ITERATORS, GENERATORS, GENERIC FUNCTIONS
9

COMPUTER SCIENCE 61A

July 22, 2014

0.1 Warmup — What Would Python Output?

1.
```
>>> a = iter(range(2))
>>> a
```

Solution: `<range_iterator object>`

```
>>> next(a)
```

Solution: `0`

```
>>> next(a)
```

Solution: `1`

```
>>> next(a)
```

Solution: `StopIteration`

1 Iterators

An **iterator** is an abstract object that represents a sequence of values. It must be able to:

1. Know how to calculate its next value.
2. Know when it has no more values left to compute.

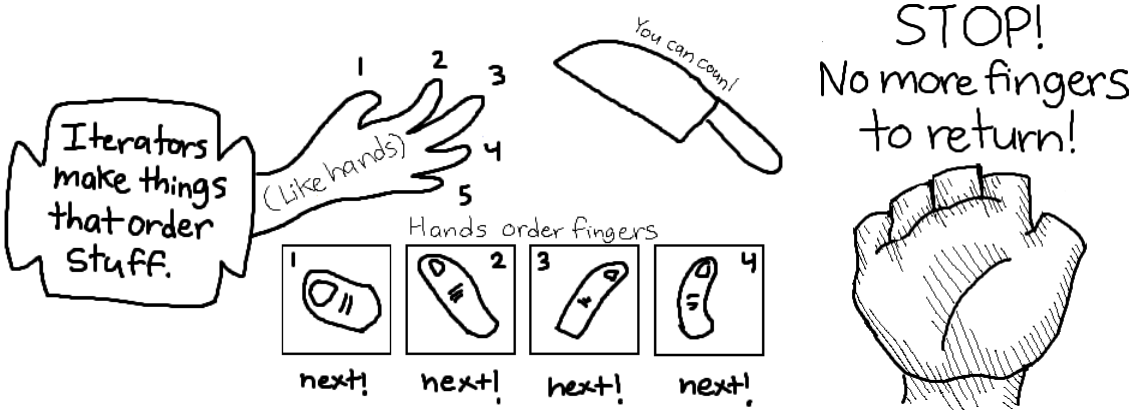
Unlike lists, an iterator *does not have its values on hand*. Instead, it computes its values on the spot when the iterator is called. An Iterator is any object that has the two methods:

`__iter__`: Method that returns an iterator that you can call `__next__` on (like hands).

`__next__`: Method that returns the next piece of data from the iterator (like fingers).


```
Most data structures  An iterator can only  That, and only once
have their values     bring up its values   per each instance.
ready to access at    in the order they
any time repeatedly.  are assigned.
```

Basically, an iterator has a series of things that it would like to return in some order.



An iterator will stop working when it has no more pieces to return. This raises the exception `StopIteration`.

Some iterators generate infinite streams, like `Naturals()`

```
>>> class NaturalsIterator:
...     def __init__(self):
...         self.current = 0
...     def __next__(self):
```

```
...             result = self.current
...             self.current += 1
...             return result
...         def __iter__(self):
...             return self
```

Here's the hand analogy in code.

```
>>> class Hand:
...         def __init__(self):
...             self.fingers = 5
...         def__iter__(self):
...             return self
...         def__next__(self):
...             if self.fingers == 0:
...                 print("No more fingers!")
...                 raise StopIteration
...             else:
...                 print("Chop chop!")
...                 self.fingers -=1
...                 return self.fingers+1
```

1.1 Questions

1. Remember your `test_dice` from the Hog Project? Produce an iterator object called `IterDice` that behaves almost as a `test_dice` does. The difference is that after returning each value once, the iterator is complete and it raises StopIteration. Remember that it should have an `__init__` and `__next__` method.

```
>>> dice = IterDice(1, 2, 3)
>>> d = iter(dice)
>>> next(d)
1
>>> next(d)
2
>>> next(d)
3
>>> next(d)
StopIteration
```

```
class IterDice(object):
    def __init__(self, *outcomes):

        self.oc = _______________
        self.start = 0

    def _____________(_____________):
        return self

    def _____________(_____________):

        if ____________________________:

            ____________________________
        self.start += 1
        return self.oc[self.start-1]
```

Solution:

```
def __init__(self, *outcomes):
    self.oc = outcomes
    self.start = 0
def __iter__(self):
    return self
def __next__(self):
    if self.start == len(self.oc):
        raise StopIteration
    self.start += 1
    return self.oc[self.start-1]
```

