OOP, Iterators, Generators, OOG

CS Scholars

March 7 and March 9, 2017

1 Inheritance

Recall that when we inherit from a class, the subclass can access all attributes and methods of the super class. However, methods and attributes can also be **overwritten** in the subclass.

1. Examine the classes below:

```
class Foo():
    baz = 5
    def __init__(self):
        baz = 7
        self.baz = baz
class Bar(Foo):
    def __init__(self):
        Foo.__init__(self)
        Foo.baz = 3
        print(baz)
```

What will Python display if we execute the following code?

```
>>> f = Foo()
>>> f.baz
7

>>> Foo.baz
5


>>> b = Bar()
```

```
Error

>>> Foo.baz
3


>>> Bar.baz
3
```

2. Now we will define `Sneaky` and `Illusion`. Note that `pass` means do nothing. So Sneaky is an empty class.

```
class Sneaky: pass
class Illusion:
    def __init__(tricky, self):
        self.tricky = tricky
        print(tricky)
    def fool(you):
        print(you.tricky + " fooled you!")
```

What will Python display if we execute the following code?

```
>>> what = Illusion("what")
Error

>>> sneaky = Sneaky()

>>> tricky = Illusion.__init__("Python", sneaky)
Python

>>> tricky is sneaky
False

>>> sneaky.fool = lambda self: print(self.tricky + "is weird")
>>> sneaky.fool()
Error

>>> Illusion.fool = lambda wat: print(wat.tricky + "is wack")
>>> Illusion.fool(sneaky)
Python is wack


>>> gullible = Sneaky()
>>> gullible.tricky = "61a"
>>> sneaky.fool(gullible)
61a is weird
```

2 Iterators

We've been using iterators all along! Examine this `for` loop.

```
>>> counts = [1, 2, 3]
>>> for item in counts:
      print(item)
1
2
3
```

That for loop actually gets unpackaged as follows:

```
>>> counts = [1, 2, 3]
>>> items = iter(counts)
>>> try:
      while True:
         item = next(items)
         print(item)
      except StopIteration
         pass # Do nothing
1
2
3
```

1. What does calling `iter` on an iterable return? Returns an iterator over the elements of an iterable value.

2. What does calling `next` on an iterator return? Returns an iterator over the elements of an iterable value.

3. What is the difference between an iterable and a iterator? Iterators have init , iter , next defined. Iterables have init and iter .

4. What methods does an iterator have? What methods does an iterable have?

5. Write an iterator class Reverse which takes in a list lst and iterates through it in the reverse direction.

```
class Reverse:
    def __init__(self, lst):
        self.list = lst
        self.current = len(lst) - 1

    def __iter__(self):
        return self



    def __next__(self):
        if self.current < 0:
        raise StopIteration()
        save = self.current
        self.current -= 1
        return self.list[save]
```

6. Write an iterator class that counts down from a given number. Raise a `StopIteration` exception after 0.

```
class Countdown:
    def __init__(self, lst):
        self.n = n




    def __iter__(self):
        return self

    def __next__(self):
        if self.n < 0:
            raise StopIteration()
        save = self.n
        self.n -= 1
        return save
```

7. (a) Tammy always brings a timer with her to take exams. Unfortunately, Tammy bought the timer before the number 6 was discovered. Her timer skips every number that contains the number 6. So it would display the first 20 seconds as follows:

```
0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 17 18 19 20 21
```

Only 19 seconds have passed, but the timer shows 21. Write an iterator class that behaves like Tammys timer.

Hint: write a helper function to determine if n contains 6.

```
def has_six(n):
   if n == 0:
      return False
   return n % 10 == 6 or has_six(n//10)
class Timer:
   def __init__(self, lst):
      self.current = 0



   def __iter__(self):
      return self



   def __next__(self):
      self.current += 1
      while has_six(self.current):
         self.current += 1
      return self.current
```

(b) Now write an iterator class that takes in a Tammy second and counts down from the actual amount of time that has passed. For example, if we pass in 21, the iterator will

count down as follows:

```
19 18 17 16 15 14 ...  5 4 3 2 1 0
```

Hint: Use the `Countdown` iterator and `has_six` function. You may also find it helpful to write new helper functions.

```
class TammyCountdown:
   def __init__(self, lst):
      Countdown.__init__(self, convert_to_real_time(
         tammy_second))


def previous(n):
   if not has_six(n-1):
      return n -1
   return previous(n-1)
def convert_to_real_time(n):
   if n == 0:
      return 0
   return 1 + convert_to_real_time(previous(n))
```

3 Generators

A generator is an iterator made by calling a generator function. A generator function is one that **yields** instead of returning a value.

Here is an example of `Countdown`:

```
class Countdown:
    def __init__(self, start):
        self.start = start
    def __iter__(self):
        v = self.start
        while v > 0
            yield v
            v -= 1
```

1. How can you tell if a function is a generator function? Is there a yield statement?

2. Given the generator function `f`, what will `f()` return? Will calling `f()` cause an error? No. It will return a generator object. Since the code will not be executed there is no error.

   ```
   def f():
       start = 0
       while start != 10:
           yield starts
           start = start / 0
           start += 1
   ```

3. Write a generator function `map` that takes in an iterator, `iter`, and a function `f`. It will yield the result of applying `f` to each element in `iter`

   ```
   def map(iter, f):
       """ Function that yields result of applying f to each
          element of iter
       >>> iter = iter([1, 2])
       >>> fn = lambda x: x + 1
       >>> map = map(iter, fn)
       >>> m.next()
       2
       >>> m.next()
       3
       >>> m.next()
       Traceback (most recent call last):
       ...
       StopIteration
   ```

```
    """
def map_gen(fn, iter1):
    for elem in iter1:
        yield fn(elem)
```

4 Orders of Growth

What is the order of growth of the following functions?

1.
```
def easy(n):
    sum, i = n, 0
    while i < n:
        sum, i = sum+n, i+1
    return sum
```

$\theta(n)$

2.
```
def hard(n):
    sum = 0
    i = 0
    while sum < n:
        while i < n:
            i += 1
        sum += n
    return sum
```

$\theta(n)$