Chapel for Python Programmers Documentation

Release 0.1

Simon A. F. Lund

Jun 07, 2018

Contents

Subtitle: How I Learned to Stop Worrying and Love the Curlybracket.

So, what is Chapel and why should you care? We all know that Python is the best thing since sliced bread. Python comes with batteries included and there is nothing that can't be expressed with Python in a short, concise, elegant, and easily readable manner. But, if you find yourself using any of these packages - Bohrium, Cython, distarray, mpi4py, threading, multiprocessing, NumPy, Numba, and/or NumExpr - you might have done so because you felt that Python's batteries needed a recharge.

You might also have started venturing deeper into the world of curlybrackets. Implementing low-level methods in C/C++ and binding them to Python. In the process you might have felt that you gained performance but lost your productivity. However, there is an alternative, it does have curlybrackets, but you won't get cut on the corners.

The alternative is Chapel, and it comes with a set of turbo-charged batteries for expressing parallelism, communication, and thereby providing performance! If such matters are important to you, and you enjoy a nice clean syntax, then you might start caring about Chapel.

CHAPTER 1

Getting Started

As a Python user, you are accustomed to running and having Python readily available on almost every machine you use. Chapel is equivalently portable (and more so). However, since Chapel is an emerging technology, it is not quite part of the standard software stack that comes bundled with your operating system. You therefore need to go ahead and download and install Chapel on your system.

If you are using a popular Linux-based operating system you will most likely be successful by running these commands:

```
# Download and unpack
cd /tmp
curl -L -O http://sourceforge.net/projects/chapel/files/chapel/1.9.0/chapel-1.9.0.tar.
↪gz
tar xzf chapel-1.9.0.tar.gz
mv /tmp/chapel-1.9.0 ~/chapel

# Build Chapel
cd ~/chapel
make

# Setup your environment, add this command to ~/.bashrc for permanent installation.
source ~/chapel/util/setchplenv.bash
```

After doing the above you should be able to:

```
# Compile an example program
chpl -o hello ~/examples/hello.chpl
# Run it
./hello
```

Running "./hello" should output:

```
Hello, world!
```

If you are running MacOSX, Windows, or for some other the reason the above commands does not work for you then consult the official quick start instructions.

1.1 Compiling

What is that!? A binary! Ohh my. . .

Chapel is currently a compiled language. However, it lets you write and compile very simple programs. There is no annoying boiler-plate needed to get going.

Python		Chapel
`print "Hello, World!"`		`writeln("Hello, World!");`

And if you like to structure your code, Chapel has neat means for doing so.

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>

```
def main():
    print "Hello, World!"

if __name__ == "__main__":
    main()
```

</td><td></td><td>

```
module Hello {
    proc main() {
            writeln(
↪"Hello, World!");
    }
}
```

</td></tr>
</table>

All examples in this tutorial / reference guide are compilable. Which means that you can take any snippet and put it into a file like *exploring.chpl* and compile it:

```
chpl -o exploring exploring.chpl
```

Which will create a binary named *exploring* to execute whatever you have written in exploring.chpl.

CHAPTER 2

Language Basics

This section provides an informal language reference. It takes you through the base language features of Python and provides an example of how an equivalent program would be expressed in Chapel.

2.1 Variables and Types

In Python, variables are *implicitly* declared and their type determined when they are assigned to. In Chapel, variable declaration is *explicit*, but the type of the variable can be inferred from its use in a manner equivalent to that of Python.

Python		Chapel

`answer = 42`
`distance = 123.45`
`computer = "Earth"`
		`var answer = 42;`
		`var distance = 123.45;`
		`var computer = "Earth";`

Types in Python are dynamic, meaning that a variable can change type during its lifetime. The type of a variable in Chapel is static and inferred at compile-time, which means that a type is assigned and cannot be changed at runtime.

2.2 Comments

Python		Chapel

`# Single-line comment`
 	
`"""`
`    Multi-line comments`
`"""`
		`// Single-line comment`
		
		`/*`
		`    Multi-line comment`
		`*/`

2.2.1 Literals

These work in much the same way that you are used to. A brief overview is provided below.

