Python integration

Mojo is still in early development and many Python features are not yet implemented. You can't currently write everything in Mojo that you can write in Python. And Mojo doesn't have its own ecosystem of packages yet.

To help bridge this gap, Mojo lets you import Python modules, call Python functions, and interact with Python objects from Mojo code. The Python code runs in a standard Python interpreter (CPython), so your existing Python code doesn't need to change.

Create a Python environment

To successfully integrate Python code with your Mojo project, your environment must have a compatible Python runtime installed along with any additional Python packages that you want to use. Currently, you can create a compatible environment in a couple of ways:

- We recommend that you use Magic, our package manager and virtual environment manager for MAX and Mojo projects. To use Magic to create and manage the virtual environment for your Mojo/Python project, first follow the instructions in Install Magic. Then you can create a new Mojo project like this:

  ```
  $ magic init my-mojo-project --format mojoproject
  ```

  After creating the project, you can enter the project and install any dependencies, for example NumPy:

  ```
  $ cd my-mojo-project
  ```

  ```
  $ magic add "numpy>=2.0"
  ```

- Alternatively, you can also add MAX and Mojo to a conda project. To do so, follow the steps in Add MAX/Mojo to a conda project.
- It's also possible to convert an existing conda project to Magic as documented in Migrate a conda project to Magic.

Import a Python module

To import a Python module in Mojo, just call `Python.import_module()` with the module name. The following shows an example of importing the standard Python NumPy package:

```
from python import Python

def main():
    # This is equivalent to Python's `import numpy as np`
    np = Python.import_module("numpy")

    # Now use numpy as if writing in Python
    array = np.array([1, 2, 3])
    print(array)
```

Running this program produces the following output:

```
[1 2 3]
```

Assuming that you have the NumPy package installed in your environment, this imports NumPy and you can use any of its features.

A few things to note:

- The `import_module()` method returns a reference to the module in the form of a `PythonObject` wrapper. You must store the reference in a variable and then use it as shown in the example above to access functions, classes, and other objects defined by the module. See Mojo wrapper objects for more information about the `PythonObject` type.
- Currently, you cannot import individual members (such as a single Python class or function). You must import the whole Python module and then access members through the module name.
- Mojo doesn't yet support top-level code, so the `import_module()` call must be inside another method. This means you may need to import a module multiple times or pass around a reference to the module. This works the same way as Python: importing the module multiple times won't run the initialization logic more than once, so you don't pay any performance penalty.
- `import_module()` may raise an exception (for example, if the module isn't installed). If you're using it inside an `fn` function, you need to either handle errors (using a `try/except` clause), or add the `raises` keyword to the function signature. You'll also see this when calling Python functions that may raise exceptions. (Raising exceptions is much more common in Python code than in the Mojo standard library, which limits their use for performance reasons.)

Note

Mojo loads the Python interpreter and Python modules at runtime, so wherever you run a Mojo program, it must be able to access a compatible Python interpreter, and to locate any imported Python modules. For more information, see

Import a local Python module

If you have some local Python code you want to use in Mojo, just add the directory to the Python path and then import the module.

For example, suppose you have a Python file named `mypython.py`:

mypython.py

```
import numpy as np

def gen_random_values(size, base):
    # generate a size x size array of random numbers between base and base+1
    random_array = np.random.rand(size, size)
    return random_array + base
```

Here's how you can import it and use it in a Mojo file:

main.mojo

```
from python import Python

def main():
    Python.add_to_path("path/to/module")
    mypython = Python.import_module("mypython")

    values = mypython.gen_random_values(2, 3)
    print(values)
```

Both absolute and relative paths work with `add_to_path()`. For example, you can import from the local directory like this:

```
Python.add_to_path(".")
```

Call Mojo from Python

As shown above, you can call out to Python modules from Mojo. However, there's currently no way to do the reverse—import Mojo modules from Python or call Mojo functions from Python.

This may present a challenge for using certain modules. For example, many UI frameworks have a main event loop that makes callbacks to user-defined code in response to UI events. This is sometimes called an "inversion of

control" pattern. Instead of your application code calling *in* to a library, the framework code calls *out* to your application code.

This pattern doesn't work because you can't pass Mojo callbacks to a Python module.

For example, consider the popular Tkinter package. The typical usage for Tkinter is something like this:

- You create a main, or "root" window for the application.
- You add one or more UI widgets to the window. The widgets can have associated callback functions (for example, when a button is pushed).
- You call the root window's `mainloop()` method, which listens for events, updates the UI, and invokes callback functions. The main loop keeps running until the application exits.

Since Python can't call back into Mojo, one alternative is to have the Mojo application drive the event loop and poll for updates. The following example uses Tkinter, but the basic approach can be applied to other packages.

First you create a Python module that defines a Tkinter interface, with a window and single button:

myapp.py

```
import tkinter as tk

class App:
    def __init__(self):
        self._root = tk.Tk()
        self.clicked = False

    def click(self):
        self.clicked = True

    def create_button(self, button_text: str):
        button = tk.Button(
            master=self._root,
            text=button_text,
            command=self.click
        )
        button.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

    def create(self, res: str):
        self._root.geometry(res)
        self.create_button("Hello Mojo!")

    def update(self):
        self._root.update()
```

You can call this module from Mojo like this:

main.mojo

```
from python import Python

def button_clicked():
    print("Hi from a Mojo🔥 fn!")

def main():
    Python.add_to_path(".")
    app = Python.import_module("myapp").App()
    app.create("800x600")

    while True:
        app.update()
        if app.clicked:
            button_clicked()
            app.clicked = False
```

Instead of the Python module calling the Tkinter `mainloop()` method, the Mojo code calls the `update()` method in a loop and checks the `clicked` attribute after each update.