**Control Structures**

**Functions**

Functions are executable units of code within a contract. Functions may only be declared within a contract’s [module scope](https://docs.vyperlang.org/en/stable/scoping-and-declarations.html#scoping-module).

**@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.

**Visibility**

All functions must include exactly one visibility decorator.

**External Functions**

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

**@external**

**def** add\_seven(a: int128) -> int128:

**return** a + 7

**@external**

**def** add\_seven\_with\_overloading(a: uint256, b: uint256 = 3):

**return** a + b

A Vyper contract cannot call directly between two external functions. If you must do this, you can use an [interface](https://docs.vyperlang.org/en/stable/interfaces.html#interfaces).

**Note**

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

**Internal Functions**

Internal functions (marked with the @internal decorator) are only accessible from other functions within the same contract. They are called via the [self](https://docs.vyperlang.org/en/stable/constants-and-vars.html" \l "constants-self) object:

**@internal**

**def** \_times\_two(amount: uint256, two: uint256 = 2) -> uint256:

**return** amount \* two

**@external**

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

**return** self.\_times\_two(amount)

**Note**

Since calling an internal function is realized by jumping to its entry label, the internal function dispatcher ensures the correctness of the jumps. Please note that for internal functions which use more than one default parameter, Vyper versions >=0.3.8 are strongly recommended due to the security advisory [GHSA-ph9x-4vc9-m39g](https://github.com/vyperlang/vyper/security/advisories/GHSA-ph9x-4vc9-m39g).

**Mutability**

You can optionally declare a function’s mutability by using a [decorator](https://docs.vyperlang.org/en/stable/control-structures.html#function-decorators). There are four mutability levels:

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

**@view**

**@external**

**def** readonly():

*# this function cannot write to state*

...

**@payable**

**@external**

**def** send\_me\_money():

*# this function can receive ether*

...

Functions default to nonpayable when no mutability decorator is used.

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

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

**Re-entrancy Locks**

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

**@external**

**@nonreentrant**("lock")

**def** make\_a\_call(\_addr: address):

*# this function is protected from re-entrancy*

...

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

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

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

**Note**

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

**Note**

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

**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](https://solidity.readthedocs.io/en/latest/contracts.html?highlight=fallback#fallback-function).

This function is always named \_\_default\_\_. It must be annotated with @external. It cannot expect any input arguments.

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

event Payment:

amount: uint256

sender: indexed(address)

**@external**

**@payable**

**def** \_\_default\_\_():

log Payment(msg.value, msg.sender)

**Considerations**

Just as in Solidity, Vyper generates a default function if one isn’t found, in the form of a REVERT call. Note that this still [generates an exception](https://github.com/ethereum/wiki/wiki/Subtleties) 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).

**The \_\_init\_\_ Function**

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

owner: address

**@external**

**def** \_\_init\_\_():

self.owner = msg.sender

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

**Decorators Reference**

All functions must include one [visibility](https://docs.vyperlang.org/en/stable/control-structures.html" \l "function-visibility) decorator (@external or @internal). The remaining decorators are optional.

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

**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**:

...

**for loops**

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

**for** i **in** <ITERABLE>:

...

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

**Array Iteration**

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

foo: int128[3] = [4, 23, 42]

**for** i **in** foo:

...

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

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

**for** i **in** [4, 23, 42]:

...

Some restrictions:

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

**Range Iteration**

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

**for** i **in** range(STOP):

...

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

**for** i **in** range(START, STOP):

...

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

**for** i **in** range(a, a + N):

...

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

# Scoping and Declarations

## Variable Declaration

The first time a variable is referenced you must declare its [type](https://docs.vyperlang.org/en/stable/types.html#types):

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

### 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).

### Declaring Immutable Variables

Variables can be marked as immutable during declaration:

DATA: immutable(uint256)

**@external**

**def** \_\_init\_\_(\_data: uint256):

DATA = \_data

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

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

### 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()

## Storage Layout

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

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

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

For example, consider upgrading the following contract:

*# old\_contract.vy*

owner: public(address)

balanceOf: public(HashMap[address, uint256])

*# new\_contract.vy*

owner: public(address)

minter: public(address)

balanceOf: public(HashMap[address, uint256])

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

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

{

"owner": {"type": "address", "slot": 0},

"minter": {"type": "address", "slot": 2},

"balanceOf": {"type": "HashMap[address, uint256]", "slot": 1}

}

For further information on generating the storage layout, see [Storage Layout](https://docs.vyperlang.org/en/stable/compiling-a-contract.html#compiler-storage-layout).

## 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.

### Module Scope

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

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

#### Accessing Module Scope from Functions

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

a: int128

**@internal**

**def** foo() -> int128

**return** 42

**@external**

**def** foo() -> int128:

b: int128 = self.foo()

**return** self.a + b

#### Name Shadowing

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

a: constant(bool) = **True**

**@external**

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

*# memory variable cannot have the same name as a constant or immutable variable*

a: bool = **False**

**return** a

a: immutable(bool)

**@external**

**def** \_\_init\_\_():

a = **True**

**@external**

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

*# input argument cannot have the same name as a constant or immutable variable*

**return** a

### Function Scope

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

**@external**

**def** foo(a: int128):

**pass**

**@external**

**def** bar(a: uint256):

**pass**

**@external**

**def** baz():

a: bool = **True**

The following examples will not compile:

**@external**

**def** foo(a: int128):

*# `a` has already been declared as an input argument*

a: int128 = 21

**@external**

**def** foo(a: int128):

a = 4

**@external**

**def** bar():

*# `a` has not been declared within this function*

a += 12

### Block Scopes

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

**@external**

**def** foo(a: bool) -> int128:

**if** a:

x: int128 = 3

**else**:

x: bool = **False**

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

**@external**

**def** foo(a: bool) -> int128:

**for** i **in** [1, 2, 3]:

**pass**

i: bool = **False**

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

**@external**

**def** foo(a: bool) -> int128:

**for** i **in** [1, 2, 3]:

a: int128 = i

a += 3