CS 61AInterfaces, Linked Lists, and Generators
Fall 2019

Discussion 7: October 16, 2019 Solutions

1 Interfaces

In computer science, an **interface** is a shared set of attributes, along with a specification of the attributes' behavior. For example, an interface for vehicles might consist of the following methods:

- `def drive(self)`: Drives the vehicle if it has stopped.
- `def stop(self)`: Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a `Car` class and a `Train` can both implement the interface described above, but the `Car` probably has a different mechanism for `drive` than the `Train`.

The power of interfaces is that other programs don't have to know *how* each data type implements the interface – only that they *have* implemented the interface. The following `travel` function can work with both `Car`s and `Train`s:

```
def travel(vehicle):
    while not at_destination():
        vehicle.drive()
    vehicle.stop()
```

Magic Methods

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- `def __add__(self, other)`: Allows objects to do `self + other`.
- `def __sub__(self, other)`: Allows objects to do `self - other`.
- `def __mul__(self, other)`: Allows objects to do `self * other`.

In addition, there is also an interface for sequences:

- `def __len__(self)`: Allows objects to do `len(self)`.
- `def __getitem__(self, index)`: Allows objects to do `self[i]`.

Questions

1.1 What would Python display?

```
class A():
    def __init__(self, x):
        self.x = x
    def __repr__(self):
         return self.x
    def __str__(self):
         return self.x * 2

class B():
    def __init__(self):
         print("boo!")
         self.a = []
    def add_a(self, a):
         self.a.append(a)
    def __repr__(self):
         print(len(self.a))
         ret = ""
         for a in self.a:
             ret += str(a)
         return ret
```

```
>>> A("one")
```

one

```
>>> print(A("one"))
```

oneone

```
>>> repr(A("two"))
```

'two'

```
>>> b = B()
```

boo!

```
>>> b.add_a(A("a"))
>>> b.add_a(A("b"))
>>> b
```

2

aabb

```
>>> c = A("c")
>>> b.add_a(c)
>>> str(b)
```

```
3
```

```
'aabbcc'
```

1.2 Write the function `is_palindrome` such that it works for any data type that implements the sequence interface.

Assume that the Link class has implemented the `__len__` method and a `__getitem__` method which takes in integers.

```
def is_palindrome(seq):
    """ Returns True if the sequence is a palindrome. A palindrome is a sequence
    that reads the same forwards as backwards
    >>> is_palindrome(Link("l", Link("i", Link("n", Link("k")))))
    False
    >>> is_palindrome(["l", "i", "n", "k"])
    False
    >>> is_palindrome("link")
    False
    >>> is_palindrome(Link.empty)
    True
    >>> is_palindrome([])
    True
    >>> is_palindrome("")
    True
    >>> is_palindrome(Link("a", Link("v", Link("a"))))
    True
    >>> is_palindrome(["a", "v", "a"])
    True
    >>> is_palindrome("ava")
    True
    """


    for i in range(len(seq)//2):
        if seq[i] != seq[len(seq) - 1 - i]:
          return False
    return True
```

2 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Implementation

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Questions

2.1 Write a function that takes in a a linked list and returns the sum of all its elements. You may assume all elements in `lnk` are integers.

```
def sum_nums(lnk):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """


    if lnk == Link.empty:
        return 0
    return lnk.first + sum_nums(lnk.rest)
```

2.2 Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
```

Recursive solution:

```
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Link`s. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Iterative solution:

```
import operator
from functools import reduce
def prod(factors):
     return reduce(operator.mul, factors, 1)

head = Link.empty
tail = head
while Link.empty not in lst_of_lnks:
    all_prod = prod([l.first for l in lst_of_lnks])
    if head is Link.empty:
        head = Link(all_prod)
        tail = head
    else:
        tail.rest = Link(all_prod)
        tail = tail.rest
    lst_of_lnks = [l.rest for l in lst_of_lnks]
return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list "backwards" as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use `head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Link`s in our list of `Link`s runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

3 Iterators and Generators

An **iterable** is a data type which contains a collection of values which can be processed one by one sequentially. Some examples of iterables we've seen include lists, tuples, strings, and dictionaries. In general, any object that can be iterated over in a **for** loop can be considered an iterable.

```
>>> a = [1, 2]
>>> a_iter = iter(a)
>>> next(a_iter)
1
>>> next(a_iter)
2
>>> next(a_iter)
StopIteration
```

While an iterable contains values that can be iterated over, we need another type of object called an **iterator** to actually retrieve values contained in an iterable. Calling the **iter** function on an iterable will create an iterator over that iterable. Each iterator keeps track of its position within the iterable. Calling the **next** function on an iterator will give the current value in the iterable and move the iterator's position to the next value.

In this way, the relationship between an iterable and an iterator is analogous to the relationship between a book and a bookmark - an iterable contains the data that is being iterated over, and an iterator keeps track of your position within that data.

Once an iterator has returned all the values in an iterable, subsequent calls to **next** on that iterable will result in a `StopIteration` exception. In order to be able to access the values in the iterable a second time, you would have to create a second iterator.

One important application of iterables and iterators is the **for** loop. We've seen how we can use **for** loops to iterate over iterables like lists and dictionaries.

This only works because the **for** loop implicitly creates an iterator using the built-in **iter** function. Python then calls **next** repeatedly on the iterator, until it raises `StopIteration`.

The code to the right shows how we can mimic the behavior of **for** loops using **while** loops.

