8

Gathering data into collections

This chapter covers working with the following types as well as some supporting types:

* Tuple
* List
* Set
* Dict
* Optional
* Variant
* Tensor

In the previous chapter we learned how to write our own custom types with structs. In this chapter we’ll focus on how to use a number of stdlib types from the *collections* module, which each can contain a number of items, that’s why they are called *collections*. The items can either be of the same type (a *homogeneous* collection) like a List, or they can be of different types (a *heterogeneous* collection) like a Tuple. All these types are also *variadic*, they store a variable number of items.

With these collection types you can build more complex data structures, which include collections of struct instances. Working with collections can greatly simplify your code, because each of them has specialized built-in methods to work with and perform operations on the collection. Knowing when and how to use a certain collection is key to writing readable and efficient programs.

With Tuple as the exception, the collections that we will explore are homogeneous,containing only one type of items. This *type* can be specified at comptime as a *parameter* like List[Float64]. That’s why these types are *generic*: you can define them for all types, like a List can contain Int16 values, or Float64, or String, or struct Person instances.

We learned how to use the Python list and dict type in § 5.4; they are known for their wide functionality. But Mojo has its own List and Dict types, that probably can satisfy your needs with much better performance. One big difference however is that List and Dict are statically typed in Mojo, that is: the types of their data must be known at comptime. List is like a workhorse in Mojo, used for many common tasks. It can grow dynamically and is packed with useful methods.

List, Dict and Set are not register-passable, they are *memory-only* types and their data is stored on the heap. The collection type is responsible for the de-allocation of data stored within it, so it happens automatically. Usually, the lifetime of the collection is also the maximum lifetime of the contained data.

Along the way, we encounter a number of more specialized types, which have their uses in certain circumstances, like Optional, Variant, VariadicList, DimList, IntTuple and InlineArray.

Finally, we discuss the properties of several Tensor types used in calculations and see how we can create and configure them.

We start off with the Tuple type, which is a bit of an outsider.

To follow along, use Magic to create a project called collections and cd into it. All code examples used in this chapter can be found in the code repo at [https://github.com/Ivo-Balbaert/Mojo\_in\_Action/tree/main/8\_Collections](https://github.com/Ivo-Balbaert/Mojo_in_Action/tree/main/7_Structs_and_Traits).

8.1 Using the Tuple type

A tuple is an ordered sequence of values, implemented in the built-in module *tuple*. More specifically, a tuple consists of zero or more, possibly heterogeneous values, separated by commas and enclosed in (), for example: (1, "ABC", True). The tuple items are indexed, starting from 0. The size of the tuple is fixed at declaration.

A Tuple is defined in the stdlib as a struct:

struct Tuple[\*element\_types: Copyable & Movable]

used for storing a small, fixed number of items of often different types on the stack.

The \* before element\_types shows us that a tuple can contain a variable number of arguments. Moreover, the items must be of a type that implements both Copyable and Movable traits (see § 7.3.3).

When and why would you use tuples? They can be useful to temporarily group a few data items that are related. In a way, you can view a tuple as a primitive struct instance, without field names for the items and without methods. But they have a few tricks upon their sleeve, like swapping values, returning more than one value from a function and unpacking values in assignments. This makes them very handy at times.

Specifically, a useful application is when you have a function that returns several items. Mojo doesn’t allow a function to return values separately, but it is allowed when they are grouped as a tuple!

Let’s first see how we declare tuples (see tuple.mojo). The Tuple type is an enumeration of the item types as parameters. So, the tuple defined in the first line of the code snippet has as type Tuple[Int, StringLiteral, Float64], while the type of tuple (1, 2, 3) is Tuple[Int, Int, Int].

@value

struct Coord:

    var x: Int

    var y: Int

var z: Int

fn main():

var tp: **Tuple[Int, StringLiteral, Float64**] = (42, "Mojo", 3.14) #A

    var tp\_alt: **(Int, StringLiteral, Float64)** = (42, "Mojo", 3.14) #B

var tp2 = (42, "Mojo", 3.14) #C

var tp3 = Tuple(1, 2, 3) #D

var tp4 = (1, 2, 3) #E

var tp5 = (Coord(5, 10, 15), 5.5, False) #F

#A Complete declaration with the Tuple type and initialization

#B An alternative simpler Tuple type notation

#C Leaving out the type to be inferred by the compiler

#D Shortened declaration, the type is inferred because of the ()-notation

#E A homogeneous Tuple

#F A heterogeneous tuple with an instance of struct Coord as first item

As you can see, explicit declaration of the Tuple type is not necessary, Mojo derives the type from the values in the ( , , , ) notation and checks them on the way.

How can you get at a specific item of a tuple? Use the index [ix]- operator (which implements the \_\_getitem\_\_ method):

    print(**tp[0]**)  # => 42

We can change a tuple item by index:

**tp[0] = 10**

    print(tp[0])  # => 10

A tuple has some useful methods, like len to get the number of items and in to check if it contains a certain item:

print(**len**(tp))  # => 3

if "Mojo" **in** tp:

        print("tp contains Mojo")

They implement their corresponding dunder-methods \_\_len\_\_ and \_\_contains\_\_.

The empty tuple, should you encounter it, is written as:

var emp = **()**

print(len(emp)) # => 0

There is no method to add an item to a tuple, because of their fixed size.

Here are the neat tricks with tuples:

1. You can assign the items of a tuple to variables, like this (discarding items you don’t need with \_):

    a, b, c = tp

    print(a, b, c)  # => 42 Mojo 3.14

 a, \_, c = tp

    print(a, c)  # => 42 3.14

This is called *unpacking* the tuple into variables a, b and c. (Why is a type declaration of a, b and c unnecessary?)

1. Here is a handy swap-method using tuples:

var u: Int = 2

    var v: Int = 3

    print(u, v)  # => 2 3

    (u, v) = (v, u)

    print(u, v)  # => 3 2

1. Here is how we can use a tuple for a function that returns more than one value:

fn full\_name() -> **(String, String)**: #A

    first\_name = String("Alice")

    last\_name = String("Doe")

    return **(first\_name, last\_name) #A**

fn main():

s1, s2 = full\_name() #B

    print(s1, s2) # => Alice Doe

    print(full\_name()[1])  # => Doe

#A A function returning a tuple of two String values

#B Unpacking the return values to separate variables

The return type of the full\_name function can be written in a typed way as:

Tuple[String, String]

Experiment 8\_1:

Does the following code compile? Explain the message

var t1: Tuple[Int, Bool]

t1 = (2.71, False)

Using the code from the example, extract the Coord struct instance from tp5.

Can you do:  for i in tp5: ? Explain the message you get.

A specialized version of tuple is the StaticTuple. As a tuple it has a size known at comptime, but this is specified as a parameter which allows for stack allocation. It is also register-passable, thus delivering higher performance. All items are of the same type, conforming to AnyTrivialRegType (see § 4.3.2), which also means the values can be handled in SIMD registers. It is defined as:

@register\_passable(trivial)

struct StaticTuple[element\_type: AnyTrivialRegType, size: Int]

Together with the fixed size, this points to the main advantage of using StaticTuple: a better performance for simpler types.

from utils import StaticTuple

alias st = StaticTuple[Int, 5](2, 7, 12, 42, 108)

print(st[2])   # => 12

print(len(st))  # => 5

StaticTuple is imported from the utils module.

