` keyword allows filtering by the setter's address.

- **Assertions with messages**: `assert _x >= 0, "No negative values"` reverts with a readable error.
- **Business logic guards**: The contract locks when the stored value reaches 100.

5.8 Name Registry

> **Warning:** This is example code for learning purposes. Do not use in production without thorough review and testing.

A minimal name registry that maps names to addresses. Once a name is registered, it cannot be changed.

```
#pragma version >0.3.10

registry: HashMap[Bytes[100], address]

@external
def register(name: Bytes[100], owner: address):
    assert self.registry[name] == empty(address)  # check name has not been set yet.
    self.registry[name] = owner


@view
@external
def lookup(name: Bytes[100]) -> address:
    return self.registry[name]
```

This pattern is useful for:

- ENS-like name services
- Service discovery
- Any first-come-first-served registration system

The `assert self.registry[name] == empty(address)` check ensures names cannot be overwritten.

5.9 ERC20 Token

> **Warning:** This is example code for learning purposes. Do not use in production without thorough review and testing.

A standard ERC20 fungible token implementation.

```
#pragma version >0.3.10

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

```

(continues on next page)

(continued from previous page)

```
# @dev example implementation of an ERC20 token
# @author Takayuki Jimba (@yudetamago)
# https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

from ethereum.ercs import IERC20
from ethereum.ercs import IERC20Detailed

implements: IERC20
implements: IERC20Detailed

name: public(String[32])
symbol: public(String[32])
decimals: public(uint8)

# NOTE: By declaring `balanceOf` as public, vyper automatically generates a 'balanceOf()' getter
#       method to allow access to account balances.
#       The _KeyType will become a required parameter for the getter and it will return _ValueType.
#       See: https://docs.vyperlang.org/en/v0.1.0-beta.8/types.html?highlight=getter#mappings
balanceOf: public(HashMap[address, uint256])
# By declaring `allowance` as public, vyper automatically generates the `allowance()` getter
allowance: public(HashMap[address, HashMap[address, uint256]])
# By declaring `totalSupply` as public, we automatically create the `totalSupply()` getter
totalSupply: public(uint256)
minter: address


@deploy
def __init__(_name: String[32], _symbol: String[32], _decimals: uint8, _supply: uint256):
    init_supply: uint256 = _supply * 10 ** convert(_decimals, uint256)
    self.name = _name
    self.symbol = _symbol
    self.decimals = _decimals
    self.balanceOf[msg.sender] = init_supply
    self.totalSupply = init_supply
    self.minter = msg.sender
    log IERC20.Transfer(sender=empty(address), receiver=msg.sender, value=init_supply)


@external
def transfer(_to : address, _value : uint256) -> bool:
    """
    @dev Transfer token for a specified address
    @param _to The address to transfer to.
    @param _value The amount to be transferred.
    """
    # NOTE: vyper does not allow underflows
    #       so the following subtraction would revert on insufficient balance
    self.balanceOf[msg.sender] -= _value
```

(continues on next page)

(continued from previous page)

```
    self.balanceOf[_to] += _value
    log IERC20.Transfer(sender=msg.sender, receiver=_to, value=_value)
    return True


@external
def transferFrom(_from : address, _to : address, _value : uint256) -> bool:
    """
     @dev Transfer tokens from one address to another.
     @param _from address The address which you want to send tokens from
     @param _to address The address which you want to transfer to
     @param _value uint256 the amount of tokens to be transferred
    """
    # NOTE: vyper does not allow underflows
    #       so the following subtraction would revert on insufficient balance
    self.balanceOf[_from] -= _value
    self.balanceOf[_to] += _value
    # NOTE: vyper does not allow underflows
    #      so the following subtraction would revert on insufficient allowance
    self.allowance[_from][msg.sender] -= _value
    log IERC20.Transfer(sender=_from, receiver=_to, value=_value)
    return True


@external
def approve(_spender : address, _value : uint256) -> bool:
    """
    @dev Approve the passed address to spend the specified amount of tokens on behalf of␣
↪msg.sender.
         Beware that changing an allowance with this method brings the risk that someone␣
↪may use both the old
         and the new allowance by unfortunate transaction ordering. One possible␣
↪solution to mitigate this
         race condition is to first reduce the spender's allowance to 0 and set the␣
↪desired value afterwards:
         https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    @param _spender The address which will spend the funds.
    @param _value The amount of tokens to be spent.
    """
    self.allowance[msg.sender][_spender] = _value
    log IERC20.Approval(owner=msg.sender, spender=_spender, value=_value)
    return True


@external
def mint(_to: address, _value: uint256):
    """
    @dev Mint an amount of the token and assigns it to an account.
         This encapsulates the modification of balances such that the
         proper events are emitted.
    @param _to The account that will receive the created tokens.
    @param _value The amount that will be created.
```

(continues on next page)

(continued from previous page)

```
    """
    assert msg.sender == self.minter
    assert _to != empty(address)
    self.totalSupply += _value
    self.balanceOf[_to] += _value
    log IERC20.Transfer(sender=empty(address), receiver=_to, value=_value)


@internal
def _burn(_to: address, _value: uint256):
    """
    @dev Internal function that burns an amount of the token of a given
         account.
    @param _to The account whose tokens will be burned.
    @param _value The amount that will be burned.
    """
    assert _to != empty(address)
    self.totalSupply -= _value
    self.balanceOf[_to] -= _value
    log IERC20.Transfer(sender=_to, receiver=empty(address), value=_value)


@external
def burn(_value: uint256):
    """
    @dev Burn an amount of the token of msg.sender.
    @param _value The amount that will be burned.
    """
    self._burn(msg.sender, _value)


@external
def burnFrom(_to: address, _value: uint256):
    """
    @dev Burn an amount of the token from a given account.
    @param _to The account whose tokens will be burned.
    @param _value The amount that will be burned.
    """
    self.allowance[_to][msg.sender] -= _value
    self._burn(_to, _value)
```

Key features:

- Implements the `IERC20` and `IERC20Detailed` interfaces from `ethereum.ercs`
- Standard `transfer`, `transferFrom`, and `approve` functions
- `mint` and `burn` functions for supply management
- Uses `HashMap` for balances and allowances

Note: This is example code. Production tokens require additional security review.

Notice how Vyper's overflow/underflow protection is built-in: the comment "vyper does not allow underflows" explains

why no explicit check is needed when subtracting from balances.

5.10 ERC721 Non-Fungible Token

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

A standard ERC721 (NFT) implementation with minting and burning.

```
#pragma version >0.3.10

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

# @dev example implementation of ERC-721 non-fungible token standard.
# @author Ryuya Nakamura (@nrryuya)
# Modified from: https://github.com/vyperlang/vyper/blob/
↪de74722bf2d8718cca46902be165f9fe0e3641dd/examples/tokens/ERC721.vy

from ethereum.ercs import IERC165
from ethereum.ercs import IERC721

implements: IERC721
implements: IERC165

# Interface for the contract called by safeTransferFrom()
interface ERC721Receiver:
    def onERC721Received(
            _operator: address,
            _from: address,
            _tokenId: uint256,
            _data: Bytes[1024]
        ) -> bytes4: nonpayable


# @dev Mapping from NFT ID to the address that owns it.
idToOwner: HashMap[uint256, address]

# @dev Mapping from NFT ID to approved address.
idToApprovals: HashMap[uint256, address]

# @dev Mapping from owner address to count of his tokens.
ownerToNFTokenCount: HashMap[address, uint256]

# @dev Mapping from owner address to mapping of operator addresses.
ownerToOperators: HashMap[address, HashMap[address, bool]]

# @dev Address of minter, who can mint a token
minter: address
```

(continues on next page)

(continued from previous page)

```
41
42  baseURL: String[53]
43
44  # @dev Static list of supported ERC165 interface ids
45  SUPPORTED_INTERFACES: constant(bytes4[2]) = [
46      # ERC165 interface ID of ERC165
47      0x01ffc9a7,
48      # ERC165 interface ID of ERC721
49      0x80ac58cd,
50  ]
51
52  @deploy
53  def __init__():
54      """
55      @dev Contract constructor.
56      """
57      self.minter = msg.sender
58      self.baseURL = "https://api.babby.xyz/metadata/"
59
60
61  @view
62  @external
63  def supportsInterface(interface_id: bytes4) -> bool:
64      """
65      @dev Interface identification is specified in ERC-165.
66      @param interface_id Id of the interface
67      """
68      return interface_id in SUPPORTED_INTERFACES
69
70
71  ### VIEW FUNCTIONS ###
72
73  @view
74  @external
75  def balanceOf(_owner: address) -> uint256:
76      """
77      @dev Returns the number of NFTs owned by `_owner`.
78           Throws if `_owner` is the zero address. NFTs assigned to the zero address are␣
    ↪considered invalid.
79      @param _owner Address for whom to query the balance.
80      """
81      assert _owner != empty(address)
82      return self.ownerToNFTokenCount[_owner]
83
84
85  @view
86  @external
87  def ownerOf(_tokenId: uint256) -> address:
88      """
89      @dev Returns the address of the owner of the NFT.
90           Throws if `_tokenId` is not a valid NFT.
91      @param _tokenId The identifier for an NFT.
```

(continues on next page)

(continued from previous page)

```
    """
    owner: address = self.idToOwner[_tokenId]
    # Throws if `_tokenId` is not a valid NFT
    assert owner != empty(address)
    return owner


@view
@external
def getApproved(_tokenId: uint256) -> address:
    """
    @dev Get the approved address for a single NFT.
         Throws if `_tokenId` is not a valid NFT.
    @param _tokenId ID of the NFT to query the approval of.
    """
    # Throws if `_tokenId` is not a valid NFT
    assert self.idToOwner[_tokenId] != empty(address)
    return self.idToApprovals[_tokenId]


@view
@external
def isApprovedForAll(_owner: address, _operator: address) -> bool:
    """
    @dev Checks if `_operator` is an approved operator for `_owner`.
    @param _owner The address that owns the NFTs.
    @param _operator The address that acts on behalf of the owner.
    """
    return (self.ownerToOperators[_owner])[_operator]


### TRANSFER FUNCTION HELPERS ###

@view
@internal
def _isApprovedOrOwner(_spender: address, _tokenId: uint256) -> bool:
    """
    @dev Returns whether the given spender can transfer a given token ID
    @param spender address of the spender to query
    @param tokenId uint256 ID of the token to be transferred
    @return bool whether the msg.sender is approved for the given token ID,
        is an operator of the owner, or is the owner of the token
    """
    owner: address = self.idToOwner[_tokenId]
    spenderIsOwner: bool = owner == _spender
    spenderIsApproved: bool = _spender == self.idToApprovals[_tokenId]
    spenderIsApprovedForAll: bool = (self.ownerToOperators[owner])[_spender]
    return (spenderIsOwner or spenderIsApproved) or spenderIsApprovedForAll


@internal
def _addTokenTo(_to: address, _tokenId: uint256):
```

(continues on next page)

(continued from previous page)

```
    """
    @dev Add a NFT to a given address
         Throws if `_tokenId` is owned by someone.
    """
    # Throws if `_tokenId` is owned by someone
    assert self.idToOwner[_tokenId] == empty(address)
    # Change the owner
    self.idToOwner[_tokenId] = _to
    # Change count tracking
    self.ownerToNFTokenCount[_to] += 1


@internal
def _removeTokenFrom(_from: address, _tokenId: uint256):
    """
    @dev Remove a NFT from a given address
         Throws if `_from` is not the current owner.
    """
    # Throws if `_from` is not the current owner
    assert self.idToOwner[_tokenId] == _from
    # Change the owner
    self.idToOwner[_tokenId] = empty(address)
    # Change count tracking
    self.ownerToNFTokenCount[_from] -= 1


@internal
def _clearApproval(_owner: address, _tokenId: uint256):
    """
    @dev Clear an approval of a given address
         Throws if `_owner` is not the current owner.
    """
    # Throws if `_owner` is not the current owner
    assert self.idToOwner[_tokenId] == _owner
    if self.idToApprovals[_tokenId] != empty(address):
        # Reset approvals
        self.idToApprovals[_tokenId] = empty(address)


@internal
def _transferFrom(_from: address, _to: address, _tokenId: uint256, _sender: address):
    """
    @dev Exeute transfer of a NFT.
         Throws unless `msg.sender` is the current owner, an authorized operator, or the␣
↪approved
         address for this NFT. (NOTE: `msg.sender` not allowed in private function so␣
↪pass `_sender`.)
         Throws if `_to` is the zero address.
         Throws if `_from` is not the current owner.
         Throws if `_tokenId` is not a valid NFT.
    """
    # Check requirements
```

(continues on next page)

(continued from previous page)

```
    assert self._isApprovedOrOwner(_sender, _tokenId)
    # Throws if `_to` is the zero address
    assert _to != empty(address)
    # Clear approval. Throws if `_from` is not the current owner
    self._clearApproval(_from, _tokenId)
    # Remove NFT. Throws if `_tokenId` is not a valid NFT
    self._removeTokenFrom(_from, _tokenId)
    # Add NFT
    self._addTokenTo(_to, _tokenId)
    # Log the transfer
    log IERC721.Transfer(sender=_from, receiver=_to, token_id=_tokenId)


### TRANSFER FUNCTIONS ###

@external
@payable
def transferFrom(_from: address, _to: address, _tokenId: uint256):
    """
    @dev Throws unless `msg.sender` is the current owner, an authorized operator, or the approved
         address for this NFT.
         Throws if `_from` is not the current owner.
         Throws if `_to` is the zero address.
         Throws if `_tokenId` is not a valid NFT.
    @notice The caller is responsible to confirm that `_to` is capable of receiving NFTs or else
            they maybe be permanently lost.
    @param _from The current owner of the NFT.
    @param _to The new owner.
    @param _tokenId The NFT to transfer.
    """
    self._transferFrom(_from, _to, _tokenId, msg.sender)


@external
@payable
def safeTransferFrom(
        _from: address,
        _to: address,
        _tokenId: uint256,
        _data: Bytes[1024]=b""
    ):
    """
    @dev Transfers the ownership of an NFT from one address to another address.
         Throws unless `msg.sender` is the current owner, an authorized operator, or the
         approved address for this NFT.
         Throws if `_from` is not the current owner.
         Throws if `_to` is the zero address.
         Throws if `_tokenId` is not a valid NFT.
         If `_to` is a smart contract, it calls `onERC721Received` on `_to` and throws if
         the return value is not `bytes4(keccak256("onERC721Received(address,address,
```

(continues on next page)

(continued from previous page)

```
↪uint256,bytes)"))`.
    @param _from The current owner of the NFT.
    @param _to The new owner.
    @param _tokenId The NFT to transfer.
    @param _data Additional data with no specified format, sent in call to `_to`.
    """
    self._transferFrom(_from, _to, _tokenId, msg.sender)
    if _to.is_contract: # check if `_to` is a contract address
        returnValue: bytes4 = extcall ERC721Receiver(_to).onERC721Received(msg.sender, _
↪from, _tokenId, _data)
        # Throws if transfer destination is a contract which does not implement
↪'onERC721Received'
        assert returnValue == method_id("onERC721Received(address,address,uint256,bytes)
↪", output_type=bytes4)


@external
@payable
def approve(_approved: address, _tokenId: uint256):
    """
    @dev Set or reaffirm the approved address for an NFT. The zero address indicates␣
↪there is no approved address.
         Throws unless `msg.sender` is the current NFT owner, or an authorized operator␣
↪of the current owner.
         Throws if `_tokenId` is not a valid NFT. (NOTE: This is not written the EIP)
         Throws if `_approved` is the current owner. (NOTE: This is not written the EIP)
    @param _approved Address to be approved for the given NFT ID.
    @param _tokenId ID of the token to be approved.
    """
    owner: address = self.idToOwner[_tokenId]
    # Throws if `_tokenId` is not a valid NFT
    assert owner != empty(address)
    # Throws if `_approved` is the current owner
    assert _approved != owner
    # Check requirements
    senderIsOwner: bool = self.idToOwner[_tokenId] == msg.sender
    senderIsApprovedForAll: bool = (self.ownerToOperators[owner])[msg.sender]
    assert (senderIsOwner or senderIsApprovedForAll)
    # Set the approval
    self.idToApprovals[_tokenId] = _approved
    log IERC721.Approval(owner=owner, approved=_approved, token_id=_tokenId)


@external
def setApprovalForAll(_operator: address, _approved: bool):
    """
    @dev Enables or disables approval for a third party ("operator") to manage all of
         `msg.sender`'s assets. It also emits the ApprovalForAll event.
         Throws if `_operator` is the `msg.sender`. (NOTE: This is not written the EIP)
    @notice This works even if sender doesn't own any tokens at the time.
    @param _operator Address to add to the set of authorized operators.
    @param _approved True if the operators is approved, false to revoke approval.
```

(continues on next page)

(continued from previous page)

```
    """
    # Throws if `_operator` is the `msg.sender`
    assert _operator != msg.sender
    self.ownerToOperators[msg.sender][_operator] = _approved
    log IERC721.ApprovalForAll(owner=msg.sender, operator=_operator, approved=_approved)


### MINT & BURN FUNCTIONS ###

@external
def mint(_to: address, _tokenId: uint256) -> bool:
    """
    @dev Function to mint tokens
         Throws if `msg.sender` is not the minter.
         Throws if `_to` is zero address.
         Throws if `_tokenId` is owned by someone.
    @param _to The address that will receive the minted tokens.
    @param _tokenId The token id to mint.
    @return A boolean that indicates if the operation was successful.
    """
    # Throws if `msg.sender` is not the minter
    assert msg.sender == self.minter
    # Throws if `_to` is zero address
    assert _to != empty(address)
    # Add NFT. Throws if `_tokenId` is owned by someone
    self._addTokenTo(_to, _tokenId)
    log IERC721.Transfer(sender=empty(address), receiver=_to, token_id=_tokenId)
    return True


@external
def burn(_tokenId: uint256):
    """
    @dev Burns a specific ERC721 token.
         Throws unless `msg.sender` is the current owner, an authorized operator, or the␣
↪approved
         address for this NFT.
         Throws if `_tokenId` is not a valid NFT.
    @param _tokenId uint256 id of the ERC721 token to be burned.
    """
    # Check requirements
    assert self._isApprovedOrOwner(msg.sender, _tokenId)
    owner: address = self.idToOwner[_tokenId]
    # Throws if `_tokenId` is not a valid NFT
    assert owner != empty(address)
    self._clearApproval(owner, _tokenId)
    self._removeTokenFrom(owner, _tokenId)
    log IERC721.Transfer(sender=owner, receiver=empty(address), token_id=_tokenId)


@view
@external
```

(continues on next page)

(continued from previous page)

```
def tokenURI(tokenId: uint256) -> String[132]:
    return concat(self.baseURL, uint2str(tokenId))
```

This implementation includes:

- `mint` and `burn` functions controlled by a minter address
- `safeTransferFrom` with receiver callback verification
- Operator approval via `setApprovalForAll`
- ERC165 interface detection

The `safeTransferFrom` function checks if the recipient is a contract and, if so, calls `onERC721Received` to ensure the recipient can handle NFTs.