Python		Chapel

`bl = True         # Booleans`
`bl = False`
 	
`ud = 42           # Unsigned digits`
`sd = -42          # Signed digits`
 	
`hd = 0x2A         # Hex-Digits`
`hd = 0X2A`
 	
`bd = 0b101010     # Binary-Digits`
`bd = 0B101010`
 	
`r = 42.0         # Reals`
 	
`s = '42'         # Strings`
 	
`s = "42"`
 	
`# Complex / imaginary`
`z = 1 + 2.0j`
 	
`# Complex accessors`
`z.real     # For the real part`
`z.imag     # For for imaginary part`
		`var bl = true;        // Booleans`
		`    bl = false;`
		
		`var ud = 42;          // Unsigned digits`
		`var sd = -42;         // Signed digits`
		
		`var hd = 0x2A;        // Hex-Digits`
		`    hd = 0X2A;`
		
		`var bd = 0b101010;    // Binary-Digits`
		`    bd = 0B101010;`
		
		`var r = 42.0;        // Reals`
		
		`var s = '42';        // Strings`
		`    s = "42";`
		
		`// Complex / imaginary`
		`var z = 1 + 2.0i;    // Common`
		`    z = (1.0, 2.0):complex; // Alternative syntax`
		
		`// Complex accessors`
		`z.re;        // For the real part`
		`z.im;        // For the imaginary part`

2.3 Console input / output

You can write to the console (standard output) using `write` and `writeln`:

Python		Chapel

`print "Hello, you."      # With a newline`
`print "Hello, you.",     # Without a newline`
		`writeln("Hello, you."); // With a newline`
		`write("Hello, you.");   // Without a newline`

You can read input from the console (standard input) using `read` and `readln`:

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>first_answer = raw_input(
    "The Answer to the ultimate question is?\n"
)
print "That is", int(first_answer) == 42

second_answer = raw_input(
    "What is the largest biological computer?\n"
)
print "That is", str(second_answer) == "Earth"</td><td></td><td>writeln("The Answer to the Ultimate Question is?");
var first_answer = read(int);

writeln("That is ", first_answer == 42);

writeln("What is the largest biological computer?");
var second_answer = read(string);

writeln("That is ", second_answer == "Earth");</td></tr>
</table>

Note: Notice that the interface for reading input is quite different, though equally simple. In Python you need to explicitly cast the input, whereas in Chapel the type of the input is provided to the `read`/`readln` functions directly.

2.4 Conditionals and Blocks

Python is famous for using an indentation guided block-structure, thereby arguably improving readability and increasing consistency of code-style. Chapel uses curly-brackets to denote the start and end of a block.

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre>
#
light = raw_input("Which color is the traffic light?\n")
</pre></td><td></td><td><pre>
writeln("Which color is the traffic light?");
var light = read(string);
</pre></td></tr>
<tr><td><pre>
if light == "green":
    print "You can cross the street now."
</pre></td><td></td><td><pre>
if light == "green" {
    writeln("You can cross the street now.");
}
</pre></td></tr>
<tr><td><pre>
if light == "green":
    print "You can cross the street now."
else:
    print "Wait for the green light."
</pre></td><td></td><td><pre>
if light == "green" {
    writeln("You can cross the street now.");
} else {
    writeln("Wait for the green light.");
}
</pre></td></tr>
<tr><td><pre>
if light == "green":
    print "You can cross the street now."
elif light == "yellow":
    print "CAUTION!"
</pre></td><td></td><td><pre>
if light == "green" {
    writeln("You can cross the street now.");
} else if light == "yellow" {
    writeln("CAUTION!");
}
</pre></td></tr>
<tr><td><pre>
if light == "green":
    print "You can cross the street now."
elif light == "yellow":
    print "CAUTION!"
else:
    print "Do not cross!"
</pre></td><td></td><td><pre>
if light == "green" {
    writeln("You can cross the street now.");
} else if light == "yellow" {
    writeln("CAUTION!");
} else {
    writeln("Do not cross!");
}
</pre></td></tr>
</table>

2.4.1 Switch / Case

Python does not support `switch-statements` and instead relies on chaining `if-elif-else` statements.