2. How could you modify the code so that the `IterDice` actually behaves like `test_dice`? i.e. don't raise a `StopIteration`? Write only your changes.

```
>>> d = iter(IterDice(3, 6))
>>> next(d)
3
>>> next(d)
6
>>> next(d)
```

```
3
```

Solution:

```
    def __next__(self):
        if self.start == len(self.oc):
            self.start = 0  # This line has been changed
        self.start += 1
        return self.oc[self.start-1]
```

3. Define an iterator that combines the elements of two input streams using some binary operator, also given as input. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

```
>>> from operator import add
>>> evens = Iter_Combiner(NaturalsIterator(), NaturalsIterator(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4

class Iter_Combiner():
    def __init__(self, iter1, iter2, combiner):
```

Solution:

```
        self.iter1 = iter1
        self.iter2 = iter2
        self.combiner = combiner
```

```
    def __next__(self):
```

Solution:

```
        return self.combiner(next(self.iter1), next(self.iter2))
```

```
    def __iter__(self):
        return self
```

4. What results from executing this sequence of commands?

```
>>> naturals = NaturalsIterator()
>>> doubled_naturals = Iter_Combiner(naturals, naturals, add)
>>> next(doubled_naturals)
```

Solution: `1`

```
>>> next(doubled_naturals)
```

Solution: `5`

2 Generators

A Generator is a type of iterator that utilizes the `yield` keyword to do what an iterator does. `yield` returns a value while saving everything that has already happened in the body, and continues the body from after the yielded value when `next()` is called.

```
>>> class Hand:
...     def __init__(self):
...         self.fingers = 5
...     def __iter__(self):
...         while True:
...             if self.fingers == 0:
...                 print("No more fingers!")
...                 raise StopIteration
...             print("Chop chop!")
...             yield self.fingers
...             self.fingers -= 1
```

Notice that `yield` replaces the need for a `__next__` method.

2.1 Questions

1. What do Python strings, tuples, lists, dictionaries and ranges all have in common? (Hint: What did we just learn about?)

Solution: They're all iterables and share the `__iter__` method.

2. What does the following code return?

```
>>> for elem in 5:
...     print(elem)
```

Solution: `TypeError: 'int' object is not iterable`

3. Is every generator an iterator? Is every iterator a generator?

Solution: Every generator implements the iterator interface, but not every object that implements the iterator interface is a generator.

4. Make a test dice using a generator. (Do not raise a StopIteration.)

```
class GenerDice:
    def __init__(self,*oc):
        self.oc = oc
        self.start = 0
    def __iter__(self):
```

Solution:

```
        while True:
            if self.start == len(self.oc):
                self.start = 0
            yield self.oc[self.start]
            self.start += 1
```

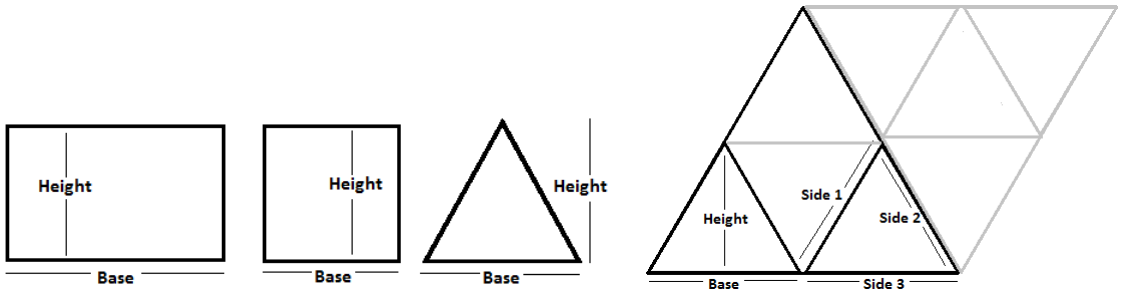
3 Generic Functions

3.1 *Do You Recall?*

Generic Operations combine together different types of objects flexibly in a large program. They aid in creating **abstract data types (ADTs)** using one of two abstraction barriers:

1. **Interface**: A shared set of data, functions, and/or messages used to manipulate various data types.
2. **Modules, Multiple Representations**: Creating independent parts (functions) permitting different design choices to coexist in a single program.

As new representations of something come along, there's no need to remove pre-existing implementations. Instead, new representations should work together with previous designs to save time by preventing programmers from starting a concept from scratch.