5.11 ERC1155 Multi-Token

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

ERC1155 supports both fungible and non-fungible tokens in a single contract. This implementation includes ownership and pause functionality.

```
#pragma version >0.3.10

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

"""
@dev example implementation of ERC-1155 non-fungible token standard ownable, with approval, OPENSEA compatible (name, symbol)
@author Dr. Pixel (github: @Doc-Pixel)
"""

############### imports ###############
from ethereum.ercs import IERC165

############### variables ###############
# maximum items in a batch call. Set to 128, to be determined what the practical limits are.
BATCH_SIZE: constant(uint256) = 128

# callback number of bytes
CALLBACK_NUMBYTES: constant(uint256) = 4096

# URI length set to 300.
MAX_URI_LENGTH: constant(uint256) = 300
# for uint2str / dynamic URI
MAX_DYNURI_LENGTH: constant(uint256) = 78
```

(continues on next page)

(continued from previous page)

```
# for the .json extension on the URL
MAX_EXTENSION_LENGTH: constant(uint256) = 5

MAX_URL_LENGTH: constant(uint256) = MAX_URI_LENGTH+MAX_DYNURI_LENGTH+MAX_EXTENSION_
↪LENGTH # dynamic URI status
dynamicUri: bool

# the contract owner
# not part of the core spec but a common feature for NFT projects
owner: public(address)

# pause status True / False
# not part of the core spec but a common feature for NFT projects
paused: public(bool)

# the contracts URI to find the metadata
baseuri: String[MAX_URI_LENGTH]
contractURI: public(String[MAX_URI_LENGTH])

# Name and symbol are not part of the ERC1155 standard. For opensea compatibility
name: public(String[128])
symbol: public(String[16])

# Interface IDs
ERC165_INTERFACE_ID: constant(bytes4)  = 0x01ffc9a7
ERC1155_INTERFACE_ID: constant(bytes4) = 0xd9b67a26
ERC1155_INTERFACE_ID_METADATA: constant(bytes4) = 0x0e89341c

# mappings

# Mapping from token ID to account balances
balanceOf: public(HashMap[address, HashMap[uint256, uint256]])

# Mapping from account to operator approvals
isApprovedForAll: public( HashMap[address, HashMap[address, bool]])

############### events ###############
event Paused:
    # Emits a pause event with the address that paused the contract
    account: address

event unPaused:
    # Emits an unpause event with the address that paused the contract
    account: address

event OwnershipTransferred:
    # Emits smart contract ownership transfer from current to new owner
    previousOwner: address
    newOwner: address

event TransferSingle:
    # Emits on transfer of a single token
```

(continues on next page)

(continued from previous page)

```
    operator:   indexed(address)
    fromAddress: indexed(address)
    to: indexed(address)
    id: uint256
    value: uint256

event TransferBatch:
    # Emits on batch transfer of tokens. the ids array correspond with the values array by their position
    operator: indexed(address) # indexed
    fromAddress: indexed(address)
    to: indexed(address)
    ids: DynArray[uint256, BATCH_SIZE]
    values: DynArray[uint256, BATCH_SIZE]

event ApprovalForAll:
    # This emits when an operator is enabled or disabled for an owner. The operator manages all tokens for an owner
    account: indexed(address)
    operator: indexed(address)
    approved: bool

event URI:
    # This emits when the URI gets changed
    value: String[MAX_URI_LENGTH]
    id: indexed(uint256)

############### interfaces ###############
implements: IERC165

interface IERC1155Receiver:
    def onERC1155Received(
       operator: address,
       sender: address,
       id: uint256,
       amount: uint256,
       data: Bytes[CALLBACK_NUMBYTES],
   ) -> bytes32: payable
    def onERC1155BatchReceived(
        operator: address,
        sender: address,
        ids: DynArray[uint256, BATCH_SIZE],
        amounts: DynArray[uint256, BATCH_SIZE],
        data: Bytes[CALLBACK_NUMBYTES],
    ) -> bytes4: payable

interface IERC1155MetadataURI:
    def uri(id: uint256) -> String[MAX_URI_LENGTH]: view

############### functions ###############

@deploy
```

(continues on next page)

(continued from previous page)

```
def __init__(name: String[128], symbol: String[16], uri: String[MAX_URI_LENGTH],␣
↪contractUri: String[MAX_URI_LENGTH]):
    """
    @dev contract initialization on deployment
    @dev will set name and symbol, interfaces, owner and URI
    @dev self.paused will default to false
    @param name the smart contract name
    @param symbol the smart contract symbol
    @param uri the new uri for the contract
    """
    self.name = name
    self.symbol = symbol
    self.owner = msg.sender
    self.baseuri = uri
    self.contractURI = contractUri

## contract status ##
@external
def pause():
    """
    @dev Pause the contract, checks if the caller is the owner and if the contract is␣
↪paused already
    @dev emits a pause event
    @dev not part of the core spec but a common feature for NFT projects
    """
    assert self.owner == msg.sender, "Ownable: caller is not the owner"
    assert not self.paused, "the contract is already paused"
    self.paused = True
    log Paused(account=msg.sender)

@external
def unpause():
    """
    @dev Unpause the contract, checks if the caller is the owner and if the contract is␣
↪paused already
    @dev emits an unpause event
    @dev not part of the core spec but a common feature for NFT projects
    """
    assert self.owner == msg.sender, "Ownable: caller is not the owner"
    assert self.paused, "the contract is not paused"
    self.paused = False
    log unPaused(account=msg.sender)

## ownership ##
@external
def transferOwnership(newOwner: address):
    """
    @dev Transfer the ownership. Checks for contract pause status, current owner and␣
↪prevent transferring to
    @dev zero address
    @dev emits an OwnershipTransferred event with the old and new owner addresses
    @param newOwner The address of the new owner.
```

(continues on next page)

(continued from previous page)

```
    """
    assert not self.paused, "The contract has been paused"
    assert self.owner == msg.sender, "Ownable: caller is not the owner"
    assert newOwner != self.owner, "This account already owns the contract"
    assert newOwner != empty(address), "Transfer to the zero address not allowed. Use␣
↪renounceOwnership() instead."
    oldOwner: address = self.owner
    self.owner = newOwner
    log OwnershipTransferred(previousOwner=oldOwner, newOwner=newOwner)

@external
def renounceOwnership():
    """
    @dev Transfer the ownership to the zero address, this will lock the contract
    @dev emits an OwnershipTransferred event with the old and new zero owner addresses
    """
    assert not self.paused, "The contract has been paused"
    assert self.owner == msg.sender, "Ownable: caller is not the owner"
    oldOwner: address = self.owner
    self.owner = empty(address)
    log OwnershipTransferred(previousOwner=oldOwner, newOwner=empty(address))

@external
@view
def balanceOfBatch(accounts: DynArray[address, BATCH_SIZE], ids: DynArray[uint256, BATCH_
↪SIZE]) -> DynArray[uint256,BATCH_SIZE]:  # uint256[BATCH_SIZE]:
    """
    @dev check the balance for an array of specific IDs and addresses
    @dev will return an array of balances
    @dev Can also be used to check ownership of an ID
    @param accounts a dynamic array of the addresses to check the balance for
    @param ids a dynamic array of the token IDs to check the balance
    """
    assert len(accounts) == len(ids), "ERC1155: accounts and ids length mismatch"
    batchBalances: DynArray[uint256, BATCH_SIZE] = []
    j: uint256 = 0
    for i: uint256 in ids:
        batchBalances.append(self.balanceOf[accounts[j]][i])
        j += 1
    return batchBalances

## mint ##
@external
def mint(receiver: address, id: uint256, amount:uint256):
    """
    @dev mint one new token with a certain ID
    @dev this can be a new token or "topping up" the balance of a non-fungible token ID
    @param receiver the account that will receive the minted token
    @param id the ID of the token
    @param amount of tokens for this ID
    """
    assert not self.paused, "The contract has been paused"
```

(continues on next page)

(continued from previous page)

```
    assert self.owner == msg.sender, "Only the contract owner can mint"
    assert receiver != empty(address), "Can not mint to ZERO ADDRESS"
    operator: address = msg.sender
    self.balanceOf[receiver][id] += amount
    log TransferSingle(operator=operator, fromAddress=empty(address), to=receiver, id=id, value=amount)


@external
def mintBatch(receiver: address, ids: DynArray[uint256, BATCH_SIZE], amounts: DynArray[uint256, BATCH_SIZE]):
    """
    @dev mint a batch of new tokens with the passed IDs
    @dev this can be new tokens or "topping up" the balance of existing non-fungible token IDs in the contract
    @param receiver the account that will receive the minted token
    @param ids array of ids for the tokens
    @param amounts amounts of tokens for each ID in the ids array
    """
    assert not self.paused, "The contract has been paused"
    assert self.owner == msg.sender, "Only the contract owner can mint"
    assert receiver != empty(address), "Can not mint to ZERO ADDRESS"
    assert len(ids) == len(amounts), "ERC1155: ids and amounts length mismatch"
    operator: address = msg.sender

    for i: uint256 in range(BATCH_SIZE):
        if i >= len(ids):
            break
        self.balanceOf[receiver][ids[i]] += amounts[i]

    log TransferBatch(operator=operator, fromAddress=empty(address), to=receiver, ids=ids, values=amounts)

## burn ##
@external
def burn(id: uint256, amount: uint256):
    """
    @dev burn one or more token with a certain ID
    @dev the amount of tokens will be deducted from the holder's balance
    @param id the ID of the token to burn
    @param amount of tokens to burnfor this ID
    """
    assert not self.paused, "The contract has been paused"
    assert self.balanceOf[msg.sender][id] > 0 , "caller does not own this ID"
    self.balanceOf[msg.sender][id] -= amount
    log TransferSingle(operator=msg.sender, fromAddress=msg.sender, to=empty(address), id=id, value=amount)

@external
def burnBatch(ids: DynArray[uint256, BATCH_SIZE], amounts: DynArray[uint256, BATCH_SIZE]):
    """
```

(continues on next page)

(continued from previous page)

```
    @dev burn a batch of tokens with the passed IDs
    @dev this can be burning non fungible tokens or reducing the balance of existing non-fungible token IDs in the contract
    @dev inside the loop ownership will be checked for each token. We can not burn tokens we do not own
    @param ids array of ids for the tokens to burn
    @param amounts array of amounts of tokens for each ID in the ids array
    """
    assert not self.paused, "The contract has been paused"
    assert len(ids) == len(amounts), "ERC1155: ids and amounts length mismatch"
    operator: address = msg.sender

    for i: uint256 in range(BATCH_SIZE):
        if i >= len(ids):
            break
        self.balanceOf[msg.sender][ids[i]] -= amounts[i]

    log TransferBatch(operator=msg.sender, fromAddress=msg.sender, to=empty(address), ids=ids, values=amounts)

## approval ##
@external
def setApprovalForAll(owner: address, operator: address, approved: bool):
    """
    @dev set an operator for a certain NFT owner address
    @param owner the NFT owner address
    @param operator the operator address
    @param approved approve or disapprove
    """
    assert owner == msg.sender, "You can only set operators for your own account"
    assert not self.paused, "The contract has been paused"
    assert owner != operator, "ERC1155: setting approval status for self"
    self.isApprovedForAll[owner][operator] = approved
    log ApprovalForAll(account=owner, operator=operator, approved=approved)

@external
def safeTransferFrom(sender: address, receiver: address, id: uint256, amount: uint256, bytes: bytes32):
    """
    @dev transfer token from one address to another.
    @param sender the sending account (current owner)
    @param receiver the receiving account
    @param id the token id that will be sent
    @param amount the amount of tokens for the specified id
    """
    assert not self.paused, "The contract has been paused"
    assert receiver != empty(address), "ERC1155: transfer to the zero address"
    assert sender != receiver
    assert sender == msg.sender or self.isApprovedForAll[sender][msg.sender], "Caller is neither owner nor approved operator for this ID"
    assert self.balanceOf[sender][id] > 0 , "caller does not own this ID or ZERO balance"
    operator: address = msg.sender
```

(continues on next page)

(continued from previous page)

```
    self.balanceOf[sender][id] -= amount
    self.balanceOf[receiver][id] += amount
    log TransferSingle(operator=operator, fromAddress=sender, to=receiver, id=id, value=amount)

@external
def safeBatchTransferFrom(sender: address, receiver: address, ids: DynArray[uint256, BATCH_SIZE], amounts: DynArray[uint256, BATCH_SIZE], _bytes: bytes32):
    """
    @dev transfer tokens from one address to another.
    @param sender the sending account
    @param receiver the receiving account
    @param ids a dynamic array of the token ids that will be sent
    @param amounts a dynamic array of the amounts for the specified list of ids.
    """
    assert not self.paused, "The contract has been paused"
    assert receiver != empty(address), "ERC1155: transfer to the zero address"
    assert sender != receiver
    assert sender == msg.sender or self.isApprovedForAll[sender][msg.sender], "Caller is neither owner nor approved operator for this ID"
    assert len(ids) == len(amounts), "ERC1155: ids and amounts length mismatch"
    operator: address = msg.sender
    for i: uint256 in range(BATCH_SIZE):
        if i >= len(ids):
            break
        id: uint256 = ids[i]
        amount: uint256 = amounts[i]
        self.balanceOf[sender][id] -= amount
        self.balanceOf[receiver][id] += amount

    log TransferBatch(operator=operator, fromAddress=sender, to=receiver, ids=ids, values=amounts)

# URI #
@external
def setURI(uri: String[MAX_URI_LENGTH]):
    """
    @dev set the URI for the contract
    @param uri the new uri for the contract
    """
    assert not self.paused, "The contract has been paused"
    assert self.baseuri != uri, "new and current URI are identical"
    assert msg.sender == self.owner, "Only the contract owner can update the URI"
    self.baseuri = uri
    log URI(value=uri, id=0)

@external
def toggleDynUri(status: bool):
    """
    @dev toggle dynamic URI
    @param status true for dynamic false for static
    """
```

(continues on next page)

(continued from previous page)

```
    assert msg.sender == self.owner
    assert status != self.dynamicUri, "already in desired state"
    self.dynamicUri = status

@view
@external
def uri(id: uint256) -> String[MAX_URL_LENGTH]:
    """
    @dev retrieve the uri. Adds requested ID when dynamic URI is active
    @param id NFT ID to retrieve the uri for.
    """
    if self.dynamicUri:
        return concat(self.baseuri, uint2str(id), '.json')
    else:
        return self.baseuri

# URI #
@external
def setContractURI(contractUri: String[MAX_URI_LENGTH]):
    """
    @dev set the contractURI for the contract. points to collection metadata file
    @dev This function is opensea specific and is required to properly show collection metadata and image
    @param contractUri the new urcontractUri for the contract
    """
    assert not self.paused, "The contract has been paused"
    assert self.contractURI != contractUri, "new and current URI are identical"
    assert msg.sender == self.owner, "Only the contract owner can update the URI"
    self.contractURI = contractUri
    log URI(value=contractUri, id=0)

@view
@external
def supportsInterface(interfaceId: bytes4) -> bool:
    """
    @dev Returns True if the interface is supported
    @param interfaceId bytes4 interface identifier
    """
    return interfaceId in [
        ERC165_INTERFACE_ID,
        ERC1155_INTERFACE_ID,
        ERC1155_INTERFACE_ID_METADATA,
    ]
```

Features beyond the base ERC1155 standard:

- **Ownable**: Only the owner can mint tokens
- **Pausable**: Owner can pause all transfers
- **Batch operations**: `mintBatch`, `burnBatch`, `safeBatchTransferFrom`
- **Dynamic URI**: Optional per-token metadata URIs

The `BATCH_SIZE` constant (128) limits array sizes for gas predictability—a Vyper requirement.

5.12 ERC4626 Tokenized Vault

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

ERC4626 standardizes yield-bearing vaults. Users deposit assets and receive shares representing their portion of the vault.

```
#pragma version >0.3.10

# NOTE: Copied from https://github.com/fubuloubu/ERC4626/blob/
↪1a10b051928b11eeaad15d80397ed36603c2a49b/contracts/VyperVault.vy

# example implementation of an ERC4626 vault

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

from ethereum.ercs import IERC20
from ethereum.ercs import IERC4626

implements: IERC20
implements: IERC4626

##### ERC20 #####

totalSupply: public(uint256)
balanceOf: public(HashMap[address, uint256])
allowance: public(HashMap[address, HashMap[address, uint256]])

NAME: constant(String[10]) = "Test Vault"
SYMBOL: constant(String[5]) = "vTEST"
DECIMALS: constant(uint8) = 18

##### ERC4626 #####

asset: public(IERC20)


@deploy
def __init__(asset: IERC20):
    self.asset = asset


@view
@external
def name() -> String[10]:
    return NAME


```

(continues on next page)

(continued from previous page)

```
@view
@external
def symbol() -> String[5]:
    return SYMBOL


@view
@external
def decimals() -> uint8:
    return DECIMALS


@external
def transfer(receiver: address, amount: uint256) -> bool:
    self.balanceOf[msg.sender] -= amount
    self.balanceOf[receiver] += amount
    log IERC20.Transfer(sender=msg.sender, receiver=receiver, value=amount)
    return True


@external
def approve(spender: address, amount: uint256) -> bool:
    self.allowance[msg.sender][spender] = amount
    log IERC20.Approval(owner=msg.sender, spender=spender, value=amount)
    return True


@external
def transferFrom(sender: address, receiver: address, amount: uint256) -> bool:
    self.allowance[sender][msg.sender] -= amount
    self.balanceOf[sender] -= amount
    self.balanceOf[receiver] += amount
    log IERC20.Transfer(sender=sender, receiver=receiver, value=amount)
    return True


@view
@external
def totalAssets() -> uint256:
    return staticcall self.asset.balanceOf(self)


@view
@internal
def _convertToAssets(shareAmount: uint256) -> uint256:
    totalSupply: uint256 = self.totalSupply
    if totalSupply == 0:
        return 0

    # NOTE: `shareAmount = 0` is extremely rare case, not optimizing for it
    # NOTE: `totalAssets = 0` is extremely rare case, not optimizing for it
    return shareAmount * staticcall self.asset.balanceOf(self) // totalSupply
```

(continues on next page)

(continued from previous page)

```
95
96
97  @view
98  @external
99  def convertToAssets(shareAmount: uint256) -> uint256:
100     return self._convertToAssets(shareAmount)
101
102
103 @view
104 @internal
105 def _convertToShares(assetAmount: uint256) -> uint256:
106     totalSupply: uint256 = self.totalSupply
107     totalAssets: uint256 = staticcall self.asset.balanceOf(self)
108     if totalAssets == 0 or totalSupply == 0:
109         return assetAmount  # 1:1 price
110
111     # NOTE: `assetAmount = 0` is extremely rare case, not optimizing for it
112     return assetAmount * totalSupply // totalAssets
113
114
115 @view
116 @external
117 def convertToShares(assetAmount: uint256) -> uint256:
118     return self._convertToShares(assetAmount)
119
120
121 @view
122 @external
123 def maxDeposit(owner: address) -> uint256:
124     return max_value(uint256)
125
126
127 @view
128 @external
129 def previewDeposit(assets: uint256) -> uint256:
130     return self._convertToShares(assets)
131
132
133 @external
134 def deposit(assets: uint256, receiver: address=msg.sender) -> uint256:
135     shares: uint256 = self._convertToShares(assets)
136     extcall self.asset.transferFrom(msg.sender, self, assets)
137
138     self.totalSupply += shares
139     self.balanceOf[receiver] += shares
140     log IERC4626.Deposit(sender=msg.sender, owner=receiver, assets=assets, shares=shares)
141     return shares
142
143
144 @view
145 @external
146 def maxMint(owner: address) -> uint256:
```