As a transition to the List type, we explore a number of more primitive List-like types which you can encounter in Mojo code.

8.2 Working with primitive List and vector types

These are more primitive List types and are defined in the builtin module *list\_literal*, so they don’t need to be imported. The ListLiteral type is kind of a hybrid between Tuple and List.

8.2.1 The ListLiteral type

This is literally a list of values, possibly of different type (that’s why it’s like a tuple), separated by commas and enclosed in []. ListLiteral is an immutable type: it only includes a get method for accessing its items, that must implement the Copyable and Movable traits. Nothing can be modified after initialization, as is the case for all literal types.

It is defined as a struct like this:

struct ListLiteral[\*Ts: Copyable & Movable]

where Ts is the type of its items.

It has one field storage of type Tuple[Ts], which is the underlying storage for the list.

Here is some code using this type (see listliteral.mojo):

fn main():

    var list = [1, 2, 3]                   #A

    var explicit\_list: ListLiteral[Int, Int, Int] = [1, 2, 3]   #B

    var list2 = [1, 5.0, "Mojo🔥"] #A

    var mixed\_list: ListLiteral[Int, Float64, StringLiteral] #B

            = [1, 5.0, "Mojo🔥"]

    print(mixed\_list.get[0, Int]()) # => 1

    print(mixed\_list.get[2, StringLiteral]())  # => Mojo🔥

    print(len(mixed\_list)) # => 3

    if 1 in mixed\_list:  print("I contain 1!")

    # => I contain 1!

#A A type-inferred ListLiteral

#B Explicitly type-specified ListLiteral

When you initialize the list, the types can be inferred or explicitly specified. However, when retrieving an item with get you need to provide the item's index as well as the type as parameters.

ListLiteral is useful for creating temporary collections of mixed types that you need to pass to functions or use in expressions. It is also used for Python interoperability: you can assign it to a PythonObject and then expand it, as shown here:

from python import PythonObject

    var py\_list: PythonObject = ["cat", 2, 3.14159, 4]  #A ListLiteral

    py\_list.append(5) #B

    print(py\_list) # => ['cat', 2, 3.14159, 4, 5]

#A Assign a ListLiteral to a PythonObject

#B Append to the Python list

When passed to Python code, it is automatically converted to a Python list. ListLiteral is also heterogeneous, unlike Mojo's native List type which requires all elements to be of the same type.

8.2.2 Variadic types

The *list\_literal* module also contains the utility types VariadicList, VariadicListMem and VariadicPack. These types are used in situations where a variable number of values and/or types can occur.

We already used the first two types in § 6.2.3, they are list types specifically made to store a variable number of function arguments or function memory arguments respectively. They can show the size of the list, which can be iterated over in a for-loop.

VariadicList items must be @register\_passable. VariadicListMem items are references to values stored in memory. Both these types accept only one type of arguments.

The VariadicPack type accepts any number of arguments of different types. It has the methods each and each\_idx to iterate through its arguments. Here is an example showing its use (see variadic\_pack.mojo):

fn count\_many\_things[\*ArgTypes: Intable](\*args: \*ArgTypes) -> Int: #A

    var total = 0

    @parameter

    fn add[Type: Intable](value: Type): #B

        total += Int(value)

    args.each[add]() #C

    return total

fn main():

    print(count\_many\_things(5, 11.7, 12))  # => 28

#A args is of type VariadicPack

#B The add function is a closure over total.

#C Use the each method to iterate over the variable number of arguments

In this code snippet, the VariadicPack args accepts only Intable arguments. These are converted to Int in the add function and then summed up in a variable total, which is captured in the closure add through @parameter.

Now we discuss the types from module collections, starting with the most used type which is List. Things they have in common are:

* They are *generic* and defined with statically known type(s).
* An empty List, Set or Dict is False when converted to Bool, and True when it is not empty.

8.3 Working with List

List is the most useful collection type; that’s why it is automatically imported by the Mojo compiler. It aims to get the same functionality as the Python list type.

A List is like a *dynamically sized* ordered array of homogeneous (same type) items: it allocates memory on the heap to store its items and can grow in size when needed.

List can be made with primitive types, Strings, struct instances, anything that conforms to the Copyable and Movable trait (see § 7.3.1).

If type T implements these traits, then we can make a var lst = List[T](), which is an empty List. A List is defined as a struct that has two fields:

1. capacity: which is an integer: the number of items the List can contain without resizing is given by lst.capacity
2. data: this is a pointer to the start of the List’s items, more precisely an UnsafePointer[T], lst.data gives the address of this pointer, for example:   print(lst.data)# => 0x1161bfe02000

8.3.1 Creating and resizing Lists

Unlike what you perhaps would expect, ListLiteral is not compatible with the List type:

var mojo\_list: List[Int] = [1, 2, 3, 4]

# => error: cannot implicitly convert 'ListLiteral[Int, Int, Int, Int]' value to 'List[Int]'

Let’s review the ways to construct a List. Declare the item type as a *parameter* to create an empty List, like this for Int items (see list1.mojo): var lst = List[**Int**]()

(As always, the () at the end are a call to the constructor of the type.)

This is a List with one item with value 8: var lst = List[Int](8)

Once one or more values are present, the type can be inferred, so this is equivalent to:

var lst = List(8)

Here is a List with four elements: var lst = List(7, 13, 42, 108)

We are using variadic arguments here to construct this List. This is made possible by the following List constructor, which uses a VariadicListMem type for the list items:

\_\_init\_\_(out self, \*, owned elements: **VariadicListMem**[T, origin])

(The parameter origin here has to do with the lifetime of elements. We’ll explain what it means in chapter 9.)

Lists are *indexed* with integers starting from 0:

var lst = List(7, 13, 42, 108)

**indices: 0 1 2 3**

    print(lst[0])  # => 7

    print(lst[2])  # => 42

    print(lst[5])  # => 0   #A

print(lst.data)  # => 0x1161bfe02000 #B

    lst3[2] = 43 #C

#A No boundaries check is done!

#B The memory address where the List data starts

#C Change item 2 by index from 42 to 43

The usual lst[ix] notation is used to access the item at index position ix. An item can be given a new value with the lst[ix]= notation. Accessing an item on an index beyond the current List doesn’t result in an error, the result is 0. This is an example of undefined behavior (UB), which probably will change to adequate error-messaging in the future.

The number of items in a List is given by its length len:

  print("Length list:", len(lst))      # => Length list: 4

Using the index in a for-loop gives us a first way to print out a List:

  for ix in range(len(lst)):

      print(lst[ix], sep=" ", end="")  # => 7 13 43 108

Besides the length, every list has a capacity, which is the number of items that can fit in the list without dynamically resizing it, for example:  print(lst.capacity) # => 4

A simple method to grow the list with one item at a time is append, the new item is appended at the back of the list:

lst.append(10)

    print(len(lst))   # => 5

    print(**lst.capacity**)  # => **8**

The length of lst is now 5, but its capacity has grown to 8! What has happened? Appending one item automatically resized the list, so it is now ready to store 8 items, with still three slots unfilled.