Chapel, on the other hand, does have `switch-statements`, specifically `select-when-otherwise` statements:

Python		Chapel

<pre>#
light = raw_input("Which␣
↪color is the traffic␣
↪light?\n")
 	
if light=="green":
    print "You can cross␣
↪the street now."
elif light=="yellow":
    print "CAUTION!"
elif light=="red":
    print "Do not cross!"
else:
    print "WARNING!␣
↪Traffic-light is broken!
↪"</pre>
		<pre>writeln("Which color is␣
		↪the traffic light?");
		var light = read(string);
		
		select(light) {
		    when "green" {
		        writeln("You can␣
		↪cross the street now.");
		    }
		    when "yellow" {
		        writeln("CAUTION!
		↪");
		    }
		    when "red" {
		        writeln("Do not␣
		↪cross!");
		    }
		    otherwise {
		        writeln("WARNING!␣
		↪Traffic-light is broken!
		↪");
		    }
		}</pre>

Note: Notice that in both Python and Chapel these forms of `switch-statements` do not **fall through**, meaning that one and only one case will be executed. Coming from Python, this might not surpise you; however, if you have ever written a `switch-statement` in other languages then this may be slightly surprising.

2.5 Ranges

In Python `range` is a list-constructor often used for driving for-loops or list comprehensions. For lowered memory consumption, Python provides the generator equivalent of `range` namely `xrange`.

In Chapel a **range** is a language construct which behaves and is used in much the same way as lists are used in Python. Where you would think about lists and slicing operations in Python, think of ranges in Chapel.

Python		Chapel

<pre>r1 = xrange(1, 10) #␣
↪yields 1, 2, 3, 4, 5, 6,
↪ 7, 8, 9
r2 = xrange(10, 1) #␣
↪yields nothing</pre>
		<pre>var ns = 1..9; // yields␣
		↪1, 2, 3, 4, 5, 6, 7, 8,␣
		↪9
		    ns = 9..1; // yields␣
		↪nothing</pre>

Note: Difference in bounds!

- In Python, `range` return values in the interval `[start, stop[`.

- In Chapel a range-expression yields values the interval `[start, stop]`.

For both languages the above is a shorthand of the wider form: `start, stop, step`.

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre>
# Values in ascending order
r1 = xrange(1, 10, 1) # yields 1, 2, 3, 4, 5, 6, 7, 8, 9
r2 = xrange(1, 10, 2) # yields 1, 3, 5, 7, 9

# Values in descending order
r3 = xrange(9, 0, -1) # yields 9, 8, 7, 6, 5, 4, 3, 2, 1
r4 = xrange(9, 0, -2) # yields 9, 7, 5, 3, 1
</pre></td><td></td><td><pre>
// Values in ascending order
var ns = 1..9 by 1; // yields 1, 2, 3, 4, 5, 6, 7, 8, 9
    ns = 1..9 by 2; // yields 1, 3, 5, 7, 9

// Values in descending order
    ns = 1..9 by -1; // yields 9, 8, 7, 6, 5, 4, 3, 2, 1
    ns = 1..9 by -2; // yields 9, 7, 5, 3, 1
</pre></td></tr>
</table>

...

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre>
No equivalent in Python
</pre></td><td></td><td><pre>
// Infinite ranges
var one_to_inf = 1..; // yields from one to infinity: 1, 2, 3, 4, 5, ...
var inf_to_one = ..1; // yields from infinity to one: ..., -5, -4, -3 , -2, -1, 0, 1
var inf_to_inf = .. ; // yields from infinity to infinity: ... , ...
</pre></td></tr>
</table>

...

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre>
# yields 10 values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
ns = xrange(10)
</pre></td><td></td><td><pre>
// yields 10 values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
var ns = 0.. # 10;
</pre></td></tr>
</table>

2.6 Loops

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>

```
# Using generators
for i in xrange(1, 10):
    print i
```

</td><td></td><td>

```
// Using ranges
for i in 1..10 {
    writeln(i);
}
```

</td></tr>
</table>

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>

```
for i, v in enumerate([
↪'running', 'with',
↪'scissors']):
    print i, v
```

</td><td></td><td>

```
for (i, v) in zip(1.. , [
↪"running", "with",
↪"scissors"]) {
    writeln(i, ' ', v);
}
```

</td></tr>
</table>

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>

```
i = 0
while i<10: # while loop
    i += 1
    print i


i = 0          # do-while␣
↪look-a-like loop
cond = True
while cond:
    i += 1
    print i
    cond = i<10
```

</td><td></td><td>

```
var i = 0;  // while loop
while i<10 {
    i += 1;
    writeln(i);
}

i = 0;       // do-while␣
↪loop
do {
    i += 1;
    writeln(i);
} while(i<10);
```

</td></tr>
</table>

2.7 Functions and Types

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td>

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

</td><td></td><td>

```
proc abs(x) {
    if (x < 0) then
        return -x;
    else
        return x;
}
```

</td></tr>
</table>

Variable arguments? Argument unpacking? Return values? Return type declaration?

