Python types

When calling Python methods, Mojo needs to convert back and forth between native Python objects and native Mojo objects. Most of these conversions happen automatically, but there are a number of cases that Mojo doesn't handle yet. In these cases you may need to do an explicit conversion, or call an extra method.

Mojo types in Python

Mojo primitive types implicitly convert into Python objects. Today we support lists, tuples, integers, floats, booleans, and strings.

For example, given this Python function that prints Python types:

```
%%python
def type_printer(value):
    print(type(value))
```

(You can ignore the `%%python` at the start of the code sample; it's explained in the note below.)

You can pass this Python function Mojo types with no problem:

```
type_printer(4)
type_printer(3.14)
type_printer(("Mojo", True))
```

```
<class 'int'>
<class 'float'>
<class 'tuple'>
```

Note

This is a simplified code example written as a set of Jupyter notebook cells. The first cell includes the `%%python` directive so it's interpreted as Python. The second cell includes top-level Mojo code. You'd need to adjust this code to run it elsewhere.

Python types in Mojo

You can also use Python objects from Mojo. For example, Mojo's `Dict` and `List` types don't natively support heterogeneous collections. One alternative is to use a Python dictionary or list.

For example, to create a Python dictionary, use the `dict()` method:

```
from python import Python

def use_dict():
    var dictionary = Python.dict()
    dictionary["item_name"] = "whizbang"
    dictionary["price"] = 11.75
    dictionary["inventory"] = 100
    print(dictionary)
```

Mojo wrapper objects

When you use Python objects in your Mojo code, Mojo adds the `PythonObject` wrapper around the Python object. This object exposes a number of common double underscore methods (dunder methods) like `__getitem__()` and `__getattr__()`, passing them through to the underlying Python object.

You can explicitly create a wrapped Python object by initializing a `PythonObject` with a Mojo literal:

```
from python import PythonObject

var py_list: PythonObject = [1, 2, 3, 4]
```

Most of the time, you can treat the wrapped object just like you'd treat it in Python. You can use Python's `[]` operators to access an item in a list, and use dot-notation to access attributes and call methods. For example:

```
var n = py_list[2]
py_list.append(5)
```

If you want to construct a Python type that doesn't have a literal Mojo equivalent, you can also use the `Python.evaluate()` method. For example, to create a Python `set`:

```
def use_py_set():
    var py_set = Python.evaluate('set([2, 3, 5, 7, 11])')
    var num_items = len(py_set)
    print(num_items, " items in set.")  # prints "5 items in set"
    print(py_set.__contains__(6))       # prints "False"
```

TODO: You should be able to use the expression `6 in py_set`. However, because of the way `PythonObject` currently works, you need to call the `__contains__()` method directly.

Some Mojo APIs handle `PythonObject` just fine, but sometimes you'll need to explicitly convert a Python value into a native Mojo value.

Currently `PythonObject` conforms to the `Intable`, `Stringable`, and `Boolable` traits, which means you can convert Python values to Mojo `Int`, `String`, and `Bool` types using the built-in `int()`, `str()`, and `bool()` functions, and print Python values using the built-in `print()` function.

`PythonObject` also provides the `to_float64()` for converting to a Mojo floating point value.

```
var i: Int = int(py_int)
var s: String = str(py_string)
var b: Bool = bool(py_bool)
var f: Float64 = py_float.to_float64()
```

Comparing Python types in Mojo

In conditionals, Python objects act like you'd expect them to: Python values like `False` and `None` evaluate as false in Mojo, too.

If you need to know the type of the underlying Python object, you can use the `Python.type()` method, which is equivalent to the Python `type()` builtin. You can compare the identity of two Python objects using the `Python.is_type()` method (which is equivalent to the Python `is` operator):

```
def python_types():
    from python import Python
    from python import PythonObject

    var value1: PythonObject = 3.7
    var value2 = Python.evaluate("10/3")
    var float_type = Python.evaluate("float")

    print(Python.type(value1))   # <class 'float'>
    print(Python.is_type(Python.type(value1), Python.type(value2)))  # True
    print(Python.is_type(Python.type(value1), float_type))           # True
    print(Python.is_type(Python.type(value1), Python.none()))        # False
```

One TODO item here: The `Python.is_type()` method is misleadingly named, since it doesn't compare *types*, but object identity.