



Mojo types & literals cheat sheet

Numbers, SIMD, conversions, and how to write values.

v1.3dev

MOJO NUMBERS ARE SIMD VECTORS

SIMD IS THE FOUNDATION

Every fixed-width number is a 1-lane SIMD. A `Float32` is a `Scalar[DType.float32]` is a `SIMD[DType.float32, 1]`.

Operations on SIMD (such as `cast()`) work on `Scalar` and items declared with named type aliases (like `Float64`, `Int16`):

```
var float: Float64 = 42.0 # 64b 42.0
var float32 = float.cast[.float32]() # 32b 42.0
var scalar = Scalar[.float64](float)
var scalar16 = scalar.cast[.float16]() # 16b 42.0
```

SIMD VECTORS HAVE FIXED WIDTHS

Widths must be powers of two and are part of the type (`SIMDLength`).

```
var vec = SIMD[.float32, 4](1.0, 2.0, 3.0, 4.0)
var double = vec * 2.0 # [2.0, 4.0, 6.0, 8.0], all lanes
vec[0] = 5.0 # write one lane [5.0, 2.0, 3.0, 4.0]
print(vec.reduce_add()) # sum of lanes (14.0)
```

DTYPE: WHAT A SIMD LANE HOLDS

```
SIMD[DType.float32, 4] # DType picks the lane type
SIMD[.float32, 4] # the same; DType inferred from context
Scalar[.int] # == Int
```

Names mirror the types: `DType.float32` ↔ `Float32`, `DType.int8` ↔ `Int8`, `DType.bool` ↔ `Bool`. A `DType` is a name, not a type. It parameterizes `SIMD`, which stores the data.

SIMD CONSTRUCTION

```
var broadcast = SIMD[.float64, 2](42.0) # broadcast all lanes
var specific = SIMD[.float64, 2](1.0, 2.0) # specific values
var zeros = SIMD[.float64, 2]() # zero-initialized vector
```

NUMBERS

INTEGERS

```
var n = 42 # Int: machine width
var u: UInt = 42 # machine width
var small: UInt8 = 255
var big: Int64 = -9_000_000_000
```

TYPE	MEANING
<code>Int</code> and <code>UInt</code>	machine word (typically 64-bit)
<code>Int8</code> .. <code>Int256</code>	sized signed
<code>UInt8</code> .. <code>UInt256</code>	sized unsigned
<code>Byte</code>	alias for <code>UInt8</code>

Use `Int` for counts and indices; sized types when bit width is part of the contract. Each is an alias for a 1-lane SIMD.

FLOATING POINT

TYPE	MEANING
<code>Float64</code>	IEEE double (default)
<code>Float32</code>	IEEE single
<code>Float16</code>	IEEE half
<code>BFloat16</code>	brain float (ML training)
<code>Float8_e4m3fn</code> ...	8-bit (GPU, ML)
<code>Float4_e2m1fn</code>	4-bit (Blackwell+)

No bare `Float` type. Each is an alias for a 1-lane SIMD.

OPERATIONS

ELEMENT OPERATIONS

TYPE	OPERATIONS
Arithmetic	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code>//</code>
Comparison	<code>==</code> , <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code>
Math functions	<code>sqrt()</code> , <code>sin()</code> , <code>cos()</code> , <code>fma()</code> , etc.
Bit operations	<code>&</code> , <code> </code> , <code>^</code> , <code>~</code> , <code><<</code> , <code>>></code>

VECTOR OPERATIONS

TYPE	OPERATIONS
Horizontal reductions	<code>reduce_add()</code> , <code>reduce_mul()</code> , <code>reduce_min()</code> , <code>reduce_max()</code>
Vector manipulation	<code>shuffle()</code> , <code>slice()</code> , <code>join()</code> , <code>split()</code>

NUMBER FACTS

BOUNDS & SPECIAL VALUES

NAME	MEANING
bit_width_of[Int]()	64 on most platforms (from std.sys.info)
UInt8.MAX	255
Int8.MIN	-128
Float32.MAX_FINITE	largest finite
Float32.MAX	inf

IEEE floats carry `inf`, `-inf`, `nan`, `-0.0`. Make them with `inf[DType.float64]()` and `nan[DType.float64]()` from `std.math`.

NUMBER CONVERSIONS ARE EXPLICIT

In addition to `cast()`, you can use named type aliases:

```
var int = 42 # Int
var f64 = Float64(int) # Int -> Float64
var i8 = Int8(int) # Int -> Int8
var back = Int(Int64(int)) # round trip
```

Without explicit typing, integer literals are `Int`.

Without explicit typing, float literals are `Float64`.

Float literals can't become integers, even with explicit typing.

