Types

All values in Mojo have an associated data type. Most of the types are *nominal* types, defined by a `struct`. These types are nominal (or "named") because type equality is determined by the type's *name*, not its *structure*.

There are a some types that aren't defined as structs:

- Functions are typed based on their signatures.
- `NoneType` is a type with one instance, the `None` object, which is used to signal "no value."

Mojo comes with a standard library that provides a number of useful types and utility functions. These standard types aren't privileged. Each of the standard library types is defined just like user-defined types—even basic types like `Int` and `String`. But these standard library types are the building blocks you'll use for most Mojo programs.

The most common types are *built-in types*, which are always available and don't need to be imported. These include types for numeric values, strings, boolean values, and others.

The standard library also includes many more types that you can import as needed, including collection types, utilities for interacting with the filesystem and getting system information, and so on.

Numeric types

Mojo's most basic numeric type is `Int`, which represents a signed integer of the largest size supported by the system—typically 64 bits or 32 bits.

Mojo also has built-in types for integer and floating-point values of various precisions:

Type name	Description
`Int8`	8-bit signed integer
`UInt8`	8-bit unsigned integer
`Int16`	16-bit signed integer
`UInt16`	16-bit unsigned integer

Type name	Description
`Int32`	32-bit signed integer
`UInt32`	32-bit unsigned integer
`Int64`	64-bit signed integer
`UInt64`	64-bit unsigned integer
`Float16`	16-bit floating point number (IEEE 754-2008 binary16)
`Float32`	32-bit floating point number (IEEE 754-2008 binary32)
`Float64`	64-bit floating point number (IEEE 754-2008 binary64)

Table 1. Numeric types with specific precision

The types in Table 1 are actually all aliases to a single type, `SIMD`, which is discussed later.

All of the numeric types support the usual numeric and bitwise operators. The `math` module provides a number of additional math functions.

You may wonder when to use `Int` and when to use the other integer types. In general, `Int` is good safe default when you need an integer type and you don't require a specific bit width. Using `Int` as the default integer type for APIs makes APIs more consistent and predictable.

Floating-point numbers

Floating-point types represent real numbers. Because not all real numbers can be expressed in a finite number of bits, floating-point numbers can't represent every value exactly.

The floating-point types listed in Table 1—`Float64`, `Float32`, and `Float16`—follow the IEEE 754-2008 standard for representing floating-point values. Each type includes a sign bit, one set of bits representing an exponent, and another set representing the fraction or mantissa. Table 2 shows how each of these types are represented in memory.

Type name	Sign	Exponent	Mantissa
`Float64`	1 bit	11 bits	52 bits

Type name	Sign	Exponent	Mantissa
`Float32`	1 bit	8 bits	23 bits
`Float16`	1 bit	5 bits	10 bits

Table 2. Details of floating-point types

Numbers with exponent values of all ones or all zeros represent special values, allowing floating-point numbers to represent infinity, negative infinity, signed zeros, and not-a-number (NaN). For more details on how numbers are represented, see IEEE 754 on Wikipedia.

A few things to note with floating-point values:

- Rounding errors. Rounding may produce unexpected results. For example, 1/3 can't be represented exactly in these floating-point formats. The more operations you perform with floating-point numbers, the more the rounding errors accumulate.
- Space between consecutive numbers. The space between consecutive numbers is variable across the range of a floating-point number format. For numbers close to zero, the distance between consecutive numbers is very small. For large positive and negative numbers, the space between consecutive numbers is greater than 1, so it may not be possible to represent consecutive integers.

Because the values are approximate, it is rarely useful to compare them with the equality operator (`==`). Consider the following example:

```
var big_num = 1.0e16
var bigger_num = big_num+1.0
print(big_num == bigger_num)
```

```
True
```

Comparison operators (`<` `>=` and so on) work with floating point numbers. You can also use the `math.isclose()` function to compare whether two floating-point numbers are equal within a specified tolerance.

Numeric literals

In addition to these numeric types, the standard libraries provides integer and floating-point literal types, `IntLiteral` and `FloatLiteral`.

These literal types are used at compile time to represent literal numbers that appear in the code. In general, you should never instantiate these types yourself.

Table 3 summarizes the literal formats you can use to represent numbers.

Format	Examples	Notes
Integer literal	`1760`	Integer literal, in decimal format.
Hexadecimal literal	`0xaa`, `0xFF`	Integer literal, in hexadecimal format. Hex digits are case-insensitive.
Octal literal	`0o77`	Integer literal, in octal format.
Binary literal	`0b0111`	Integer literal, in binary format.
Floating-point literal	`3.14`, `1.2e9`	Floating-point literal. Must include the decimal point to be interpreted as floating-point.