Resizing is a costly operation. It makes sense to provide enough storage space from the start. Say you know your list will have to contain maximum 100 items, you can make a List of that size:

**var lst = List[Int](capacity=100)**

    print(len(lst))  # => 0

    print(lst.capacity)  # => 100

This saves you some 5 resize operations. If during execution you need to add another 100 items, you can make a reserve to provide the new storage space:

lst.**reserve**(200)

    print(lst.capacity)  # => 200

If this is greater than the current capacity, new storage for 200 items will be allocated, and the items stored will be moved to the new location. If it is smaller, nothing is done.

A similar operation called resize exists, but it behaves very differently. It takes an argument new\_size for the list. When new\_size is greater than the current length, you must also provide a value to fill in the new memory slots. Suppose our List lst contains 5 items:

**lst.resize(10, 0) #A**

    print(len(lst))  # => 10

    print(lst.capacity)  # => 10

    for ix in range(len(lst)):

        print(lst[ix], end=" ")  # => 7 13 43 108 10 0 0 0 0 0

    print()

#A Resize lst to size 10, and pad 5 slots with 0 values

After resizing we see that len and capacity have grown to the new size, and the new memory is filled up with the value 0 we specified.

But if it has only one argument for the new\_size of the List, this must be smaller than the current size. The capacity stays the same, but this effectively shrinks the list to the new smaller size and the superfluous items at the end are discarded:

**lst.resize(4)**

    print(len(lst))   # => 4

    print(lst.capacity)  # => 10

    for ix in range(len(lst)):

        print(lst[ix], end=" ")

# => 7 13 43 108

You can also add an entire List to an existing list with the extend method:

  var lst5 = List(256, 365, 421, 508)

    lst3**.extend(lst5)**

    for ix in range(len(lst3)):

        print(lst3[ix], end=" ")

    # => 7 13 43 108 256 365 421 508

This is kind of an append of a List instead of a value.

8.3.2 Useful methods for Lists

The List functionality contains a lot of methods, and new ones are regularly added (see the complete list in the docs <https://docs.modular.com/mojo/stdlib/collections/list/List/>.

Pop and insert

Just as the append method adds an item to the back of a List, the pop method gets that item (so you can assign it to another variable) and removes it from the list (see list2.mojo):

 var lst = List(7, 13, 42, 108)

print(**lst.pop()**)  # => 108

    print(len(lst))   # => 3

    for i in range(len(lst)):

    print(lst[i], end=" ")

# => 7 13 42

If you give an index as argument, you can use pop to get and remove the lists value at that index:

print(**lst.pop(1)**)   # => 13

    print(len(lst))   # => 2

    for i in range(len(lst)):

        print(lst[i], end=" ")

# => 7 42

Let’s add the removed item back with the insert method, which uses an index and a value:

**lst.insert(1, 13)**

 for i in range(len(lst)):

     print(lst[i], end=" ")

# => 7 13 42

Printing and for-loop

Printing out a List with one statement is not yet possible. The closest you can get is:

    print(lst.\_\_str\_\_())   # => [7, 13, 42]

    print(lst.\_\_repr\_\_())   # => [7, 13, 42]

This works, but only if the List items type implements the \_\_str\_\_ and \_\_repr\_\_ method.

What about a for-loop? The docs tell us that List implements the \_\_iter\_\_ method. However, this simple form does not work:

 for li in lst:

        print(li) # => error!

It gives the compiler error:

invalid call to 'print': could not deduce parameter 'Ts' of callee 'print'

Recalling the explanation in § 4.4 of how the for loop is implemented: the li

iterator variable is a pointer in Mojo. To get the value we must dereference the pointer using []. So, this works:

  for li in lst:

        print(**li[]**, end=" ")  # => 7 13 42

Contains

List also implements a \_\_contains\_\_ method, so you can test if a List contains a certain item with the in operator:

 if 13 in lst: print("lst contains 13")

# => lst contains 13

Equality

Two Lists are equal if their items are of the same type and are the same:

  var l1 = List[Int](7, 13, 42)

    var l2 = List[Int](7, 13, 42)

    var l3 = List[Int](13, 7, 42)

    if l1 == l2: print("l1 and l2 are equal")

    # => l1 and l2 are equal

    if l1 != l3: print("l1 and l3 are not equal")

    # l1 and l3 are not equal

Slicing

As we did for Strings in § 4.4.3, we can also take a slice from a List:

var lst = List[Int](7, 13, 42, 108, 177, 255, 312)

    var lspl = lst[2:6]

    for i in range(len(lspl)):

        print(lspl[i], end=" ")

    # => 42 108 177 255

Here lspl is a slice of List lst, which is a *copy* of part of the List.

Shuffle

To randomly shuffle the items of a List, changing the List in-place, use the shuffle function from module random:

from random import shuffle

shuffle(lst)

    for li in lst:

        print(li[], end=" ")

    # => 42 255 177 13 108 312 7

Experiment 8\_2

Show for yourself that when a List is copied to another variable, you now have two independent lists. In other words: copying a List is a deep copy.

Try out the List methods: reverse and swap\_elements. Show that the byte length of the List is equal to the length of the List \* sizeof[T](), where T is its item type.

How is it possible that a String can grow dynamically? We can now explain this.

8.3.3 Constructing a String from a List

The String object has an internal buffer field \_buffer. This is a List[Byte] field for efficiency reasons, which holds the bytes that make up the string. A List stores its contents in dynamically allocated memory on the heap, so a String can grow or shrink. The String itself doesn't have any special destructor logic, but when Mojo destroys a String, it calls the destructor for the List of Byte field, which de-allocates the memory.

We can build a String by constructing such a List (called buf in the example string\_list.mojo):

Listing 8.1 Building a String from a List

fn main():

    var buf = List[Byte](2) #A

    buf.append(65)

    buf.append(79)

    buf[0] = 78

    buf.append(32)

    buf.append(ord("H"))

    buf.append(ord("i"))

    buf.**append(0)** #B

    for i in range(len(buf)):

        print(buf[i], end=" ")

    print()

    # => 78 65 79 32 72 105 0

    var str = **String(buffer=buf)** #C

    print(str)  # => NAO Hi

#A Start building a List of Byte items (same as UInt8)

#B Terminate it with a 0 byte

#C Construct a String, taking as buffer the List buf

The List buf must be terminated with a null (0) byte. We can then make a String out of this List by using the following String constructor:

\_\_init\_\_(out self, \*, owned buffer: List[SIMD[uint8, 1], hint\_trivial\_type])

From this we can see that buffer is a keyword argument (it comes after \*), that’s why we needed to use String(buffer = buf).

A String can grow dynamically because it contains a field buffer for its data, which is a resizable List.

