---

title: Smart contract languages description: An overview and comparison of the two main smart contract languages – Solidity and Vyper. lang: en

---

A great aspect about Ethereum is that smart contracts can be programmed using relatively developer-friendly languages. If you're experienced with Python or any curly-bracket language, you can find a language with familiar syntax.

The two most active and maintained languages are:

- Solidity
- Vyper

More experienced developers also might want to use Yul, an intermediate language for theEthereum Virtual Machine, or Yul+, an extension to Yul.

If you're curious and like to help test new languages that are still under heavy development you can experiment with Fe, an emerging smart contract language which is currently still in its infancy.

Prerequisites {#prerequisites}

Previous knowledge of programming languages, especially of JavaScript or Python, can help you make sense of differences in smart contract languages. We also recommend you understand smart contracts as a concept before digging too deep into the language comparisons. Intro to smart contracts.

Solidity {#solidity}

- Object-oriented, high-level language for implementing smart contracts.
- Curly-bracket language that has been most profoundly influenced by C++.
- Statically typed (the type of a variable is known at compile time).
- Supports:
- Inheritance (you can extend other contracts).
- Libraries (you can create reusable code that you can call from different contracts – like static functions in a static class in other object oriented programming languages).
- Complex user-defined types.

Important links {#important-links}

- Documentation
- Solidity Language Portal
- Solidity by Example
- GitHub
- Solidity Gitter Chatroom bridged to Solidity Matrix Chatroom
- Cheat Sheet
- Solidity Blog
- Solidity Twitter

Example contract {#example-contract}

```solidity // SPDX-License-Identifier: GPL-3.0 pragma solidity >= 0.7.0;

contract Coin { // The keyword "public" makes variables // accessible from other contracts address public minter; mapping (address => uint) public balances;

```
// Events allow clients to react to specific
// contract changes you declare
event Sent(address from, address to, uint amount);

// Constructor code is only run when the contract
```
```

```
// is created
constructor() {
    minter = msg.sender;
}

// Sends an amount of newly created coins to an address
// Can only be called by the contract creator
function mint(address receiver, uint amount) public {
    require(msg.sender == minter);
    require(amount < 1e60);
    balances[receiver] += amount;
}

// Sends an amount of existing coins
// from any caller to an address
function send(address receiver, uint amount) public {
    require(amount <= balances[msg.sender], "Insufficient balance.");
    balances[msg.sender] -= amount;
    balances[receiver] += amount;
    emit Sent(msg.sender, receiver, amount);
}
```

} ```

This example should give you a sense of what Solidity contract syntax is like. For a more detailed description of the functions and variables, see the docs.

Vyper {#vyper}

- Pythonic programming language
- Strong typing
- Small and understandable compiler code
- Efficient bytecode generation
- Deliberately has less features than Solidity with the aim of making contracts more secure and easier to audit. Vyper does not support:
- Modifiers
- Inheritance
- Inline assembly
- Function overloading
- Operator overloading
- Recursive calling
- Infinite-length loops
- Binary fixed points

For more information, read the Vyper rationale.

Important links {#important-links-1}

- Documentation
- Vyper by Example
- More Vyper by Example
- GitHub
- Vyper community Discord chat
- Cheat Sheet
- Smart contract development frameworks and tools for Vyper
- VyperPunk - learn to secure and hack Vyper smart contracts
- VyperExamples - Vyper vulnerability examples
- Vyper Hub for development
- Vyper greatest hits smart contract examples
- Awesome Vyper curated resources

Example {#example}

```python

# Open Auction

# Auction params

# Beneficiary receives money from the highest bidder

beneficiary: public(address) auctionStart: public(uint256) auctionEnd: public(uint256)

# Current state of auction

highestBidder: public(address) highestBid: public(uint256)

# Set to true at the end, disallows any change

ended: public(bool)

# Keep track of refunded bids so we can follow the withdraw pattern

pendingReturns: public(HashMap[address, uint256])

# Create a simple auction with `_bidding_time`

# seconds bidding time on behalf of the

# beneficiary address `_beneficiary`.

@external def **init**(_beneficiary: address, _bidding_time: uint256): self.beneficiary = _beneficiary self.auctionStart = block.timestamp self.auctionEnd = self.auctionStart + _bidding_time

# 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 is over. assert block.timestamp < self.auctionEnd # Check if bid is high enough assert msg.value > self.highestBid # Track the refund for the previous high bidder self.pendingReturns[self.highestBidder] += self.highestBid # Track new high bid self.highestBidder = msg.sender self.highestBid = msg.value
```

Withdraw a previously refunded bid. The withdraw pattern is

used here to avoid a security issue. If refunds were directly

sent as part of bid(), a malicious bidding contract could block

those refunds and thus block new higher bids from coming in.

