Control flow

Mojo includes several traditional control flow structures for conditional and repeated execution of code blocks.

The `if` statement

Mojo supports the `if` statement for conditional code execution. With it you can conditionally execute an indented code block if a given boolean expression evaluates to `True`.

```
temp_celsius = 25
if temp_celsius > 20:
    print("It is warm.")
    print("The temperature is", temp_celsius * 9 / 5 + 32, "Fahrenheit." )
```

```
It is warm.
The temperature is 77.0 Fahrenheit.
```

You can write the entire `if` statement as a single line if all you need to execute conditionally is a single, short statement.

```
temp_celsius = 22
if temp_celsius < 15: print("It is cool.") # Skipped because condition is False
if temp_celsius > 20: print("It is warm.")
```

```
It is warm.
```

Optionally, an `if` statement can include any number of additional `elif` clauses, each specifying a boolean condition and associated code block to execute if `True`. The conditions are tested in the order given. When a condition evaluates to `True`, the associated code block is executed and no further conditions are tested.

Additionally, an `if` statement can include an optional `else` clause providing a code block to execute if all conditions evaluate to `False`.

```
temp_celsius = 25
if temp_celsius <= 0:
    print("It is freezing.")
elif temp_celsius < 20:
    print("It is cool.")
elif temp_celsius < 30:
    print("It is warm.")
else:
    print("It is hot.")
```

```
It is warm.
```

TODO

Mojo currently does not support the equivalent of a Python `match` or C `switch` statement for pattern matching and conditional execution.

Short-circuit evaluation

Mojo follows short-circuit evaluation semantics for boolean operators. If the first argument to an `or` operator evaluates to `True`, the second argument is not evaluated.

```
def true_func() -> Bool:
    print("Executing true_func")
    return True

def false_func() -> Bool:
    print("Executing false_func")
    return False

print('Short-circuit "or" evaluation')
if true_func() or false_func():
    print("True result")
```

```
Short-circuit "or" evaluation
Executing true_func
True result
```

If the first argument to an `and` operator evaluates to `False`, the second argument is not evaluated.

```
print('Short-circuit "and" evaluation')
if false_func() and true_func():
    print("True result")
```

```
Short-circuit "and" evaluation
Executing false_func
```

Conditional expressions

Mojo also supports conditional expressions (or what is sometimes called a *ternary conditional operator*) using the syntax

```
true_result if boolean_expression else false_result
```

, just as in Python. This is most often used as a concise way to assign one of two different values to a variable, based on a boolean condition.

```
temp_celsius = 15
forecast = "warm" if temp_celsius > 20 else "cool"
print("The forecast for today is", forecast)
```

```
The forecast for today is cool
```

The alternative, written as a multi-line `if` statement, is more verbose.

```
if temp_celsius > 20:
    forecast = "warm"
else:
    forecast = "cool"
print("The forecast for today is", forecast)
```

```
The forecast for today is cool
```

The `while` statement

The `while` loop repeatedly executes a code block while a given boolean expression evaluates to `True`. For example, the following loop prints values from the Fibonacci series that are less than 50.

```
fib_prev = 0
fib_curr = 1

print(fib_prev, end="")
while fib_curr < 50:
    print(",", fib_curr, end="")
    fib_prev, fib_curr = fib_curr, fib_prev + fib_curr
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

A `continue` statement skips execution of the rest of the code block and resumes with the loop test expression.

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        continue
    print(n, end=", ")
```

```
1, 2, 4, 5,
```

A `break` statement terminates execution of the loop.

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        break
    print(n, end=", ")
```

```
1, 2,
```

Optionally, a `while` loop can include an `else` clause. The body of the `else` clause executes when the loop's boolean condition evaluates to `False`, even if it occurs the first time tested.

```
n = 5

while n < 4:
    print(n)
    n += 1
else:
    print("Loop completed")
```

```
Loop completed
```

Note

The `else` clause does *not* execute if a `break` or `return` statement exits the `while` loop.

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        break
    print(n)
else:
    print("Executing else clause")
```

```
1
2
```

The `for` statement

The `for` loop iterates over a sequence, executing a code block for each element in the sequence. The Mojo `for` loop can iterate over any type that implements an `__iter__()` method that returns a type that defines `__next__()` and `__len__()` methods.

Iterating over Mojo collections

All of the collection types in the `collections` module support `for` loop iteration. See the Collection types documentation for more information on Mojo collection types.

TODO

Iterating over Mojo native collections currently assigns the loop index variable a `Reference` to each item, not the item itself. You can access the item using the dereference operator, `[]`, as shown in the examples below. This may change in a future version of Mojo.