8.3.4 Sorting a List

Sorting is commonplace in data manipulations. Mojo has a built-in sort method that comes in a few versions and that sorts a List in-place, reducing the amount of memory operations that are needed. Here is a simple example (sorting1.mojo):

fn main():

var v = List[Int](108)

v.append(20)

v.append(10)

v.append(70)

sort(v) #A

for li in v:

print(li[], end=" ") # => 10 20 70 108

#A The list v is sorted in place

Let’s see how this function works. Looking at the header of the general sort function (omitting some arguments that are not relevant here),

sort[…, **type: Copyable & Movable**, … //, **cmp\_fn: fn(type, type) capturing -> Bool**, \*, stable: Bool = False](**span: Span[type, …])**

we spot the following:

* The type of the elements to be sorted is Copyable & Movable, but in other versions it can also be a SIMD[type, 1], SIMD[DType, 1] or an Int.
* The function with which to sort is: cmp\_fn: fn(type, type) capturing -> Bool. This means you can insert your own sorting function when calling sort, as long as it has this signature. The sort method is a higher-order function (see § 6.4.3).
* The object that is sorted is of type Span[type](see § 9.4.3).

8.3.5 Implementing \_\_contains\_\_ in a list-type struct field

Here is an example of a struct implementing the in operator, which is backed up by the dunder method \_\_contains\_\_, on a List field inside a struct (see in\_list.mojo):

Listing 8.2 Implementing the in operator

struct LiStruct:

    var ints: List[Int] #A

    fn \_\_init\_\_(out self, ints: List[Int]): #B

        self.ints = ints

**fn \_\_contains\_\_**(self, value: Int) -> Bool: #C

        for i in self.ints:

            if i[] == value:

                return True

        return False

fn main():

  var struct1 = LiStruct(List(1, 2, 3)) #D

    print(1 in struct1)  # => True #E

    print(5 in struct1)  # => False #F

print(struct1.\_\_contains\_\_(5)) # => False #F

#A The List contains a field which is a List[Int]

#B The constructor \_\_init\_\_ takes a List[Int] as argument

#C The \_\_contains\_\_ method is implemented

#D Creating a struct instance struct1

#E This works here because we implemented \_\_contains\_\_

#F These are two equivalent tests

8.4 Working with the Set type

The Set type represents *an unordered, dynamically growable* set of *unique* *homogeneous* values and must be imported from *collections*. You can add and remove elements from the set, test whether a value exists in the set, and much more.

Sets are generic, declared as Set[Type], and the item Type must conform to the KeyElement trait, which entails the CopyAble, Movable, Hashable, and EqualityComparable traits. Here is some code that gets you started with Sets (see set.mojo):

Listing 8.3 Constructing Sets and Set operations

from collections import Set #A

fn main():

    var set1 = **Set[Int](**42, 0) #B

    set1.**add**(13) #C

    print(**len**(set1))  # => 3

    set1.add(0) #C

    print(len(set1))  # => 3 #D

**for item in set1:**

        print(**item[]**, end=" ")  # => 42 0 13 #E

    print()

    var fruits = Set("orange", "pineapple", "banana", "apple") #F

    var i\_eat = Set("pizza", "banana", "salad", "apple")

    var i\_eat\_fruit = **fruits.intersection(i\_eat)** #G

    # var i\_eat\_fruit = fruits **&** i\_eat #G

    print("I eat fruit:")

    for item in i\_eat\_fruit:

        print(item[], end=" - ")

    # => I eat fruit:

    # => banana - apple -

#A Importing Set from collections

#B Declaring a Set of Int and initializing it with 2 items 0 and 42

#C Adding an element

#D Only unique values are accepted

#E Use [ ] to get the value

#F Type of the Set is inferred as String

#G These two lines work the same way

Note that we can only add items that don’t yet exist in the set. Using a for loop to iterate over a Set works, but again we must use the [] operator to get the value.

Sets allow a huge number of operations like ==, issubset, issuperset, clear all items, discard an item, and so on. You can also perform set algebra operations, like union (|) , intersection(&) and difference (-) between two sets. You either use them like fruits.intersection(i\_eat) or with & as a binary operator in fruits & i\_eat (where fruits and i\_eat are sets).

8.5 Working with the Dict type

A dictionary (type Dict) is a collection of key-value pairs, where you can use the key to look up the corresponding value. For example, a dictionary could hold country names as key, and their capitals as value. This Dict would be defined as (see dict.mojo):

from collections import Dict

var country\_capital = Dict[String, String]()

This is still an *empty* dictionary. To start filling it up with key-value pairs, you can write:

country\_capital["United Kingdom"] = "London"

country\_capital["India"] = "New Delhi"

country\_capital["China"] = "Beijing"

The keys have the same type, and the values also; but the types of keys and values can be different, like here: var hashes = Dict[String, Int]()

A key type must implement the KeyElement trait (which implies CopyAble, Movable, Hashable, and EqualityComparable: a key is first hashed to a number, and when searching for a key, their respective hashes are compared). Value types must conform to CopyAble and Movable.

To get the value associated with a certain key, use dict[key] like this:

print(country\_capital["India"])  # => New Delhi

You can view a key as a generalized index.

If you ask for a key that doesn’t exist, you get a KeyError:

    print(country\_capital["Belgium"])

    # => Unhandled exception caught during execution: KeyError

We can avoid such an error by checking whether a Dict contains a certain key with the in operator:

if "Belgium" **in** country\_capital:

        print("Belgium has as capital ", country\_capital["Belgium"])

    else:

        print("We don't know the capital of Belgium")

# => We don't know the capital of Belgium

To change the value associated with a key, use an assignment like when first creating the key-value pair:

country\_capital["India"] = "Mumbai"

If the key already exists, its value is changed in the pair. If the key does not exist, the (key, value) pair is added to the dictionary.

Like a List, Dict is also a dynamically growing collection, reallocating its items in memory when needed. If you expect to have many key-value pairs in the Dict, you can initialize it with a reserved capacity from the start, for example:

var x = Dict[Int, Int](**power\_of\_two\_initial\_capacity = 256**)

This allows us to insert (2/3 of 256 = 170) entries without reallocation.

Although still a bit awkward, you can print out a Dict as follows:

dict\_as\_string = country\_capital.\_\_str\_\_()  
 print(dict\_as\_string)

# => {'United Kingdom': 'London', 'India': 'New Delhi', 'China': 'Beijing'}

The simplest way to iterate over a dictionary in a for-loop works with the items() method:

for item in country\_capital**.items()**:

    print(**item[].key** + ", " + **item[].value**, end=" - ")

print(**item[].hash**)

# =>

United Kingdom, London - 16399761064420479588

India, New Delhi - 14418127593282858532

China, Beijing - 5887465646640935972

The item iterator variable here is an *immutable reference*, that must be dereferenced as item[] to get its value. Then use the key and value property to get their values. We also showed the hash property, just to give an idea (you won’t have to use it in code yourself).

