**Environment Variables and Constants**

**Environment Variables**

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

**Block and Transaction Properties**

| **Name** | **Type** | **Value** |
| --- | --- | --- |
| block.coinbase | address | Current block miner’s address |
| block.difficulty | uint256 | Current block difficulty |
| block.prevrandao | uint256 | Current randomness beacon provided by the beacon chain |
| block.number | uint256 | Current block number |
| block.prevhash | bytes32 | Equivalent to blockhash(block.number - 1) |
| block.timestamp | uint256 | Current block epoch timestamp |
| chain.id | uint256 | Chain ID |
| msg.data | Bytes | Message data |
| msg.gas | uint256 | Remaining gas |
| msg.sender | address | Sender of the message (current call) |
| msg.value | uint256 | Number of wei sent with the message |
| tx.origin | address | Sender of the transaction (full call chain) |
| tx.gasprice | uint256 | Gas price of current transaction in wei |

**Note**

block.prevrandao is an alias for block.difficulty. Since block.difficulty is considered deprecated according to [EIP-4399](https://eips.ethereum.org/EIPS/eip-4399) after “The Merge” (Paris hard fork), we recommend using block.prevrandao.

**Note**

msg.data requires the usage of [**slice**](https://docs.vyperlang.org/en/stable/built-in-functions.html#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](https://docs.vyperlang.org/en/stable/built-in-functions.html" \l "len" \o "len)**.

**The self Variable**

self is an environment variable used to reference a contract from within itself. Along with the normal [address](https://docs.vyperlang.org/en/stable/types.html#address) members, self allows you to read and write to state variables and to call private functions within the contract.

| **Name** | **Type** | **Value** |
| --- | --- | --- |
| self | address | Current contract’s address |
| self.balance | uint256 | Current contract’s balance |

**Accessing State Variables**

self is used to access a contract’s [state variables](https://docs.vyperlang.org/en/stable/structure-of-a-contract.html#structure-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](https://docs.vyperlang.org/en/stable/control-structures.html#structure-functions-internal) within a contract:

**@internal**

**def** \_times\_two(amount: uint256) -> uint256:

**return** amount \* 2

**@external**

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

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

**Custom Constants**

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

TOTAL\_SUPPLY: constant(uint256) = 10000000

total\_supply: public(uint256)

**@external**

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

self.total\_supply = TOTAL\_SUPPLY

# Statements

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

## Control Flow

### break

The break statement terminates the nearest enclosing for loop.

**for** i **in** [1, 2, 3, 4, 5]:

**if** i == a:

**break**

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

### continue

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

**for** i **in** [1, 2, 3, 4, 5]:

**if** i != a:

**continue**

...

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

### 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**

### 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.

## Event Logging

### log

The log statement is used to log an event:

log MyEvent(...)

The event must have been previously declared.

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

## Assertions and Exceptions

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

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

### 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.

### 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"