HPE

If it walks like Python and quacks like Python, it must be....Chapel?

Jade Abraham and Lydia Duncan

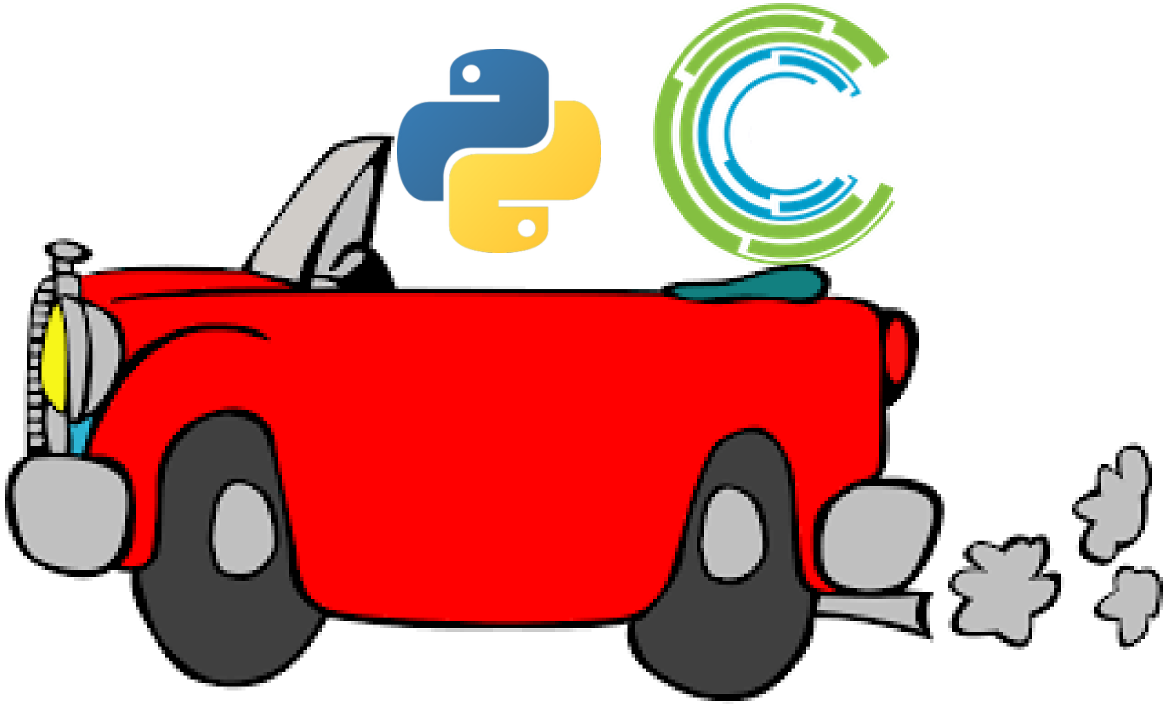
October 10, 2025

What are we talking about?

- For a long time you could call Chapel from Python
  - Compile Chapel to a shared library, load it from Python
  - Python is the “driver”

- Can we put Chapel in the driver seat?
  - Yes!

- Outline
  - Motivating Example
  - Integrated Example
  - Internals
  - Next Steps



Python isn’t a high-performance language, why?

- Dynamic execution!
  - Change your program without recompiling
  - Embed a fully fledged interpreter in your application
  - Do funky dynamic execution not possible in Chapel

- Massive ecosystem of easy-to-use libraries
  - pytorch
  - tensorflow
  - scikit-learn
  - your-favorite-ai-library

Example Time!

Motivating Example

```
proc square(x: int): int do return x * x;
```

A Chapel function to "apply" to an array

```
use BlockDist;
config const n = 16;
var myArr = blockDist.createArray(1..n, int);
myArr = myArr.domain;
```

Create a block-distributed array named 'myArray'

```
writeln("myArr before: ", myArr);

forall x in myArr do
  myArr[x] = square(x);

writeln("myArr after: ", myArr);
```

For each element in the array, call 'square( )'

The computation is both distributed and parallel

How do we make this dynamic?

Dynamic Execution Part 1

```
const square = """
  def square(x):
    return x * x
""".dedent();

import Python;
record funcPerLocale {
  var i = new Python.Interpreter();
  forwarding var func = i.createModule(square).get("square");
}

// ...

coforall l in myArr.targetLocales() do on l {
  var f = new funcPerLocale();
  for i in myArr.localSubdomain() do
    myArr[i] = f(myArr[i]): int;
}
```

The Python equivalent function, embedded as a Chapel string in the source code

A Chapel widget that handles the book-keeping needed to load and call Python

Create a new Python module and get 'square( )' as a handle to a Python Value

Explicitly distributes execution like 'forall', creating a function 'f( )' for each locale

The computation is distributed, not parallel

Calls 'f( )' like any normal Chapel function!

