**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.

**Version Pragma**

Vyper supports a version pragma to ensure that a contract is only compiled by the intended compiler version, or range of versions. Version strings use [NPM](https://docs.npmjs.com/about-semantic-versioning) style syntax.

*# @version ^0.2.0*

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

**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](https://docs.vyperlang.org/en/stable/types.html#types-initial) for their type.

storedData: int128

State variables are accessed via the [self](https://docs.vyperlang.org/en/stable/constants-and-vars.html#constants-self) object.

self.storedData = 123

See the documentation on [Types](https://docs.vyperlang.org/en/stable/types.html#types) or [Scoping and Declarations](https://docs.vyperlang.org/en/stable/scoping-and-declarations.html#scoping) for more information.

**Functions**

Functions are executable units of code within a contract.

**@external**

**def** bid():

...

Functions may be called internally or externally depending on their [visibility](https://docs.vyperlang.org/en/stable/control-structures.html#function-visibility). Functions may accept input arguments and return variables in order to pass values between them.

See the [Functions](https://docs.vyperlang.org/en/stable/control-structures.html#control-structures-functions) documentation for more information.

**Events**

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

event Payment:

amount: int128

sender: indexed(address)

total\_paid: int128

**@external**

**@payable**

**def** pay():

self.total\_paid += msg.value

log Payment(msg.value, msg.sender)

See the [Event](https://docs.vyperlang.org/en/stable/event-logging.html#event-logging) documentation for more information.

**Interfaces**

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

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

interface FooBar:

**def** calculate() -> uint256: view

**def** test1(): nonpayable

**from** **foo** **import** FooBar

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

**@external**

**def** test(some\_address: address):

FooBar(some\_address).calculate()

See the [Interfaces](https://docs.vyperlang.org/en/stable/interfaces.html#interfaces) documentation for more information.

**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](https://docs.vyperlang.org/en/stable/types.html#types-struct) documentation for more information.

# 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.

## Value Types

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

### Boolean

**Keyword:** bool

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

#### Values

The only possible values are the constants True and False.

#### Operators

| **Operator** | **Description** |
| --- | --- |
| 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.

### Signed Integer (N bit)

**Keyword:** intN (e.g., int128)

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

#### Values

Signed integer values between -2N-1 and (2N-1 - 1), inclusive.

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

#### Operators

##### Comparisons

Comparisons return a boolean value.

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

x and y must both be of the same type.

##### Arithmetic Operators

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

x and y must both be of the same type.

##### Bitwise Operators

| **Operator** | **Description** |
| --- | --- |
| x & y | Bitwise and |
| x | y | Bitwise or |
| x ^ y | Bitwise xor |

x and y must be of the same type.

##### Shifts

| **Operator** | **Description** |
| --- | --- |
| x << y | Left shift |
| x >> y | Right shift |

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

**Note**

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

### Unsigned Integer (N bit)

**Keyword:** uintN (e.g., uint8)

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

#### Values

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

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

**Note**

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

#### Operators

##### Comparisons

Comparisons return a boolean value.

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

x and y must be of the same type.

##### Arithmetic Operators

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

x and y must be of the same type.

##### Bitwise Operators

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

x and y must be of the same type.

**Note**

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

##### Shifts

| **Operator** | **Description** |
| --- | --- |
| x << y | Left shift |
| x >> y | Right shift |

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

**Note**

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

### Decimals

**Keyword:** decimal

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

#### Values

A value with a precision of 10 decimal places between -18707220957835557353007165858768422651595.9365500928 (-2167 / 1010) and 18707220957835557353007165858768422651595.9365500927 ((2167 - 1) / 1010).

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

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

#### Operators

##### Comparisons

Comparisons return a boolean value.

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

x and y must be of the type decimal.

##### Arithmetic Operators

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

x and y must be of the type decimal.

### Address

**Keyword:** address

The address type holds an Ethereum address.

#### Values

An address type can hold an Ethereum address which equates to 20 bytes or 160 bits. Address literals must be written in hexadecimal notation with a leading 0x and must be [checksummed](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md).

##### Members

| **Member** | **Type** | **Description** |
| --- | --- | --- |
| balance | uint256 | Balance of an address |
| codehash | bytes32 | Keccak of code at an address, EMPTY\_BYTES32 if no contract is deployed |
| codesize | uint256 | Size of code deployed at an address, in bytes |
| is\_contract | bool | Boolean indicating if a contract is deployed at an address |
| code | Bytes | Contract bytecode |

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

**Note**

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

**Note**

\_address.code requires the usage of [**slice**](https://docs.vyperlang.org/en/stable/built-in-functions.html#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.

