



Mojo basics cheat sheet

Syntax essentials: the core you reach for first.

v1.3dev

HELLO, MOJO (APPLICATION)

Basic structure of a Mojo application.

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

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

Build it: `mojo build hello.mojo`. Creates a `hello` executable.

COMMENTS & DOCSTRINGS

How to write comments and docstrings in Mojo.

```
"""Docstring: what this module (file) does.  
  
Further information goes here.  
"""  
  
# Single-line comment  
  
def greet():  
    """Docstring: what greet does."""  
    print("hi")
```

Triple quotes make multi-line strings.

DECLARE VARIABLES: VAR OWNS, REF REFERS

How to declare and use variables in Mojo.

```
var count = 0 # owned value, `Int` inferred  
var name: String = "Mojo" # explicit `String`  
count = count + 1 # var is mutable  
  
var data: List = [1, 2, 3]  
ref view = data[0] # ref to an element, no copy  
view = 99 # writes through to `data`'s value  
  
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. When mutable, changes update the value.

BUILT-IN NUMBER TYPES

Overview of Mojo's numeric types.

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

TYPE	MEANING
Int	machine-word integer; default index type
UInt	machine-word unsigned integer
Int8 ... Int64	sized and signed integers
Float64	default floating point
Float16, Float32	half and single precision; FP8 and FP4 formats also available.
Bool	True / False (uppercase)
SIMD[dt, n]	n-wide numeric vector; for n = 1: <code>Scalar[dt]</code>

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

For types that are known, you can drop the type name before the dot. For example, `SIMD[.float32, 8]` is equivalent to `SIMD[DType.float32, 8]`.

`SIMD` vectors always use a power-of-two width.

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

OPERATORS

 Common operators in Mojo.

OP	MEANING
+ - * /	add, subtract, multiply, divide
// %	floor divide, modulo
**	power (2 ** 10), also <code>pow(2, 10)</code> , but not <code>^</code>
== !=	equality and inequality
< <= > >=	comparisons (chainable)
and or not	logical, short-circuit
+= -=	compound assign (*=, /=, ...)
a < b <= c # chains to (a < b) and (b <= c)	

OTHER STANDARD TYPES

Overview of other standard types in Mojo.

TYPE	MEANING
String	UTF-8 with Unicode grapheme support
Array[T, n]	fixed-size, homogeneous sequence
Dict[K, V]	key-value mapping, homogeneous
List[T]	growable, homogeneous sequence

Use arrays for accelerator work and crossing foreign function boundaries. Arrays are fixed-size. You can't append to them.

Types are *PascalCase* (`Array[Int, 1024]`); names are *lower_snake_case* (`count`, `max_value`).

STRINGS & PRINTING

How to work with strings and printing in Mojo.

```
var who = "Mojo"
print("Hi, " + who) # concatenation
print(t"Hi, {who}!") # interpolation, `t` prefix
print(1, 2, 3, sep=": ") # keyword args, '1: 2: 3'

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

`print()` takes `t`-strings directly. Cast with `String(...)` to use one elsewhere.

CONTROL FLOW

Conditional statements and pattern matching in Mojo.

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

# Pattern matching syntax
match x:
    case 2: # Match equality
        print("is exactly two")
    case _ if x.is_power_of_two(): # Match with
condition
        print("power of two")
    case _:
        print("not power of two") # Universal
catch-all match

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

Cases are checked top to bottom, and only the first match runs.

LOOPS

 Looping constructs in Mojo.

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

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

var n = 4
while n > 0: # loop while true
    print(n) # 4, then 3, 2, 1
    n -= 1
```

Repeat `n` times with `for _ in range(n):`. Using the discard pattern `_` ignores the value.

FUNCTIONS, CLOSURES, AND LAMBDA EXPRESSIONS

Defining and using functions, closures, and lambda expressions in Mojo.

```
def add_two(a: Int, b: Int) -> Int: # basic
    return a + b
print(add_two(2, 3)) # 5

var c = 10
def add_c(a: Int) {imm c} -> Int: # closure
    return a + c
print(add_c(5)) # 15

var anon = lambda (a: Int) {imm c} -> Int: a + c
print(anon(10)) # 20, the same as add_c

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

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

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

No return arrow means a function returns `None`.

`def` can raise only if marked `raises`, so callers can see it coming.

IMPORTS

 How to import modules and names in Mojo.

```
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

 Working with lists in Mojo.

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

Always type annotate `List`. The default type for a bracket list is the fixed-size `Array`.

DICTIONARIES

 Working with dictionaries in Mojo.

```
var xs: Dict = {"a": 1, "b": 2}
xs["c"] = 3
print(xs.get("a", 0)) # 1
print(len(xs)) # 3
```

When reading, `get()` with a default fallback won't raise. Direct reads, like `xs["d"]`, raise an error when the key is missing.

STRUCTS

Defining and using structs in Mojo.

```
@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 needing a return arrow.

Structs support `comptime` members and static methods (`@staticmethod`).

ERRORS

Error handling in Mojo.

```
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 DYNAMIC LANGUAGES

Key differences when transitioning from dynamic languages to Mojo.

Every value has a fixed type. No implicit numeric conversion: write `Float64(n)`, `Int(x)`.

Assignment copies the value implicitly or explicitly (adding `.copy()`); it is not a shared reference.

Declare with `var` (and `ref`) rather than a bare `x = 5`.

Template strings use `t"..."`, not `f"..."`.

COMING FROM SYSTEMS LANGUAGES

Key differences when transitioning from systems languages to Mojo.

Python-style syntax: indentation and `def`, no braces around blocks, no headers, parentheses for line continuation.

Value semantics with transfer: `^` transfers ownership of a value; `__deinit__()` gives RAI cleanup.

Reusable behavior comes from `traits`, not class inheritance or templates.