The following shows an example of iterating over a Mojo `List`.

```
from collections import List

states = List[String]("California", "Hawaii", "Oregon")
for state in states:
    print(state[])
```

```
California
Hawaii
Oregon
```

The same technique works for iterating over a Mojo `Set`.

```
from collections import Set

values = Set[Int](42, 0)
for item in values:
    print(item[])
```

```
42
0
```

There are two techniques for iterating over a Mojo `Dict`. The first is to iterate directly using the `Dict`, which produces a sequence of the dictionary's keys.

```
from collections import Dict

capitals = Dict[String, String]()
capitals["California"] = "Sacramento"
capitals["Hawaii"] = "Honolulu"
capitals["Oregon"] = "Salem"

for state in capitals:
    print(capitals[state[]] + ", " + state[])
```

```
Sacramento, California
Honolulu, Hawaii
Salem, Oregon
```

The second approach to iterating over a Mojo `Dict` is to invoke its `items()` method, which produces a sequence of `DictEntry` objects. Within the loop body, you can then access the `key` and `value` fields of the entry.

```
for item in capitals.items():
    print(item[].value + ", " + item[].key)
```

```
Sacramento, California
Honolulu, Hawaii
Salem, Oregon
```

Another type of iterable provided by the Mojo standard library is a *range*, which is a sequence of integers generated by the `range()` function. It differs from the collection types shown above in that it's implemented as a generator, producing each value as needed rather than materializing the entire sequence in memory. Additionally, each value assigned to the loop index variable is simply the `Int` value rather than a `Reference` to the value, so you should not use the dereference operator on it within the loop. For example:

```python
for i in range(5):
    print(i, end=", ")
```

```
0, 1, 2, 3, 4,
```

`for` loop control statements

A `continue` statement skips execution of the rest of the code block and resumes the loop with the next element of the collection.

```python
for i in range(5):
    if i == 3:
        continue
    print(i, end=", ")
```

```
0, 1, 2, 4,
```

A `break` statement terminates execution of the loop.

```python
for i in range(5):
    if i == 3:
        break
    print(i, end=", ")
```

```
0, 1, 2,
```

Optionally, a `for` loop can include an `else` clause. The body of the `else` clause executes after iterating over all of the elements in a collection.

```python
for i in range(5):
    print(i, end=", ")
else:
    print("\nFinished executing 'for' loop")
```

```
0, 1, 2, 3, 4,
Finished executing 'for' loop
```

The `else` clause executes even if the collection is empty.

```python
from collections import List
```

```
empty = List[Int]()
for i in empty:
    print(i[])
else:
    print("Finished executing 'for' loop")
```

```
Finished executing 'for' loop
```

Note

The `else` clause does *not* execute if a `break` or `return` statement terminates the `for` loop.

```
from collections import List

animals = List[String]("cat", "aardvark", "hippopotamus", "dog")
for animal in animals:
    if animal[] == "dog":
        print("Found a dog")
        break
else:
    print("No dog found")
```

```
Found a dog
```

Iterating over Python collections

The Mojo `for` loop supports iterating over Python collection types. Each item retrieved by the loop is a `PythonObject` wrapper around the Python object. Refer to the Python types documentation for more information on manipulating Python objects from Mojo.

The following is a simple example of iterating over a mixed-type Python list.

```
from python import Python

# Create a mixed-type Python list
py_list = Python.evaluate("[42, 'cat', 3.14159]")
for py_obj in py_list:  # Each element is of type "PythonObject"
    print(py_obj)
```

```
42
cat
3.14159
```

TODO

Iterating over a Mojo collection currently assigns the loop index variable a `Reference` to each element, which then requires you to use the dereference operator within the loop body. In contrast, iterating over a Python collection assigns a `PythonObject` wrapper for the element, which does *not* require you to use the dereference operator.

There are two techniques for iterating over a Python dictionary. The first is to iterate directly using the dictionary, which produces a sequence of its keys.

```
from python import Python

# Create a mixed-type Python dictionary
py_dict = Python.evaluate("{'a': 1, 'b': 2.71828, 'c': 'sushi'}")
for py_key in py_dict:  # Each element is of type "PythonObject"
    print(py_key, py_dict[py_key])
```

```
a 1
b 2.71828
c sushi
```

The second approach to iterating over a Python dictionary is to invoke its `items()` method, which produces a sequence of 2-tuple objects. Within the loop body, you can then access the key and value by index.

```
for py_tuple in py_dict.items():  # Each element is of type "PythonObject"
    print(py_tuple[0], py_tuple[1])
```

```
a 1
b 2.71828
c sushi
```

Was this page helpful?

Edit this page