(continues on next page)

(continued from previous page)

```
    return max_value(uint256)


@view
@external
def previewMint(shares: uint256) -> uint256:
    assets: uint256 = self._convertToAssets(shares)

    # NOTE: Vyper does lazy eval on `and`, so this avoids SLOADs most of the time
    if assets == 0 and staticcall self.asset.balanceOf(self) == 0:
        return shares  # NOTE: Assume 1:1 price if nothing deposited yet

    return assets


@external
def mint(shares: uint256, receiver: address=msg.sender) -> uint256:
    assets: uint256 = self._convertToAssets(shares)

    if assets == 0 and staticcall self.asset.balanceOf(self) == 0:
        assets = shares  # NOTE: Assume 1:1 price if nothing deposited yet

    extcall self.asset.transferFrom(msg.sender, self, assets)

    self.totalSupply += shares
    self.balanceOf[receiver] += shares
    log IERC4626.Deposit(sender=msg.sender, owner=receiver, assets=assets, shares=shares)
    return assets


@view
@external
def maxWithdraw(owner: address) -> uint256:
    return max_value(uint256)  # real max is `self.asset.balanceOf(self)`


@view
@external
def previewWithdraw(assets: uint256) -> uint256:
    shares: uint256 = self._convertToShares(assets)

    # NOTE: Vyper does lazy eval on and, so this avoids SLOADs most of the time
    if shares == assets and self.totalSupply == 0:
        return 0  # NOTE: Nothing to redeem

    return shares


@external
def withdraw(assets: uint256, receiver: address=msg.sender, owner: address=msg.sender) ->
↪ uint256:
    shares: uint256 = self._convertToShares(assets)
```

(continues on next page)

(continued from previous page)

```
    # NOTE: Vyper does lazy eval on `and`, so this avoids SLOADs most of the time
    if shares == assets and self.totalSupply == 0:
        raise  # Nothing to redeem

    if owner != msg.sender:
        self.allowance[owner][msg.sender] -= shares

    self.totalSupply -= shares
    self.balanceOf[owner] -= shares

    extcall self.asset.transfer(receiver, assets)
    log IERC4626.Withdraw(sender=msg.sender, receiver=receiver, owner=owner, assets=assets, shares=shares)
    return shares


@view
@external
def maxRedeem(owner: address) -> uint256:
    return max_value(uint256)  # real max is `self.totalSupply`


@view
@external
def previewRedeem(shares: uint256) -> uint256:
    return self._convertToAssets(shares)


@external
def redeem(shares: uint256, receiver: address=msg.sender, owner: address=msg.sender) -> uint256:
    if owner != msg.sender:
        self.allowance[owner][msg.sender] -= shares

    assets: uint256 = self._convertToAssets(shares)
    self.totalSupply -= shares
    self.balanceOf[owner] -= shares

    extcall self.asset.transfer(receiver, assets)
    log IERC4626.Withdraw(sender=msg.sender, receiver=receiver, owner=owner, assets=assets, shares=shares)
    return assets


@external
def DEBUG_steal_tokens(amount: uint256):
    # NOTE: This is the primary method of mocking share price changes
    # do not put in production code!!!
    extcall self.asset.transfer(msg.sender, amount)
```

The vault implements:

- `deposit` / `withdraw`: Exchange assets for shares
- `mint` / `redeem`: Exchange shares for assets
- Share price calculation based on `totalAssets / totalSupply`

Note: The `DEBUG_steal_tokens` function is for testing share price changes. Do not include in production code.

5.13 On-Chain Market Maker

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

A simple automated market maker (AMM) using the constant product formula (x * y = k).

```
#pragma version >0.3.10

from ethereum.ercs import IERC20


totalEthQty: public(uint256)
totalTokenQty: public(uint256)
# Constant set in `initiate` that's used to calculate
# the amount of ether/tokens that are exchanged
invariant: public(uint256)
token: IERC20
owner: public(address)
finalized: bool

# Sets the on chain market maker with its owner, initial token quantity,
# and initial ether quantity
@external
@payable
def initiate(token_addr: address, token_quantity: uint256):
    assert self.invariant == 0
    self.token = IERC20(token_addr)
    extcall self.token.transferFrom(msg.sender, self, token_quantity)
    self.owner = msg.sender
    self.totalEthQty = msg.value
    self.totalTokenQty = token_quantity
    self.invariant = msg.value * token_quantity
    assert self.invariant > 0

# Sells ether to the contract in exchange for tokens (minus a fee)
@external
@payable
def ethToTokens():
    assert not self.finalized

```

(continues on next page)

(continued from previous page)

```
    fee: uint256 = msg.value // 500
    eth_in_purchase: uint256 = msg.value - fee
    new_total_eth: uint256 = self.totalEthQty + eth_in_purchase
    new_total_tokens: uint256 = self.invariant // new_total_eth
    extcall self.token.transfer(msg.sender, self.totalTokenQty - new_total_tokens)
    self.totalEthQty = new_total_eth
    self.totalTokenQty = new_total_tokens

# Sells tokens to the contract in exchange for ether
@external
def tokensToEth(sell_quantity: uint256):
    assert not self.finalized

    extcall self.token.transferFrom(msg.sender, self, sell_quantity)
    new_total_tokens: uint256 = self.totalTokenQty + sell_quantity
    new_total_eth: uint256 = self.invariant // new_total_tokens
    eth_to_send: uint256 = self.totalEthQty - new_total_eth
    send(msg.sender, eth_to_send)
    self.totalEthQty = new_total_eth
    self.totalTokenQty = new_total_tokens

# Owner can withdraw their funds and stop the exchange
@external
def ownerWithdraw():
    assert self.owner == msg.sender

    self.finalized = True

    extcall self.token.transfer(self.owner, self.totalTokenQty)

    if self.balance > 0:
        send(self.owner, self.balance)
```

How it works:

1. Owner calls `initiate()` with initial ETH and tokens, setting the invariant (k = ETH * tokens)
2. Users swap ETH for tokens via `ethToTokens()`
3. Users swap tokens for ETH via `tokensToEth()`
4. The invariant is maintained: more ETH in = fewer tokens out

The 0.2% fee (`msg.value // 500`) on ETH-to-token swaps goes to the liquidity provider.

Note: Production AMMs need price oracles, slippage protection, and liquidity management. This example demonstrates the core swap mechanism only.

5.14 Factory Pattern

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

The factory pattern deploys and registers multiple contract instances. This example shows a factory that registers exchanges and routes trades between them.

5.14.1 Factory Contract

```
#pragma version >0.3.10

from ethereum.ercs import IERC20

interface Exchange:
    def token() -> IERC20: view
    def receive(_from: address, _amt: uint256): nonpayable
    def transfer(_to: address, _amt: uint256): nonpayable


exchange_codehash: public(bytes32)
# Maps token addresses to exchange addresses
exchanges: public(HashMap[IERC20, Exchange])


@deploy
def __init__(_exchange_codehash: bytes32):
    # Register the exchange code hash during deployment of the factory
    self.exchange_codehash = _exchange_codehash


# NOTE: Could implement fancier upgrade logic around self.exchange_codehash
#       For example, allowing the deployer of this contract to change this
#       value allows them to use a new contract if the old one has an issue.
#       This would trigger a cascade effect across all exchanges that would
#       need to be handled appropriately.


@external
def register():
    # Verify code hash is the exchange's code hash
    assert msg.sender.codehash == self.exchange_codehash
    # Save a lookup for the exchange
    # NOTE: Use exchange's token address because it should be globally unique
    # NOTE: Should do checks that it hasn't already been set,
    #       which has to be rectified with any upgrade strategy.
    exchange: Exchange = Exchange(msg.sender)
    self.exchanges[staticcall exchange.token()] = exchange


```

(continues on next page)

(continued from previous page)

```
@external
def trade(_token1: IERC20, _token2: IERC20, _amt: uint256):
    # Perform a straight exchange of token1 to token 2 (1:1 price)
    # NOTE: Any practical implementation would need to solve the price oracle problem
    extcall self.exchanges[_token1].receive(msg.sender, _amt)
    extcall self.exchanges[_token2].transfer(msg.sender, _amt)
```

5.14.2 Exchange Contract

```
#pragma version >0.3.10

from ethereum.ercs import IERC20


interface Factory:
    def register(): nonpayable


token: public(IERC20)
factory: Factory


@deploy
def __init__(_token: IERC20, _factory: Factory):
    self.token = _token
    self.factory = _factory


@external
def initialize():
    # Anyone can safely call this function because of EXTCODEHASH
    extcall self.factory.register()


# NOTE: This contract restricts trading to only be done by the factory.
#       A practical implementation would probably want counter-pairs
#       and liquidity management features for each exchange pool.


@external
def receive(_from: address, _amt: uint256):
    assert msg.sender == self.factory.address
    success: bool = extcall self.token.transferFrom(_from, self, _amt)
    assert success


@external
def transfer(_to: address, _amt: uint256):
    assert msg.sender == self.factory.address
    success: bool = extcall self.token.transfer(_to, _amt)
    assert success
```

How the pattern works:

1. Deploy the Exchange code and record its `codehash`
2. Deploy the Factory with the exchange codehash
3. Deploy Exchange instances (one per token)
4. Each Exchange calls `factory.register()` to register itself
5. The Factory verifies the caller's codehash matches the expected exchange code
6. Users can now trade between any registered tokens via `factory.trade()`

The `msg.sender.codehash` check ensures only legitimate exchange contracts can register.

5.15 Multi-Signature Wallet

Warning: This is example code for learning purposes. Do not use in production without thorough review and testing.

A multi-signature wallet requiring multiple owner approvals to execute transactions.

```
#pragma version >0.3.10

################################################################################
## THIS IS EXAMPLE CODE, NOT MEANT TO BE USED IN PRODUCTION! CAVEAT EMPTOR!
################################################################################

# An example of how you can implement a wallet in Vyper.

# A list of the owners addresses (there are a maximum of 5 owners)
owners: public(address[5])
# The number of owners required to approve a transaction
threshold: int128
# The number of transactions that have been approved
seq: public(int128)


@deploy
def __init__(_owners: address[5], _threshold: int128):
    for i: uint256 in range(5):
        if _owners[i] != empty(address):
            self.owners[i] = _owners[i]
    self.threshold = _threshold


@external
def testEcrecover(h: bytes32, v:uint8, r:bytes32, s:bytes32) -> address:
    return ecrecover(h, v, r, s)


# `@payable` allows functions to receive ether
```

(continues on next page)

(continued from previous page)

```
@external
@payable
def approve(_seq: int128, to: address, _value: uint256, data: Bytes[4096], sigdata:␣
↪uint256[3][5]) -> Bytes[4096]:
    # Throws if the value sent to the contract is less than the sum of the value to be␣
↪sent
    assert msg.value >= _value
    # Every time the number of approvals starts at 0 (multiple signatures can be added␣
↪through the sigdata argument)
    approvals: int128 = 0
    # Starts by combining:
    # 1) The number of transactions approved thus far.
    # 2) The address the transaction is going to be sent to (can be a contract or a␣
↪user).
    # 3) The value in wei that will be sent with this transaction.
    # 4) The data to be sent with this transaction (usually data is used to deploy␣
↪contracts or to call functions on contracts, but you can put whatever you want in it).
    # Takes the keccak256 hash of the combination
    h: bytes32 = keccak256(concat(convert(_seq, bytes32), convert(to, bytes32), convert(_
↪value, bytes32), data))
    # Then we combine the Ethereum Signed message with our previous hash
    # Owners will have to sign the below message
    h2: bytes32 = keccak256(concat(b"\x19Ethereum Signed Message:\n32", h))
    # Verifies that the caller of approve has entered the correct transaction number
    assert self.seq == _seq
    # # Iterates through all the owners and verifies that there signatures,
    # # given as the sigdata argument are correct
    for i: uint256 in range(5):
        if sigdata[i][0] != 0:
            # If an invalid signature is given for an owner then the contract throws
            assert ecrecover(h2, sigdata[i][0], sigdata[i][1], sigdata[i][2]) == self.
↪owners[i]
            # ecrecover handles multiple types
            assert ecrecover(h2, convert(sigdata[i][0], uint8), convert(sigdata[i][1],␣
↪bytes32), convert(sigdata[i][2], bytes32)) == self.owners[i]
            # For every valid signature increase the number of approvals by 1
            approvals += 1
    # Throw if the number of approvals is less then the number of approvals required␣
↪(the threshold)
    assert approvals >= self.threshold
    # The transaction has been approved
    # Increase the number of approved transactions by 1
    self.seq += 1
    # Use raw_call to send the transaction
    return raw_call(to, data, max_outsize=4096, gas=3_000_000, value=_value)


@external
@payable
def __default__():
    pass
```

Key concepts:

- **Threshold signatures**: Requires `threshold` out of 5 owners to approve
- **Signature verification**: Uses `ecrecover` to verify owner signatures
- **Replay protection**: `seq` counter prevents transaction replay
- **Arbitrary calls**: `raw_call` executes any transaction once approved

The approval process:

1. Owners sign a hash of (sequence number, destination, value, data)
2. Anyone can call `approve()` with the collected signatures
3. If enough valid signatures are provided, the transaction executes

---

Note: This demonstrates signature verification patterns. Production multisigs need additional safeguards like time locks and nonce management.

CHAPTER

SIX

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.

6.1 Pragmas

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

6.1.1 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.

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

```
#pragma version ^0.4.0
```

Note: Both pragma directive versions `#pragma` and `# pragma` are supported.

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

```
# @version ^0.4.0
```

In the above examples, the contract will only compile with Vyper versions `0.4.x`.

6.1.2 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.

6.1.3 EVM Version

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

6.1.4 Experimental Code Generation

The new experimental code generation feature can be activated using the following directive:

```
#pragma experimental-codegen
```

Alternatively, you can use the alias `"venom-experimental"` instead of `"experimental-codegen"` to enable this feature.

6.2 Imports

Import statements allow you to import *Modules* or *Interfaces* with the `import` or `from ... import` syntax.

6.2.1 Imports via `import`

You may import modules (defined in `.vy` files) and interfaces (defined in `.vyi` or `.json` files) via `import` statements. You may use plain or `as` variants.

```
# without an alias
import foo

# with an alias
import my_package.foo as bar
```

6.2.2 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 my_package import foo

# with an alias
from my_package import foo as bar
```

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
```

Further higher directories can be accessed with `...`, `....` etc., as in Python.

6.2.3 Searching For Imports

When looking for a file to import, Vyper will first search relative to the same folder as the contract being compiled. It then checks for the file in the provided search paths, in the precedence provided. Vyper checks for the file name with a `.vy` suffix first, then `.vyi`, then `.json`.

When using the *vyper CLI*, the search path defaults to the current working directory, plus the python syspath. You can append to the search path with the `-p` flag, e.g.:

```
$ vyper my_project/contracts/my_contract.vy -p ../path/to/other_project
```

In the above example, the `my_project` folder is set as the root path.

Note: Including the python syspath on the search path means that any Vyper module in the current `virtualenv` is discoverable by the Vyper compiler, and Vyper packages can be published to and installed from PyPI and accessed via `import` statements with no additional configuration. Keep in mind that best practice is always to install packages *within* a `virtualenv` and not globally!

You can additionally disable the behavior of adding the syspath to the search path with the CLI flag `--disable-sys-path`:

```
$ vyper --disable-sys-path my_project/my_contract.vy
```

When compiling from a *.vyz archive file* or *standard json input*, the search path is already part of the bundle, it cannot be changed from the command line.

6.3 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.

6.4 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.

6.5 Modules

A module is a set of function definitions and variable declarations which enables code reuse. Vyper favors code reuse through composition, rather than inheritance.

Broadly speaking, a module contains:

- function definitions
- state variable declarations
- type definitions

Therefore, a module encapsulates

- functionality (types and functions), and
- state (variables), which may be tightly coupled with that functionality

Modules can be added to contracts by importing them from a `.vy` file. Any `.vy` file is a valid module which can be imported into another contract! This is a very powerful feature which allows you to assemble contracts via other contracts as building blocks.

```
# my_module.vy

def perform_some_computation() -> uint256:
    return 5

@external
def some_external_function() -> uint256:
    return 6
```

```
import my_module

exports: my_module.some_external_function

@external
def foo() -> uint256:
    return my_module.perform_some_computation()
```

Modules are opt-in by design. That is, any operations involving state or exposing external functions must be explicitly opted into using the `exports`, `uses` or `initializes` keywords. See the *Modules* documentation for more information.

6.6 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
```

(continues on next page)

(continued from previous page)

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

See the *Event* documentation for more information.

6.7 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 `.vyi` 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):
    x: uint256 = staticcall FooBar(some_address).calculate()
```

See the *Interfaces* documentation for more information.

6.8 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

SEVEN

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.

Unlike in some other languages, there are no sub-categories of types. Values are always copied, both when assigned to a variable and when passed to a function (also known as call-by-value). This means a calling function never needs to worry about a callee modifying the data of a passed structure.

Note: However neither parameters of *Internal Functions* nor variables are immutable. They can be reassigned to values of the same type. Furthermore, some types (for example arrays and structs) have operations that modify them in place, usually by assigning to their members directly (for example `my_array[0] = 42`).

Parameters of *External Functions* are immutable. They can neither be reassigned nor modified in place.

7.1 Boolean

Keyword: `bool`

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

7.1.1 Values

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

7.1.2 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.

7.2 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).

7.2.1 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.

7.2.2 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`	Integer 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`.

Note: Integer division has different rounding semantics than Python for negative numbers: Vyper rounds towards zero, while Python rounds towards negative infinity. For example, `-1 // 2` will return `-1` in Python, but `0` in Vyper. This preserves the spirit (but not the text!) of the reasoning for Python's round-towards-negative-infinity behavior, which is that the behavior of `//` combined with the behavior of `%` preserves the following identity no matter if the quantities are negative or non-negative: `(x // y) * y + (x % y) == x`.

7.3 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).

7.3.1 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)`.

7.3.2 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`	Integer 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 an unsigned 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`.

7.4 Decimals

Keyword: `decimal`

A decimal is a type to store a decimal fixed point value. As of v0.4.0, decimals must be enabled with the CLI flag `--enable-decimals`.

7.4.1 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 `int168`.

7.4.2 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`	Decimal division
`x % y`	Modulo

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

7.5 Address

Keyword: `address`

The address type holds an Ethereum address.

7.5.1 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`	`uint2`	Balance of an address
`codehash`	`bytes`	Keccak of code at an address, `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a47` if no contract is deployed (see EIP-1052)
`codesize`	`uint2`	Size of code deployed at an address, in bytes
`is_contr`	`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`.

7.6 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
```

7.6.1 Operators

Keyword	Description
`keccak256(x)`	Return the keccak256 hash as bytes32.
`concat(x, ...)`	Concatenate multiple inputs.
`slice(x, start, length)`	Return a slice of `length` bytes starting at `start`.

Where `x` is a byte array and `start` as well as `length` are integer values.

7.7 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 or as hex strings.