The result is a generic Value, the explicit cast extracts the integer value

Dynamic Execution Part 2

```
config const modName = "func";
config const funcName = "func";

import Python;
record funcPerLocale {
  var i = new Python.Interpreter();
  forwarding var func = i.importModule(modName).get(funcName);
}

// ...

coforall l in myArr.targetLocales() do on l {
  var f = new funcPerLocale();
  for i in myArr.localSubdomain() do
    myArr[i] = f(myArr[i]): int;
}
```

Instead of embedding the Python code in the program, extract it out to a file

func.py

```
def func(x):
  return x

def square(x):
  return x * x

def cube(x):
  return square(x) * x
```

The behavior is now fully editable at runtime

```
$ ./example -nl4

$ ./example -nl4 --funcName=cube

$ ./example -nl4 --modName=numpy --funcName=negative
```

Integrated Example

Integrated Example

- We started by showing you calling out to a Python file
  - This next example is going to show you how to intermingle Python calls with your Chapel code using Parquet
  - It will show a variety of features available in the Python module:
    - Importing modules
    - Calling functions
    - Accessing fields on Python types
    - Using both generic and specific Python types
      - Including Python/NumPy arrays
    - Casting to supported Chapel types
    - Iterating over Python types
    - Interoperating with NumPy arrays through Parquet

- We're not going to focus on the details of reading a Parquet file

Integrated Example

- To start, we’ll want to import the Parquet module:

```
var pa = interp.importModule("pyarrow");
var pq = interp.importModule("pyarrow.parquet");
```

- We’ll open a Parquet file…

```
var parquet_file = pq.call("ParquetFile", filename);
```

- We can get the columns in that file using field accesses via the ‘get( )’ call
  - We then can cast the Python-type result to a Chapel list of strings

```
var columns = parquet_file.get("schema").get("names"): list(string);
```

Integrated Example

- We will then iterate over the file and store its contents into an array of Chapel lists of ambiguous Python types

```
var data_chunks: [0..<columns.size] list(owned Value?);

for batch in parquet_file.call("iter_batches", kwargs=["batch_size" => 300]) {
  for (col, idx) in zip(columns, 0..) {
    data_chunks[idx].pushBack(batch.call("__getitem__", col));
  }
}
```

Create the array of lists

Iterate over the Python Parquet file type

Store the Python Value into the relevant Chapel list

Integrated Example

- Next, we'll iterate over the columns and determine the sum

```
var num_rows = parquet_file.get("metadata").get("num_rows"): int;
var schema_arrow = parquet_file.get("schema_arrow");

for (col, idx) in zip(columns, 0..) {
  var rowType = schema_arrow.call("field", col).get("type");

  if pa.call("int64") == rowType {
    var arr = getArray(int(64), data_chunks[idx], num_rows);
    writeln("Column: ", col, " Sum: ", + reduce arr);
  } else if pa.call("float64") == rowType {
    var arr = getArray(real(64), data_chunks[idx], num_rows);
    writeln("Column: ", col, " Sum: ", + reduce arr);
  }
}
```

Get a string description of the Python type stored

Iterate over the Chapel list type

If statement because Python's type is dynamic

Calls to a Chapel function that will use numpy

Integrated Example

- Finally, let's dive into 'getArray( )'

```
proc getArray(type eltType, ref data_chunks: list(owned Value?), num_rows: int) {
  var arr: [0..<num_rows] eltType;
  var i = 0;
  for chunk in data_chunks {
    var chunk_arr = chunk!.call(owned PyArray(eltType, 1), "to_numpy",
                                kwargs=["zero_copy_only" => false,
                                        "writable" => true]);
    arr[i..#chunk_arr.size] = chunk_arr.array();
    i += chunk_arr.size;
  }
  return arr;
}
```

Traverse the Chapel list of Python Values

Call a Python method, returning a handle to a NumPy array

'.array( )' returns a reference, but storing into the Chapel array is a copy

Internals

Internals

- The Python interpreter is embedded into the Chapel application
  - Manages the GIL, when present
  - Increments and decrements references to Python objects, ensuring they stay alive and get cleaned up

- Library itself actively converts between Chapel and Python types
  - Arguments to library calls can be of either type, library need to convert to the appropriate type
  - Can register custom types, enabling their use
  - Python objects are referenced rather than copied, including arrays

- Vast majority of implementation is Chapel module code
  - Relies on C interoperability to access Python's C API
  - Some additional Chapel-specific C functions and macros were necessary
    - But otherwise Chapel language features were sufficient
  - Could potentially be used as a blueprint for interoperability with other interpreted languages?

What’s Next?

What’s Next?

- New features in both Chapel and Python can enable better multi-threaded performance
  - GIL-less Python – Python is inherently single threaded because of the GIL, removing it is “free” performance
  - Chapel function pointer support can enable tight integration with Python JIT libraries like ‘numba’

- GPU’s
  - Using Chapel’s GPU support, we can call Python GPU functions, like kernels from ‘cupy’

- Shave off the sharp edges
  - Multiple libraries could all try to start Python interop, this will fall over
  - Interoperability with some Python libraries can fall over
    - This is due to internal Python C API incompatibilities
    - We may never be able to make this better

- Investigate ways to improve the syntax for interoperating with Python

Thank You