Another way is to work with the Dict name itself in the for loop, and using an iterator variable like country, the key is then country[] and the capital is given by country\_capital[country[]].

  print("capital, country:")

    for country in country\_capital:

        print(country\_capital[country[]] + ", " + country[], end=" - ")

    # => capital,  country:

    # London, United Kingdom - New Delhi, India - Beijing, China -

Here is a Dict with different types for key and value:

var d1 = **Dict[String, Float64]**()

    d1["pi"] = 3.1415

    d1["e"] = 2.71

    print(**len**(d1))   # => 2

    print(d1["pi"])   # => 3.1415

    print(d1.**pop**("e"))  # => 2.71

    print(len(d1))   # => 1

The len method returns the number of key-value pairs in the dictionary. The pop(key1) method removes the key-value pair (key1,value1) from the Dict and returns value1. How can you check whether a key exists in a dictionary? Use the in operator:

 if "pi" **in** d1:

        print("We have pi in our Dict!")

    # => We have pi in our Dict!

if **not** ("e" **in** d1):

        print("We have no e in our Dict!")

    # => We have no e in our Dict!

Apart from the in operator, there is also a find method. Type d1.find("pi") and you will see the warning message:

Optional[SIMD[float64, 1]]' value is unused

This means we have to assign the returned value to a variable:

    var opt1 = d1.find("pi")

    var empty\_opt = d1.find("e")

When the key is present, find returns an Optional with the value in it, otherwise it returns an empty Optional. This is clearly a new type, let’s explore it now.

* 1. Working with the Optional type

In many circumstances (like we saw in the find method above) a method may return a value, or nothing, in other words: a value may or may not be present. In the last case, Mojo says it has value None. This None value replaces a runtime error, as is the case when we try to get the Dict value for a key which does not exist. In many languages in such cases a *null* value is returned, which often results in runtime errors.

Mojo has a special *safe type* called Optional to be used in such cases. This can have two kinds of contents: a real value, or None. This needs to be checked, and the value must be explicitly extracted. Listing 8.4 illustrates how to work with this type (see optional.mojo):

Listing 8.4 Working with the Optional type

from collections import Optional

fn main():

    var opt1 = Optional(5)   #A

    var opt2: Optional[Int] = 5 #A

  var opt3 = Optional[Int]()   #B

    var opt4: Optional[Int] = None #B

print(opt4.value()) #C

# error: .value() on empty Optional

    var opt: Optional[String] = String("Testing")

**if opt**:   #D

        print(**opt.value**())  # => Testing

    if opt4: #E

        print(opt4.value())

    else:

        print("opt4 has value None")

# => opt4 has value None

    var custom\_greeting: Optional[String] = None

    print(custom\_greeting**.or\_else("Hello")**)  # => Hello    #F

    custom\_greeting = String("Hi")

    var d = custom\_greeting.or\_else("Hello")  #G

    print(d)  # => Hi

#A Two ways to initialize an Optional with a value

#B Two ways to initalize an Optional with no value

#C .value() on None gives an error

#D Test if opt contains a value and if so get it with opt.value()

#E opt4 is None , so if branch is not executed

#F or\_else provides an alternative to take when the Optional is None

#G If the Optional has a value, or\_else has no effect

Checking can be done with if, or a dedicated or\_else method. The value is extracted with a .value() method. Optional effectively provides  a *type-safe nullable pattern*: if a type can be None (null), handle it with if and or\_else methods to avoid problems.

An Optional opt can be tested with if, when this returns:

* True, then get the value with opt.value()
* False, then opt is None, opt.value() doesn’t do anything

In fact, opt.value() has undefined behavior (UB) in the None case, that’s the reason why opt.value() must always be tested inside an if statement or another conditional.

Another way is the or\_else method, which provides an alternative value that is returned if the optional contains None. In case the optional contains a value, that value is returned.

Like the other collection types, Optional is generic, and can hold any type T that conforms to the Copyable and Moveable traits: Optional[T].

Experiment 8\_3 Working with find in a dictionary

Use a dictionary from § 8.4 and experiment how to work with the find method.

The Optional type is about a value, or no value. But what if you need a type that can hold a value of type Int, *or* a value of type Float64, *or* other types, but *only one type at a given time*? Mojo has you covered with the Variant type.

* 1. Working with the Variant type

The Variant type a is run-time (dynamic) type. It is capable of holding values of different types, but only one value at any given time. It is imported with: from utils import Variant.

Its struct declaration: struct Variant[\*Ts: Copyable & Movable]

shows that it can hold a variable number of types, that all conform to the Copyable and Movable traits.

For example, a Variant[Int, String] can hold a value that is either an Int or a String.

Its storage in memory requires the largest size of any of its variants, plus a 16-bit tag which contains the current Type. The Variant type (or something similar) exists in many other languages under diverse names, such as *tagged union*, *sum* type, *choice* type, *enum* (in Rust), and so on.

Its main advantage is that you can write code that depends on the type of the value contained in the Variant.

Here are some examples of declarations (see variant.mojo):

**alias** IntOrString = **Variant[Int, String]** #A

var v1 = Variant[Int, Float64](1) #B

    var v2 = Variant[Int, Float64]("Mojo") #C

# => comptime error

var v3 = IntOrString(String("Mojo")) #D

 print(v1) #E

    # => invalid call to 'print'

#A Define a shortcut alias for a Variant type

#B A Variant of types Int or Float64 is initialized

#C This gives an error: constraint failed

#D Declaring a Variant with an alias

The Variant that defines v1 must contain a value of type Int or Float64, and it is initialized with value 1 which is ok. Variable v2 of the same type is initialized with a String, which isn’t in the list of Variant types, so the compiler rejects this:

error: constraint failed: not a union element type

But we have defined an alias IntOrString as a shortcut, which can contain a String, so using this works.

Printing out a Variant value gives an error:

invalid call to 'print': could not deduce parameter 'Ts' of callee 'print'

Why is this? Because print doesn’t know the type of the Variant. But this works:

 print(**v1[Int]**)  # => 1

We must find out the type first, which is done with the isa method:

if v3.**isa[String]**():

        print(v3[String])  # => Mojo

The set method changes a Variant value, even switching to another type of the Variant:

v1**.set[Int]**(12)

    print(v1[Int])  # => 12

    v1**.set[Float64]**(3.14)

    print(v1[Float64])  # => 3.14

But this doesn’t work:

v1.set[String]("Mojo")

    # => error: constraint failed: not a union element type

So, how can we ask if a type is supported in the Variant? This works as follows:

if v1**.is\_type\_supported[Float64]**():

        v1 = Float64(2.71)

    print(v1[Float64])  # => 2.71

Testing type correctness at run-time is vital for the Variant methods. If not, the program could crash or show undefined behavior, returning unsafe and garbage data! (That’s why you need fn main() raises for this code.)

Finally, combining List with Variant shows how to make lists with different type values:

fn print\_variant(value: Variant[Int, Float64]):

    if value.isa[Int]():

        print(value[Int], end=" / ")

    else:

        print(value[Float64], end=" / ")

var a = **List[Variant[Int, Float64**]](1, 2.5, 3, 4.5, 5)

    var b = **List[Variant[Int, StringLiteral]]**(1, "Hi", 3, "Hello", 5)

    for i in range(len(a)):

        print\_variant(a[i])

    # => 1 / 2.5 / 3 / 4.5 / 5 /

Experiment 8\_4 Variant

Write a to\_string function for the type Variant[Int, String] and test it out.