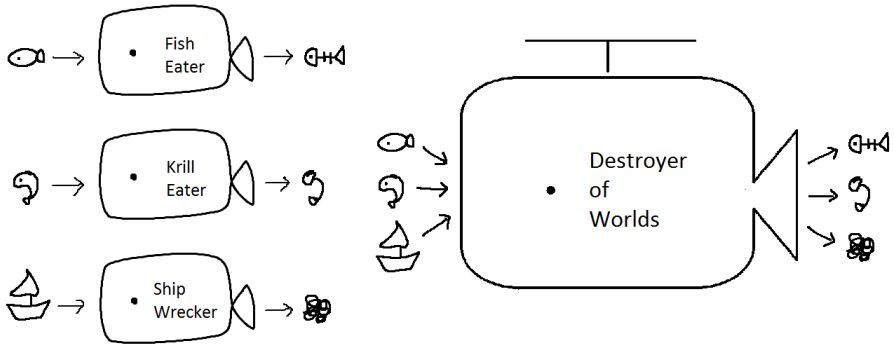
Neither triangle is wrong; different representations are better for some functions than others, such as finding the perimeter and finding the area.

3.2 Generic Functions: For Real

A generic function is a function that can take in different object types and operate on all of them. Some ways to create one include:

Type Dispatching: Creating a function for each different combination of data types for which an argument is valid, then choose one to use by checking the types of arguments passed in.

Coercion: Converting one object(1) into another object(2) so that (2) can use (1)'s methods.



If each kind of whale represents a different function, a generic function is equivalent to a whale which can eat krill, fish, ships, etc. and output the appropriate result for each different object.

3.3 Questions

1. Consider the following ways of implementing the generic `eat` function. Say whether we are using interfaces, type dispatching, data directed programming, or coercion.

   (a) You have many different whales, and they all have an `eat` method. So, you write the generic `eat` function as follows:

   ```
   def eat(whale, food):
       return whale.eat(food)
   ```

   > **Solution:** Since all of the whales have an `eat` method, we are using *interfaces* here.

   (b) Each whale has a different method for eating:

   ```
   def eat(whale, food):
       if type(whale) == FishEater:
           return whale.eat_fish(food)
       elif type(whale) == KrillEater:
           return whale.eat_krill(food)
       return whale.destroy_ship(food)
   ```

   > **Solution:** Since we check what type of whales we have in order to do the right thing, this is an example of *type dispatching*.

2. In Pokémon, you have a bag that can store many different items. While each item performs different actions, all of them can be selected for use by pressing an A Button.

   ```
   class Berry:
       def __init__(self, amount):
           self.amount =  amount
       def eat(self):
            if self.amount > 0:
                self.amount -= 1
            return self.amount

   class BattleItem:
       def __init__(self, message):
           self.message = message
       def  use(self):
            return self.message
   ```

   ```
   class KeyItem:
       def __init__(self, on, off):
           self.on = on
           self.off = off
           self.active = False

       def switch(self):
           if not self.active:
               self.active = True
               return self.on
           else:
               self.active = False
               return self.off
   ```

We want to write the code for `press_a` such that it can model selecting and using various items. The `press_a` function has been given to you; however, it depends on the dictionary `item_actions`. Fill in the `item_actions` dictionary so that `press_a` works as intended.

```
def press_a(item):
    key = type(item)
    return item_actions[key](item)

# Make the item_actions dictionary
# which is used in press_a above.
# Hint: Domain and Range
# Look at press_a and figure out
# what the keys and values of
# item_actions should be.

item_actions = {




}
```

```
>>> berry = Berry(3)
>>> flute = BattleItem(\
"played a familiar tune")
>>> bike = KeyItem( \
"using bike", \
"stopped using bike")
>>> press_a(berry)
2
>>> press_a(berry)
1
>>> press_a(flute)
'played a familiar tune'
>>> press_a(berry)
0
>>> press_a(bike)
'using bike'
>>> press_a(flute)
'played a familiar tune'
>>> press_a(bike)
'stopped using bike'
```

Solution:

```
item_actions = {
  Berry: lambda x: x.eat(),
  BattleItem: lambda x: x.use(),
  KeyItem: lambda x: x.switch()
}
```

Extended Question: What Color is it?