2.8 Lists, Arrays, Tuples, and Dicts

In Python, lists are an essential built-in datastructure. You might be frightened to learn that lists are not particularly useful in Chapel. However, fear not. Many of the uses of lists in Python are handled by ranges, such as driving loops. So if that is your primary concern, then take another look at the description of ranges above.

If you need the ability to have elements of different types in a container such as:

```
stuff = ['a string', 42, ['another', 'list', 'with', 'strings']]
```

Then take a look at tuples in the following section.

If you use lists for processing various forms of data of the same type, then what you need are Chapel arrays. Yes, that is correct, Chapel actually has arrays as first-class citizens in the language. Chapel is, to a great extent, all about arrays.

2.8.1 Tuples

Tuples work in ways quite familiar to a Python programmer. Tuples are among other things useful for packing and unpacking return-values from functions and having sequences of varying types.

Python		Chapel

coord = ('47.606165', '-122.332233');   # Assignment
print "coord =", coord   ## Tuple item access
    # - Indexing
print "Latitude =", coord[0], \
    ", Longitude =", coord[1]
 	
(latitude, longitude) = coord;   # - Unpacking
 	
print "Latitude =", latitude, \
    ", Longitude =", longitude
		var coord = (47.606165, -122.332233); // Assignment
		writeln("coord = ", coord); /// Tuple item access
		    // - Indexing
		writeln(
		    "Latitude = ", coord(1),
		  ", Longitude = ", coord(2)
		);
		
		var (latitude, longitude) = coord; // - Unpacking
		
		writeln(
		    "Latitude = ", latitude,
		  ", Longitude = ", longitude
		);

Note: Indexing scheme of tuples.

- In Python, tuple-indexing is 0-based.

- In Chapel, tuple-indexing is 1-based.

Note: Mutability of tuples.

- In Python, tuples are immutable.
- In Chapel, tuples are mutable.

2.8.2 Arrays

This section only scratches the surface of Arrays in Chapel. The use of arrays and concepts related to them are described in greater detail in the section on data parallelism.

Since Python does not support arrays within the language, a comparison to the widespread and popular array-library NumPy is used as a reference instead. The first example below illustrates the creation and iteration over a `10x10` array containing 64-bit floating point numbers.

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre>
import numpy as np

A = np.zeros((10, 10), dtype=np.float64)

for a in np.nditer(A):   # Element iteration
    print a

for i in xrange(0, 10): # Index iteration
    for j in xrange(0, 10):
        print "(%d,%d) = %f" % (i, j, A[i,j])
</pre></td><td></td><td><pre>
// No need to import, arrays are built-in

var A: [0..9, 0..9] real;

for a in A {   // Element iteration
    writeln(a);
}
                        // Index iteration
for (i, j) in A.domain {
    writeln("(",i,",",j,") = ",A[i,j]);
}
</pre></td></tr>
</table>

Note: `Domains` an unfamiliar concept!

The array syntax and semantics should be easy to follow. The interesting thing to notice is the use of `.domain` when doing indexed iteration. A `domain` is a powerful concept and you will be very pleased with it once you get to know it. However, it does require an introduction.

A `domain` defines a set of indexes. When iterating over the domain associated with an array, as in the example above, you effectively iterate over all the indexes of all elements in the array. You might be accustomed to `0-based` indexing from Python when using lists and tuples. With Chapel you can define whether you want your arrays to be `0-based` or `1-based`. In the example above, the array is `0-based` since the indexes are defined by the range `0..9`. If you would prefer `1-based` arrays you would define it using the range `1..10` instead.

This is quite a powerful feature. When using arrays as abstractions for matrices, you might find it useful to use `1-based` indexing and in other situations a different indexing scheme. With Chapel you can define the index-set and scheme that is most convenient for the domain you are working within.

Initialization

Python		Chapel

import numpy as np
 	
A = np.arange(1, 11, dtype=np.float64)
 	
print A
		// No need to import, arrays are built-in
		
		var A: [1..10] real = 1..10;
		
		writeln(A);

Whole-array operations.