```
bytes_string: Bytes[100] = b"\x01"
bytes_string: Bytes[100] = x"01"
```

7.8 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"
```

7.9 Flags

Keyword: `flag`

Flags are custom defined types. A flag 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 a flag with two members
flag Roles:
    ADMIN
    USER

# Declaring a flag variable
role: Roles = Roles.ADMIN

# Returning a member
return Roles.ADMIN
```

7.9.1 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

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

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

```
flag Roles:
    MANAGER
    ADMIN
    USER
```

(continues on next page)

(continued from previous page)

```
# Check for membership
@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 flag objects are simultaneously set, while == checks that two flag objects are bit-for-bit equal.

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

```
@external
def add_user(a: Roles) -> Roles:
    ret: Roles = a
    ret |= Roles.USER  # set the USER bit to 1
    return ret

@external
def revoke_user(a: Roles) -> Roles:
    ret: Roles = a
    ret &= ~Roles.USER  # set the USER bit to 0
    return ret

@external
def flip_user(a: Roles) -> Roles:
    ret: Roles = a
    ret ^= Roles.USER  # flip the user bit between 0 and 1
    return ret
```

7.10 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 flags.

```
# 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

# 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.

7.11 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: uint256 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.

7.12 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
```

7.13 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.

7.14 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.

7.15 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).
- Flags 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

EIGHT

ENVIRONMENT VARIABLES AND CONSTANTS

8.1 Environment Variables

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

8.1.1 Block and Transaction Properties

Name	Type	Value
`block.coinbase`	`address`	Current block miner's address
`block.difficulty`	`uint256`	Current block difficulty
`block.prevrandao`	`bytes32`	Current randomness beacon provided by the beacon chain
`block.number`	`uint256`	Current block number
`block.gaslimit`	`uint256`	Current block's gas limit
`block.basefee`	`uint256`	Current block's base fee
`block.blobbasefee`	`uint256`	Current block's blob gas base fee
`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.mana`	`uint256`	Remaining gas (alias for `msg.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 the `block.difficulty` opcode. 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`*.

8.1.2 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 internal 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)
```

8.2 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)

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

CHAPTER

NINE

STATEMENTS

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

9.1 Control Flow

9.1.1 break

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

```
for i: uint256 in [1, 2, 3, 4, 5]:
    if i == a:
        break
```

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

9.1.2 continue

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

```
for i: uint256 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`.

9.1.3 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
```

9.1.4 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.

9.2 Event Logging

9.2.1 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.

Warning: The evaluation order of arguments passed to `log` is undefined. The compiler may evaluate them in any order. Therefore, arguments with side effects should be evaluated in separate statements before the `log` call to ensure predictable behavior.

9.3 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 happens 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.

9.3.1 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.

Custom errors can also be raised. They are declared at module scope and are encoded with a 4-byte selector followed by ABI-encoded arguments:

```
error Unauthorized:
    caller: address
    expected: address

raise Unauthorized(caller=msg.sender, expected=owner)
```

Custom errors are included in the generated ABI with `type: "error"`.

9.3.2 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

TEN

CONTROL STRUCTURES

10.1 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.

10.1.1 Visibility

You can optionally declare a function's visibility by using a *decorator*. There are three visibility levels in Vyper:

- `@external`: exposed in the selector table, can be called by an external call into this contract
- `@internal` (default): can be invoked only from within this contract. Not available to external callers
- `@deploy`: constructor code. This is code which is invoked once in the lifetime of a contract, upon its deploy. It is not available at runtime to either external callers or internal call invocations. At this time, only the *__init__() function* may be marked as `@deploy`.

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) -> uint256:
    return a + b
```

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

External functions can use the `@raw_return` decorator to return raw bytes without ABI-encoding:

```
@external
@payable
@raw_return
def proxy_call(target: address) -> Bytes[128]:
    # Forward a call and return the raw response without ABI-encoding
    return raw_call(
        target,
        msg.data,
        is_delegate_call=True,
        max_outsize=128,
        value=msg.value
    )
```

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. Consequently, the ABI signature for a function (this includes interface functions) excludes optional arguments when their default values are used in the function call.

```
from ethereum.ercs import IERC4626

@external
def foo(x: IERC4626):
    extcall x.withdraw(0, self, self)   # keccak256("withdraw(uint256,address,address)
↪")[:4] = 0xb460af94
    extcall x.withdraw(0)               # keccak256("withdraw(uint256)")[:4] = 0x2e1a7d4d
```

Internal Functions

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

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

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

Or for internal functions which are defined in *imported modules*, they are invoked by prefixing the name of the module to the function name:

```
import calculator_library

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

Marking an internal function as `payable` specifies that the function can interact with `msg.value`. A `nonpayable` internal function can be called from an external `payable` function, but it cannot access `msg.value`.

```
@payable
def _foo() -> uint256:
    return msg.value % 2
```

Note: As of v0.4.0, the `@internal` decorator is optional. That is, functions with no visibility decorator default to being `internal`.

Note: Please note that for `internal` functions which use more than one default parameter, Vyper versions >=`0.3.8` are recommended due to the security advisory GHSA-ph9x-4vc9-m39g.

10.1.2 The `__init__` Function

The `__init__()` function, also known as the constructor, is a special initialization function that is only called at the time of deploying a contract. It can be used to set initial values for storage or immutable variables. It must be declared with the `@deploy` decorator. A common use case is to set an `owner` variable with the creator of the contract:

```
owner: address

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

Additionally, *immutable variables* may only be set within the constructor.

10.1.3 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` (default): 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 and access Ether via `msg.value`.

```
@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.

Note: The `@nonpayable` decorator is not strictly enforced on `internal` functions when they are invoked through an `external payable` function. As a result, an `external payable` function can invoke an `internal nonpayable` function. However, the `nonpayable internal` function cannot have access to `msg.value`.

10.1.4 Nonreentrancy Locks

The `@nonreentrant` decorator places a global nonreentrancy lock on a function. An attempt by an external contract to call back into any other `@nonreentrant` function causes the transaction to revert.

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

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.

You can put the `@nonreentrant` decorator on a `__default__` function, but keep in mind that this will result in the contract rejecting ETH payments from callbacks.

You can view where the nonreentrant key is physically laid out in storage by using `vyper` with the `-f layout` option (e.g., `vyper -f layout foo.vy`). Unless it is overridden, the compiler will allocate it at slot `0`.

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.

Note: Prior to 0.4.0, nonreentrancy keys took a "key" argument for fine-grained nonreentrancy control. As of 0.4.0, only a global nonreentrancy lock is available.

10.1.5 The nonreentrant pragma

Beginning in 0.4.2, the `#pragma nonreentrancy on` pragma is available, and it enables nonreentrancy on all external functions and public getters (except for `constants` and `immutables`) in the file. This is to prepare for a future release, probably in the 0.5.x series, where nonreentrant locks will be enabled by default language-wide.

When the pragma is on, to re-enable reentrancy for a specific function, add the `@reentrant` decorator. For getters, add the `reentrant()` modifier. Here is an example:

```
# pragma nonreentrancy on

x: public(uint256)  # this is protected from view-only reentrancy
y: public(reentrant(uint256))  # this is not protected from view-only reentrancy

@external
def make_a_call(addr: address):
    # this function is protected from re-entrancy
    ...

@external
@reentrant
def callback(addr: address):
    # this function is allowed to be reentered into
    ...

@external
def __default__():
    # this function is nonreentrant!
    ...
```

The default is `#pragma nonreentrancy off`, which can be used to signal specifically that nonreentrancy protection is off in this file.

Note that the same caveats about nonreentrancy on `__default__()` as mentioned in the previous section apply here, since the `__default__()` function will be nonreentrant by default with the pragma on.

With the pragma on, internal functions remain unlocked by default but can still use the `@nonreentrant` decorator. External `view` functions are protected by default (as before, checking the lock upon entry but only reading its state). External `pure` functions do not interact with the lock.

Internal functions, `__init__` function and getters for `constants` and `immutables` can be marked `reentrant`. Reentrant behavior is the default for these structures anyway, and this feature can be used to explicitly highlight the fact.

Note: All the protected functions share the same, global lock.

Note: Vyper disallows calling a `nonreentrant` function from another `nonreentrant` function, since the compiler implements nonreentrancy as a global lock which is acquired at function entry.

Note: The `nonreentrancy on/off` pragma is scoped to the current file. If you import a file without the `nonreentrancy on` pragma, the functions in that file will behave as the author intended, that is, they will be reentrant unless marked otherwise.

Note: The `constant` and `immutable` state variable getters don't check the lock because the value of the variables can't change.

10.1.6 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 rolls back state changes, 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`).

10.1.7 Decorators Reference

Decorator	Description
`@external`	Function can only be called externally, it is part of the runtime selector table
`@internal`	Function can only be called within current contract
`@deploy`	Function is called only at deploy time
`@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`	Function cannot be called back into during an external call
`@raw_return`	Function returns raw bytes without ABI-encoding (`@external` functions only)
`@abstract`	Function body must be `...`; an `@override` must provide the implementation (see *Abstract Modules*)
`@override(module)`	Function provides the implementation for an `@abstract` function in `module` (see *Abstract Modules*)

10.1.8 Raw Return

The `@raw_return` decorator allows a function to return raw bytes without ABI-encoding. This is particularly useful for proxy contracts and other helper contracts where you want to forward the exact output bytes from another contract call without adding an additional layer of ABI-encoding.

```
@external
@payable
@raw_return
def forward_call(target: address) -> Bytes[1024]:
    # Returns the raw bytes from the external call without ABI-encoding
    return raw_call(target, msg.data, max_outsize=1024, value=msg.value, is_delegate_
↪call=True)
```

The `@raw_return` decorator has the following restrictions:

- It can only be used on `@external` functions
- The function must have a `Bytes[N]` return type
- It cannot be used on `@deploy` (constructor) functions (you can however use it in the `__default__()` function)
- It cannot be used on `@internal` functions

When a function is marked with `@raw_return`, the compiler directly returns the bytes value using the EVM `RETURN` opcode, bypassing the normal ABI-encoding that would wrap the bytes in a `(bytes)` tuple.

Note: The `@raw_return` decorator cannot be used in interface definitions (`.vyi` files). Note that to call a `@raw_return` function from another contract, you should use `raw_call` instead of an interface call, since the return data may not be ABI-encoded.

Warning: When using `@raw_return`, ensure all return paths in your function use raw bytes. Having multiple return statements where some use ABI-encoded data and others don't can lead to decoding errors.

10.2 `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:
    ...
```

10.3 `for` loops

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

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

The iterated value can be a static array, a dynamic array, or generated from the built-in `range` function.

10.3.1 Array Iteration

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

```
foo: int128[3] = [4, 23, 42]
for i: int128 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 the annotated type is valid for each item in the array:

```
for i: int128 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.

10.3.2 Range Iteration

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

```
for i: uint256 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`. `i` must be of the same type as `STOP`.

```
for i: uint256 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. `i`, `stop` and `N` must be of the same type.

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

```
for i: uint256 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`. `i`, `START` and `STOP` must be of the same type.

Finally, it is possible to use `range` with runtime *start* and *stop* values as long as a constant *bound* value is provided. In this case, Vyper checks at runtime that *end* - *start* <= *bound*. `N` must be a compile-time constant. `i`, `stop` and `N` must be of the same type.

```
for i: uint256 in range(start, end, bound=N):
    ...
```

CHAPTER

ELEVEN

SCOPING AND DECLARATIONS

11.1 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

11.1.1 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)`.

11.1.2 Declaring Immutable Variables

Variables can be marked as `immutable` during declaration:

```
DATA: immutable(uint256)

@deploy
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.

11.1.3 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()
```

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

When creating a custom storage layout, you must also include `n_slots` for each storage variable. This tells the compiler how many 32 byte slots to allocate from the `slot` storage offset.

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

11.3 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.

11.3.1 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.

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 bar() -> 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
    a: bool = False
    return a
```

```
a: immutable(bool)

@deploy
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
```

11.3.2 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
```

(continues on next page)

(continued from previous page)

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

11.3.3 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: int128 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: int128 in [1, 2, 3]:
        a: int128 = i
    a += 3
```

CHAPTER

TWELVE

BUILT-IN FUNCTIONS

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

12.1 Bitwise Operations

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.

Note: The functions `bitwise_and`, `bitwise_or`, `bitwise_xor` and `bitwise_not` have been deprecated from version 0.3.4., and removed in version 0.4.2. Please use their operator versions instead: `&`, `|`, `^`, `~`.

12.2 Chain Interaction

Vyper has four built-ins for contract creation; the first 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`
- **`raw_create(initcode: Bytes[...], ...)`**
  - Low-level create. Takes the given initcode, along with the arguments to be abi-encoded, and deploys the initcode after concatenating the abi-encoded arguments.

`create_minimal_proxy_to`(*target: address*, *value: uint256 = 0*, *revert_on_failure: bool = True*[, *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)
- `revert_on_failure`: If `False`, instead of reverting when the create operation fails, return the zero address (Optional, default `True`)
- `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*, *revert_on_failure: bool = True*[, *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)
- `revert_on_failure`: If `False`, instead of reverting when the create operation fails, return the zero address (Optional, default `True`)
- `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 = 3*, *revert_on_failure: bool = True*[, *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 3)
- `revert_on_failure`: If `False`, instead of reverting when the create operation fails, return the zero address (Optional, default `True`)
- `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
```

(continues on next page)

(continued from previous page)

```
    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.

Note: Prior to Vyper version `0.4.0`, the `code_offset` parameter defaulted to `0`.

Warning: It is recommended to deploy blueprints with an ERC-5202 preamble like `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 (and defaults to the ERC-5202 default of 3) 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_create(*initcode: Bytes[...], *args, value: uint256 = 0, revert_on_failure: bool = True[, salt: bytes32]*) → address

Create a contract using the given `initcode`. Provides low-level access to the `CREATE` and `CREATE2` opcodes.

- `initcode`: Initcode bytes
- `value`: The wei value to send to the new contract address (Optional, default 0)
- `*args`: Constructor arguments to forward to the initcode.
- `revert_on_failure`: If `False`, instead of reverting when the create operation fails, return the zero address (Optional, default `True`)
- `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.

```
@external
def foo() -> address:
    # create the bytes of an empty vyper contract
    return raw_create(x
↪"0x61000361000f60003961000360000f35f5ffd855820cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f
↪")
```

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)
    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: 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:** As of the Cancun hardfork (EIP-6780), this opcode no longer deletes contract code unless called in the same transaction as contract creation. It only transfers the contract's ETH balance to the specified address.

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`.

> **Warning:** The `gas` parameter defaults to `0`. When transferring a **non-zero amount of ETH**, the EVM automatically grants the callee a 2300-gas stipend (`GAS_STIPEND`). However, no stipend is added when `value == 0`.
>
> As a result, `send(to, 0)` forwards no gas and will only succeed if the recipient requires no execution gas (for example, an account with no code).

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

12.3 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,
    1932153376655236886094655243748051544141683003977791163791341882495166776176 1,
]
```

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)
```

Alternatively, `v`, `r`, and `s` can be passed as `uint256`:

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

```
>>> ExampleContract.bar(
↪'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
```

12.4 Data Manipulation

concat(*a*, *b*, **args*) → 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: Bytes | bytes32 | String*, *start: uint256*, *length: uint256*) → 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"
```

12.5 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. The rounding mode is to round down to the nearest epsilon. For instance, `sqrt(0.9999999998) == 0.9999999998`.

Note: `sqrt` has been moved to the `math` stdlib module as part of the `0.4.2` release (see PR #4520). See *Math Module*. Import it with `import math` and call `math.sqrt(d)`.

```
import math

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

Note: `isqrt` has been moved to the `math` stdlib module as part of the `0.5.0` release (see PR #4923). See *Math Module*. Import it with `import math` and call `math.isqrt(x)`.

```
import math

@external
@view
def foo(x: uint256) -> uint256:
    return math.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
@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
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).

12.6 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
```

Note: When `as_wei_value` is given some `decimal`, the result might be rounded down to the nearest integer, for example, the following is true: `as_wei_value(12.2, "wei") == 12`.

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
```

blobhash(*index: uint256*) → bytes32

Return the versioned hash of the `index`-th BLOB associated with the current transaction.

Note: A versioned hash consists of a single byte representing the version (currently `0x01`), followed by the last 31 bytes of the `SHA256` hash of the KZG commitment (EIP-4844). For the case `index >= len(tx.blob_versioned_hashes)`, `blobhash(index: uint256)` returns `empty(bytes32)`.

```
@external
@view
def foo(index: uint256) -> bytes32:
    return blobhash(index)
```

```
>>> ExampleContract.foo(0)
0xfd28610fb309939bfec12b6db7c4525446f596a5a5a66b8e2cb510b45b2bbeb5

>>> ExampleContract.foo(6)
0x0000000000000000000000000000000000000000000000000000000000000000
```

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: 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]*) → 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*, *method_id: Bytes[4] = None*) → 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"
```

Note: Prior to v0.4.0, this function was named `_abi_encode`.

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
```

Note: Prior to v0.4.0, this function was named `_abi_decode`.

`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.

Warning: In Vyper, as of v0.4.0, the order of argument evaluation of builtins is not defined. That means that the compiler may choose to reorder evaluation of arguments. For example, `extract32(x(), y())` may yield unexpected results if `x()` and `y()` both touch the same data. For this reason, it is best to avoid calling functions with side-effects inside of builtins. For more information, see GHSA-g2xh-c426-v8mf and issue #4019.

CHAPTER

THIRTEEN

MATH MODULE

The `math` module is a standard library module that provides mathematical functions. Import it with:

```
import math
```

Functions are then called as `math.isqrt(x)`, `math.sqrt(d)`, etc.

Note: `import math` always refers to the standard library module. To import a local file named `math.vy`, use a relative import: `from . import math`.

13.1 Functions

sqrt(*d: decimal*) → decimal

Return the square root of the provided decimal number, using the Babylonian square root algorithm. The rounding mode is to round down to the nearest epsilon. For instance, `math.sqrt(0.9999999998) == 0.9999999998`.

```
import math

@external
@view
def foo(d: decimal) -> decimal:
    return math.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, `math.isqrt(101) == 10`.

```
import math

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

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

CHAPTER

FOURTEEN

MODULES

A module is a set of function definitions and variable declarations which enables code reuse. Vyper favors code reuse through composition, rather than inheritance. A module encapsulates everything needed for code reuse, from type and function declarations to state. It is important to note that functions which make use of defined state must be initialized in order to use that state, whereas functions that are "pure" do not require this.

14.1 Declaring and using modules

The simplest way to define a module is to write a contract. In Vyper, any contract is a valid module! For example, the following contract is also a valid module.