Art professors Sheet and Screen are having a dispute over whether teal (a blueish-greenish color) is a kind of blue or green. After a week, they realize that they may have been arguing over two different hues. The decide they need a way to exactly identify their colors. The problem is, Professor Screen uses RGB (red, green, and blue) values, while Professor Sheet uses CMYK (Cyan, Magenta, Yellow, Black).

A favorite student of both (because CS students have great style), you are asked to

develop a program that will tell them what color their combination of pigment values generates. You decide to identify color **using the html color codes** (hex), a base-16 representation. Professor Screen provides you with code from a prior project:

```
class Color: # treat this as an interface, not a class
    base16 = {0:'0', 1:'1', 2:'2', 3:'3', 4:'4', 5:'5', 6:'6', 7:'7',\
        8:'8', 9:'9', 10:'A', 11:'B', 12:'C', 13:'D', 14:'E', 15:'F'}
class Color_rgb(Color):
    def __init__(self, r, g, b):
        assert (r>=0 and r<256) and (g>=0 and g<256) \
           and (b>=0 and b<256), "Values must be from 0 to 255."
        self.color_list = [r, g, b]
    @property
    def red(self):
        return self.color_list[0]
    # Similar properties for green and blue
def what_color_rgb(rgb): # hex code
    hex_color = "#"
    for hue in rgb.color_list:
        hex_color += Color.base16[hue // 16]
        hex_color += Color.base16[hue % 16]
    return hex_color
```

You make a `Color_cmyk` object for Professor Sheet:

```
class Color_cmyk(Color):
    def __init__(self, c, m, y, k):
        assert (c>=0 and c<=1) and (m>=0 and m<=1) and (y>=0 and y<=1)\
            and (k>=0 and k<=1), "Values must be from 0 to 1."
        self.color_list = [c, m, y, k]
    @property
    def cyan(self):
        return self.color_list[0]
    # Similar properties for magenta, yellow, black

def what_color_cmyk(cmyk):
    hex_color = "#"
    for hue in cmyk.color_list[:-1]: # ignore black
        num = int(255 * (1-hue) * (1-cmyk.black))
        hex_color += Color.base16[num // 16]
        hex_color += Color.base16[num % 16]
    return hex_color
```

3. Make a generic `what_color` method using type dispatching.

```
def what_color(color):
    if type(color) == Color_rgb:
```

Solution:

```
        return what_color_rgb(color)
```

```
    if type(color) == Color_cmyk:
```

Solution:

```
        return what_color_cmyk(color)
```

Later, you decide it would be better if the two classes of color should have a shared **interface** and give `Color_cmyk` property methods `red`, `green`, and `blue`.

```
# This is now part of the Color_cmyk class.
    @property
    def red(self):
        return int(255 * (1-self.cyan) * (1 - self.black))
    # Similar properties for green and blue.
```

4. Rewrite `what_color` to take advantage of the interface. (Assume that `what_color_rgb` and `what_color_cmyk` don't exist any more.)

```
def what_color(color):
    hex_color = "#"
    color_list = []
```

Solution:

```
    color_list += [color.red, color.green, color.blue]
or
    color_list.append(color.red)
    color_list.append(color.green)
    color_list.append(color.blue)
```

```
    for hue in color_list:
        hex_color += Color.base16[hue // 16]
```

```
            hex_color += Color.base16[hue % 16]
        return hex_color
```

5. Professor Sheet and Professor Screen were so pleased with your function that they've decided to collaborate on a new art project together! They've asked for your assistance in designing a function `mix_color` that will allow them to combine any two colors in equal proportions using their own system of producing color. That is, **create a `mix_color` function that will return the hex-code of blending two colors regardless of what representations are given**.

   You decide to **coerce CMYK values to RGB values with a `cmyk_to_rgb` method** to avoid type-dispatching. (What if we introduce another kind of color? There's just too many possible mixing combinations!)

   ```
   def cmyk_to_rgb(color):
   ```

   Solution:

   ```
       assert type(color) == Color_cmyk:
       return Color_rgb(color.red, color.green, color.blue)
   ```

   Time to write `mix_color`.

   ```
   def mix_color(c1,c2):
   ```

   Solution:

   ```
   if type(c1) == Color_cmyk:
       c1 = cmyk_to_rgb(c1)
   if type(c2) == Color_cmyk:
       c2 = cmyk_to_rgb(c2)
   return what_color_g(
       Color_rgb((c1.red+c2.red) // 2,
                 (c1.green+c2.green) // 2,
                 (c1.blue+c2.blue) // 2))
   ```