Python		Chapel

import numpy as np
 	
B = np.random.random((10, 10))
C = np.random.random((10, 10))
 	
A = B + 2.0 * C
 	
for a in np.nditer(A):
    print a
		use Random;
		
		config const mySeed = SeedGenerator.currentTime; // Allow caller to set seed
		
		var A, B, C: [1..10, 1..10] real;
		fillRandom(B, mySeed); // Fill with random values
		fillRandom(C, mySeed);
		
		A = B + 2.0 * C; // Whole-array operations
		
		for a in A { // Print the result
		    writeln(a);
		}

Reductions and scans

Python		Chapel

import numpy as np
 	
A = np.arange(1, 11, dtype=np.float64)
 	
print np.sum(A)   # Reduction
 	
print np.cumsum(A)   # Scan
		// No need to import, arrays are built-in
		
		var A: [1..10] real = 1..10;
		
		writeln(+reduce(A)); // Reduction
		
		writeln(+scan(A)); // Scan

Function promotion

<table>
<tr><th>Python</th><th></th><th>Chapel</th></tr>
<tr><td><pre><code>import numpy as np

def unary(element):
    return element*3

def binary(e1, e2):
    return (e1+e2)*3

A = np.arange(1, 11, dtype=np.float64)
B = np.arange(1, 11, dtype=np.float64)

print np.sqrt(A)   # Rely on NumPy ufuncs
print map(unary, A)   # Or mapping functions
print map(binary, A, B)   # Or mapping functions
</code></pre></td><td></td><td><pre><code>// No need to import, arrays are built-in

proc unary(element) {   // User-defined functions
    return element*3;
}

proc binary(e1, e2) {
    return (e1+e2)*3;
}

var A, B: [1..10] real = 1..10;

writeln(sqrt(A));   // Promotion of built-in
writeln(unary(A));   // Promotion of userdef unary
writeln(binary(A, B));   // Promotion of userdef binary
</code></pre></td></tr>
</table>

2.8.3 Dictionaries (Associative Arrays)

Dict-comprehension?

2.9 Classes and Objects

In Python, everything is an object and all objects have a textual representation defined by the object.str(), etc. is there equivalent functionality in Chapel?

Python

```python
class Stoplight:

    def __init__(self, color):
        self.color = color




sl = Stoplight("Green")

print sl.color
```

Chapel

```chapel
class Stoplight {
    var color: string;

    proc Stoplight(color: string) {
        this.color = color;
    }
}

var sl = new Stoplight("Green");

writeln(sl.color);
```

2.10 Organizing Code

Python names modules implicitly via the filename convention. Chapel allows you to use the filename, but also allows you to define it explicitly through the "module" directive. You can also define and use submodules, or modules defined within the scope of another module.

Python

```python
def main():
    pass

if __name__ == "__main__":
    main()
```

Chapel

```chapel
module Hello {
    proc main() {

    }
}
```

Python

```python
from random import *

# Other means of importing
import random
assert random.Random
from random import Random
```

Chapel

```chapel
use Random;

// There are no equivalent means of
// of importing where the namespaces
// are maintained.
```

CHAPTER 3

Parallelism

Parallelism in Chapel is provided by the language itself in contrast to Python, which relies on modules and libraries. This section contains fewer side-by-side examples, as most of these features are harder to come by in Python. Instead, reference to libraries will be provided.

3.1 Task Parallelism

In Chapel, orchestration of parallel execution is provided by the built-in keywords: *begin*, *sync*, *cobegin*, and atomic variables (*atomic*). Task parallelism in Python is provided through libraries such as: *multiprocessing*, *threading*, *thread*, *Queue*, *queue*, *Mutex*, and *mutex*.

If you are used to the *multiprocessing* and *threading* libraries, then think of a Chapel task as either a *multiprocessing.Process* or a *threading.Thread*.

3.1.1 begin and sync

The examples below implement equivalent programs in Python and Chapel: a function is executed in parallel, arguments are passed to the function and the main program waits for the function to finish.