Now we leave the traditional collection types and start discussing collection types that are used in calculations, like Tensor. Let’s first look at some supporting types.

* 1. Supporting types in calculations

Mojo has special types to work with dimensions. Think about the (x, y, z) 3-dimensional axis system to get an idea of what a dimension is. A List has one dimension, like a Vector in other languages. A matrix has two dimensions: rows and columns.

8.8.1 Index and IndexList

An IndexList is a struct that represents N-dimensional indices. It's commonly used for working with multi-dimensional data structures and coordinates.

Import it as: from utils import IndexList

IndexList is frequently used in tensor operations, matrix manipulations and layout transformations where you need to represent positions in N-dimensional space and other contexts where you need to represent multi-dimensional coordinates or shapes. Here is an example (see indexlist.mojo):

from utils import Index, IndexList

fn main():

    var index1d = Index(5) #A

    var index2d = Index(2, 4) #B

    var index3d = Index(1, 2, 3) #C

#A Create a 1D index

#B Create a 2D index

#C Create a 3D index

You create an IndexList using the Index function with different dimensions, like Index(5), or Index(2, 4) or Index(1, 2, 3). index1d, index2d and index3d are all of type IndexList. For example: index2d has two dimensions, in the first dimension it has 2 items, and in the second dimension 4 items.

8.8.2 Dim and DimList

A dimension can be represented as a value of type Dim. This struct can hold up to three Intable values, in essence *integer* values and thus register-passable. When a value is present at comptime, it is a static value, otherwise it gets bound at run-time and is dynamic (see dim.mojo):

from buffer import Dim, DimList #A

fn main():

    var dim1 = Dim(5) #B

    print(dim1.has\_value())  # => True

    print(dim1.get())   # => 5

    var dim\_list = DimList(2, 4) #C

    var index\_list = dim\_list**.into\_index\_list[rank=2]()** #D

print(index\_list)   # => (2, 4)

#A Import Dim from module buffer

#B Create dim1 with a single indexable (from 0 to 4) value for x; y and z are set to 1

#C Create a DimList with two dimensions

#D Convert a DimList to an IndexList

The integer that a Dim can contain is the size or extent (the number of elements) in that specific dimension. In our example, dim1 would contain 5 elements in the x-dimension, and 1

Dim is commonly used to represent grid and block dimensions in GPU computing (see chapter ??).

The DimList type from module *buffer* represents a *list of dimensions*, each of type Dim, which is mostly used in computations with tensors, which are like matrices but with more and higher dimensions. It closely resembles the IndexList, to which it can be converted. DimList has a value field which is of type VariadicList[Dim].

8.8.3 IntTuple

IntTuple is a specialized tuple type which can contain nested tuples, but the only value type allowed are integers. Its implementation is highly memory-efficient (through memory sharing and zero-copy views) optimized for tensor shapes, stride- and index operations in high-performance computing. You create them like this:

**from layout import IntTuple**  #A

**from layout.int\_tuple import compact\_order, flatten, size**

fn main():

    var shape = **IntTuple(2, IntTuple(3, 4), 5)** #B

    print(shape)  # => (2, (3, 4), 5)

    var flat = **flatten**(shape)   #C

    print(flat)  # => (2, 3, 4, 5)

    var total\_size = **size**(shape) #D

    print(total\_size)  # => 120

    var inttup = IntTuple(1, IntTuple(2, 3), 4)

    shape**.append**(inttup) #E

    print(shape)  # => (2, (3, 4), 5, (1, (2, 3), 4))

print(len(shape) # => 4

#A IntTuple and its functionality are imported from module layout

#B Created (nested) IntTuples

#C Flatten a nested tuple

#D Calculate total size (the product of all elements)

#E Append another IntTuple

The layout.int\_tuple module contains a huge number of functions to do calculations with IntTuples. In our example we use

* flatten, which removes nested tuples so that only one IntTuple remains.
* size, which multiplies all elements.
* append, which appends another IntTuple value.
* len, which returns the number of items in the tuple (a nested IntTuple is one item).

8.8.4 InlineArray

The InlineArray from module collections is imported automatically. It is a sequence with a *fixed size*, and all items have the *same type* which must conform to Copyable and Moveable. The array size cannot be changed and is bounds-checked at comptime, which means you can never use an index out of range. Being stack-allocated, InlineArrays offer very good performance for small, fixed-size arrays due to their (stack) locality and direct access. You cannot use a for-loop on this kind of array, because it doesn’t implement the \_\_iter\_\_ method.

Here is how to create, access and change them(see inline\_array.mojo):

fn main():

    var arr1 = **InlineArray[Int, 4](8)** #A

    var arr2 = InlineArray[Int, 3](1, 2, 3) #B

    var filled = InlineArray[Int, 5](fill=42) #C

    # Access elements

    print(arr1[0])  # => 8

    print(arr1[1])  # =>

    arr1[1] = 13

    print(arr1[1])  # => 13

    print(arr1.unsafe\_get(2))  # => 8

    print(len(arr1))  # => 4

    # print(arr1[4])   #D

    # Assert Error: InlineArray index out of bounds:

    # index (4) valid range: -4 <= index < 4

#A Create an array of 4 identical integers with value 8

#B Create an array of 3 integers with the given values

#C Create an array filled with value 42

#D An out-of-bounds access crashes the program

To gain performance, you can use the unsafe\_get(index) to read elements, because this method skips bounds-checking. As the name indicates, this is an unsafe method, which can result in undefined behavior with an invalid index.

With these supporting types in our belt, let’s explore tensors in Mojo. But first, we’ll have to learn some more details what tensors are.

8.9 Some properties of tensors

A typical vector type like a SIMD vector (see § 4.3) has only one *dimension*. But matrices have 2 dimensions. *Tensors* are a generalized form of matrices that can have an arbitrary number of *dimensions*. They are the basic building blocks used to represent data in AI and are omnipresent in deep-learning frameworks like TensorFlow and PyTorch.

But first, we need to understand some concepts that come up again and again when working with tensors. We’ll see concrete code examples in the next section:

* The *rank* of a Tensor indicates *the number of dimensions* present in a tensor, for example: rank 0 are scalars (single values), rank 1 are vectors (arrays), rank 2 are matrices, and rank 3 and above are tensors. A rank is just an integer.
* The *shape* (or layout or structure) of a Tensor defines the *rank* and the *size of each dimension* in the tensor, in other words: how many elements there are along each axis (dimension) of the tensor. Mojo contains a specific type TensorShape for this, but a DimList can also store this info.
* The *size* of a Tensor is the product of all its dimension sizes, in other words: the *total number of elements* in a tensor, which is also of integer type.
* The shape of a Tensor, together with the type of its elements, is called the *specification (spec)*, or the TensorSpec.

For example, in Listing 8.5 we’ll define a tensor with *shape* (10, 10, 3), so its *rank* is 3 (3 dimensions). In the first and second dimension its size is 10, in the third the size is 3. The tensor has *size* is 10x10x3 = 300. If we choose a type Int32 for its items, then its spec is: 10x10x3xint32. The space it takes up in memory is 4 bytes x 300 = 200 bytes.

