Introduction to Cython - Week 3 (Alternate: Python Development)

Richard Killam

Monday 2nd November, 2015

Outline

1. Basic Data Structures
   - list
   - set
   - dict
2. Language Constructs
   - Context Managers
   - Decorators
   - Generators

List Creation

- List multiplication (works for strings too)

```
1  my_list = [[1] * 2] * 2 ⇒ [[1, 1],
2                              [1, 1]]
```

List Creation

- List multiplication (works for strings too)

```
1  my_list = [[1] * 2] * 2 ⇒ [[1, 1],
2                              [1, 1]]
```

- List comprehension

```
1  [2**i for i in xrange(5)] ⇒ [1, 2, 4, 8, 16]
```

List Indexing

- my_list[my_list.length - 1]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
-

```
1  for(int i = 1; i < my_list.length; i += 2)
```

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
-

```
1  for(int i = 1; i < my_list.length; i += 2)
```

v.s. my_list[1::2]

List Indexing

- my_list[my_list.length - 1] v.s. my_list[-1]
-

```
1  for(int i = 1; i < my_list.length; i += 2)
```

v.s. my_list[1::2]

- range(5)[1:] ⇒ [1, 2, 3, 4]
  range(5)[:-1] ⇒ [0, 1, 2, 3]
  range(5)[1:3] ⇒ [1, 2]
  range(10)[::2] ⇒ [0, 2, 4, 6, 8]
  range(10)[1::2] ⇒ [1, 3, 5, 7, 9]

Sets

- Remove duplicates:

```
1  set([1, 2, 3, 4, 1, 2, 3, 4]) ⇒ {1, 2, 3, 4}
```

Sets

- Remove duplicates:

```
1  set([1, 2, 3, 4, 1, 2, 3, 4]) ⇒ {1, 2, 3, 4}
```

-

```
1  if x in my_set:
```

Avg: **O(1)**
Worst: O(n)

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))
- set1 | set2 == set1.union(set2)
  O(len(set1)+len(set2))

Set Operations

- set1 - set2 == set1.difference(set2)
  O(len(set1))
- set1 & set2 == set1.intersection(set2)
  Avg: O(min(len(set1, set2)))
- set1 | set2 == set1.union(set2)
  O(len(set1)+len(set2))
- set1 ^ set2 == set1.symmetric_difference(set2)
  Avg: O(len(set1))

Dictionaries

- Built-in hash map
- O(1) lookups:

```
1  my_dict.get(key) # None if key does not exist
```

- dict comprehension:

```
1  {obj.name: obj.data for obj in my_objs}
```

- Easy looping:

```
1  for key, val in my_dict.items():
2      print('{}: {}'.format(key, val))
```

What are Context Managers?

Handles allocation and release of a resource

This:

```
1 f = None
2 try:
3     f = open('f.txt', 'r')
4     # Do stuff...
5 finally:
6     if f is not None:
7         f.close()
```

Becomes:

What are Context Managers?

Handles allocation and release of a resource

This:

```
1 f = None
2 try:
3     f = open('f.txt', 'r')
4     # Do stuff...
5 finally:
6     if f is not None:
7         f.close()
```

Becomes:

```
1 with open('f.txt', 'r')
2     as f:
3     # Do stuff...
```

Why use Context Managers?

- Avoid verbose repeat code
- Ensure release is handled properly
- Variable scope retention

Context Manager Uses

- Ensure successful db transaction before commit
- Holding some I/O
- Locking a thread
- Opening a file

What are Decorators?

Classes or higher order functions that wrap a given function or class

This:

```
1 def my_f(...):
2     Do stuff...
3     ret = f(...)
4     Do more stuff...
5     return ret
```

Becomes:

What are Decorators?

Classes or higher order functions that wrap a given function or class

This:

```
1 def my_f(...):
2     Do stuff...
3     ret = f(...)
4     Do more stuff...
5     return ret
```

Becomes:

```
1 @my_decorator
2 def f(...):
3     ...
```

Why use Decorators?

- Avoid verbose repeat code
- Closures allow for state retention:

  aggregation, memoization, etc

Decorator Uses

- Argument/return checking
- Function timeout
- Logging (decorate class)
- Memoization
- Thunkifying (Parallelizing)

What are Generators?

An easy way to support iterations

This:

```
class Test(object):
    def __init__(sf,s,e):
        sf.c = s
        sf.e = e
    def __iter__(sf):
        return sf
    def next(sf):
        if sf.c>=sf.e:
            raise StopIter
        r = sf.c
        sf.c += 1
        return r
```

Becomes:

What are Generators?

An easy way to support iterations

This:

```
1  class Test(object):
2      def __init__(sf,s,e):
3          sf.c = s
4          sf.e = e
5      def __iter__(sf):
6          return sf
7      def next(sf):
8          if sf.c>=sf.e:
9              raise StopIter
10         r = sf.c
11         sf.c += 1
12         return r
```

Becomes:

```
1  def test(start, end):
2      while start < end:
3          yield start
4          start += 1
```

Why use Generators?

- Lazy evaluation
- Less memory usage

Generator Uses

- Co-routines (producer/consumer) using two-way generators
- Interpolations/regressions (unknown number of iterations)
- Process text files