**Built-in Functions**

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

**Bitwise Operations**

**bitwise\_and(*x: uint256*, *y: uint256*)→ uint256**

Perform a “bitwise and” operation. Each bit of the output is 1 if the corresponding bit of **x** AND of **y** is 1, otherwise it is 0.

**@external**

**@view**

**def** foo(x: uint256, y: uint256) -> uint256:

**return** bitwise\_and(x, y)

**>>>** ExampleContract.foo(31337, 8008135)

12353

**Note**

This function has been deprecated from version 0.3.4 onwards. Please use the & operator instead.

**bitwise\_not(*x: uint256*)→ uint256**

Return the bitwise complement of **x** - the number you get by switching each 1 for a 0 and each 0 for a 1.

**@external**

**@view**

**def** foo(x: uint256) -> uint256:

**return** bitwise\_not(x)

**>>>** ExampleContract.foo(0)

115792089237316195423570985008687907853269984665640564039457584007913129639935

**Note**

This function has been deprecated from version 0.3.4 onwards. Please use the ~ operator instead.

**bitwise\_or(*x: uint256*, *y: uint256*)→ uint256**

Perform a “bitwise or” operation. Each bit of the output is 0 if the corresponding bit of **x** AND of **y** is 0, otherwise it is 1.

**@external**

**@view**

**def** foo(x: uint256, y: uint256) -> uint256:

**return** bitwise\_or(x, y)

**>>>** ExampleContract.foo(31337, 8008135)

8027119

**Note**

This function has been deprecated from version 0.3.4 onwards. Please use the | operator instead.

**bitwise\_xor(*x: uint256*, *y: uint256*)→ uint256**

Perform a “bitwise exclusive or” operation. Each bit of the output is the same as the corresponding bit in **x** if that bit in **y** is 0, and it is the complement of the bit in **x** if that bit in **y** is 1.

**@external**

**@view**

**def** foo(x: uint256, y: uint256) -> uint256:

**return** bitwise\_xor(x, y)

**>>>** ExampleContract.foo(31337, 8008135)

8014766

**Note**

This function has been deprecated from version 0.3.4 onwards. Please use the ^ operator instead.

**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.

**Chain Interaction**

Vyper has three built-ins for contract creation; all 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

**create\_minimal\_proxy\_to(*target: address*, *value: uint256 = 0*[, *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)
* **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*[, *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)
* **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*, *code\_offset=0*[, *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)
* **code\_offset**: The offset to start the **EXTCODECOPY** from (Optional, default 0)
* **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

arg2: String = "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.

**Warning**

It is recommended to deploy blueprints with the ERC-5202 preamble 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 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\_call(*to: address*, *data: Bytes*, *max\_outsize: uint256 = 0*, *gas: uint256 = gasLeft*, *value: uint256 = 0*, *is\_delegate\_call:***[***bool***](https://docs.python.org/3.8/library/functions.html#bool)***= False*, *is\_static\_call:***[***bool***](https://docs.python.org/3.8/library/functions.html#bool)***= False*, *revert\_on\_failure:***[***bool***](https://docs.python.org/3.8/library/functions.html#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. If not set, all remaining gas is forwarded.
* **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

**raw\_log(*topics: bytes32[4]*, *data: Union[Bytes, bytes32]*)→**[**None**](https://docs.python.org/3.8/library/constants.html#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**](https://docs.python.org/3.8/library/constants.html#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**](https://docs.python.org/3.8/library/constants.html#None)

Trigger the **SELFDESTRUCT** opcode (**0xFF**), causing the contract to be destroyed.

* **to**: Address to forward the contract’s ether balance to

**Warning**

This method deletes the contract from the blockchain. All non-ether assets associated with this contract are “burned” and the contract is no longer accessible.

**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**](https://docs.python.org/3.8/library/constants.html#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**.

**@external**

**def** foo(\_receiver: address, \_amount: uint256, gas: uint256):

send(\_receiver, \_amount, gas=gas)

**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,

19321533766552368860946552437480515441416830039777911637913418824951667761761,

]

**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 **0** on error.

**@external**

**@view**

**def** foo(hash: bytes32, v: uint8, r:bytes32, s:bytes32) -> address:

**return** ecrecover(hash, v, r, s)

**@external**

**@view**

**def** foo(hash: bytes32, v: uint256, r:uint256, s:uint256) -> address:

**return** ecrecover(hash, v, r, s)

**>>>** ExampleContract.foo('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

**Data Manipulation**

**concat(*a*, *b*, *\*args*)→ Union[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](https://docs.vyperlang.org/en/stable/types.html#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 (**bytes32**, **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: Union[Bytes, bytes32, String]*, *start: uint256*, *length: uint256*)→ Union[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"

**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

**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.

**@external**

**@view**

**def** foo(d: decimal) -> decimal:

**return** 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**.

**@external**

**@view**

**def** foo(x: uint256) -> uint256:

**return** 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).

**Utilities**

**as\_wei\_value(*\_value*, *unit:***[***str***](https://docs.python.org/3.8/library/stdtypes.html#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

**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

**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: Union[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***](https://docs.python.org/3.8/library/functions.html#type)***= Bytes[4]*)→ Union[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***](https://docs.python.org/3.8/library/functions.html#bool)***= True*)→ 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"

**\_abi\_decode(*b: Bytes*, *output\_type: type\_*, *unwrap\_tuple:***[***bool***](https://docs.python.org/3.8/library/functions.html#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

**print(*\*args*, *hardhat\_compat=False*)→**[**None**](https://docs.python.org/3.8/library/constants.html#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.