NUMBER LITERALS

LITERAL	MEANING
42	decimal Int
0xFF 0o52 0b1010	hex, octal, binary
1_000_000	underscores group digits
3.14 .5 2. 2.5e-3	floats
2 ** 200	comptime IntLiteral, comptime arbitrary precision

Leading zeros on base-10 integers are rejected. At runtime literals materialize to `Int` / `Float64`.

COLLECTIONS

COLLECTION LITERALS

COLLECTION TYPES

```
[1, 2, 3] # Array, length in the type
{"id": 1, "qty": 9} # Dict
(1, "a", 2.0) # Tuple, mixed types
{1, 2, 3} # Set
```

Bracket literals default to `Array`. For lists, use: `var x: List = [ ... ]` or `var x: List[Type] = [ ... ]`.

EMPTY COLLECTIONS

For empty collections, specify explicit types before the literal.

```
var list: List[ElementType] = []

var dict: Dict[KeyType, ValueType] = {}

var set: Set[ElementType] = {}
```

An empty `Array` has no use; arrays are fixed size.
You must import for the empty `Set`: `from std.collections import Set`

OPTIONALS

Optionals represent values that may or may not be present.

```
# Initialize
var foo: Optional[Int] = None
var bar: Optional[Int] = 42

# Access with default fallback
print(foo.or_else(0))
print(bar.or_else(0))

# Check then access the value
if foo:
    print(foo.value())
if bar:
    print(bar.value())
```

STRINGS

STRING LITERALS

Mojo strings use UTF-8 encoding. A `String` is mutable.

```
"double" 'single'

# triple quotes: newlines and indentation included
"""line one
   line two"""

r"C:\raw\path" # raw: no escape processing

"\u20AC" # lowercase \u, 4 digits: € (EURO)
"\U0001F44B" # uppercase \U, 8 digits: 🍗 (above U+FFFF)

# adjacent literals join, same line or across lines:
"Hello" " world!" # -> "Hello world!"
"Content of line 1. "
"Content of line 2."
```

Escapes:

ESCAPE	MEANING
\n \t \" \\	newline, tab, quote, backslash
\xFF	byte (2 hex digits)
\UHHHH	Unicode (4 hex digits)
\UHHHHHHHH	Unicode (8 hex digits)

`\u` and `\U` reject surrogate code points (U+D800 to U+DFFF).
Code points above U+FFFF need `\u`, not a surrogate pair.

STRING LENGTH

```
var s = "héllö"
s.byte_length() # 6: UTF-8 bytes
len(s.codepoints()) # 5: Unicode code points
len(s.graphemes()) # 5: user-visible characters
```

`len(string)` won't compile. Name the strategy you mean:

```
var text = "café"
text.byte_length() # 5, é is 2 bytes
text.count_codepoints() # 4, é is 1 codepoint
text.count_graphemes() # 4, é is 1 user-visible
character
```

Counting codepoints and graphemes are $O(n)$ operations, where n is the string length in bytes.

TSTRINGS

TStrings are not strings. They are templates that generate strings at runtime by interpolating expressions within curly braces:

```
var who = "Mojo"
t"Hi, {who}!" # interpolation
t"sum = {1 + 2}" # any expression
t"{{literal braces}}" # -> {literal braces}
rt"raw\path {who}" # raw TString: \ literal
                    # still interpolates
String(t"x = {who}") # cast to String
```

Other than `print()`, cast to `String` for any context expecting a string value.

WORTH KNOWING

NO CHARACTER TYPE

'A' is a one-letter string, the same as "A". Use `ord("A")` (65) and `chr(66)` ("B") for code points.

FORMATTING

TStrings don't take format specifiers: `t"{x:.2f}"` won't compile. Use `std.math` utilities like `round()` or `String` utilities like `ascii_rjust()` instead.

DIVISION

`/` is true division on floats and truncates toward zero on integer values.

`//` floors for all numbers. With `a = -7`, `a / 2` is `-3` and `a // 2` is `-4`.

The literal expression `7 / 2` is `3.5`.

SHARP EDGES

Int width is platform-dependent: use `Int64` for a fixed width.

Integer overflows wrap: `Int8(127) + 1` is `-128`. Shifts at or above the bit width are undefined. Integer literals wrap silently: `var b: Int8 = 300` is `44`, and `var u: UInt8 = -1` is `255`.

Float-to-integer truncates toward zero: `Int(Float64(3.9))` is `3`.

`Int128 / Int256` are software-emulated.

TAKEAWAYS

OTHER THINGS

NAME	MEANING
<code>True</code> <code>False</code>	boolean values
<code>None</code>	the only <code>NoneType</code> value
<code>Self</code>	the enclosing type
<code>-</code>	discard a value in assignment
<code>...</code>	marks a required trait method