Iterators

Announcements

Iterators (Bonus Material)

Iterators

A container can provide an iterator that provides access to its elements in order

`iter(iterable)`: Return an iterator over the elements of an iterable value

`next(iterator)`: Return the next element in an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```

(Demo)

Discussion Question

What will be printed?

```
a = [1, 2, 3]
b = [a, 4]
c = iter(a)
d = c
print(next(c))
print(next(d))
print(b)
```

Higher Order Functions, Revisited
Map, Filter

(Demo)

Functions that return iterables

map, filter, zip

These objects are **not** sequences.

They are *iterables*. A "stream" of data we can iterate over.

Why?

- Can't directly slice into them.
- Don't know their length
- If we want to see all the elements at once, we need to explicitly collect them, by using list() or tuple(), or use next()

```
data = map(lambda x: x*x, range(5))

# Iterate with for loops

for num in data:

    print(num)

data = map(lambda x: x*x, range(5))

next(data) # returns 0

next(data) # returns 1 …

next(data) # eventually raises StopIteration error
```

How do we build iterators?

What's an Iterator? [Docs]

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead.

iterable

An object capable of returning its members one at a time. Examples of include all sequence types and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements sequence semantics.

Next element in generator iterable

- Iterables work because they implement some "magic methods" on them. We saw magic methods when we learned about classes,
  - e.g., __init__, __repr__ and __str__.
- The first one we see for iterables is `__next__`

- `iter()` – transforms a sequence into an iterator
  - Usually this is not necessary, but can be useful.

Iterators: The `iter` protocol [Docs]

- In order to be iterable, a class must implement the iter protocol
- The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:
  - __iter__: Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
    - This method returns an iterator object (which can be self)
  - __next__ : Return the next item from the container. If there are no further items, raise the StopIteration exception.

The Iter Protocol In Practice

- Classes get to define how they are iterated over by defining these methods
  - containers (objects like lists, tuples, etc) typically define a Container class and a separate ContainterIterator class.
- Lists, Ranges, etc are *not* directly iterators
  - We cannot call next() on them.
  - We can all iter(list), iter(range), etc if needed.
- However, they implement an `__iter__` method, and `list_iterator`, `range_iterator` class, etc.

Making Our Own Range

```
class myrange:
    def __init__(self, n, step=0):
        self.i = 0
        self.n = n
        self.step = step

    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            current = _________________
            self.i += _________________
            return current
        else:
  raise StopIteration()
```

Making Our Own Range

```
class myrange:
    def __init__(self, n, step=0):
        self.i = 0
        self.n = n
        self.step = step

    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            current = self.i
            self.i += _________________
            return current
        else:
  raise StopIteration()
```

Making Our Own Range

```
class myrange:
    def __init__(self, n, step=0):
        self.i = 0
        self.n = n
        self.step = step

    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            current = self.i
            self.i += self.step
            return current
        else:
  raise StopIteration()
```

Range HOF!

What if range() accepted a HOF argument?

```
class rangehof:
    """
    >>> x = rangehof(0, 3, lambda x: x+1)
    >>> list(x)
    [1, 2, 3]
    """
    …
    def __next__(self):
        if self.i < self.n:
            current = _____________________________
            self.i = _____________________________
            return current
        else:
            raise StopIteration()
```

What if range() accepted a HOF argument?

```
class rangehof:
    """
    >>> x = rangehof(0, 3, lambda x: x+1)
    >>> list(x)
    [1, 2, 3]
    """
    def __init__(self, start, stop, function):
        …
        self.function = function
    def __next__(self):
        if self.i < self.n:
            current = self.function(self.i)
            self.i =  current
            return current
        else:
            raise StopIteration()
```

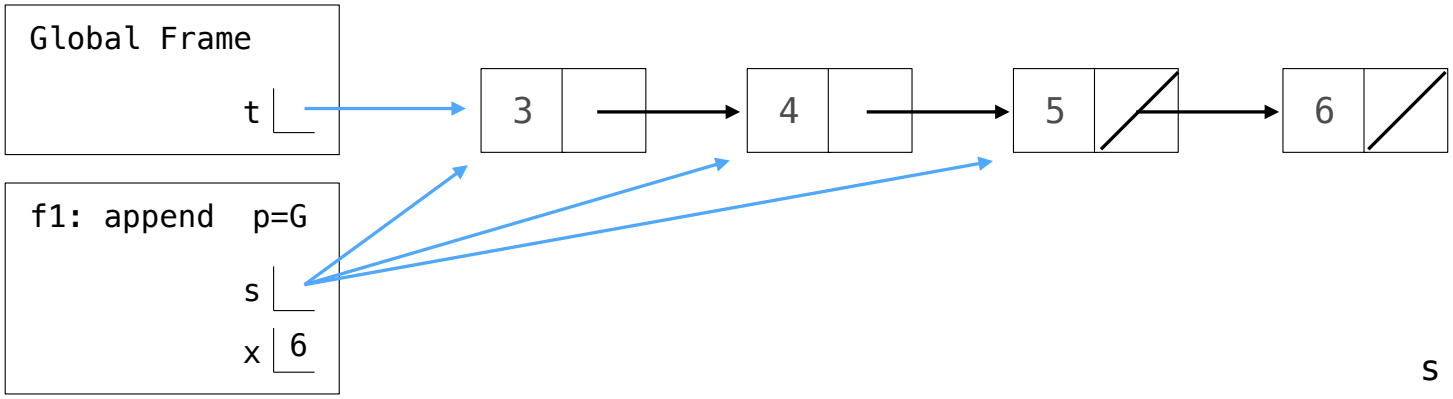
Optional Linked List Practice

Linked List Mutation

To change the contents of a linked list, assign to first and rest attributes

Example: Append x to the end of non-empty s

```
>>> t = Link(3, Link(4, Link(5)))
>>> append(t, 6)
>>> t
Link(3, Link(4, Link(5, Link(6))))
```




```
s = s.rest

s.rest = Link(x)
```

Recursion and Iteration

Many linked list processing functions can be written both iteratively and recursively

Recursive approach:

- What recursive call do you make?
- What does this recursive call do/return?
- How is this result useful in solving the problem?

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)  # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    if s.rest is not Link.empty:
        append(s.rest, x)
    else:
        s.rest = Link(x)
```

Iterative approach:

- Describe a process that solves the problem.
- Figure out what additional names you need to carry out this process.
- Implement the process using those names.

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)  # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    while s.rest is not Link.empty:
        s = s.rest
    s.rest = Link(x)
```

Example: Pop

Implement pop, which takes a linked list s and positive integer i. It removes and returns the element at index i of s (assuming s.first has index 0).

```
def pop(s, i):
    """Remove and return element i from linked list s for positive i.
    >>> t = Link(3, Link(4, Link(5, Link(6))))
    >>> pop(t, 2)
    5
    >>> pop(t, 2)
    6
    >>> pop(t, 1)
    4
    >>> t
    Link(3)
    """
    assert i > 0 and i < length(s)
    for x in range( i - 1 ):
        s = s.rest
    result = s.rest.first
    s.rest = s.rest.rest
    return result
```