Mojo conversions

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

Mojo language reference
v1.0.0b2

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

NUMBERS

INT machine-width integer; a 1-lane SIMD

MAKE

```
42 # IntLiteral
Int(x) # any Intable: Bool, Int8 …
Int(s) # any Intable that's raising: String
Int(Int32(5)) # any numeric scalar, cross-dtype
```

CONVERT

```
Float64(i), UInt(i) # to float, to unsigned (same bits)
String(i), Bool(i) # to text, True if non-zero
i.cast[DType.int8]() # to another dtype
```

Float to Int truncates toward zero; cross-dtype casts wrap (two's complement). **Int** is **Scalar[DType.int]**, so **.cast** works like any SIMD scalar.

Int and **UInt** share bits: Int(-1) is UInt 2^64-1.

FLOAT64 IEEE double; a 1-lane SIMD

MAKE

```
3.14 # FloatLiteral
Float64(x) # any Floatable: Int, Bool
Float64(s) # any Floatable that's raising: String
Float32(x).cast[DType.float64]() # widen a scalar
```

CONVERT

```
Int(f) # truncates toward zero
f.cast[DType.float32]() # narrow (precision loss)
Bool(f), String(f) # True if non-zero, to text
```

No bare **Float**, and no Float128/256. Convert all other floats (including special purpose) with **.cast()**.

BOOL true / false, and what's truthy

MAKE

```
Bool(x) # any Boolable
```

CONVERT

```
Int(b), Float64(b), String(b) # 0 or 1, 0.0 or 1.0, "True" or "False"
```

TRUTHY for if / while / and / or / Bool()

Numbers	non-zero
Strings & collections	non-empty
Optional	None False, else True
PythonObject	Python's own rules

Bool to **Int** via **__as_int__** (the one implicit numeric coercion). Any type with a **__bool__** is truthy.

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.

TEXT

STRING owned, growable UTF-8 text

MAKE

```
String(x) # any StringSlice, StringLiteral, Writable (Int, Float64, Bool …)
String(t"{x}") # Not needed for print()
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", ""
Float64(s) # parse (1e3, inf ok); raises "", garbage
Bool(s) # True if non-empty
```

ACCESS (BY BYTE / CODEPOINT / GRAPHEME)

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

POINTERS

UNSAFEPOINTER[T] to backing buffer

Use **unsafe_ptr()** to access: **List**, **String**, **StringSlice**, **InlineArray**, and **Span**.

ACCESS THROUGH A POINTER

```
buf.unsafe_ptr() # -> UnsafePointer[T]
p[i] # deref one element
(p + i)[] # pointer arithmetic, then deref
p.load[width=N]() # read N lanes -> SIMD[T, N]
list.steal_data() # take a List's buffer (consumes it)
```

VECTORIZE A BUFFER (THE ESCAPE HATCH)

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

UnsafePointer is non-null by design. Use **Optional[UnsafePointer]** for a nullable pointer.

COLLECTIONS

LIST[T] growable, homogeneous sequence

MAKE

```
[a, b, c] # list literal, all one type
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.steal_data() # take the buffer as an UnsafePointer, consuming the list
```

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.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
```