```
counts = [1, 2, 3]

for i in counts:
    print(i)

items = iter(counts)
while True:
    try:
        i = next(items)
        print(i)
    except StopIteration:
        break #Exit the while loop
```

Note that most iterators are also iterables - that is, calling **iter** on them will return an iterator. This means that we can use them inside **for** loops. However, calling **iter** on most iterators will not create a new iterator - instead, it will simply return the same iterator.

We can also iterate over iterables in a list comprehension or pass in an iterable to the built-in function **list** in order to put the items of an iterable into a list.

In addition to the sequences we've learned, Python has some built-in ways to create iterables and iterators. Here are a few useful ones:

- **range**(`start, end`) returns an iterable containing numbers from start to end-1. If `start` is not provided, it defaults to 0.
- **map**(`f, iterable`) returns a new iterator containing the values resulting from applying `f` to each value in `iterable`.
- **filter**(`f, iterable`) returns a new iterator containing only the values in `iterable` for which `f(value)` returns `True`.

Questions

3.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, and if another error occurs, write `Error`.

```
>>> lst = [6, 1, "a"]
>>> next(lst)
```

Error

```
>>> lst_iter = iter(lst)
>>> next(lst_iter)
```

6

```
>>> next(lst_iter)
```

1

```
>>> next(iter(lst))
```

6

```
>>> [x for x in lst_iter]
```

["a"]

Generators

A **generator function** is a special kind of Python function that uses a **yield** statement instead of a **return** statement to report values. *When a generator function is called, it returns a generator object, which is a type of iterator.* To the right, you can see a function that returns an iterator over the natural numbers. The **yield** statement is similar to a **return** statement. However, while a **return** statement closes the current frame after the function exits, a **yield** statement causes the frame to be saved until the next time **next** is called, which allows the generator to automatically keep track of the iteration state.

```
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

Once **next** is called again, execution resumes where it last stopped and continues until the next **yield** statement or the end of the function. A generator function can have multiple **yield** statements.

Including a **yield** statement in a function automatically tells Python that this function will create a generator. When we call the function, it returns a generator object instead of executing the body. When the generator's **next** method is called, the body is executed until the next **yield** statement is executed.

When **yield from** is called on an iterator, it will **yield** every value from that iterator. It's similar to doing the following:

```
for x in an_iterator:
    yield x
```

The example to the right demonstrates different ways of computing the same result.

```
>>> square = lambda x: x*x
>>> def many_squares(s):
...     for x in s:
...         yield square(x)
...     yield from map(square, s)
...
>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9]
```

Questions

3.1 Implement `filter_link`, which takes in a linked list `link` and a function `f` and returns a generator which yields the values of `link` for which `f` returns `True`.

Try to implement this both using a while loop and without using any form of iteration.

```
def filter_link(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> g = filter_link(link, lambda x: x % 2 == 0)
    >>> next(g)
    2
    >>> next(g)
    StopIteration
    >>> list(filter_link(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    while ___________________________:

        if ________________________:

            ______________________

        ___________________________
```

```
def filter_link(link, f):
    while link is not Link.empty:
        if f(link.first):
            yield link.first
        link = link.rest
```

```
def filter_no_iter(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> list(filter_no_iter(link, lambda x: x % 2 != 0))
    [1, 3]
    """
```

```
    if ____________________________:

        return

    elif ____________________________:

        ____________________________

    ________________________________
```

```
def filter_no_iter(link, f):
    if link is Link.empty:
        return
    elif f(link.first):
        yield link.first
    yield from filter_no_iter(link.rest, f)
```

3.2 Implement `sum_paths_gen`, which takes in a Tree instance `t` and and returns a generator which yields the sum of all the nodes from a path from the root of a tree to a leaf.

You may yield the sums in any order.

```
def sum_paths_gen(t):
    """
    >>> t1 = Tree(5)
    >>> next(sum_paths_gen(t1))
    5
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(9)])
    >>> sorted(sum_paths_gen(t2))
    [6, 7, 10]
    """

    if _____________________________:

        yield ____________________

    for ____________________________:

        for ____________________________:

            yield ____________________
```

```
def sum_paths_gen(t):
    if t.is_leaf():
        yield t.label
    for b in t.branches:
```

```
        for s in sum_paths_gen(b):
            yield s + t.label
```