* The *strides* of a Tensor: These depend on the memory layout of the tensor: in other words, how its items are allocated in memory slots. By default, the items of a tensor are allocated one after another in contiguous memory locations in *row-major order* (or *layout*), so row1, row2, row3, and so on, see Figure 8.1 for a 2-dimensional tensor:

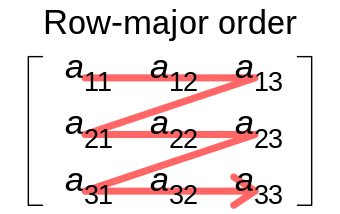


Figure 8.1 – The items of a tensor are stored in memory in row-major order: first all items of the first row, then the second row, and so on.

The elements of the tensor in Figure 8.1 are stored in contiguous memory slots in this order:

a11 - a12 - a13 - a21 - a22 - a23 - a31 - a32 - a33.

This makes clear that row-major storage layout is optimized for accessing rows as iterating row by row is faster, because we're accessing subsequent memory locations.

The *stride* is the number of elements to skip over in memory to move from one position along a particular dimension to jump to the next element along a specific dimension of the tensor. In a multi-dimensional tensor, each dimension has its own stride. Let’s make this concrete.

Consider the 3 x 3 tensor below:

|  |  |  |
| --- | --- | --- |
| 11 | 12 | 13 |
| 14 | 15 | 16 |
| 17 | 18 | 19 |

In memory its elements are stored in a one-dimensional block of memory like this (row-major):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

The rows are dimension 0. To jump from position 11 to the next element of a row, which is position 14, requires skipping 3 elements in memory storage, so the stride for the rows is 3:

stride[0] = 3

The columns are dimension 1. To jump from position 11 to the next element of a column, which is position 12, requires skipping only 1 element in memory storage, so the stride for the columns is 1:

stride[1] = 1

Tensors are stored in 1 dimension in computer memory, but they can be accessed as multi-dimensional tensors based on their shape and strides. Knowing the strides of a tensor is an important aspect to accessing the tensor elements efficiently in memory.

*Column-major* order can also be used where the elements are laid out in memory in the order column1 (a11, a21, a31…) , column2 (a12, a22, a32…) , and so on.

Row-major and column-major are two examples of possible *memory layout* strategies for tensors. With LayoutTensor discussed in §8.10.3, you can specify the memory layout of data.

Matrices are also tensors, and MAX, which is the main AI library using Mojo, contains a Tensor type which has full tensor functionality. For example, in Ai applications, a tensor can be allocated on a GPU, or created on the CPU, and then move from the CPU to the GPU.

Now we’ll start using tensors. Listing 8.5 demonstrates some of the concepts we just discussed, like shape, spec and rank.

8.10 Using Tensor types

Because tensors are so popular in AI, the definitions and functionality for tensors are contained in the MAX Mojo API, more specifically in modules *max.tensor* and *max.driver*.

8.10.1 The Tensor type from max.tensor

A tensor is a higher-dimensional matrix, the Tensor type is defined in module max.tensor.

We already used this tensor type in § 2.4.3, running our Stable Diffusion model with MAX in chapter 2.

The Tensor type manages its own data, that is:  it performs its own memory allocation and freeing. It owns its underlying data and takes a DType value as parameter. We used it in our Stable Diffusion Model in chapter 2. It is defined as a struct like this:

struct Tensor[type: DType]

The following Listing 8.5 shows different ways of creating tensors, using some of the many overloading constructors that exist (see tensor1.mojo):

Listing 8.5 Using a Tensor

from tensor import Tensor, TensorShape, TensorSpec   #A

fn main():

    var shape = **TensorShape(10, 10, 3)**   #B

    print(shape.**rank()**)  # => 3

    print(shape.**num\_elements()**)  # => 300

    print(shape)  # => 10x10x3

    print(shape[0])  # => 10

    var spec = **TensorSpec(DType.int32, shape)**   #C

    print(spec.**dtype()**)  # => int32

    print(spec.**bytecount()**)  # => 1200 #D

    print(spec)  # => 10x10x3xint32

    var spec2 = TensorSpec(DType.int32, (3, 3))

    var ts1 = **Tensor[DType.float32](shape)** #E

var ts2 = **Tensor[DType.float32](spec)** #F

var tdim1 = **Tensor[DType.int32](10)** #G

print(tdim1)  # => Tensor([[0, 0, 0, ..., 0, 0, 0]],

dtype=int32, shape=10)

    var tdim2= Tensor[DType.int32](spec2)   #H

    print(tdim2)  # =>

    # Tensor([[0, 0, 0],

    # [0, 0, 0],

    # [0, 0, 0]], dtype=int32, shape=3x3)

#A Tensor is imported from module max.tensor

#B A TensorShape initialized with a Tuple (10, 10, 3)

#C A TensorSpec needs a DType value and a TensorShape

#D The bytecount specifies the memory the Tensor occupies

#E Creating a Tensor from a DType parameter and a TensorShape argument

#F Creating a Tensor from a DType parameter and a TensorSpec argument

#G tdim1 is a 1-dimensional tensor (a vector) with 10 elements

#H tdim2 is a 2-dimensional tensor (a matrix)

A TensorShape is nothing more than a series of integers. It can be created from a Tuple, a List[Int], a VariadicList[Int], or a variable number of integers, for example: TensorShape(10, 10, 3)

Each integer represents a dimension. The number of integers is the rank of the tensor that is going to be created, so our shape has rank 3; the tensor created will be 3-dimensional. The product of all the integers is the size (num\_elements) of the tensor. When printed out, a TensorShape nicely shows the dimensional structure, like in this case: 10x10x3.

A TensorSpec combines a tensor shape with the dtype of the tensor’s items, for example: TensorSpec(DType.int32, shape)

It has the same properties as TensorShape, and a dtype() method (which gives int32 here) and bytecount() (which is 1200 = 10 x 10 x 3 x 4 bytes per integer) method.

A TensorSpec prints out to this format: 10x10x3xint32.

There are more than 10 overloading constructors for creating a Tensor. In most cases a TensorShape or TensorSpec is needed. The data can be provided through a variable number or a List of SIMD values or can be referenced by a pointer (see chpter 9). A new tensor is initialized with zeros (0).