Python		Chapel

<pre>from multiprocessing␣
↪import Process
 	
def f(name):
    print('Hello, '+ name)
 	
if __name__ == '__main__':
    p = Process(target=f,␣
↪args=('bob',))
    p.start()
    p.join()</pre>
		<pre>proc f(name) {
		    writeln("Hello, ",␣
		↪name);
		}
		
		proc main() {
		    sync begin f("bob");
		}</pre>

In Chapel, the spawning of a task is done by using the *begin* statement, while Python requires the instantiation of a Process targeting a function and invoking the *start* method. Waiting for the parallel execution to finish is done by applying the *sync* statement in Chapel and invoking the *join* method in Python.

Spawning a task in Chapel does not require specifying a target function, blocks of code can be used:

Chapel

```
var name = "Bob";
writeln("Let us make ", name, " feel welcome.");
begin {
    writeln("Hi ", name);
    writeln("Pleased to meet you.");
}
writeln("Done welcoming ", name);
```

Which also illustrates how you can share data between tasks. Data within scope is available to the task and it is therefore not nescesarry to pass it argument via a function-call.

If you try to execute the example above you might notice that the spawning task prints out “Done welcoming …” prematurely (prior to the spawned task printing “Welcome, …”.

This is just to emphasize the use of the *sync* statement which blocks until the parallel execution finishes. So to ensure the correct ordering, apply the *sync* statement as done below:

Chapel

```
var name = "Bob";
writeln("Let us make ", name, " feel welcome.");
sync begin {
    writeln("Hi ", name);
    writeln("Pleased to meet you.");
}
writeln("Done welcoming ", name);
```

3.1.2 cobegin

begin spawns off given statement as a single task, the *cobegin* statement spawns off multiple tasks; one for each statement in the given block of statements.

Chapel

```
var name = "Bob";
writeln("Let us all say hi. ");
cobegin {
    writeln("Hi ", name, "i am Alice");
    writeln("Hi ", name, "i am John.");
    writeln("Hi ", name, "i am Jane.");
    writeln("Hi ", name, "i am Richard.");
    writeln("Hi ", name, "i am Norma.");
}
writeln("Done welcoming ", name);
```

In addition to spawning a task for each statement within the block, the *cobegin* also implicitly syncs. That is, it waits for all the statements within the block to finish executing. The above could also be expressed in terms of *begin* and *sync* by:

Chapel

```
var name = "Bob";
writeln("Let us all say hi. ");
sync {
    begin writeln("Hi ", name, "i am Alice");
    begin writeln("Hi ", name, "i am John.");
    begin writeln("Hi ", name, "i am Jane.");
    begin writeln("Hi ", name, "i am Richard.");
    begin writeln("Hi ", name, "i am Norma.");
}
writeln("Done welcoming ", name);
```

3.1.3 Synchronization Variables

sync, *single*, and *atomic*

3.2 Data Parallelism

forall, domains, arrays, reduce, scan . . .

3.2.1 Locality

locale, on

3.2.2 Domain Maps

CHAPTER 4

NumPy

CHAPTER 5

Batteries

Python is well-known for having "batteries-included". The cPython interpreter comes packaged with a rich standard library for functionality. This section gives a brief overview of how a subset of the Python standard library maps to Chapel language features and libraries.

What is the equivalent of "https://docs.python.org/2/library/" ?

5.1 argparse

Config variables. Param and config.

5.2 multiprocessing

. . .

5.3 threading

See multiprocessing.

5.4 time

CHAPTER 6

Keywords

You might stumble over keywords in Chapel that you did not see coming. The following code might look harmless for a Python programmer:

```
var begin = 1;
var end   = 10;
for n in begin..end {
  write(n);
}
writeln(".");
```

However, in Chapel `begin` is a keyword for one of the task-parallelism features of the language. The above will therefore produce an error along the lines of `syntax error: near 'begin'`. Chapel uses the following keywords:

```
_          align      atomic     begin      break
by         class      cobegin    coforall   config
const      continue   delete     dmapped    do
domain     else       enum       export     extern
for        forall     if         in         index
inline     inout      iter       label      let
local      module     new        nil        on
otherwise  out        param      proc       record
reduce     ref        return     scan       select
serial     single     sparse     subdomain  sync
then       type       union      use        var
when       where      while      yield      zip
```

CHAPTER 7

Pythonic Module

For those transitioning from Python, curious about Chapel, the `Pythonic` module might be nice to take a look at. It contains a set of helper functions mimicing the functionality of some of the functions built into Python such as `enumerate`, `xrange`, `range`, among others.