```
# ownable.vy

owner: address

@deploy
def __init__():
    self.owner = msg.sender

def _check_owner():
    assert self.owner == msg.sender

@pure
def _times_two(x: uint256) -> uint256:
    return x * 2

@external
def update_owner(new_owner: address):
    self._check_owner()

    self.owner = new_owner
```

This contract basically has two bits of functionality which can be reused upon import, the `_check_owner()` function and the `update_owner()` function. The `_check_owner()` is an internal function which can be used as a helper to check ownership in importing modules, while the `update_owner()` is an external function which an importing module can itself *export* as an externally facing piece of functionality.

You can use this module's functionality simply by importing it, however any functionality that you do not use from a module will not be included in the final compilation target. For example, if you don't use the `initializes` statement to declare a module's location in the storage layout, you cannot use its state. Similarly, if you don't explicitly `export` an external function from a module, it will not appear in the runtime code.

14.2 Importing a module

A module can be imported using `import` or `from ... import` statements. The following are all equivalent ways to import the above module:

```
import ownable               # accessible as `ownable`
import ownable as ow         # accessible as `ow`
from . import ownable        # accessible as `ownable`
from . import ownable as ow  # accessible as `ow`
```

When importing using the `as` keyword, the module will be referred to by its alias in the rest of the contract.

The `_times_two()` helper function in the above module can be immediately used without any further work since it is "pure" and doesn't depend on initialized state.

```
import ownable as helper
@external
def my_function(x: uint256) -> uint256:
    return helper._times_two(x)
```

The other functions cannot be used yet, because they touch the `ownable` module's state. There are two ways to declare a module so that its state can be used.

14.3 Using a module as an interface

A module can be used as an interface with the `__at__` syntax.

```
import ownable

an_ownable: ownable.__interface__

def call_ownable(addr: address):
    self.an_ownable = ownable.__at__(addr)
    self.an_ownable.transfer_ownership(...)
```

14.4 Initializing a module

In order to use a module's state, it must be "initialized". A module can be initialized with the `initializes` keyword. This declares the module's location in the contract's *Storage Layout*. It also creates a requirement to invoke the module's *__init__() function*, if it has one. This is a well-formedness requirement, since it does not make sense to access a module's state unless its `__init__()` function has been called.

```
import ownable

initializes: ownable

@deploy
def __init__():
    ownable.__init__()
```

(continues on next page)

(continued from previous page)

```
@external
def my_access_controlled_function():
    ownable._check_owner()  # reverts unless msg.sender == ownable.owner

    ... # do things that only the owner can do
```

It is a compile-time error to invoke a module's `__init__()` function more than once!

A module's state can be directly accessed just by prefixing the name of a variable with the module's alias, like follows:

```
@external
def get_owner() -> address:
    return ownable.owner
```

The `initializes` statement is also required when overriding the abstract methods of an *abstract module*.

14.5 The uses statement

Another way of using a contract's state without directly initializing it is to use the `uses` keyword. This is a more advanced usage which is expected to be mostly utilized by library designers. The `uses` statement allows a module to use another module's state but defer its initialization to another module in the compilation tree (most likely a user of the library in question).

This is best illustrated with an example:

```
# ownable_2step.vy
import ownable

uses: ownable

# does not export ownable.transfer_ownership!

pending_owner: address  # the pending owner in the 2-step transfer process

@deploy
def __init__():
    self.pending_owner = empty(address)

@external
def begin_transfer(new_owner: address):
    ownable._check_owner()

    self.pending_owner = new_owner

@external
def accept_transfer():
    assert msg.sender == self.pending_owner

    ownable.owner = self.pending_owner
    self.pending_owner = empty(address)
```

Here, the `ownable_2step` module does not want to seal off access to calling the `ownable` module's `__init__()` function. So, it utilizes the `uses: ownable` statement to get access to the `ownable` module's state, without the re-

quirement to initialize it. Note that this is a valid module, but it is not a valid contract (that is, it cannot produce bytecode) because it does not initialize the `ownable` module. To make a valid contract, the user of the `ownable_2step` module would be responsible for initializing the `ownable` module themselves (as in the next section: *initializing dependencies*).

The `uses` statement is also required when calling abstract methods of another module. See *Calling abstract methods* for details.

Whether to `use` or `initialize` a module is a choice which is left up to the library designer.

14.5.1 Technical notes on the design

This section contains some notes on the design from a language design perspective. It can be safely skipped if you are just interested in how to use modules, and not necessarily in programming language theory.

The design of the module system takes inspiration from (but is not directly related to) the rust language's borrow checker. In the language of type systems, module initialization is modeled as an affine constraint which is promoted to a linear constraint if the module's state is touched in the compilation target. In practice, what this means is:

- A module must be "used" or "initialized" before its state can be accessed in an import
- A module may be "used" many times
- A module which is "used" or its state touched must be "initialized" exactly once

To read more about the design background of Vyper's module system, please see its original design document.

14.6 Initializing a module with dependencies

Sometimes, you may encounter a module which itself `uses` other modules. Vyper's module system is designed to allow this, but it requires you make explicit the access to the imported module's state. The above `ownable_2step.vy` contract is an example of this. If you wanted to initialize the `ownable_2step` module, it would use the special `:=` (aka "walrus") syntax, and look something like this:

```
import ownable
import ownable_2step

initializes: ownable

# ownable is explicitly declared as a state dependency of `ownable_2step`
initializes: ownable_2step[ownable := ownable]

@deploy
def __init__():
    ownable.__init__()
    ownable_2step.__init__()

# export all external functions from ownable_2step
exports: ownable_2step.__interface__
```

Warning: In normal usage, you should make sure that `__init__()` functions are called in dependency order. In the above example, you can get unexpected behavior if `ownable_2step.__init__()` is called before `ownable.__init__()`! The compiler may enforce this behavior in the future.

14.7 Exporting functions

In Vyper, `@external` functions are not automatically exposed (i.e., included in the runtime code) in the importing contract. This is a safety feature, it means that any externally facing functionality must be explicitly defined in the top-level of the compilation target.

So, exporting external functions from modules is accomplished using the `exports` keyword. In Vyper, functions can be exported individually, or, a wholesale export of all the functions in an interface can be done. The special interface `module.__interface__` is a compiler-defined interface, which automatically includes all the functions in a module.

The following are all ways of exporting functions from an imported module.

```
# export a single function from `ownable_2step`
exports: ownable_2step.transfer_ownership

# export multiple functions from `ownable_2step`, being explicit about
# which specific functions are being exported
exports: (
    ownable_2step.transfer_ownership,
    ownable_2step.accept_ownership,
)

# export all IERC20 functions from `base_token`
exports: base_token.IERC20

# export all external functions from `ownable_2step`
exports: ownable_2step.__interface__
```

Note: Any exported interfaces must be implemented by the module. For example, in the above example, `base_token` must contain `implements: IERC20`, or else the compiler will raise an error.

CHAPTER

FIFTEEN

ABSTRACT MODULES

An abstract module is a special kind of *module* which offers points at which its logic can be customized. This takes the form of **abstract methods**, *internal* methods decorated with `@abstract`. These methods can then be overridden to supply the custom logic. Here is an example:

```
# base_token.vy
# This is the abstract module

balances: public(HashMap[address, uint256])

# Abstract method is declared here
@abstract
def _before_transfer(sender: address, recipient: address, amount: uint256):
    ...

@external
def transfer(recipient: address, amount: uint256):
    # and used here
    self._before_transfer(msg.sender, recipient, amount)
    self.balances[msg.sender] -= amount
    self.balances[recipient] += amount
```

The `base_token` module defines a transfer hook, `_before_transfer()`, as an abstract method. It is called during every transfer, but has no implementation — that is left to whoever initializes this module. This lets this module focus on *where* custom logic runs, while the overriding module decides *what* it does.

To supply an implementation, a module imports and `initializes` the abstract module, then provides an `@override` for each abstract method:

```
# pausable_token.vy

import base_token

initializes: base_token

exports: base_token.transfer

paused: public(bool)

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    assert not self.paused, "transfers are paused"
```

Now every call to `base_token.transfer()` will check the pause flag before moving tokens. The override is resolved at compile time — there is no runtime dispatch or inheritance involved.

As you can see in the example above, what makes a module "abstract" is only the presence of abstract methods. Furthermore, that is the only difference, everything that can be done in a concrete (i.e. non-abstract) module can also be done in an abstract module.

15.1 Abstract methods

An abstract method is an *internal method* decorated with `@abstract`. Its body must consist of an Ellipsis literal (`...`) potentially preceded by a docstring and comments.

```
@abstract
def an_abstract_method(x: uint256) -> uint256:
    """This is a docstring"""
    ...
```

There are no other restrictions on abstract methods, they can have any signature, can take any decorator (provided they are valid on internal methods), and be called like concrete methods:

```
@pure
@abstract
def name() -> String[10]:
    ...

@payable
@nonreentrant
@abstract
def foo(bar: DynArray[Bytes[20], 10]) -> DynArray[String[15], 38]:
    ...

def concrete_calls():
    _name: String[10] = self.name()
    complex_expression: String[15] = self.foo([]).pop()
```

15.2 Overriding an abstract module

As abstract modules are by essence incomplete, it is necessary for another module to complete them, by providing implementations for its abstract methods:

```
import base_token

initializes: base_token

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    assert not self.paused, "transfers are paused"
```

For module M1 to override module M2:

1. M1 must *initialize* M2

2. M1 must override each abstract method of M2, see *Overriding abstract methods*

Note: So there is no choice to be made about which override to choose, abstract modules can only be overridden once. This is guaranteed by the initialization system.

15.3 Overriding abstract methods

To override abstract method `_before_transfer` of module `base_token`, the overriding module must define an internal method `_before_transfer` with an `@override(my_abstract_module)` decorator:

```
@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    assert not self.paused, "transfers are paused"
```

Since any call to `base_token._before_transfer` will be replaced at compile-time to a call to the overriding `_before_transfer`, it is necessary that any call to the former is a valid call to the latter:

```
# abstract_m.vy
@pure
@abstract
def compute(x: uint256) -> uint256:
    ...
```

```
import abstract_m
initializes: abstract_m

# Valid: signature matches exactly
@pure
@override(abstract_m)
def compute(x: uint256) -> uint256:
    return x + 1
```

```
import abstract_m
initializes: abstract_m

# Invalid: parameter and return type don't match the ones in abstract_m.compute
@pure
@override(abstract_m)
def compute(x: uint8) -> uint8:
    return x + 1
```

```
import abstract_m
initializes: abstract_m

# Invalid: parameter name doesn't match
@pure
@override(abstract_m)
def compute(y: uint256) -> uint256:
    return y + 1
```

```
import abstract_m
initializes: abstract_m

# Invalid: Has looser mutability: nonpayable vs the original which is pure
@nonpayable
@override(abstract_m)
def compute(y: uint256) -> uint256:
    return y + 1
```

A good heuristic is the following; the override must have:

- The same parameters as the abstract method: same name, same type, and same default value.
- The same return type as the abstract method.
- The same decorators as the abstract method, except `@override` and `@abstract`.

Following the rules above will always result in a valid override. However these might prove too restrictive in your use-case, for this reason the actual rules are more flexible:

- Each parameter of the abstract method must appear in the override at the same position, with the same name.
- If the abstract method defines a default argument for a parameter, the value of the default argument must match in the override. Any value matches an ellipsis (`...`).
- The override may add extra parameters to the right, as long as they have default values.
- The override may add a default value to a parameter that was mandatory in the abstract method.
- Each parameter type in the override must be a super-type of the corresponding parameter type in the abstract method.
- The return type of the override must be a sub-type of the abstract method's return type.
- The override's *mutability* must be the same (or stricter) than the abstract's.
- The override's *reentrancy* must match exactly the one of the abstract method.

15.4 Calling abstract methods

Abstract methods are called in the same way concrete methods are. Additionally, calling the abstract methods of another module requires *using* it.

```
# base_token.vy

@abstract
def _before_transfer(sender: address, recipient: address, amount: uint256):
    ...

@external
def transfer(recipient: address, amount: uint256):
    # call to abstract method in same module
    self._before_transfer(msg.sender, recipient, amount)
    self.balances[msg.sender] -= amount
    self.balances[recipient] += amount
```

```
import base_token

# required by the call below
uses: base_token

def simulate_transfer(sender: address, recipient: address, amount: uint256):
    # call to abstract method in different module
    base_token._before_transfer(sender, recipient, amount)
```

All calls to abstract methods are resolved at compile time to the concrete override: there is no runtime dispatch.

When a module overrides an abstract method, the compiler requires callers to use the most concrete path available. In particular, if `self._before_transfer` overrides `base_token._before_transfer`, any call within that module must go through `self`, not through `base_token`:

```
import base_token

initializes: base_token

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    assert not self.paused, "transfers are paused"

def foo():
    # Valid: uses the override directly
    self._before_transfer(msg.sender, msg.sender, 0)

    # Invalid: base_token._before_transfer is overridden by
    # self._before_transfer, call that instead.
    base_token._before_transfer(msg.sender, msg.sender, 0)
```

15.5 Advanced Uses

In this section we will explain consequences of the above specification which might not jump to mind, and are useful in advanced contexts.

15.5.1 Abstract overriding modules

An overriding module can itself be abstract. In other words, `@abstract` and `@override` can co-exist in the same module (and even *on the same method*):

```
# checked_token.vy

import base_token

initializes: base_token

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    assert self.check_address(sender), "Invalid sender"
```

(continues on next page)

(continued from previous page)

```
    assert self.check_address(recipient), "Invalid recipient"
    assert self.check_cap(amount), "Invalid amount"

@abstract
def check_cap(amount: uint256) -> bool:
    ...

@abstract
def check_address(addr: address) -> bool:
    ...
```

Here `checked_token.vy` provides a concrete `_before_transfer` for `base_token`, but it is itself an abstract module because it introduces `check_cap` and `check_address`. A final module completes the chain by overriding those:

```
# my_token.vy
import checked_token

initializes: checked_token

MAX_AMOUNT: constant(uint256) = 10**24
total_supply: uint256

@override(checked_token)
def check_cap(amount: uint256) -> bool:
    return 0 < amount and amount <= MAX_AMOUNT

@override(checked_token)
def check_address(addr: address) -> bool:
    return addr != empty(address)
```

15.5.2 Overriding multiple modules

A single module can initialize more than one abstract module, providing overrides for each independently. This is how you compose unrelated concerns, for instance combining transfer validation from one module with fee configuration from another.

Using `base_token` from the *earlier example*, consider a second abstract module that manages access control:

```
# access_control.vy

@abstract
def _is_allowed(user: address) -> bool:
    ...

def check_allowed(user: address):
    assert self._is_allowed(user), "access denied"
```

A contract can initialize both modules and override each abstract method:

```
# my_token.vy

import base_token
```

(continues on next page)

(continued from previous page)

```
import access_control

initializes: base_token
initializes: access_control

exports: base_token.transfer

allowed: HashMap[address, bool]

@override(access_control)
def _is_allowed(user: address) -> bool:
    return self.allowed[user]

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    access_control.check_allowed(sender)
```

The overriding module weaves both concerns together: `_before_transfer` delegates to the access control module to gate who can send tokens.

15.5.3 Overriding multiple abstract methods

When a module overrides multiple others, it might happen that the abstract modules share a method name. To handle this case, it is possible for a single method to override multiple, by adding multiple `@override` decorators. Note however that the method must be a *valid override* for every abstract method:

```
# minter.vy

import my_roles

@abstract
def get_role(user: address) -> my_roles.ROLE:
    ...
```

```
# authentication_provider.vy

import my_roles

@abstract
def get_role(user: address, default_role: my_roles.ROLE) -> my_roles.ROLE:
    ...
```

```
import minter
import authentication_provider
import my_roles
initializes: minter
initializes: authentication_provider

roles: HashMap[address, my_roles.ROLE]

# By having `default_role` as an optional parameter, it's a valid override of both
```

(continues on next page)

(continued from previous page)

```
# `minter.get_role` which does not have that parameter, and
# `authentication_provider.get_role` which has it as a mandatory parameter

@override(minter)
@override(authentication_provider)
def get_role(user: address, default_role: my_roles.ROLE = empty(my_roles.ROLE)) -> my_
↪roles.ROLE:
    role: my_roles.ROLE = self.roles[user]
    if role == empty(my_roles.ROLE):
        return default_role
    return role
```

15.5.4 Deferring overriding

In some cases a module might not want to override every method of another, if such is the case, `@abstract` and `@override` can be combined to defer providing an implementation:

```
# base.vy

@abstract
def method_a() -> uint256:
    ...

@abstract
def method_b() -> uint256:
    ...
```

```
# middle.vy
import base

initializes: base

# Concrete override
@override(base)
def method_a() -> uint256:
    return 0

# Delegates
@abstract
@override(base)
def method_b() -> uint256:
    ...
```

```
# top.vy
import middle

initializes: middle

@override(middle)
def method_b() -> uint256:
    return 42
```

Any call to `base.method_b` resolves to a call to the concrete implementation: `top.method_b`.

15.5.5 Default Implementation

It's sometimes desirable to provide a default implementation for an abstract method, this can be done by defining a separate method which holds this logic:

```
# base_token.vy

@abstract
def _before_transfer(sender: address, recipient: address, amount: uint256):
    ...

def _before_transfer_default(sender: address, recipient: address, amount: uint256):
    assert sender != empty(address), "transfer from zero address"
    assert recipient != empty(address), "transfer to zero address"
```

This allows accepting the default implementation downstream as:

```
# simple_token.vy

import base_token

initializes: base_token

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):
    base_token._before_transfer_default(sender, recipient, amount)
```

And even allows using the default implementation while adding extra logic:

```
# pausable_token.vy

import base_token

initializes: base_token

paused: public(bool)

@override(base_token)
def _before_transfer(sender: address, recipient: address, amount: uint256):

    # Use default check
    base_token._before_transfer_default(sender, recipient, amount)

    # Add custom logic
    assert not self.paused, "transfers are paused"
```

Recommendations:

1. The default implementation should have the same name as the abstract method followed by `_default`.
2. The default implementation should not be called anywhere (except in the override), as this defeats the purpose of making the method abstract in the first place.

CHAPTER

SIXTEEN

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.

16.1 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):
    extcall foobar.test1()

@external
def test2(foobar: FooBar) -> uint256:
    return staticcall 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

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

@external
def test():
    extcall self.foobar_contract.test1()
```

Specifying `payable` or `nonpayable` annotation in the interface indicates that the call made to the external contract will be able to alter storage, whereas `view` and `pure` calls 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.

Either the `extcall` or `staticcall` keyword is required to precede the external call to distinguish it from internal calls. The keyword must match the visibility of the function, `staticcall` for `pure` and `view` functions, and `extcall` for `payable` and `nonpayable` functions. Additionally, the output of a `staticcall` must be assigned to a result.

Warning: If the signature in an interface does not match the actual signature of the called contract, you can get runtime errors or undefined behavior. For instance, if you accidentally mark a `nonpayable` function as `view`, calling that function may result in the EVM reverting execution in the called contract.

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

@external
def test(foobar: FooBar):
    s: uint256 = staticcall foobar.calculate()  # cannot change storage
    s = staticcall foobar.query()  # cannot change storage, but reads itself
    extcall foobar.update()  # storage can be altered
    extcall 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` check (but keep `RETURNDATASIZE` check)
`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:

```
extcall IERC20(USDT).transfer(msg.sender, 1, default_return_value=True) # returns True
extcall IERC20(USDT).transfer(msg.sender, 1) # reverts because nothing returned
```

16.2 Built-in Interfaces

Vyper includes common built-in interfaces such as IERC20 and IERC721. These are imported from `ethereum.ercs`:

```
from ethereum.ercs import IERC20

implements: IERC20
```

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

16.3 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.vyi` (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.

