



Mojo conversions cheat sheet

Make one, convert one, access one. Remember to cast.

v1.3dev

CONVERSION

ESTABLISHING TYPES sideways conversions

To initialize across types, use a value that conforms to `Intable`, `Floatable`, `Boolable`, or (for string destinations) `Writable`. Use a raising form when conversions can fail: `IntableRaising`, `FloatableRaising`.

```
Int(32), Int16(32) # explicit typing from literal expressions
```

```
Int64(f), Float64(i), String(v) # assign new forms across types
```

```
Int(s), Float64(s) # conversions can fail; raising
```

Each trait defines the types it builds:

TRAIT	BUILDS
<code>Intable</code>	any integer type: <code>Int</code> , <code>Int16</code> , <code>UInt8</code> , ...
<code>Floatable</code>	<code>Float64</code> only
<code>IntableRaising</code>	<code>Int</code> only
<code>FloatableRaising</code>	<code>Float64</code> only

For numeric parsing, strings convert only to `Int` and `Float64`, so `Int16(s)` won't compile. Convert to `Int` or `Float64`, then narrow—for example, `Int16(Int(s))`.

String conversions can fail. For example: `Int("two")` raises from `IntableRaising`.

TEXT

STRING owned, growable UTF-8 text

MAKE

```
String(x) # any StringSpan, StringLiteral, Writable  
String("x") # realize the string
```

```
String(from_utf8=bytes) # raises on bad UTF-8  
String(from_utf8_lossy=bytes) # replaces bad bytes
```

CONVERT

```
Int(s) # base-10 parse; raises on "3.5", "0xff",  
      "", overflow  
Float64(s) # parse (1e3, inf ok); raises "",  
          garbage  
Bool(s) # True if non-empty
```

ACCESS (BY BYTE / CODEPOINT / GRAPHEME)

```
# also for codepoint and single index grapheme  
s[byte=i], s[byte=i:j]  
  
# byte Span; iterators  
s.as_bytes(), s.codepoints(), s.graphemes()
```

NUMBERS

NUMERIC CREATION create numeric values of known types

Scalars are 1-lane SIMD values: `SIMD[DType, 1] == Scalar[DType]`

`Int` is the machine-width integer; `Float64` is the default floating point

Integer-like literal expressions are `Int` by default, floating-point are `Float64`

No bare `Float`, and no `Float128/256`

Initialize literals to set the destination type (`Int32(42)`, `Float16(3.14)`)

CASTING convert between numeric types

A same-width conversion between signed and unsigned keeps the bits exactly. Only the interpretation changes. For example, `UInt(Int(-1))` is `UInt.MAX`, the machine-width unsigned maximum.

```
# convert a value, scalar, or vector with `.cast()`  
f64.cast[.int32]() # convert: 1.0 -> 1
```

```
# reinterpret existing bits  
from std.memory import bitcast  
bitcast[.uint32](f32)
```

Numbers don't convert implicitly:

```
var result = f64 + i32 # no  
var result = f64 + Float64(i32) # yes
```

Mixed bitwidths require explicit casts. Narrowing may degrade precision. Widening preserves precision:

```
Float64(f32) + f64 # widen one side to match
```

Using `.cast()` converts between dtypes, including `Int` and `Float64`

Integer casts wrap with two's complement

`Int` and `UInt` share their machine-dependent width: converting `Int(-1)` produces `UInt.MAX`

Float to integer truncates toward zero

```
i64.cast[DType.int8]() # to another integer dtype;  
wraps  
fp64.cast[DType.float32]() # narrows (precision  
loss)
```

BOOL True / False, and what's truthy

TRUTHY IS FOR IF / WHILE / AND / OR / BOOL().

KIND	TRUTHY WHEN
Numbers	non-zero
Strings & collections	non-empty
Optional	None False, else True
PyObject	Python's own rules

Converting a Bool to a number is explicit: Int(b). A type is truthy when it conforms to Boolable and implements __bool__().

SIMD[T, N] N lanes of one dtype; Scalar = SIMD[T, 1]

MAKE

```
SIMD[T, N](x) # splat one value to all lanes
SIMD[T, 4](a, b, c, d) # per-lane
```

CONVERT

```
v.cast[DType.x]() # new dtype, same lane count
SIMD[T, N](scalar) # splat a Scalar up to N lanes
```

N is the lane count, not bit width; a lane's bit width is its dtype. Int, Float64, Int8 ... are all SIMD scalars.

COLLECTIONS

COLLECTION LITERALS what a bracket or brace makes

LITERAL	TYPE	ANNOTATE?
[v1, v2, ...]	Array	No
[v1, v2, ...]	List	Yes: var x: List = [...]
{k1: v1, k2: v2, ...}	Dict	No

Optional has no literal shorthand.