### M-byte-wide Fixed Size Byte Array

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

**Example:**

*# Declaration*

hash: bytes32

*# Assignment*

self.hash = \_hash

some\_method\_id: bytes4 = 0x01abcdef

#### Operators

| **Keyword** | **Description** |
| --- | --- |
| keccak256(x) | Return the keccak256 hash as bytes32. |
| concat(x, ...) | Concatenate multiple inputs. |
| slice(x, start=\_start, len=\_len) | Return a slice of \_len starting at \_start. |

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

### Byte Arrays

**Keyword:** Bytes

A byte array with a max size.

The syntax being Bytes[maxLen], where maxLen is an integer which denotes the maximum number of bytes. On the ABI level the Fixed-size bytes array is annotated as bytes.

Bytes literals may be given as bytes strings.

bytes\_string: Bytes[100] = b"**\x01**"

### 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"

### Enums

**Keyword:** enum

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

*# Defining an enum with two members*

enum Roles:

ADMIN

USER

*# Declaring an enum variable*

role: Roles = Roles.ADMIN

*# Returning a member*

**return** Roles.ADMIN

#### Operators

##### Comparisons

Comparisons return a boolean value.

| **Operator** | **Description** |
| --- | --- |
| x == y | Equals |
| x != y | Does not equal |
| x in y | x is in y |
| x not in y | x is not in y |

##### Bitwise Operators

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

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

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

enum Roles:

MANAGER

ADMIN

USER

*# Check for membership*

**@external**

**def** foo(a: Roles) -> bool:

**return** a **in** (Roles.MANAGER | Roles.USER)

*# Check not in*

**@external**

**def** bar(a: Roles) -> bool:

**return** a **not** **in** (Roles.MANAGER | Roles.USER)

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

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

## Reference Types

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

**Note**

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

### Fixed-size Lists

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

Lists can be declared with \_name: \_ValueType[\_Integer], except Bytes[N], String[N] and enums.

*# Defining a list*

exampleList: int128[3]

*# Setting values*

exampleList = [10, 11, 12]

exampleList[2] = 42

*# Returning a value*

**return** exampleList[0]

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

A two dimensional list can be declared with \_name: \_ValueType[inner\_size][outer\_size]. Elements can be accessed with \_name[outer\_index][inner\_index].

*# Defining a list with 2 rows and 5 columns and set all values to 0*

exampleList2D: int128[5][2] = empty(int128[5][2])

*# Setting a value for row the first row (0) and last column (4)*

exampleList2D[0][4] = 42

*# 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.

### Dynamic Arrays

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

*# Defining a list*

exampleList: DynArray[int128, 3]

*# Setting values*

exampleList = []

*# exampleList.pop() # would revert!*

exampleList.append(42) *# exampleList now has length 1*

exampleList.append(120) *# exampleList now has length 2*

exampleList.append(356) *# exampleList now has length 3*

*# exampleList.append(1) # would revert!*

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

*# myValue = exampleList[2] # would revert!*

*# Returning a value*

**return** exampleList[0]

**Note**

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

**Note**

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

**for** item **in** self.my\_array:

self.my\_array[0] = item

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

**Note**

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

### 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

### Mappings

Mappings are [hash tables](https://en.wikipedia.org/wiki/Hash_table) 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](https://docs.vyperlang.org/en/stable/types.html#types-initial).

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.

## 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.

## Type Conversions

All type conversions in Vyper must be made explicitly using the built-in convert(a: atype, btype) function. Type conversions in Vyper are designed to be safe and intuitive. All type conversions will check that the input is in bounds for the output type. The general principles are:

* Except for conversions involving decimals and bools, the input is bit-for-bit preserved.
* Conversions to bool map all nonzero inputs to 1.
* When converting from decimals to integers, the input is truncated towards zero.
* address types are treated as uint160, except conversions with signed integers and decimals are not allowed.
* Converting between right-padded (bytes, Bytes, String) and left-padded types, results in a rotation to convert the padding. For instance, converting from bytes20 to address would result in rotating the input by 12 bytes to the right.
* Converting between signed and unsigned integers reverts if the input is negative.
* Narrowing conversions (e.g., int256 -> int128) check that the input is in bounds for the output type.
* Converting between bytes and int types results in sign-extension if the output type is signed. For instance, converting 0xff (bytes1) to int8 returns -1.
* Converting between bytes and int types which have different sizes follows the rule of going through the closest integer type, first. For instance, bytes1 -> int16 is like bytes1 -> int8 -> int16 (signextend, then widen). uint8 -> bytes20 is like uint8 -> uint160 -> bytes20 (rotate left 12 bytes).
* Enums can be converted to and from uint256 only.

A small Python reference implementation is maintained as part of Vyper’s test suite, it can be found [here](https://github.com/vyperlang/vyper/blob/c4c6afd07801a0cc0038cdd4007cc43860c54193/tests/parser/functions/test_convert.py#L318). The motivation and more detailed discussion of the rules can be found [here](https://github.com/vyperlang/vyper/issues/2507).