Mojo Types & Literals

Numbers, SIMD, conversions, and how to write values

Mojo language reference
v1.0.0b2

keyword type built-in "string" number operator @decorator True/False/None # comment

SIMD IS THE FOUNDATION

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

var v = SIMD[DType.float32, 4](1.0, 2.0, 3.0, 4.0)
var d = v * 2.0 # [2, 4, 6, 8], all lanes
v[0] = 5.0 # write one lane
print(v.reduce_add()) # sum of lanes (14.0)
```

Width must be a power of two and is part of the type; its parameter type is **SIMDSize**.

DTYPE: WHAT A LANE HOLDS

```
SIMD[DType.float32, 4] # DType picks the lane type
Scalar[DType.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.

INTEGERS

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

`Int / UInt`	machine word (typically 64-bit)
`Int8 … Int256`	sized signed
`UInt8 … UInt256`	sized unsigned
`Byte`	alias for UInt8

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

`Float64`	IEEE double (default)
`Float32`	IEEE single
`Float16`	IEEE half
`BFloat16`	brain float (ML training)
`Float8_e4m3fn …`	8-bit (GPU, ML)
`Float4_e2m1fn`	4-bit (Blackwell+)

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

BOUNDS & SPECIAL VALUES

`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`	may be inf

IEEE floats carry **inf**, **-inf**, **nan**, **-0.0**.

CONVERSIONS ARE EXPLICIT

```
var i = 42
var f = Float64(i) # Int -> Float64
var s = Int8(i) # Int -> Int8
var back = Int(Int64(i)) # round trip

# between SIMD-based types: .cast[]
var g = f.cast[DType.int32]()
```

Variables never convert implicitly; the compiler enforces it. Literals convert only when it can prove the result is exact.

NUMBER LITERALS

`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**.

COLLECTION LITERALS

```
[1, 2, 3] # List
{"id": 1, "qty": 9} # Dict
(1, "a", 2.0) # Tuple, mixed types
```

Element types are inferred. Annotate when they can't be: **var x: List[Int] = [1, 2, 3]**.

STRING LITERALS

```
"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:

`\n \t`	newline, tab
`\" \\`	quote, backslash
`\xHH`	byte (2 hex digits)
`\uHHHH`	Unicode (4 hex digits)
`\UHHHHHHHH`	Unicode (8 hex digits)

Source is UTF-8. **\u** and **\U** reject surrogate code points (U+D800 to U+DFFF); code points above U+FFFF need **\U**, not a surrogate pair.

T-STRINGS

```
var who = "Mojo"

t"Hi, {who}!" # interpolation

t"sum = {1 + 2}" # any expression

t"{{literal braces}}" # -> {literal braces}

rt"raw\path {who}" # raw t-string: \ literal, still
      interpolates

String(t"x = {who}") # cast to String
```

Interpolations evaluate at runtime.

OTHER THINGS

`True False`	boolean values
`None`	the only NoneType value
`Self`	the enclosing type
`_`	discard a value in assignment
`...`	marks a required trait method

SHARP EDGES

- **Int width is platform-dependent** — use Int64 for a fixed width.
- **Integer overflow wraps** — Int8(127) + 1 is -128.
- **Float-to-int truncates toward zero** — Int(Float64(3.9)) is 3.
- **nan == nan is False.**
- **Float8 needs a GPU** — no runtime CPU arithmetic.
- **Int128 / Int256 are software-emulated.**

COMING FROM PYTHON

- No implicit numeric conversion: `Int + Float64` is an error. Cast with `Float64(n)`, `Int(x)`.
- Numbers are fixed-width SIMD scalars, not arbitrary-precision `int`; only a **comptime IntLiteral** is unbounded.
- No bare `float` or `int` — pick **Int**, **Float64**, or a sized type.

COMING FROM C++ / RUST

- Every scalar is a 1-lane **SIMD**; vectorizing widens the lane count, not a new type.
- Integer overflow **wraps** (defined), not C++ undefined behavior or a Rust debug panic.
- **DType** is a value-level tag that parameterizes **SIMD**, not a type alias.
- **Int** is C++ `ssize_t` / Rust `isize`, not C++ `int`, which is usually 32-bit.