Multiple `implements` statements can be grouped into one:

```
implements: Foo
implements: Bar

# Equivalent to:

implements: (
    Foo,
    Bar,
)
```

Note: Functions involving length-bounded types (`Bytes`, `DynArray`, `String`) follow standard variance rules:

- **Return types are covariant**: if the interface declares `String[10]`, the implementation's declared maximum must be **at most** 10 (it may be narrower).
- **Parameter types are contravariant**: if the interface declares `Bytes[10]`, the implementation's declared maximum must be **at least** 10 (it may be wider).

For example, given the interface:

```
interface IFoo:
    def foo(x: Bytes[10]) -> String[10]: view
```

the following implementation is accepted:

```
implements: IFoo

@external
@view
def foo(x: Bytes[20]) -> String[5]:  # wider parameter, narrower return
    return ""
```

Matching bounds are also valid: an implementation declared as `def foo(x: Bytes[10]) -> String[10]` satisfies the same interface.

Think of a function as a funnel. To swap it for another, the input has to be at least as wide (the new function must accept everything the original did, and may accept more), and the output has to be no wider (whatever it produces must still fit where the original output went). A narrower input would let some valid arguments fall through, and a wider output would exceed what the caller is prepared to receive.

To declare "any length" in an interface, use the wildcard `...` with one of the length-bounded types: `Bytes[...]`, `DynArray[uint256, ...]`, or `String[...]`. The wildcard is only valid inside interface declarations and only for these types; static arrays (e.g. `uint256[5]`) still require a concrete length.

At the call site, a wildcard in a return type resolves against the type Vyper infers from context (e.g. an assignment target). If there is no such type (for example when the return value is discarded), it resolves to an unbounded length.

Because `...` leaves the actual bound up to the implementation, the call-site type check accepts arguments of any declared length. If the value the caller passes is longer than the implementation's declared bound, the callee reverts when decoding the argument. For example, given an interface `def foo(x: String[...])` and an implementation `def foo(x: String[2])` at address `addr`, calling `IFoo(addr).foo("hello")` passes the call-site type check but reverts at the callee, because the value's length (5) exceeds the implementation's bound (2). A concrete bound in the interface (e.g. `String[5]`) is checked at compile time instead. The compiler rejects arguments whose declared type allows lengths greater than 5 before the contract is deployed. In this sense `...` is less safe than a concrete bound, where "safe" means "program might revert unexpectedly", not "program is vulnerable to attackers".

Note: Prior to v0.4.0, `implements` required that events defined in an interface were re-defined in the "implementing" contract. As of v0.4.0, this is no longer required because events can be used just by importing them. Any events used in a contract will automatically be exported in the ABI output.

Note: An interface function with default parameters (e.g. `deposit(assets: uint256, receiver: address = msg.sender)`) implies that the contract being interfaced with supports these default arguments via the ABI-encoded function signatures (e.g. `keccak256("deposit(uint256,address)")[:4]` and `keccak256("deposit(uint256)")[:4]`). It is the responsibility of the callee to implement the behavior associated with these defaults.

16.4 Standalone Interfaces

Standalone interfaces are written using a variant of standard Vyper syntax. The body of each function must be an ellipsis (`...`). Interface files must have a `.vyi` suffix in order to be found by an import statement.

```
# ISomeInterface.vyi

@external
def test1():
    ...

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

16.5 Extracting Interfaces

Vyper has a built-in format option to allow you to easily export a Vyper interface from a pre-existing contract.

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

# Functions

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

# ...
```

If you want to export it as an inline interface, Vyper provides a utility to extract that as well.

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

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

The output can then easily be copy-pasted directly in a regular vyper file.

CHAPTER

SEVENTEEN

EVENT LOGGING

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

17.1 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
@external
def transfer(_to : address, _value : uint256) -> bool:

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

   # All done, log the event for listeners
   log Transfer(sender=msg.sender, receiver=_to, value=_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);
```

(continues on next page)

(continued from previous page)

```
var myToken = MyToken.at("0x1234...ab67" /* address */);

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

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

17.2 Declaring Events

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

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

The EVM currently has five opcodes for emitting event logs: `LOG0`, `LOG1`, `LOG2`, `LOG3`, and `LOG4`. These opcodes can be used to create log records, where each log record consists of both **topics** and **data**. Topics are 32-byte ''words" that are used to describe what is happening in an event. While topics are searchable, data is not. Event data is however not limited, which means that you can include large or complicated data like arrays or strings. Different opcodes (`LOG0` through `LOG4`) allow for different numbers of topics. For instance, `LOG1` includes one topic, `LOG2` includes two topics, and so on. 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 (topics), 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 up to four indexed arguments (`LOG4`), 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 (data), which are passed through to listeners. You can have any number of value arguments and they can have arbitrary names.

Note that the first topic of a log record consists of the signature of the name of the event that occurred, including the types of its parameters. It is also possible to create an event with no arguments. In this case, use the `pass` statement:

```
event Foo: pass
```

17.3 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 are stored in transaction logs rather than contract state storage, making them significantly cheaper than storage operations. However, the drawback is that events are not available to contracts, only to clients.

Logging events is done using the `log` statement:

```
log Transfer(sender=msg.sender, receiver=_to, value=_value)
```

The types of arguments given must match those used when declaring the event. When using keyword arguments (as shown above), the order does not matter.

17.4 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

EIGHTEEN

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).

18.1 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
    """
```

18.2 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*.

18.3 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
```

18.3.1 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.

18.3.2 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

NINETEEN

COMPILING A CONTRACT

19.1 Command-Line Compiler Tools

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

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

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

19.1.1 vyper

`vyper` provides CLI 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,abi_python,bytecode,bytecode_runtime,blueprint_bytecode,cfg,cfg_runtime,
↪interface,external_interface,ast,annotated_ast,integrity,ir,ir_json,ir_runtime,asm,
↪opcodes,opcodes_runtime,source_map,source_map_runtime,archive,solc_json,method_
↪identifiers,userdoc,devdoc,metadata,combined_json,layout yourFileName.vy
```

Note: The `opcodes` and `opcodes_runtime` output of the compiler has been returning incorrect opcodes since `0.2.0` due to a lack of 0 padding (patched via PR 3735). If you rely on these functions for debugging, please use the latest patched versions.

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 Imports* 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.

19.1.2 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.

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

19.2 Online Compilers

19.2.1 Try VyperLang!

Try VyperLang! is a JupyterHub 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.

19.2.2 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.

19.3 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.

19.4 Enabling Experimental Code Generation

When compiling, you can use the CLI flag `--experimental-codegen` (or its alias `--venom-experimental`) to activate the new Venom IR. Venom IR is inspired by LLVM IR and enables new advanced analysis and optimizations.

19.5 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 “prague” fork of the EVM.

```
#pragma evm-version prague
```

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]"
    }
}
```

19.5.1 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. In general, the compiler team maintains an informal policy that the compiler will support 3 years of hard fork rulesets, but this policy may be revisited as appropriate.

london

paris

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

shanghai

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

cancun

- 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.

prague(*default*)

19.6 Controlling Warnings

Vyper allows suppression of warnings via the CLI flag `-Wnone`, or promotion of (all) warnings to errors via the `-Werror` flag.

```
$ vyper -Wnone foo.vy   # suppress warnings
```

```
$ vyper -Werror foo.vy   # promote warnings to errors
```

19.7 Integrity Hash

To help tooling detect whether two builds are the same, Vyper provides the `-f integrity` output, which outputs the integrity hash of a contract. The integrity hash is recursively defined as the sha256 of the source code with the integrity hashes of its dependencies (imports) and storage layout overrides (if provided).

19.8 Vyper Archives

A Vyper archive is a compileable bundle of input sources and settings. Technically, it is a ZIP file, with a special structure to make it useable as input to the compiler. It can use any suffix, but the convention is to use a `.zip` suffix or `.vyz` suffix. It must contain a `MANIFEST/` folder, with the following directory structure.

```
MANIFEST
├── cli_settings.txt
├── compilation_targets
├── compiler_version
├── integrity
├── settings.json
├── searchpaths
└── storage_layout.json [OPTIONAL]
```

- `cli_settings.txt` is a text representation of the settings that were used on the compilation run that generated this archive.
- `compilation_targets` is a newline separated list of compilation targets. Currently only one compilation is supported
- `compiler_version` is a text representation of the compiler version used to generate this archive
- `integrity` is the *integrity hash* of the input contract
- `searchpaths` is a newline-separated list of the search paths used on this compilation run
- `settings.json` is a json representation of the settings used on this compilation run. It is 1:1 with `cli_settings.txt`, but both are provided as they are convenient for different workflows (typically, manually vs automated).
- `storage_layout.json` is a json representation of the storage layout overrides to be used on this compilation run. It is optional.

A Vyper archive file can be produced by requesting the `-f archive` output format. The compiler can also produce the archive in base64 encoded form using the `--base64` flag. The Vyper compiler can accept both `.vyz` and base64-encoded Vyper archives directly as input.

```
$ vyper -f archive my_contract.vy -o my_contract.vyz  # write the archive to my_contract.
↪vyz
$ vyper -f archive my_contract.vy --base64 > my_contract.vyz.b64  # write the archive,␣
↪as base64-encoded text
$ vyper my_contract.vyz  # compile my_contract.vyz
$ vyper my_contract.vyz.b64  # compile my_contract.vyz.b64
```

19.9 Compiler Input and Output JSON Description

JSON input/output is provided for compatibility with solidity, however, the recommended way is to use the aforementioned *Vyper archives*. So-called "standard json" input can be generated from a contract using the `vyper -f solc_json` output format.

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

19.9.1 Input JSON Description

The following example describes the expected input format of `vyper-json`. (Comments are not normally permitted in JSON and are used here 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": {
            // 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
    // Sources given here are made available for import by the contracts
    // that are compiled. If the suffix is ".vy", the compiler will expect
    // Vyper syntax. If the suffix is "abi" the compiler will expect an
    // ABI object.
    "interfaces": {
        "contracts/bar.vy": {
            "content": ""
        },
        "contracts/baz.json": {
            "abi": []
        }
    },
    // Optional
    // Storage layout overrides for the contracts that are compiled
    "storage_layout_overrides": {
        "contracts/foo.vy": {
            "a": {"type": "uint256", "slot": 1, "n_slots": 1},
            "b": {"type": "uint256", "slot": 0, "n_slots": 1},
        }
    },
    // Required
    "settings": {
        "evmVersion": "prague",  // EVM version to compile for. Can be london, paris,␣
↪shanghai, cancun or prague (default).
        // optional, optimization mode
```

(continues on next page)

(continued from previous page)

```
        // 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,
        // optional, whether to use the experimental venom pipeline
        // defaults to false
        "experimentalCodegen": false,
        // the search paths to use for resolving imports
        "search_paths": [],
        // 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.bytecode.sourceMap - Source mapping (useful for debugging)
        //    evm.deployedBytecode.object - Deployed bytecode object
        //    evm.deployedBytecode.opcodes - Deployed opcodes list
        //    evm.deployedBytecode.sourceMap - Deployed source mapping (useful for␣
↪debugging)
        //    evm.methodIdentifiers - The list of function hashes
        //    layout - Storage layout of the contract
        //
        // Using `evm`, `evm.bytecode`, etc. will select every target part of that output.
        // Additionally, `*` can be used as a wildcard to request everything.
        // Note that the sourceMapFull.pc_ast_map is the recommended source map to use;
        // the other types are included for legacy and compatibility reasons.
        //
        "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
        }
    }
}
```

19.9.2 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-0.4.0",
    // 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"
        }
    ],
    // Optional: not present if there are no storage layout overrides
    "storage_layout_overrides": {
        "contracts/foo.vy": {
            "a": {"type": "uint256", "slot": 1, "n_slots": 1},
            "b": {"type": "uint256", "slot": 0, "n_slots": 1},
        }
    },
    // This contains the file-level outputs. Can be limited/filtered by the␣
↪outputSelection settings.
    "sources": {
        "source_file.vy": {
            // Identifier of the source (used in source maps)
            "id": 0,
            // The AST object
            "ast": {},
        }
    },
    // This contains the contract-level outputs. Can be limited/filtered by the␣
↪outputSelection settings.
    "contracts": {
        "source_file.vy": {
            // The contract name will always be the file name without a suffix
            "source_file": {
```

(continues on next page)

(continued from previous page)

```
                // The Ethereum Contract ABI.
                // See https://docs.soliditylang.org/en/latest/abi-spec.html
                "abi": [],
                // Natspec developer documentation
                "devdoc": {},
                // Intermediate representation (the IR node tree as a JSON object)
                "ir": {},
                // Natspec user documentation
                "userdoc": {},
                // Storage layout of the contract
                "layout": {
                    "storage_layout": {
                        "variableName": {
                            "type": "uint256",
                            "slot": 0,
                            "n_slots": 1
                        }
                    }
                },
                // EVM-related outputs
                "evm": {
                    "bytecode": {
                        // The bytecode as a hex string.
                        "object": "00fe",
                        // Opcodes list (string)
                        "opcodes": "",
                        // The creation source mapping.
                        "sourceMap": {
                            "breakpoints": [],
                            "error_map": {},
                            "pc_ast_map": {},
                            "pc_ast_map_item_keys": [],
                            "pc_breakpoints": [],
                            "pc_jump_map": {},
                            "pc_pos_map": {},
                            // The creation source mapping as a string.
                            "pc_pos_map_compressed": ""
                        }
                    },
                    "deployedBytecode": {
                        // The deployed bytecode as a hex string.
                        "object": "00fe",
                        // Deployed opcodes list (string)
                        "opcodes": "",
                        // The deployed source mapping.
                        "sourceMap": {
                            "breakpoints": [],
                            "error_map": {},
                            "pc_ast_map": {},
                            "pc_ast_map_item_keys": [],
                            "pc_breakpoints": [],
                            "pc_jump_map": {},
```

(continues on next page)

(continued from previous page)

```
                        "pc_pos_map": {},
                        // The deployed source mapping as a string.
                        "pc_pos_map_compressed": ""
                    }
                },
                // 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

TWENTY

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.

```
#pragma enable-decimals

@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`.

```
@external
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 incompatible types.

```
#pragma enable-decimals

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

    if baz + bar > 4:
        pass
```

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

exception UndeclaredDefinition

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

exception VariableDeclarationException

Raises on an invalid variable declaration.

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

exception VersionException

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

exception ZeroDivisionException

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

20.1 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

TWENTYONE

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 deploy it through geth or another Ethereum client:

```
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 Titanoboa:

```
import boa
boa.set_network_env(<RPC URL>)
from eth_account import Account
# in a real codebase, always load private keys safely from an encrypted store!
boa.env.add_account(Account(<a private key>))
deployer = boa.load_partial("yourFileName.vy")
deployer.deploy()
```

- Use the development environment provided at https://try.vyperlang.org to compile and deploy your contract on your net of choice. try.vyperlang.org comes "batteries-included", with Titanoboa pre-installed, and browser signer integration as well.

CHAPTER

TWENTYTWO

TESTING A CONTRACT

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

22.1 Titanoboa

Titanoboa is a Vyper interpreter maintained by the Vyper team. It provides:

- Fast execution for testing
- Pretty tracebacks for debugging
- Forking capabilities
- Deployment features

Getting Started:

- Official Titanoboa Documentation

Note: For comprehensive examples and best practices, refer to the official Titanoboa documentation linked above.

22.2 Moccasin

Moccasin is a fast, Pythonic smart contract testing and development framework built on top of Titanoboa. It provides:

- ZKsync built-in support
- Named contracts for cleaner address management
- Custom pytest markers for staging tests
- Encrypted wallet support (no private keys in `.env` files)
- GitHub and Python dependency installation

Getting Started:

- Official Moccasin Documentation

CHAPTER

TWENTYTHREE

OTHER RESOURCES AND LEARNING MATERIAL

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

23.1 General

- Cyfrin Updraft - Python & Vyper by Cyfrin
- Ape Academy – Learn how to build Vyper projects by ApeWorX
- More Vyper by Example by Smart Contract Engineer
- Vyper greatest hits smart contract examples
- A curated list of Vyper resources, libraries, tools, and more

23.2 Frameworks and tooling

- Titanoboa – A Vyper interpreter with pretty tracebacks, forking, debugging features and more
- Moccasin - A smart contract development and testing framework for Vyper and Python, built on Titanoboa
- ApeWorX – The Ethereum development framework for Python Developers, Data Scientists, and Security Professionals
- VyperDeployer – A helper smart contract to compile and test Vyper contracts in Foundry
- snekmate – Vyper smart contract building blocks
- Smart contract development frameworks and tools for Vyper on Ethereum.org

23.3 Security

- VyperPunk – learn to secure and hack Vyper smart contracts

23.4 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)

23.5 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)
- Foundry x Vyper – Foundry template to compile Vyper contracts
- Vyper Hub for development (not maintained since 2021)

CHAPTER

TWENTYFOUR

DEEP VERIFICATION

Formal verification is becoming substantially more practical for smart contracts. Better tooling and accessible machine-checked semantics now make it feasible to prove properties on production code that were previously out of reach.

However, formal verification only proves that a program satisfies a specified property *within its formal model*. In practice, most verification work today establishes that property at only one layer of the stack. Everything else (from the accuracy and completeness of the model itself, through the compiler, to the EVM execution semantics), is taken on trust.

This distance between what is formally proved and what actually executes on chain is the **verification gap**.

Deep verification is the discipline of reducing that gap by formally bridging as many layers as possible. The fewer unverified assumptions remain in the chain, the stronger the resulting guarantee.

24.1 Verification Depth and Breadth

Two concepts help clarify the verification gap:

- **Verification depth** — how far the proof carries through the stack, from a high-level security property through source semantics and compiler correctness down to the EVM execution model.
- **Verification breadth** — how much of the contract's actual behavior is covered: which functions, which properties, which sequences of operations, and which environment assumptions.

Every verification effort has some gap. Deep verification is the discipline of reducing it to its irreducible minimum.

24.2 The Layers

A source-level proof establishes a property under the language's formal semantics. A bytecode-level proof establishes the analogous statement against a formal EVM model. A verified compiler connects the two. The fewer layers a guarantee skips by assumption, the smaller the verification gap.

24.3 The Technical Foundation

Two of those layers already exist as public, machine-checked artifacts developed in the Verifereum project using HOL4:

- A formal semantics for the Vyper source language.
- A formal semantics for the EVM.

Together they provide the mathematical foundation against which source-level and bytecode-level proofs about Vyper programs can be stated. The work is fully open and available in the vyper-hol repository.

24.4 Current Status

Source-level verification on Vyper contracts and libraries is available today. Compiler verification (the critical link that connects source semantics to deployed bytecode) is in active development, with substantial proof engineering already underway.

No complete deep verification pipeline exists yet for any widely deployed smart-contract language. Vyper is one of the places where the pieces are actively being assembled, and the stack is already in place for meaningful source-level work.

See also:

vyper-hol on GitHub
: The open-source repository containing the formal semantics for Vyper and the EVM.

Verifereum project
: The project developing the HOL4 semantics and compiler verification work.

CHAPTER

TWENTYFIVE

RELEASE NOTES

Release audits can be found here.