ARRAY[T, N]

fixed-size, homogeneous; the bracket-literal default

MAKE

```
[1, 2, 3] # unannotated literal -> Array[Int, 3]
Array[T, n](fill=x) # n copies of x
Array[T, n](fill_with=f) # f(i) for each index
Array[T, n]() # default values (T: Defaultable)
Array[T, n](uninitialized=True) # unsafe;
initialize slots with unsafe_write()
```

CONVERT

```
List(a) # Array -> List: copies onto the heap
Array[T, n](
  fill_with=lambda (i: Int) {imm l} -> T: l[i]
) # List -> Array; you pick n
Span(a), Span(1) # view either one, no copy
a.concat(b) # moves both into Array[T, n + m]
a.copy() # explicit; no implicit copy
```

ACCESS

```
a[i] # ref; a constant index is checked at compile time
a.unsafe_ptr() # -> Pointer[T] to inline, contiguous storage
a.unsafe_ptr().unsafe_bitcast[NoneType]() # type-erased, for C void *
```

Elements sit inline and back to back, like a C array, with no heap. Pass one to C or to any library that takes a raw block. For mixed types, use Tuple.

Array: size in the type, stored inline, for C blocks, tables, and GPU arguments. List: size at runtime, on the heap, and it grows.

LIST[T] growable, homogeneous sequence

MAKE

```
var x: List[T] = [a, b, c] # annotate: unannotated defaults to Array
List[T](capacity=n) # empty; initial room for n
List[T](length=n, fill=x) # n copies of x (T: Copyable)
List(range(n)) # materialize a range
List(iterable) # from any iterator / iterable
```

ACCESS

```
list[i], list[i:j] # element by ref, Span view (no copy)
list.unsafe_take_allocation() # hand off the buffer as an Allocation[T]
```

SIMD, ARRAY, AND LIST

move whole vectors in and out of memory

FROM → TO	CODE	NOTE
Array → List	List(a)	copies
List → Array	1.unsafe_ptr().unsafe_bitcast[Array[T, N]]().copy()	T: Copyable
Array → SIMD	a.unsafe_ptr().unsafe_load[width=N]()	numeric, N a power of 2
SIMD → Array	a.unsafe_ptr().unsafe_store(v)	into an existing Array
List → SIMD	1.unsafe_ptr().unsafe_load[width=N]()	numeric, N a power of 2
SIMD → List	1.unsafe_ptr().unsafe_store(v)	needs len(1) >= N

N must be known at compile time: compiletime N = type_of(a).length, not len(a). Nothing checks a List's length; a short one reads or writes past its end.

Elements are Scalar[dtype], such as Float32. A load or store moves all N lanes at once and checks nothing: a List shorter than N reads past its end.

DICT[K, V] hash map; keys Hashable + Equatable + Movable

MAKE

```
Dict[K, V]() # empty; fill with d[k] = v
Dict[K, V](capacity=n) # empty; initial room for n
Dict.fromkeys(keys, v) # every key maps to v
```

ACCESS

```
d.setdefault(key, default) # ref; inserts default
if absent
d.get(key), d.find(key) # Optional[V]
d.get(key, default) # value, or default; never
raises
d.keys() d.values() # lazy iterators
d.items() # iterator of DictEntry (.key / .value)
d.pop(key) # value, removes it
```

OPTIONAL[T] a value, or nothing

MAKE

```
Optional(x) # from a value (T inferred from x)
Optional[T](), Optional[T](None) # empty
```

ACCESS

```
o.value(), o.take(), o[] # ref, move out, ref
(abort, abort, raise)
o.or_else(default) # value, or default
```

POINTERS

POINTER[T] to backing buffer

Use `unsafe_ptr()` to access: `List`, `String`, `StringSpan`, `Array`, and `Span`.

ACCESS THROUGH A POINTER

```
buf.unsafe_ptr() # -> Pointer[T]
p[unsafe_offset=i] # deref one element
p.unsafe_offset(i)[] # pointer arithmetic, then
deref
p.unsafe_load[width=N]() # read N lanes -> SIMD[T,
N]
```

VECTORIZE A BUFFER (THE ESCAPE HATCH)

```
var v = data.unsafe_ptr().unsafe_load[width=8]() #
8 elements -> one SIMD
var total = v.reduce_add() # SIMD-wide reduce
```

POINTER SAFETY

Unsafe operations carry the `unsafe_` prefix or an `unsafe_` keyword

`unsafe_ptr()` has contiguous storage and is never null, even when empty. Check lengths before access

`Pointer` is non-null by design; use `OptionalPointer` when the pointer may be null

The pointer's origin keeps its owner alive until its last use, but it doesn't protect against reallocation. After a `List` grows or a `String` changes, reacquire the pointer before using it

`String.unsafe_ptr()` is read-only