The following Listing shows different methods for reading and writing values to a Tensor (see tensor2.mojo):

Listing 8.6 – Reading and writing values

from tensor import Tensor

from pathlib import Path

from utils.index import Index

fn main():

var t = Tensor[DType.int32](2, 2) #A

print(t.**load()**)  # => 0 #B

print(t.**load[width=4]()**) # => [0, 0, 0, 0] #C

t.**store[4](0, 1)** #D

t.store[4](2, 3) #E

print(t)

# =>

# Tensor([[1, 0],[3, 0]], dtype=int32, shape=2x2)

  print(t.load(2))  # => 3 #F

 print(t[0, 0])  # => 1 #G

    print(t[1, 0])  # => 1

    t[Index(0, 0)] = 3 #H

 var t2 = Tensor[DType.float32](3, 10, 10)

    t2**[Index(2, 9, 9)]** = 1

    print(t2[Index(2, 9, 9)])  # => 1.0 #G

t**.tofile(Path("tensor1")**) #I

    print(**t.fromfile(Path("tensor1"))**) #J

    # => Tensor([[1, 0, 3, 0]], dtype=int32, shape=4)

 print(t2.**reshape(TensorShape(4, 75))**) #K

    # =>

    # Tensor([[0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],

    # [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],

    # [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],

    # [0.0, 0.0, 0.0, ..., 0.0, 0.0, 1.0]], dtype=float32, shape=4x75)

 t2.**astype[DType.int32]() #L**

#A Tensor t has shape 2 x 2

#B Load 1 element, which is 0

#C Load 4 int32 numbers which are all 0

#D Stores value 1 on index 0

#E Stores value 3 on index 2

#F Load item on index 2

#G Reading with index notation or Index function

#H Changing a value with the Index function

#I Write the tensor to a file

#J Read the tensor from a file

#K Reshaping a tensor from 3x10x10 to 4x75

#L Changing the type of a tensor

If your load or store has a parameter in [], this is the number of elements to be loaded or stored, otherwise only one item is read or written. You can use the normal [ix] notation for reading a value. An item from a tensor can be also retrieved (or changed) with the t[Index(x, y, z)] notation, using the Index function from § 8.8.1.

You can also write a tensor to a file with the tofile method (it is not in readable format) and read it back again with the fromfile method. As you can see, the items are stored in *row-major* order.

A tensor can be reshaped to another form given a TensorShape argument, on condition that the number of elements stays the same. The astype method can change the type of all values of a Tensor, given a new type that the current type can be converted to.

8.10.2 The Tensor type from max.driver

The module *max.driver* also contains a Tensor definition, which is like the Tensor type from the previous section an owned, indexed buffer type. This tensor type however provides primitives for working with heterogeneous hardware systems (GPUs and CPUs). It allows you to allocate on-device memory, transfer data between host and device, and query device stats. The max.driver.Tensor is designed to be hardware-aware and can be explicitly associated with specific devices.

One difference is that with the max.driver Tensor type, you can specify the rank as a parameter, like this (see tensor3.mojo):

struct Tensor[type: DType, rank: Int]

Mojo can execute code on the CPU as well as on GPU’s. So, you will want to be able to specify on which device your tensor is placed, as in the following snippet (see tensor3.mojo):

from max.driver import Tensor, cpu #A

from tensor import TensorShape, TensorSpec

def main():

    tensor = Tensor[DType.float32, **rank=2**](TensorShape(1, 2)) #B

    tensor[0, 0] = 1.0

 print(tensor)  # =>

    # Tensor([[1.0, 0.0]], dtype=float32, shape=1x2)

**device = cpu()**  #C

    tensor2 = Tensor[DType.float32, rank=2](TensorShape(1, 2), **device**)#B

    tensor2[0, 0] = 1.0

  tensor3 = tensor.**move\_to(device)** #D

#A Import the device tensor from max.driver

#B Specify the tensor with rank and/or device on which to store the tensor

#C device can take on the values cpu() or gpu()

#D Move tensor explicitly to CPU

8.10.3 The LayoutTensor type

This is a tensor type which provides precise control over memory layout and organization. It supports various memory layouts (row-major, column-major, tiled), hardware-specific optimizations, and efficient parallel access patterns. It can also be allocated on the stack. All these possibilities make it possible to create high-performance tensors.

The following Listing gives a first impression of how to use LayoutTensor in practice:

Listing 8.7 – Using LayoutTensor

from gpu.host import DeviceContext, DeviceBuffer

from layout import Layout, LayoutTensor, IntTuple #A

from layout.layout import blocked\_product

alias dtype = DType.float32

alias layout = **Layout.row\_major(4, 4)** #B

fn main() raises:

var layout0 = Layout.row\_major(3, 4) #C

var memory\_idx = layout(IntTuple(1, 2)) #D

var matrix = Layout.row\_major(2, 3) #E

var block = Layout.row\_major(2, 2) #F

var blocked = **blocked\_product(block, matrix)** #G

var tiled = blocked\_product(layout, Layout(IntTuple(2, 2))) #H

var ctx = **DeviceContext()**

var dev\_buf = ctx.enqueue\_create\_buffer[dtype](8)

var tensor = **LayoutTensor[dtype, layout](dev\_buf)** #I

var storage = InlineArray[Scalar[DType.float32], 5 \* 4](uninitialized=True)

var tensor\_5x4 = **LayoutTensor[DType.float32, Layout.row\_major(5, 4)](**

**storage** #J

**)**

#A Import LayoutTensor from module layout

#B Define a layout at comptime

#C Create a 3x4 row-major layout at run-time

#D Access the memory location for logical coordinates (1, 2)

#E Create a 2x3 matrix layout

#F Define 2x2 blocks

#G Create a blocked layout with 2x2 blocks

#H Create a tiled layout for blocked matrix multiplication

#I Create a LayoutTensor from a DeviceBuffer

#I Create a LayoutTensor with an InlineArray

The blocked\_product function creates a blocked layout. In our example this means that in any of the 6 (2 x 3) positions of the matrix layout, a block of 2 x 2 items is placed.

In calculations, a *device* can be any kind of CPU, GPU, or other execution hardware.

The term *accelerator* is used to denote any non-CPU device, like any kind of GPU, or TPU, and so on. Functions that are executed on an accelerator are also called *kernels*.

A DeviceContext is a low-level software interface from MAX to a particular accelerator.

It provides methods for allocating buffers on the device, copying data between host and device, and for compiling and running kernels on the device. The enqueue\_create\_buffer

method takes a DType value and creates a DeviceBuffer dev\_buf on the device. DeviceBuffer is a block of storage on that device. For GPU’s, this is allocated in the device's global memory.

A LayoutTensor can then be created with a Dtype value and layout as parameters, and a DeviceBuffer or an InlineArray as argument.

The Stable Diffusion model with MAX we saw in action in § 2.4.3 contains lots of manipulations and computations with tensors. But we still need to learn more about memory operations and unsafe pointers to understand that code in detail. Finally, in the next chapter, we’ll be able to do that!

Summary

* A Tuple is immutable and contains index-ordered heterogeneous values enclosed in (). Use it when you want to return a few different-type values from a function, which you can then unpack at the call site.
* A List is a mutable index-ordered, dynamically sized collection of homogeneous items.
* A Set is an unordered, dynamically sized collection of unique homogeneous values.
* A Dict is a mutable associative array of key-value pairs.
* An Optional represents a value that may or may not be present. It is a safe, nullable type and must be checked with if or or\_else to avoid compile errors.
* A Variant can contain one value at a time, but it can hold different types. Check the type with isa.
* Supporting types for tensor calculations contain Dim, Index, DimList, IndexList, IntTuple and InlineArray.
* Tensors of all ranks are the most widely used structure in AI calculations. Mostly they come as a higher-dimensional matrix of homogeneous items. Different types of tensors exist in the current Mojo and MAX libraries. They will probably converge to the LayoutTensor type, which can be configured with a highy-flexible layout and can adapt to device characteristics.