@external def withdraw(): pending_amount: uint256 = self.pendingReturns[msg.sender] self.pendingReturns[msg.sender] = 0 send(msg.sender, pending_amount)

End the auction and send the highest bid

to the beneficiary.

@external def endAuction(): # It is a good guideline to structure functions that interact # with other contracts (i.e. they call functions or send ether) # into three phases: # 1. checking conditions # 2. performing actions (potentially changing conditions) # 3. interacting with other contracts # If these phases are mixed up, the other contract could call # back into the current contract and modify the state or cause # effects (ether payout) to be performed multiple times. # If functions called internally include interaction with external # contracts, they also have to be considered interaction with # external contracts.

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

# 2. Effects
self.ended = True

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

```

```

This example should give you a sense of what Vyper contract syntax is like. For a more detailed description of the functions and variables, see the docs.

## Yul and Yul+ {#yul}

If you're new to Ethereum and haven't done any coding with smart contract languages yet, we recommend getting started with Solidity or Vyper. Only look into Yul or Yul+ once you're familiar with smart contract security best practices and the specifics of working with the EVM.

**Yul**

- Intermediate language for Ethereum.
```

- Supports the EVM and Ewasm, an Ethereum flavored WebAssembly, and is designed to be a usable common denominator of both platforms.
- Good target for high-level optimisation stages that can benefit both EVM and Ewasm platforms equally.

Yul+

- A low-level, highly efficient extension to Yul.
- Initially designed for an optimistic rollup contract.
- Yul+ can be looked at as an experimental upgrade proposal to Yul, adding new features to it.

Important links {#important-links-2}

- Yul Documentation
- Yul+ Documentation
- Yul+ Playground
- Yul+ Introduction Post

Example contract {#example-contract-2}

The following simple example implements a power function. It can be compiled using`solc --strict-assembly --bin input.yul`. The example should be stored in the input.yul file.

```
{ function power(base, exponent) -> result { switch exponent case 0 { result := 1 } case 1 { result := base }
default { result := power(mul(base, base), div(exponent, 2)) if mod(exponent, 2) { result := mul(base, result) }
} } let res := power(calldataload(0), calldataload(32)) mstore(0, res) return(0, 32) }
```

If you are already well experienced with smart contracts, a full ERC20 implementation in Yul can be foundhere.

Fe {#fe}

- Statically typed language for the Ethereum Virtual Machine (EVM).
- Inspired by Python and Rust.
- Aims to be easy to learn -- even for developers who are new to the Ethereum ecosystem.
- Fe development is still in its early stages, the language had its alpha release in January 2021.

Important links {#important-links-3}

- GitHub
- Fe Announcement
- Fe 2021 Roadmap
- Fe Discord Chat
- Fe Twitter

Example contract {#example-contract-3}

The following is a simple contract implemented in Fe.

``` type BookMsg = bytes[100]

contract GuestBook: pub guest_book: map

```
event Signed:
    book_msg: BookMsg

pub def sign(book_msg: BookMsg):
    self.guest_book[msg.sender] = book_msg

    emit Signed(book_msg=book_msg)

pub def get_msg(addr: address) -> BookMsg:
    return self.guest_book[addr].to_mem()
```
```

```

## How to choose {#how-to-choose}

As with any other programming language, it's mostly about choosing the right tool for the right job as well as personal preferences.

Here are a few things to consider if you haven't tried any of the languages yet:

### What is great about Solidity? {#solidity-advantages}

- If you are a beginner, there are many tutorials and learning tools out there. See more about that in the Learn by Coding section.
- Good developer tooling available.
- Solidity has a big developer community, which means you'll most likely find answers to your questions quite quickly.

### What is great about Vyper? {#vyper-advatages}

- Great way to get started for Python devs that want to write smart contracts.
- Vyper has a smaller number of features which makes it great for quick prototyping of ideas.
- Vyper aims to be easy to audit and maximally human-readable.

### What is great about Yul and Yul+? {#yul-advantages}

- Simplistic and functional low-level language.
- Allows to get much closer to raw EVM, which can help to optimize the gas usage of your contracts.

## Language comparisons {#language-comparisons}

For comparisons of basic syntax, the contract lifecycle, interfaces, operators, data structures, functions, control flow, and more check out this cheatsheet by Auditless

## Further reading {#further-reading}

- Solidity Contracts Library by OpenZeppelin
- Solidity by Example
```