Table 3. Numeric literal formats

At compile time, the literal types are arbitrary-precision (also called infinite-precision) values, so the compiler can perform compile-time calculations without overflow or rounding errors.

At runtime the values are converted to finite-precision types— `Int` for integer values, and `Float64` for floating-point values. (This process of converting a value that can only exist at compile time into a runtime value is called *materialization*.)

The following code sample shows the difference between an arbitrary-precision calculation and the same calculation done using `Float64` values at runtime, which suffers from rounding errors.

```
var arbitrary_precision = 3.0 * (4.0 / 3.0 - 1.0)
# use a variable to force the following calculation to occur at runtime
var three = 3.0
var finite_precision = three * (4.0 / three - 1.0)
print(arbitrary_precision, finite_precision)
```

```
1.0 0.99999999999999978
```

`SIMD` and `DType`

To support high-performance numeric processing, Mojo uses the `SIMD` type as the basis for its numeric types. SIMD (single instruction, multiple data) is a processor technology that allows you to perform an operation on an entire set of operands at once. Mojo's `SIMD` type abstracts SIMD operations. A `SIMD` value represents a SIMD *vector*—that is, a fixed-size array of values that can fit into a processor's register. SIMD vectors are defined by two *parameters*:

- A `DType` value, defining the data type in the vector (for example, 32-bit floating-point numbers).
- The number of elements in the vector, which must be a power of two.

For example, you can define a vector of four `Float32` values like this:

```
var vec = SIMD[DType.float32, 4](3.0, 2.0, 2.0, 1.0)
```

Math operations on SIMD values are applied *elementwise*, on each individual element in the vector. For example:

```
var vec1 = SIMD[DType.int8, 4](2, 3, 5, 7)
var vec2 = SIMD[DType.int8, 4](1, 2, 3, 4)
var product = vec1 * vec2
print(product)
```

```
[2, 6, 15, 28]
```

Scalar values

The `SIMD` module defines several *type aliases* that are shorthand for different types of `SIMD` vectors. In particular, the `Scalar` type is just a `SIMD` vector with a single element. The numeric types listed in Table 1, like `Int8` and `Float32` are actually type aliases for different types of scalar values:

```
alias Scalar = SIMD[size=1]
alias Int8 = Scalar[DType.int8]
alias Float32 = Scalar[DType.float32]
```

This may seem a little confusing at first, but it means that whether you're working with a single `Float32` value or a vector of float32 values, the math operations go through exactly the same code path.

The `DType` type

The `DType` struct describes the different data types that a `SIMD` vector can hold, and defines a number of utility functions for operating on those data types. The `DType` struct defines a set of aliases that act as identifiers for the different data types, like `DType.int8` and `DType.float32`. You use these aliases when declaring a `SIMD` vector:

```
var v: SIMD[DType.float64, 16]
```

Note that `DType.float64` isn't a *type*, it's a value that describes a data type. You can't create a variable with the type `DType.float64`. You can create a variable with the type `SIMD[DType.float64, 1]` (or `Float64`, which is the same thing).

```
from utils.numerics import max_finite, min_finite

def describeDType[dtype: DType]():
    print(dtype, "is floating point:", dtype.is_floating_point())
    print(dtype, "is integral:", dtype.is_integral())
    print("Min/max finite values for", dtype)
    print(min_finite[dtype](), max_finite[dtype]())

describeDType[DType.float32]()
```

```
float32 is floating point: True
float32 is integral: False
Min/max finite values for float32
-3.4028234663852886e+38 3.4028234663852886e+38
```

There are several other data types in the standard library that also use the `DType` abstraction.

Strings

Mojo's `String` type represents a mutable string. (For Python programmers, note that this is different from Python's standard string, which is immutable.) Strings support a variety of operators and common methods.

```
var s: String = "Testing"
s += " Mojo strings"
print(s)
```

```
Testing Mojo strings
```

Most standard library types conform to the `Stringable` trait, which represents a type that can be converted to a string. Use `str(value)` to explicitly convert a value to a string:

```
var s = str("Items in list: ") + str(5)
print(s)
```

```
Items in list: 5
```

String literals

As with numeric types, the standard library includes a string literal type used to represent literal strings in the program source. String literals are enclosed in either single or double quotes.

