Mojo Basics

Syntax essentials: the core you reach for first

Mojo language reference
v1.0.0b2

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

HELLO, MOJO

```
def main():
    print("Hello, Mojo!")
```

Run it: **mojo hello.mojo**. Every program starts at **main()**.

COMMENTS & DOCSTRINGS

```
# Line comment

def greet():
    """Docstring: what greet does."""
    print("hi")
```

VAR OWNS, REF REFERS

```
var count = 0 # owned, inferred Int
var name: String = "Mojo"
count = count + 1 # var is mutable

var data = [1, 2, 3]
ref view = data[0] # ref to an element, no copy
view = 99 # writes through to data

comptime PI = 3.14159 # compile-time const
```

var declares an owned value, mutable by default. **ref** is a reference to a value it doesn't own. Mutable changes update the original value.

BUILT-IN TYPES

```
Float32 == Scalar[DType.float32]
        == SIMD[DType.float32, 1]
```

Int	machine-word integer; default index type
UInt	machine-word
Int8 … Int64	sized integers
Float64	default floating point (also Float32, Float16, and special-purpose 8-bit FP8 and 4-bit FP4 floats)
Bool	True / False
String	UTF-8, supports Unicode graphemes
List[T]	array, fixed size or growing
SIMD[dt, n]	n-wide numeric vector

Numeric types are **SIMD** vectors under the hood: scaling to vector math is built in.

No implicit numeric conversion: cast explicitly with **Float64(n)**, **Int(x)**, **String(v)**. Use **.cast** for SIMD vectors.

Types are **PascalCase** (**Int**); names are **lower_snake_case**.

OPERATORS

+ - * /	add, subtract, multiply, divide
// %	floor divide, modulo
**	power (2 ** 10), also pow(2, 10)
== !=	equal, not equal
< <= > >=	comparisons (chainable)
and or not	logical, short-circuit
+= -=	compound assign (*=, /=, ...)

```
a < b < c # chains to (a < b) and (b < c)
```

STRINGS & PRINTING

```
var who = "Mojo"
print("Hi, " + who) # concatenation
print(t"Hi, {who}!") # interpolation
print(1, 2, 3, end=": ") # keyword args

var s = String(t"x = {1 + 1}") # to String
var raw = r"C:\path" # raw string
```

print takes a t-string directly; cast with **String(...)** to use one elsewhere. Triple quotes make multi-line strings.

CONTROL FLOW

```
if x > 0:
    print("positive")
elif x == 0:
    print("zero")
else:
    print("negative")

# ternary
var kind = "even" if x % 2 == 0 else "odd"
```

LOOPS

```
for i in range(5): # 0 1 2 3 4
    print(i)

for item in [10, 20, 30]: # iterate an array
    if item == 20:
        continue # skip to next
    if item == 30:
        break # stop the loop
    print(item)

while n > 0: # loop while true
    print(n)
    n -= 1
```

Repeat n times with **for _ in range(n)** (_ discards the value).

FUNCTIONS

```
def add(a: Int, b: Int) -> Int:
    return a + b

def greet(name: String = "world"): # default
    print(t"Hi, {name}")

def risky() raises: # may raise
    raise Error("boom")

def nothing():
    pass # do-nothing body
```

No `->` means the function returns **None**. `def` can raise only if marked **raises**, so callers can see it coming.

IMPORTS

```
from std.math import sqrt # one name
from std.math import sqrt as root # aliased
```

Built-ins like **Int**, **String**, **List**, and **print** are in the Mojo prelude, no import needed.

LISTS

```
var xs: List[Int] = [1, 2, 3]
xs.append(4)
print(xs[0]) # 1
print(len(xs)) # 4
```

List[T] needs its element type; omit it and inference can surprise you.

STRUCTS

```
@fieldwise_init # synthesizes __init__
struct Point:
    var x: Int
    var y: Int

struct Counter:
    var n: Int
    def __init__(out self): # builds self
        self.n = 0
    def bump(mut self): # modifies self
        self.n += 1

def main():
    var p = Point(3, 4)
    print(p.x, p.y) # 3 4
```

Every instance method takes **self** as its first argument. The **out** convention returns the initialized **self** without a return arrow. Structs also support **comptime** constants and static methods (**@staticmethod**).

ERRORS

```
def risky() raises:
    raise Error("boom")

def main():
    try:
        risky()
    except e:
        print("caught:", e)
```

Mark a raising function **raises**; **raise** signals, **try**/**except** catches. **except e** binds the error.

COMING FROM PYTHON

- Every value has a fixed type. No implicit numeric conversion: write `Float64(n)`, `Int(x)`.
- Assignment copies the value; it is not a shared reference.
- String interpolation is `t"…"`, not `f"…"`.
- Declare with **var** (and **ref**) rather than a bare `x = 5`.
- No **match** yet: use `if/elif/else`.
- A `def` with no `->` returns **None**, same as Python.

COMING FROM C++ / RUST

- Python-style syntax: indentation and `def`, no braces, semicolons, or headers.
- Value semantics with moves: `^` transfers ownership (like `std::move` or a Rust move); `__del__` gives RAII cleanup.
- Behavior comes from **traits**, not class inheritance or templates (like Rust traits or C++20 concepts).
- No `?:` ternary or Elvis operator: write `a if cond else b`.
- No **switch** yet: use `if/elif/else`.