25.1 v0.4.3 ("Buttermilk Racer")

25.1.1 Date released: 2025-06-19

v0.4.3 introduces the `@raw_return` decorator which allows contracts to return bytes directly without ABI-encoding, which enables new proxy contract use cases. The default EVM version has been updated to `prague`, and several improvements have been made to the Venom optimizer pipeline.

Audits: ChainSecurity, Anatomist

Breaking changes

- feat[tool]!: make `prague` the default evm version (#4633)

Other new features and improvements

- feat[lang]: `@raw_return` decorator (#4568)
- fix[lang]: disallow `@raw_return` in interfaces (#4700)

Tooling / CLI

- fix[tool]: fix invalid quotes in `-f cfg` output (#4672)
- fix[tool]: add metadata for ctor functions (#4668)

Venom improvements

- fix[venom]: fix incorrect write reordering in `DFTPass` (#4695)
- feat[venom]: rewrite grammar to LALR(1) (#4687)
- fix[venom]: reduce single-use expansion (#4667)
- fix[venom]: fix function inliner `clone` function (#4696)

Docs

- feat[docs]: add docs for `@raw_return` decorator (#4699)
- fix[docs]: clarify `n_slots` requirement in custom layout file (#4641)
- fix[docs]: fix build status in README.md (#4680)
- chore[docs]: document the copy maxbound heuristic reasoning (#4593)

Test suite and CI improvements

- refactor[test]: rename venom `param` instruction in tests (#4689)
- chore[ci]: update gitignore (#4690)

25.2 v0.4.2 ("Lernaean Hydra")

25.2.1 Date released: 2025-05-31

v0.4.2 includes a new `raw_create()` builtin which allows users to build more generic factories in Vyper. It also moves the `sqrt()` builtin to a pure Vyper module, involving a refactor which will allow more stdlib functionality to be written in Vyper in the future.

Additionally, Venom has undergone more improvements, including a CSE elimination pass, dead-store elimination pass, as well as moving more items in the calling convention to the stack in the venom pipeline. Benchmark contracts are now typically 5% smaller.

Two low severity GHSAs have been patched in this release.

Audits: ChainSecurity, Anatomist

Breaking and notable changes

- feat[lang]!: move sqrt to new stdlib `math` module (#4520)
- fix[lang]!: ban calling nonreentrant functions from nonreentrant functions (#4574)
- fix[lang]!: forbid calling `__default__` (#4371)
- feat[lang]!: remove deprecated bitwise builtins (#4552)
- feat[tool]!: rename `--venom` to `--venom-experimental` (#4662)
- refactor[tool]!: update storage layout format (#4495)

Other new features and improvements

- feat[lang]: nonreentrancy by default (#4563)
- feat[lang]: add `raw_create()` builtin (#4204)
- feat[lang]: remove one-warning limit from builtins (#4542)
- feat[lang]: enable bitwise ops for `bytesM` types (#4538)
- fix[lang]: extend `as_wei_value` to all numeric types (#3498)

- feat[lang]: bubble up create revertdata (#4540)
- fix[codegen]: fix overcopying of bytes in `make_setter` (#4419)
- fix[ux]: don't warn for logs with 0 positional args (#4501)

Tooling / CLI

- feat[tool]: add settings dict to `combined_json` output (#4541)
- fix[tool]: fix output bundle construction (#4654)
- feat[tool]: add venom artifacts into `solc_json` output (#4637)

Bugfixes

- fix[codegen]: disallow `slice()` with length 0 for ad-hoc locations (#4645)
- fix[codegen]: fix bytes copying routines for 0-length case (#4649)
- fix[codegen]: fix removal of side effects in concat (#4644)
- fix[codegen]: interleaved effects eval for some builtins (#4156)
- fix[lang]: block modules in structs (#4566)
- fix[lang]: filter oob array access during folding (#4571)
- fix[lang]: disallow some builtins in `pure` functions (#3157)
- fix[lang]: only reserve builtins at the top level (#4578)
- fix[tool]: fix layout export with nonreentrancy pragma on (#4621)
- fix[lang]: disallow staticcall in pure context (#4619)
- fix[lang]: disallow duplicate getter annotations (#4623)
- fix[lang]: disable nonreentrant behavior of immutable and constant getters (#4622)
- fix[lang]: fix invalid memory read in `raw_create` (#4624)
- fix[parser]: block value assignment in for targets (#4492)
- fix[ux]: fold `keccak` and `sha256` of constant hexbytes (#4536)
- fix[ux]: typechecking for loop annotation of list variable (#4550)

Patched security advisories (GHSAs)

- `concat()` builtin may elide side-effects for zero-length arguments (GHSA-vgf2-gvx8-xwc3)
- `slice()` may elide side-effects when output length is 0 (GHSA-4w26-8p97-f4jp)

Venom improvements

- feat[venom]: add dead store elimination pass (#4556)
- feat[venom]: generalize DSE to more locations (#4652)
- feat[venom]: improve memmerge pass (#4422)
- feat[venom]: implement new calling convention (#4482)
- perf[venom]: improve CSE elimination speed (#4607)
- feat[venom]: disable legacy optimizer in venom pipeline (#4411)
- feat[venom]: mark loads as non-volatile (#4388)
- feat[venom]: improve `dload`/`mstore` merging (#4570)
- refactor[venom]: add dom walk property (#4546)
- feat[venom]: add memory SSA analysis (#4555)
- feat[venom]: improvements for iszero handling and memmerge (#4469)
- refactor[venom]: update more passes to use `InstUpdater` (#4516)
- fix[venom]: fix var defined check for unreachable blocks (#4518)
- feat[venom]: tune function inliner (#4584)
- feat[venom]: add assert optimizer (#4585)
- fix[venom]: revert-to-assert should invalidate dfg (#4586)
- feat[venom]: add common subexpression elimination (#4241)
- fix[venom]: fix `.name` invalidation in `MakeSSA` (#4545)
- refactor[venom]: add annotation to `append_instruction` (#4583)
- fix[venom]: fix handling of params in `venom_to_assembly` (#4587)
- refactor[venom]: extract liveness and cfg data structures (#4595)
- fix[venom]: fix `DominatorTreeAnalysis.dominates()` (#4615)
- fix[venom]: fix callsite phis after inlining (#4666)
- refactor[venom]: rename “store”-related passes (#4627)
- feat[venom]: add phi simplification pass (#4628)
- feat[venom]: add `calloca` instruction (#4376)
- feat[venom]: allow SCCP to run without removing allocas (#4655)
- feat[venom]: improve phi elimination pass (#4635)
- feat[venom]: allow labels to be assigned to variables (#4514)
- perf[venom]: optimize time spent in `SimplifyCFG` (#4658)
- feat[venom]: remove `dload` from list of volatile instructions (#4659)
- perf[venom]: improve performance of `MakeSSA` (#4491)
- feat[test]: add tests for venom dload lowering pass (#4471)
- feat[venom]: add varname freshener (#4484)
- refactor[test]: add more tests with hevm venom harness (#4493)

- feat[venom]: add basic semantic check machinery to venom (#4483)
- fix[venom]: fix sccp resolution of truthy `jnz` (#4505)
- feat[test]: add hevm to `get_contract` harness (#4499)
- feat[venom]: parse hex literals in text format (#4532)
- refactor[venom]: simplify `SimplifyCFG` pass (#4528)
- refactor[venom]: use `InstUpdater` in more passes (#4508)
- feat[venom]: make `revert` a bb terminator (#4529)

Docs

- fix[docs]: update `skip_contract_check` docs (#4511)
- fix[docs]: fix venom examples (#4475)
- chore[docs]: remove dead link from internal documentation (#4543)
- feat[docs]: document order of evalution of arguments of `log` (#4617)
- fix[docs]: fix code block rst formatting (#4618)

Test suite and CI improvements

- feat[lang]: remove `@external` decorator from builtin interfaces (#4562)
- fix[test]: fix warnings in thirdparty tests (#4547)
- chore[tool]: widen version bounds for `packaging` (#4590)
- refactor[test]: remove `selfdestruct` from example contracts (#4537)
- fix[ci]: suppress hypothesis health check (#4533)
- feat[test]: add more example contracts (#4500)
- feat[test]: add negative hevm tests (#4504)
- chore[tool]: widen version bounds for `asttokens` (#4592)
- chore[test]: add test for struct member names (#3527)

Misc / Refactor

- refactor[parser]: refactor pragma parsing (#4530)
- refactor[parser]: put settings on Module AST node (#4569)
- chore[lang]: remove `sha3` and `importlib_metadata` imports (#4588)
- refactor[stdlib]: refactor `math.sqrt` implementation (#4575)
- refactor[codegen]: refactor `get_type_for_exact_size()` (#4632)
- refactor[lang]: refactor decorator parsing (#4490)

25.3 v0.4.1 ("Tokara Habu")

25.3.1 Date released: 2025-03-01

v0.4.1 is primarily a polishing release, focusing on bug fixes, UX improvements, and security-related fixes (with four low-to-moderate severity GHSA reports published). However, a substantial amount of effort has also been invested in improving the Venom pipeline, resulting in better performance and code generation from the Venom pipeline. Venom can be enabled by passing the `--venom` or `--experimental-codegen` flag to the Vyper compiler (they are aliases of each other). Venom code can now also be compiled directly, using the `venom` binary (included in this release).

Breaking changes

- feat[lang]!: make `@external` modifier optional in `.vyi` files (#4178)
- feat[codegen]!: check `returndatasize` even when `skip_contract_check` is set (#4148)
- fix[stdlib]!: fix `IERC4626` signatures (#4425)
- fix[lang]!: disallow absolute relative imports (#4268)

Other new features and improvements

- feat[lang]: add `module.__at__()` to cast to interface (#4090)
- feat[lang]: use keyword arguments for event instantiation (#4257)
- feat[lang]: add native hex string literals (#4271)
- feat[lang]: introduce `mana` as an alias for `gas` (#3713)
- feat[lang]: support top level `"abi"` key in json interfaces (#4279)
- feat[lang]: support flags from imported interfaces (#4253)
- feat[ux]: allow "compiling" `.vyi` files (#4290)
- feat[ux]: improve hint for events kwarg upgrade (#4275)

Tooling / CLI

- feat[tool]: add `-Werror` and `-Wnone` options (#4447)
- feat[tool]: support storage layouts via `json` and `.vyz` inputs (#4370)
- feat[tool]: add integrity hash to initcode (#4234)
- fix[ci]: fix commithash calculation for pypi release (#4309)
- fix[tool]: include structs in `-f interface` output (#4294)
- feat[tool]: separate import resolution pass (#4229)
- feat[tool]: add all imported modules to `-f annotated_ast` output (#4209)
- fix[tool]: add missing internal functions to metadata (#4328)
- fix[tool]: update VarAccess pickle implementation (#4270)
- fix[tool]: fix output formats for .vyz files (#4338)
- fix[tool]: add missing user errors to error map (#4286)

- fix[ci]: fix README encoding in `setup.py` (#4348)
- refactor[tool]: refactor `compile_from_zip()` (#4366)

Bugfixes

- fix[lang]: add `raw_log()` constancy check (#4201)
- fix[lang]: use folded node for typechecking (#4365)
- fix[ux]: fix error message for "staticall" typo (#4438)
- fix[lang]: fix certain varinfo comparisons (#4164)
- fix[codegen]: fix `abi_encode` buffer size in external calls (#4202)
- fix[lang]: fix == and != bytesM folding (#4254)
- fix[lang]: fix `.vyi` function body check (#4177)
- fix[venom]: invalid jump error (#4214)
- fix[lang]: fix precedence in floordiv hint (#4203)
- fix[lang]: define rounding mode for sqrt (#4486)
- fix[codegen]: disable augassign with overlap (#4487)
- fix[codegen]: relax the filter for augassign oob check (#4497)
- fix[lang]: fix panic in call cycle detection (#4200)
- fix[tool]: update `InterfaceT.__str__` implementation (#4205)
- fix[tool]: fix classification of AST nodes (#4210)
- fix[tool]: keep `experimentalCodegen` blank in standard json input (#4216)
- fix[ux]: fix relpath compiler panic on windows (#4228)
- fix[ux]: fix empty hints in error messages (#4351)
- fix[ux]: fix validation for `abi_encode()` `method_id` kwarg (#4369)
- fix[ux]: fix false positive for overflow in type checker (#4385)
- fix[ux]: add missing filename to syntax exceptions (#4343)
- fix[ux]: improve error message on failed imports (#4409)
- fix[parser]: fix bad tokenization of hex strings (#4406)
- fix[lang]: fix encoding of string literals (#3091)
- fix[codegen]: fix assertions for certain precompiles (#4451)
- fix[lang]: allow `print()` schema larger than 32 bytes (#4456)
- fix[codegen]: fix iteration over constant literals (#4462)
- fix[codegen]: fix gas usage of iterators (#4485)
- fix[codegen]: cache result of iter eval (#4488)
- fix[lang]: fix recursive interface imports (#4303)
- fix[tool]: roll back OS used to build binaries (#4494)

Patched security advisories (GHSAs)

- success of certain precompiles not checked (GHSA-vgf2-gvx8-xwc3)
- AugAssign evaluation order causing OOB write within object (GHSA-4w26-8p97-f4jp)
- `sqrt` doesn't define rounding behavior (GHSA-2p94-8669-xg86)
- multiple eval in `for` list iterator (GHSA-h33q-mhmp-8p67)

Venom improvements

- feat[venom]: add venom parser (#4381)
- feat[venom]: new `DFTPass` algorithm (#4255)
- feat[venom]: only `stack_reorder` before join points (#4247)
- feat[venom]: add function inliner (#4478)
- feat[venom]: add binop optimizations (#4281)
- feat[venom]: offset instruction (#4180)
- feat[venom]: make dft-pass commutative aware (#4358)
- perf[venom]: add `OrderedSet.last()` (#4236)
- feat[venom]: improve liveness computation time (#4086)
- fix[venom]: fix invalid `phi`s after SCCP (#4181)
- fix[venom]: clean up sccp pass (#4261)
- refactor[venom]: remove `dup_requirements` analysis (#4262)
- fix[venom]: remove duplicate volatile instructions (#4263)
- fix[venom]: fix `_stack_reorder()` routine (#4220)
- feat[venom]: store expansion pass (#4068)
- feat[venom]: add effects to instructions (#4264)
- feat[venom]: add small heuristic for cleaning input stack (#4251)
- refactor[venom]: refactor module structure (#4295)
- refactor[venom]: refactor sccp pass to use dfg (#4329)
- refactor[venom]: update translator for `deploy` instruction (#4318)
- feat[venom]: make cfg scheduler "stack aware" (#4356)
- feat[venom]: improve liveness computation (#4330)
- refactor[venom]: optimize lattice evaluation (#4368)
- perf[venom]: improve OrderedSet operations (#4246)
- fix[venom]: promote additional memory locations to variables (#4039)
- feat[venom]: add codesize optimization pass (#4333)
- fix[venom]: fix unused variables pass (#4259)
- refactor[venom]: move commutative instruction set (#4307)

- fix[venom]: add `make_ssa` pass after algebraic optimizations (#4292)
- feat[venom]: reduce legacy opts when venom is enabled (#4336)
- fix[venom]: fix duplicate allocas (#4321)
- fix[venom]: add missing extcodesize+hash effects (#4373)
- refactor[ux]: add `venom` as `experimental-codegen` alias (#4337)
- feat[venom]: allow alphanumeric variables and source comments (#4403)
- feat[venom]: cleanup variable version handling (#4404)
- feat[venom]: merge memory writes (#4341)
- refactor[venom]: make venom repr parseable (#4402)
- feat[venom]: propagate `dload` instruction to venom (#4410)
- feat[venom]: remove special cases in store elimination (#4413)
- feat[venom]: update text format for data section (#4414)
- feat[venom]: add load elimination pass (#4265)
- fix[venom]: fix `MakeSSA` with existing phis (#4423)
- refactor[venom]: refactor mem2var (#4421)
- fix[venom]: fix store elimination pass (#4428)
- refactor[venom]: add `make_nop()` helper function (#4470)
- feat[venom]: improve load elimination (#4407)
- refactor[venom]: replace `bb.mark_for_removal` with `make_nop` (#4474)

Docs

- chore[docs]: add `method_id` to `abi_encode` signature (#4355)
- chore[docs]: mention the `--venom` flag in venom docs (#4353)
- feat[docs]: add bug bounty program to security policy (#4230)
- feat[docs]: add installation via pipx and uv (#4274)
- chore[docs]: add binary installation methods (#4258)
- chore[docs]: update `sourceMap` field descriptions (#4170)
- chore[docs]: remove experimental note for cancun (#4183)
- chore[venom]: expand venom docs (#4314)
- chore[docs]: abi function signature for default arguments (#4415)
- feat[docs]: add Telegram badge to README.md (#4342)
- chore[docs]: update readme about testing (#4448)
- chore[docs]: `nonpayable internal` function behaviour (#4416)
- chore[docs]: add `FUNDING.json` for drips funding (#4167)
- chore[docs]: add giveth to `FUNDING.yml` (#4466)
- chore[tool]: update `FUNDING.json` for optimism RPGF (#4218)

- chore[tool]: mention that output format is comma separated (#4467)

Test suite improvements

- refactor[venom]: add new venom test machinery (#4401)
- feat[ci]: use `coverage combine` to reduce codecov uploads (#4452)
- feat[test]: add hevm harness for venom passes (#4460)
- fix[test]: fix test in grammar fuzzer (#4150)
- chore[test]: fix a type hint (#4173)
- chore[ci]: add auto-labeling workflow (#4276)
- fix[test]: fix some clamper tests (#4300)
- refactor[test]: add some sanity checks to `abi_decode` tests (#4096)
- chore[ci]: enable Python `3.13` tests (#4386)
- chore[ci]: update codecov github action to v5 (#4437)
- chore[ci]: bump upload-artifact action to v4 (#4445)
- chore[ci]: separate codecov upload into separate job (#4455)
- chore[ci]: improve coverage jobs (#4457)
- chore[ci]: update ubuntu image for `build` job (#4473)

Misc / Refactor

- refactor[parser]: remove `ASTTokens` (#4364)
- refactor[codegen]: remove redundant `IRnode.from_list` (#4151)
- feat[ux]: move exception hint to the end of the message (#4154)
- fix[ux]: improve error message for bad hex literals (#4244)
- refactor[lang]: remove translated fields for constant nodes (#4287)
- refactor[ux]: refactor preparser (#4293)
- refactor[codegen]: add profiling utils (#4412)
- refactor[lang]: remove VyperNode `__hash__()` and `__eq__()` implementations (#4433)

25.4 v0.4.0 (“Nagini”)

25.4.1 Date released: 2024-06-20

v0.4.0 represents a major overhaul to the Vyper language. Notably, it overhauls the import system and adds support for code reuse. It also adds a new, experimental backend to Vyper which lays the foundation for improved analysis, optimization and integration with third party tools.

