COSC 2306
Data Programming

OOP and Class

Iterators

```
numbers = [1, 2, 3]
print(next(numbers))
TypeError: 'list' object is not an iterator

print(dir(numbers))
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

List is iterable but not an iterator object

Iterators

```
iter_nums = numbers.__iter__() #or iter_nums =iter(numbers)
print(next (iter_nums))
>>>1
```

```
print(dir(iter_nums))
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__length_hint__', '__lt__', '__ne__',
'__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setstate__', '__sizeof__', '__str__', '__subclasshook__']
```

__iter__ of iter_nums is the same iterator (itself)

```
print(iter_nums)
<list_iterator object at 0x1089360e0>
```

Iterators

Create your own __iter__() and __next__() methods, e.g., create an iterator that returns numbers starting from 1 and increments by 1:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self
    def __next__(self):
        x = self.a
        self.a += 1
        return x
```

```
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))  # 1
print(next(myiter))  # 2
print(next(myiter))  # 3
```

Iterators

Summary:

- You get an iterator from an iterable (e.g., list, tuple, string) using the built-in iter() function
- An iterator is an object that implements two special methods:
  - __iter__() returns the iterator object itself
  - __next__() returns the next item in the sequence, and when there are no more items, it raises StopIteration
- Use iterator:
  - through a *for* loop -- automatically handles calling iter() and next(), stops when StopIteration is raised
  - retrieve items from an iterator one by one using the next()
  - Create your own __iter__() and __next__() methods

Iterators

In-class programming:

For a given list: my_list = ['apple', 'banana', 'cherry'], please use both *next() method* and *for method* to get items in my_list

Iterators

Next method:

```
my_iterator = iter(my_list)  # get an iterator from the list
print(next(my_iterator))  # outputs: apple
print(next(my_iterator))  # outputs: banana
print(next(my_iterator))  # outputs: cherry
```

For method:

```
for fruit in my_list:
    print(fruit)
```

Generators

- Generators are a simple and powerful tool for creating iterators
- They are written like regular functions but use the yield statement whenever they want to return data (like a *continuable* return, make a function fruitful)
- Each time the next() is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed)
- Saves memory: only one value is generated and held in memory at a time

Generators

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(3)
print(next(counter))  # 1
print(next(counter))  # 2 #save count state between calls
print(next(counter))  # 3
# if call next(counter) again, it raises StopIteration

for num in count_up_to(3):
    print(num)
print(num) # 3
```

Generators

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

for char in reverse('golf'):
    print(char)
print(char)
```

```
f
l
o
g
g
```

Generators

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
x = reverse('golf')
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
f
l
o
g
```

Generators

Iterator count to max using __iter__ and __next__

```
class CountIterator:
    def __init__(self, max):
        self.current = 1
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.max:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

c = CountIterator(3)
for num in c:
    print(num)
```

Iterator count to max using generator

```
def count_generator(max):
    current = 1
    while current <= max:
        yield current
        current += 1

gen = count_generator(3)
for num in gen:
    print(num)
```

Generators

Summary:

- Anything that can be done with generators can also be done with class based iterators
- What makes generators so compact is that the __iter__() and next() methods are created automatically
- Another key feature is that the local variables and execution state are automatically saved between calls
- Make it easy to create iterators with no more effort than writing a regular function with yield statement