Adjacent literals are concatenated together, so you can define a long string using a series of literals broken up over several lines:

```
var s = "A very long string which is "
        "broken into two literals for legibility."
```

To define a multi-line string, enclose the literal in three single or double quotes:

```
var s = """
Multi-line string literals let you
enter long blocks of text, including
newlines."""
```

Note that the triple double quote form is also used for API documentation strings.

Unlike `IntLiteral` and `FloatLiteral`, `StringLiteral` doesn't automatically materialize to a runtime type. In some cases, you may need to manually convert `StringLiteral` values to `String` using the built-in `str()` method.

For example, if you want to concatenate string literals to other types, you need to first convert `StringLiteral` to `String` values. This is because many types can be implicitly converted to `String`, but not to `StringLiteral`.

```
# print("Strings play nicely with others: " + True)
# Error: ... right hand side cannot be converted from Bool to StringLiteral
print(str("Strings play nicely with others: ") + str(True))
```

```
Strings play nicely with others: True
```

Booleans

Mojo's `Bool` type represents a boolean value. It can take one of two values, `True` or `False`. You can negate a boolean value using the `not` operator.

```
var conditionA = False
var conditionB: Bool
conditionB = not conditionA
print(conditionA, conditionB)
```

```
False True
```

Many types have a boolean representation. Any type that implements the `Boolable` trait has a boolean representation. As a general principle, collections evaluate as True if they contain any elements, False if they are empty; strings evaluate as True if they have a non-zero length.

Collection types

The Mojo standard library also includes a set of basic collection types that can be used to build more complex data structures:

- `List`, a dynamically-sized array of items.
- `Dict`, an associative array of key-value pairs.
- `Set`, an unordered collection of unique items.
- `Optional` represents a value that may or may not be present.

The collection types are *generic types*: while a given collection can only hold a specific type of value (such as `Int` or `Float64`), you specify the type at compile time using a parameter. For example, you can create a `List` of `Int` values like this:

```
var l = List[Int](1, 2, 3, 4)
# l.append(3.14) # error: FloatLiteral cannot be converted to Int
```

You don't always need to specify the type explicitly. If Mojo can *infer* the type, you can omit it. For example, when you construct a list from a set of integer literals, Mojo creates a `List[Int]`.

```
# Inferred type == Int
var l1 = List(1, 2, 3, 4)
```

Where you need a more flexible collection, the `Variant` type can hold different types of values. For example, a `Variant[Int32, Float64]` can hold either an `Int32` *or* a `Float64` value at any given time. (Using `Variant` is not covered in this section, see the API docs for more information.)

The following sections give brief introduction to the main collection types.

List

`List` is a dynamically-sized array of elements. List elements need to conform to the `CollectionElement` trait, which just means that the items must be copyable and movable. Most of the common standard library primitives, like `Int`, `String`, and `SIMD` conform to this trait. You can create a `List` by passing the element type as a parameter, like this:

```
var l = List[String]()
```

The `List` type supports a subset of the Python `list` API, including the ability to append to the list, pop items out of the list, and access list items using subscript notation.

```
from collections import List

var list = List(2, 3, 5)
list.append(7)
list.append(11)
print("Popping last item from list: ", list.pop())
for idx in range(len(list)):
      print(list[idx], end=", ")
```

```
Popping last item from list:  11
2, 3, 5, 7,
```

Note that the previous code sample leaves out the type parameter when creating the list. Because the list is being created with a set of `Int` values, Mojo can *infer* the type from the arguments.

There are some notable limitations when using `List`:

- You can't currently initialize a list from a list literal, like this:

  ```
  # Doesn't work!
  var list: List[Int] = [2, 3, 5]
  ```

  But you can use variadic arguments to achieve the same thing:

  ```
  var list = List(2, 3, 5)
  ```

- You can't `print()` a list, or convert it directly into a string.

  ```
  # Does not work
  print(list)
  ```

  As shown above, you can print the individual elements in a list as long as they're a `Stringable` type.

- Iterating a `List` currently returns a `Reference` to each item, not the item itself. You can access the item using the dereference operator, `[]`:

```
#: from collections import List
var list = List(2, 3, 4)
```

```
for item in list:
      print(item[], end=", ")
```

```
2, 3, 4,
```

Subscripting in to a list, however, returns the item directly—no need to dereference:

```
#: from collections import List
#: var list = List[Int](2, 3, 4)
for i in range(len(list)):
    print(list[i], end=", ")
```

```
2, 3, 4,
```

Dict

The `Dict` type is an associative array that holds key-value pairs. You can create a `Dict` by specifying the key type and value type as parameters, like this:

```
var values = Dict[String, Float64]()
```

The dictionary's key type must conform to the `KeyElement` trait, and value elements must conform to the `CollectionElement` trait.