If it is useful it should probably be made available in a more convenient form, than this.

```
module Pythonic {

iter enumerate(iterable) {
    for zipped in zip(1.. , iterable) {
        yield zipped;
    }
}

iter xrange(nelements: int) {
    for i in 0..nelements-1 by 1 {
        yield i;
    }
}

iter xrange(start: int, stop: int) {
    for i in start..stop-1 by 1 {
        yield i;
    }
}

iter xrange(start: int, stop: int, step: int) {
    for i in start..stop-1 by step {
        yield i;
    }
}

//
// Python equivalents
```

(continues on next page)

(continued from previous page)

```
//

// These should return 1D arrays?
proc range(nelements) {

}

proc range(start, stop) {

}

proc range(start, stop, step) {

}

//
//  NumPy Equivalents
//
iter arange(start, stop, step) {
    yield 1;
}

//
// Hmmm how about parallel iterators? Should the above instead be forall?
// How about parallel zipped iterators?

}
```

CHAPTER 8

Python and Chapel

SciPy and its accompanying software stack[2] provides a powerful environment for scientific computing in Python. The fundamental building block of SciPy is the multidimensional arrays provided by NumPy[1]. NumPy expands Python by providing a means of doing array-oriented programming using array-notation with slicing and whole-array operations.

The high-level abstractions, however, fails the user in the quest for high performance. In which case the user must take control and choose between porting to another language or integrate with low-level APIs.

The following project ideas seek to cover some ground when choosing to port a Python/NumPy application to Chapel, or use Chapel as a backend for Python/NumPy both implicitly and explicitly.

8.1 Chapel for Python/NumPy Users

The output of this project is an introduction to the Chapel language and concepts from the perspective of a NumPy user. The introduction is written to answer the question "I am used to doing X in NumPy, how would I express X in Chapel?".

8.2 npbackend / Hidden Chapel

The strong suits of Python/NumPy are high-level abstractions, convenient array-notation and a rich environment/software stack. It would be interesting to explore how to treat NumPy as a DSL and map array operations transparently to Chapel.

Thereby maintaining abstractions, environment, existing Python/NumPy sourcecode but somehow transparently delegating parallelization to Chapel. Using and possibly expanding upon the experiences gained from the previously described project and applying sensible default strategies for mapping to domains and locales. Strategies which would to a great extent rely on implicit data-parallelism of array operations.

The work can build upon experiences from our integration of Bohrium and NumPy and would involve factoring out the glue between NumPy and Bohrium into a self-contained component which could be retargeted to Chapel.

8.3 pyChapel

The pyChapel implementation is now deprecated in favor of an approach utilizing Cython. This is a work in progress effort, but should hopefully come online shortly.

CHAPTER 9

Miscellaneous Notes

This documentation is hosted on readthedocs.org.

9.1 Development

For development, install the Python packages listed in the `requirements.txt` file.

```
# From the root of the git repo:
pip install -r requirements.txt
```

9.2 Introspection

writeln(typeToString(something.type))

CompilerWarning

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

Appendix

11.1 If Chapel had a band

One of their songs might be a cover of “Hot Chocolate - Every 1’s a Winner”:

```
"Every 1's An Iterator"

Never could believe the things you do to me
Never could believe the way you are
Every day I bless the day that you got through to me
'Cause baby, I believe that you're a star

Everyone's an iterator, that's the truth (yes, the truth)
Making loops with you is such a thrill
Everyone's an iterator, that's no lie (yes, no lie)
You never fail to satisfy (satisfy)
Let's do it again

[Instrumental]

Never could explain just what was happening to me
Just one yield from you and I'm a flame
Baby, it's amazing just how wonderful it is
That the things we like to do are just the same

Everyone's an iterator, that's the truth (yes, the truth)
Making loops with you is such a thrill
Everyone's an iterator, that's no lie (yes, no lie)
You never fail to satisfy (satisfy)

[Instrumental]

Let's do it again
```

(continues on next page)

(continued from previous page)

```
Everyone's an iterator, that's the truth (yes, the truth)
Making loops with you is such a thrill
Everyone's an iterator, that's no lie (yes, no lie)
You never fail to satisfy (satisfy)

Oh, baby
Oh, baby
Oh, baby...
```

TODO: Example of implementing the above as an iterator…

CHAPTER 12

Links