Audits: ChainSecurity, ChainSecurity 2nd audit, ChainSecurity 3rd audit, Statemind, OtterSec, OtterSec 2nd audit

Breaking Changes

- feat[tool]!: make cancun the default evm version (#4029)
- feat[lang]: remove named reentrancy locks (#3769)
- feat[lang]!: change the signature of `block.prevrandao` (#3879)
- feat[lang]!: change ABI type of `decimal` to `int168` (#3696)
- feat[lang]: rename `_abi_encode` and `_abi_decode` (#4097)
- feat[lang]!: add feature flag for decimals (#3930)
- feat[lang]!: make internal decorator optional (#4040)
- feat[lang]: protect external calls with keyword (#2938)
- introduce floordiv, ban regular div for integers (#2937)
- feat[lang]: use keyword arguments for struct instantiation (#3777)
- feat: require type annotations for loop variables (#3596)
- feat: replace `enum` with `flag` keyword (#3697)
- feat: remove builtin constants (#3350)
- feat: drop istanbul and berlin support (#3843)
- feat: allow range with two arguments and bound (#3679)
- fix[codegen]: range bound check for signed integers (#3814)
- feat: default code offset = 3 (#3454)
- feat: rename `vyper.interfaces` to `ethereum.ercs` (#3741)
- chore: add prefix to ERC interfaces (#3804)
- chore[ux]: compute natspec as part of standard pipeline (#3946)
- feat: deprecate `vyper-serve` (#3666)

Module system

- refactor: internal handling of imports (#3655)
- feat: implement “stateless” modules (#3663)
- feat[lang]: export interfaces (#3919)
- feat[lang]: singleton modules with ownership hierarchy (#3729)
- feat[lang]: implement function exports (#3786)
- feat[lang]: auto-export events in ABI (#3808)
- fix: allow using interface defs from imported modules (#3725)
- feat: add support for constants in imported modules (#3726)
- fix[lang]: prevent modules as storage variables (#4088)
- fix[ux]: improve initializer hint for unimported modules (#4145)
- feat: add python `sys.path` to vyper path (#3763)
- feat[ux]: improve error message for importing ERC20 (#3816)

- fix[lang]: fix importing of flag types (#3871)
- feat: search path resolution for cli (#3694)
- fix[lang]: transitive exports (#3888)
- fix[ux]: error messages relating to initializer issues (#3831)
- fix[lang]: recursion in `uses` analysis for nonreentrant functions (#3971)
- fix[ux]: fix `uses` error message (#3926)
- fix[lang]: fix `uses` analysis for nonreentrant functions (#3927)
- fix[lang]: fix a hint in global initializer check (#4089)
- fix[lang]: builtin type comparisons (#3956)
- fix[tool]: fix `combined_json` output for CLI (#3901)
- fix[tool]: compile multiple files (#4053)
- refactor: reimplement AST folding (#3669)
- refactor: constant folding (#3719)
- fix[lang]: typecheck hashmap indexes with folding (#4007)
- fix[lang]: fix array index checks when the subscript is folded (#3924)
- fix[lang]: pure access analysis (#3895)

Venom

- feat: implement new IR for vyper (venom IR) (#3659)
- feat[ir]: add `make_ssa` pass to venom pipeline (#3825)
- feat[venom]: implement `mem2var` and `sccp` passes (#3941)
- feat[venom]: add store elimination pass (#4021)
- feat[venom]: add `extract_literals` pass (#4067)
- feat[venom]: optimize branching (#4049)
- feat[venom]: avoid last `swap` for commutative ops (#4048)
- feat[venom]: “pickaxe” stack scheduler optimization (#3951)
- feat[venom]: add algebraic optimization pass (#4054)
- feat: Implement target constrained venom jump instruction (#3687)
- feat: remove `deploy` instruction from venom (#3703)
- fix[venom]: liveness analysis in some loops (#3732)
- feat: add more venom instructions (#3733)
- refactor[venom]: use venom pass instances (#3908)
- refactor[venom]: refactor venom operand classes (#3915)
- refactor[venom]: introduce `IRContext` and `IRAnalysisCache` (#3983)
- feat: add utility functions to `OrderedSet` (#3833)
- feat[venom]: optimize `get_basic_block()` (#4002)

- fix[venom]: fix branch eliminator cases in sccp (#4003)
- fix[codegen]: same symbol jumpdest merge (#3982)
- fix[venom]: fix eval of `exp` in sccp (#4009)
- refactor[venom]: remove unused method in `make_ssa.py` (#4012)
- fix[venom]: fix return opcode handling in mem2var (#4011)
- fix[venom]: fix `cfg` output format (#4010)
- chore[venom]: fix output formatting of data segment in `IRContext` (#4016)
- feat[venom]: optimize mem2var and store/variable elimination pass sequences (#4032)
- fix[venom]: fix some sccp evaluations (#4028)
- fix[venom]: add `unique_symbols` check to venom pipeline (#4149)
- feat[venom]: remove redundant store elimination pass (#4036)
- fix[venom]: remove some dead code in `venom_to_assembly` (#4042)
- feat[venom]: improve unused variable removal pass (#4055)
- fix[venom]: remove liveness requests (#4058)
- fix[venom]: fix list of volatile instructions (#4065)
- fix[venom]: remove dominator tree invalidation for store elimination pass (#4069)
- fix[venom]: move loop invariant assertion to entry block (#4098)
- fix[venom]: clear `out_vars` during calculation (#4129)
- fix[venom]: alloca for default arguments (#4155)
- Refactor ctx.add_instruction() and friends (#3685)
- fix: type annotation of helper function (#3702)
- feat[ir]: emit `djump` in dense selector table (#3849)
- chore: move venom tests to `tests/unit/compiler` (#3684)

Other new features

- feat[lang]: add `blobhash()` builtin (#3962)
- feat[lang]: support `block.blobbasefee` (#3945)
- feat[lang]: add `revert_on_failure` kwarg for create builtins (#3844)
- feat[lang]: allow downcasting of bytestrings (#3832)

Docs

- chore[docs]: add docs for v0.4.0 features (#3947)
- chore[docs]: `implements` does not check event declarations (#4052)
- docs: adopt a new theme: `shibuya` (#3754)
- chore[docs]: add evaluation order warning for builtins (#4158)
- Update `FUNDING.yml` (#3636)
- docs: fix nit in v0.3.10 release notes (#3638)
- docs: add note on `pragma` parsing (#3640)
- docs: retire security@vyperlang.org (#3660)
- feat[docs]: add more detail to modules docs (#4087)
- docs: update resources section (#3656)
- docs: add script to help working on the compiler (#3674)
- docs: add warnings at the top of all example token contracts (#3676)
- docs: typo in `on_chain_market_maker.vy` (#3677)
- docs: clarify `address.codehash` for empty account (#3711)
- docs: indexed arguments for events are limited (#3715)
- docs: Fix typos (#3747)
- docs: Upgrade dependencies and fixes (#3745)
- docs: add missing cli flags (#3736)
- chore: fix formatting and docs for new struct instantiation syntax (#3792)
- docs: floordiv (#3797)
- docs: add missing `annotated_ast` flag (#3813)
- docs: update logo in readme, remove competition reference (#3837)
- docs: add rationale for floordiv rounding behavior (#3845)
- chore[docs]: amend `revert_on_failure` kwarg docs for create builtins (#3921)
- fix[docs]: fix clipped `endAuction` method in example section (#3969)
- refactor[docs]: refactor security policy (#3981)
- fix: edit link to style guide (#3658)
- Add Vyper online compiler tooling (#3680)
- chore: fix typos (#3749)

Bugfixes

- fix[codegen]: fix `raw_log()` when topics are non-literals (#3977)
- fix[codegen]: fix transient codegen for `slice` and `extract32` (#3874)
- fix[codegen]: bounds check for signed index accesses (#3817)
- fix: disallow `value=` passing for delegate and static raw_calls (#3755)
- fix[codegen]: fix double evals in sqrt, slice, blueprint (#3976)
- fix[codegen]: fix double eval in dynarray append/pop (#4030)
- fix[codegen]: fix double eval of start in range expr (#4033)
- fix[codegen]: overflow check in `slice()` (#3818)
- fix: concat buffer bug (#3738)
- fix[codegen]: fix `make_setter` overlap with internal calls (#4037)
- fix[codegen]: fix `make_setter` overlap in `dynarray_append` (#4059)
- fix[codegen]: `make_setter` overlap in the presence of `staticcall` (#4128)
- fix[codegen]: fix `_abi_decode` buffer overflow (#3925)
- fix[codegen]: zero-length dynarray `abi_decode` validation (#4060)
- fix[codegen]: recursive dynarray oob check (#4091)
- fix[codegen]: add back in `returndatasize` check (#4144)
- fix: block memory allocation overflow (#3639)
- fix[codegen]: panic on potential eval order issue for some builtins (#4157)
- fix[codegen]: panic on potential subscript eval order issue (#4159)
- add comptime check for uint2str input (#3671)
- fix: dead code analysis inside for loops (#3731)
- fix[ir]: fix a latent bug in `sha3_64` codegen (#4063)
- fix: `opcodes` and `opcodes_runtime` outputs (#3735)
- fix: bad assertion in expr.py (#3758)
- fix: iterator modification analysis (#3764)
- feat: allow constant interfaces (#3718)
- fix: assembly dead code eliminator (#3791)
- fix: prevent range over decimal (#3798)
- fix: mutability check for interface implements (#3805)
- fix[codegen]: fix non-memory reason strings (#3877)
- fix[ux]: fix compiler hang for large exponentiations (#3893)
- fix[lang]: allow type expressions inside pure functions (#3906)
- fix[ux]: raise `VersionException` with source info (#3920)
- fix[lang]: fix `pow` folding when args are not literals (#3949)
- fix[codegen]: fix some hardcoded references to `STORAGE` location (#4015)

Patched security advisories (GHSAs)

- Bounds check on built-in `slice()` function can be overflowed (GHSA-9x7f-gwxq-6f2c)
- `concat` built-in can corrupt memory (GHSA-2q8v-3gqq-4f8p)
- `raw_call` `value=` kwargs not disabled for static and delegate calls (GHSA-x2c2-q32w-4w6m)
- negative array index bounds checks (GHSA-52xq-j7v9-v4v2)
- `range(start, start + N)` reverts for negative numbers (GHSA-ppx5-q359-pvwj)
- incorrect topic logging in `raw_log` (GHSA-xchq-w5r3-4wg3)
- double eval of the `slice` start/length args in certain cases (GHSA-r56x-j438-vw5m)
- multiple eval of `sqrt()` built in argument (GHSA-5jrj-52x8-m64h)
- double eval of raw_args in `create_from_blueprint` (GHSA-3whq-64q2-qfj6)
- `sha3` codegen bug (GHSA-6845-xw22-ffxv)
- `extract32` can read dirty memory (GHSA-4hwq-4cpm-8vmx)
- `_abi_decode` Memory Overflow (GHSA-9p8r-4xp4-gw5w)
- External calls can overflow return data to return input buffer (GHSA-gp3w-2v2m-p686)

Tooling

- feat[tool]: archive format (#3891)
- feat[tool]: add source map for constructors (#4008)
- feat: add short options `-v` and `-O` to the CLI (#3695)
- feat: Add `bb` and `bb_runtime` output options (#3700)
- fix: remove hex-ir from format cli options list (#3657)
- fix: pickleability of `CompilerData` (#3803)
- feat[tool]: validate AST nodes early in the pipeline (#3809)
- feat[tool]: delay global constraint check (#3810)
- feat[tool]: export variable read/write access (#3790)
- feat[tool]: improvements to AST annotation (#3829)
- feat[tool]: add `node_id` map to source map (#3811)
- chore[tool]: add help text for `hex-ir` CLI flag (#3942)
- refactor[tool]: refactor storage layout export (#3789)
- fix[tool]: fix cross-compilation issues, add windows CI (#4014)
- fix[tool]: star option in `outputSelection` (#4094)

Performance

- perf: lazy eval of f-strings in IRnode ctor (#3602)
- perf: levenshtein optimization (#3780)
- feat: frontend optimizations (#3781)
- feat: optimize `VyperNode.deepcopy` (#3784)
- feat: more frontend optimizations (#3785)
- perf: reimplement `IRnode.__deepcopy__` (#3761)

Testing suite improvements

- refactor[test]: bypass `eth-tester` and interface with evm backend directly (#3846)
- feat: Refactor assert_tx_failed into a context (#3706)
- feat[test]: implement `abi_decode` spec test (#4095)
- feat[test]: add more coverage to `abi_decode` fuzzer tests (#4153)
- feat[ci]: enable cancun testing (#3861)
- fix: add missing test for memory allocation overflow (#3650)
- chore: fix test for `slice` (#3633)
- add abi_types unit tests (#3662)
- refactor: test directory structure (#3664)
- chore: test all output formats (#3683)
- chore: deduplicate test files (#3773)
- feat[test]: add more transient storage tests (#3883)
- chore[ci]: fix apt-get failure in era pipeline (#3821)
- chore[ci]: enable python3.12 tests (#3860)
- chore[ci]: refactor jobs to use gh actions (#3863)
- chore[ci]: use `--dist workersteal` from latest `xdist` (#3869)
- chore: run mypy as part of lint rule in Makefile (#3771)
- chore[test]: always specify the evm backend (#4006)
- chore: update lint dependencies (#3704)
- chore: add color to mypy output (#3793)
- chore: remove tox rules for lint commands (#3826)
- chore[ci]: roll back GH actions/artifacts version (#3838)
- chore: Upgrade GitHub action dependencies (#3807)
- chore[ci]: pin eth-abi for decode regression (#3834)
- fix[ci]: release artifacts (#3839)
- chore[ci]: merge mypy job into lint (#3840)
- test: parametrize CI over EVM versions (#3842)

- feat[ci]: add PR title validation (#3887)
- fix[test]: fix failure in grammar fuzzing (#3892)
- feat[test]: add `xfail_strict`, clean up `setup.cfg` (#3889)
- fix[ci]: pin hexbytes to pre-1.0.0 (#3903)
- chore[test]: update hexbytes version and tests (#3904)
- fix[test]: fix a bad bound in decimal fuzzing (#3909)
- fix[test]: fix a boundary case in decimal fuzzing (#3918)
- feat[ci]: update pypi release pipeline to use OIDC (#3912)
- chore[ci]: reconfigure single commit validation (#3937)
- chore[ci]: downgrade codecov action to v3 (#3940)
- feat[ci]: add codecov configuration (#4057)
- feat[test]: remove memory mocker (#4005)
- refactor[test]: change fixture scope in examples (#3995)
- fix[test]: fix call graph stability fuzzer (#4064)
- chore[test]: add macos to test matrix (#4025)
- refactor[test]: change default expected exception type (#4004)

Misc / refactor

- feat[ir]: add `eval_once` sanity fences to more builtins (#3835)
- fix: reorder compilation of branches in stmt.py (#3603)
- refactor[codegen]: make settings into a global object (#3929)
- chore: improve exception handling in IR generation (#3705)
- refactor: merge `annotation.py` and `local.py` (#3456)
- chore[ux]: remove deprecated python AST classes (#3998)
- refactor[ux]: remove deprecated `VyperNode` properties (#3999)
- feat: remove Index AST node (#3757)
- refactor: for loop target parsing (#3724)
- chore: improve diagnostics for invalid for loop annotation (#3721)
- refactor: builtin functions inherit from `VyperType` (#3559)
- fix: remove .keyword from Call AST node (#3689)
- improvement: assert descriptions in Crowdfund finalize() and participate() (#3064)
- feat: improve panics in IR generation (#3708)
- feat: improve warnings, refactor `vyper_warn()` (#3800)
- fix[ir]: unique symbol name (#3848)
- refactor: remove duplicate terminus checking code (#3541)
- refactor: `ExprVisitor` type validation (#3739)

- chore: improve exception for type validation (#3759)
- fix: fuzz test not updated to use TypeMismatch (#3768)
- chore: fix StringEnum._generate_next_value_ signature (#3770)
- chore: improve some error messages (#3775)
- refactor: `get_search_paths()` for vyper cli (#3778)
- chore: replace occurrences of 'enum' by 'flag' (#3794)
- chore: add another borrowship test (#3802)
- chore[ux]: improve an exports error message (#3822)
- chore: improve codegen test coverage report (#3824)
- chore: improve syntax error messages (#3885)
- chore[tool]: remove `vyper-serve` from `setup.py` (#3936)
- fix[ux]: replace standard strings with f-strings (#3953)
- chore[ir]: sanity check types in for range codegen (#3968)

25.5 v0.3.10 ("Black Adder")

25.5.1 Date released: 2023-10-04

v0.3.10 is a performance focused release that additionally ships numerous bugfixes. 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).

Audits: OtterSec, CodeHawks, ChainSecurity

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)

25.6 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.

Audits: ChainSecurity

Fixes:

- initcode codesize blowup (#3450)

- add back global calldatasize check for contracts with default fn (#3463)

25.7 v0.3.8

Date released: 2023-05-23

Audits: ChainSecurity, ChainSecurity 2nd audit

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)

25.8 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.

25.9 v0.3.6

Date released: 2022-08-07

Bugfixes:

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

25.10 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)

25.11 v0.3.4

Date released: 2022-07-27

Audits: ChainSecurity

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!

25.12 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)

25.13 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.

25.14 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!

25.15 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!

25.16 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!

25.17 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)

25.18 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)

25.19 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)

25.20 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)

25.21 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

25.22 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`

25.23 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)!

25.24 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)

25.25 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)

25.26 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 between `address` and `Bytes[20]` (#2191)

25.27 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)

25.28 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)

25.29 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)

25.30 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)

25.31 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 declaration 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.

25.32 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.

25.33 v0.1.0-beta.16

Date released: 09-01-2020

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

25.34 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!

25.35 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 subtraction on unsigned types (#1638)
- Disallow `orelse` syntax in `for` loops (#1633)
- Increased clarity and efficiency of zero-padding (#1605)

25.36 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)

25.37 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!

25.38 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!

25.39 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.

25.40 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)

25.41 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 annotated 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

TWENTYSIX

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.

26.1 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

26.2 How to Suggest Improvements

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

26.3 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.

26.4 Fix Bugs

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

26.5 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.

26.6 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.

26.6.1 Commit Messages

For guidance on writing good commit messages, see How to Write a Git Commit Message.

To ensure your commit message is properly formatted (wrapped at 72 characters), use the included formatter:

1. Write your commit message in a text file
2. Run `python fmt_commit_msg.py commitmsg.txt` (this formats the file in-place)
3. Paste the formatted message in your PR description, wrapped in triple backticks

The maintainer will use this message when squash-merging your PR.

26.6.2 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

TWENTYSEVEN

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.

27.1 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.

27.2 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.

27.2.1 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 retrievals 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.

27.2.2 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 **should** 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
```

27.2.3 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.

27.2.4 Strings

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

27.2.5 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.

27.3 Tests

We use the pytest framework for testing.

27.3.1 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.

27.3.2 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/`.

27.3.3 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.

27.3.4 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.

27.4 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.

27.4.1 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.

27.4.2 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.

27.4.3 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: `#`, `=`, `-`, `*`, `^`.

27.5 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.

27.6 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.

27.6.1 Conventional Commits

Commit messages **should** adhere to the Conventional Commits standard. A conventional 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.

27.6.2 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

TWENTYEIGHT

VYPER VERSIONING GUIDELINE

28.1 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	Integrating 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.

28.2 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.

28.2.1 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

28.2.2 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

28.2.3 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

28.2.4 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*.

28.2.5 Pre-Release Versions

Any 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.

28.3 Pull Requests

Pull Requests should be opened against the `master` branch.

28.4 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

D

E

F

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z