You can insert and remove key-value pairs, update the value assigned to a key, and iterate through keys, values, or items in the dictionary.

The `Dict` iterators all yield references, so you need to use the dereference operator `[]` as shown in the following example:

```
from collections import Dict

var d = Dict[String, Float64]()
d["plasticity"] = 3.1
d["elasticity"] = 1.3
d["electricity"] = 9.7
for item in d.items():
    print(item[].key, item[].value)
```

```
plasticity 3.1000000000000001
elasticity 1.3
electricity 9.6999999999999993
```

Set

The `Set` type represents a set of unique values. You can add and remove elements from the set, test whether a value exists in the set, and perform set algebra operations, like unions and intersections between two sets.

Sets are generic and the element type must conform to the `KeyElement` trait.

```
from collections import Set

i_like = Set("sushi", "ice cream", "tacos", "pho")
you_like = Set("burgers", "tacos", "salad", "ice cream")
we_like = i_like.intersection(you_like)

print("We both like:")
for item in we_like:
    print("-", item[])
```

```
We both like:
- ice cream
- tacos
```

Optional

An `Optional` represents a value that may or may not be present. Like the other collection types, it is generic, and can hold any type that conforms to the `CollectionElement` trait.

```
# Two ways to initialize an Optional with a value
var opt1 = Optional(5)
var opt2: Optional[Int] = 5
# Two ways to initialize an Optional with no value
var opt3 = Optional[Int]()
var opt4: Optional[Int] = None
```

An `Optional` evaluates as `True` when it holds a value, `False` otherwise. If the `Optional` holds a value, you can retrieve a reference to the value using the `value()` method. But calling `value()` on an `Optional` with no value results in undefined behavior, so you should always guard a call to `value()` inside a conditional that checks whether a value exists.

```
var opt: Optional[String] = str("Testing")
if opt:
    var value_ref = opt.value()
    print(value_ref)
```

```
Testing
```

Alternately, you can use the `or_else()` method, which returns the stored value if there is one, or a user-specified default value otherwise:

```
var custom_greeting: Optional[String] = None
print(custom_greeting.or_else("Hello"))

custom_greeting = str("Hi")
print(custom_greeting.or_else("Hello"))
```

```
Hello
Hi
```

Register-passable, memory-only, and trivial types

In various places in the documentation you'll see references to register-passable, memory-only, and trivial types. Register-passable and memory-only types are distinguished based on how they hold data:

- Register-passable types are composed exclusively of fixed-size data types, which can (theoretically) be stored in a machine register. A register-passable type can include other types, as long as they are also register-passable. `Int`, `Bool`, and `SIMD`, for example, are all register-passable types. So a register-passable `struct` could include `Int` and `Bool` fields, but not a `String` field. Register-passable types are declared with the `@register_passable` decorator.

  Register-passable types are always passed by value (that is, the values are copied).

- Memory-only types consist of any types that *don't* fit the description of register-passable types. In particular, these types usually have pointers or references to dynamically-allocated memory. `String`, `List`, and `Dict` are all examples of memory-only types.

Our long-term goal is to make this distinction transparent to the user, and ensure all APIs work with both register-passable and memory-only types. But right now you will see some standard library types that only work with register-passable types or only work with memory-only types.

In addition to these two categories, Mojo also has "trivial" types. Conceptually a trivial type is simply a type that doesn't require any custom logic in its lifecycle methods. The bits that make up an instance of a trivial type can be copied or moved without any knowledge of what they do. Currently, trivial types are declared using the `@register_passable(trivial)` decorator. Trivial types shouldn't be limited to only register-passable types, so in the future we intend to separate trivial types from the `@register_passable` decorator.

`AnyType` and `AnyTrivialRegType`

Two other things you'll see in Mojo APIs are references to `AnyType` and `AnyTrivialRegType`. These are effectively *metatypes*, that is, types of types.

- `AnyType` represents any Mojo type. Mojo treats `AnyType` as a special kind of trait, and you'll find more discussion of it on the Traits page.
- `AnyTrivialRegType` is a metatype representing any Mojo type that's marked register passable.

You'll see them in signatures like this:

```
fn any_type_function[ValueType: AnyTrivialRegType](value: ValueType):
    ...
```

You can read this as `any_type_function` has an argument, `value` of type `ValueType`, where `ValueType` is a register-passable type, determined at compile time.

There is still some code like this in the standard library, but it's gradually being migrated to more generic code that doesn't distinguish between register-passable and memory-only types.