Type to enter text

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 6: LINKED LISTS AND IMMUTABILITY

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

6.
LINKED LISTS AND IMMUTABILITY

BY FRED ANNEXSTEIN, PHD

CHAPTER 6 OVERVIEW

In this module we will be covering the abstract data types of linked lists. We will examine the implementation, creation, and processing of these objects using a functional programming style using recursion. We illustrate how to create an immutable sequence data-type through the specification of a linked list.

CHAPTER 6 OBJECTIVES

By the end of this module, you will be able to...

- Understand the notion of an abstract data type using the example abstractions of linked lists.
- Understand a functional approach to the implementation of creating and processing linked lists.
- Understand how to use recursion to effectively process linked lists.
- Understand how to design an immutable sequence data type using linked lists.

MUTABILITY

As programmers we often need to know whether an object is mutable - this means that the internal state of the object is changeable. Immutable means that the data type doesn't allow any change in the object's state once it has been created. Objects of built-in type that are mutable include Lists, Set, and Dictionaries. Objects of built-in type that are immutable include Ints, Floats, Strings, and Tuples.

In this chapter we look at designing and implementing a sequence type that is immutable, even though we make use of a mutable underlying data type. We can do this by designing the specification so that no method or interface function designed for the new type modifies any existing object of that type.

LINKED LISTS AND SEQUENCES

Python has many built-in types of sequences: lists, ranges, and strings, to name a few. We design and and implement our own sequence data type we call a linked list. A linked list is a simple type of sequence that is composed of multiple links that are connected.

The key idea for the link abstraction is that each link is a pair where the first element is an item-value in the linked list, and the second element is another link. Links can be also be empty, that is a special value marking the end of the linked list.

CONSTRUCTORS

We design a function to construct a linked link as follows.

```
empty = 'empty'

def link(first, rest=empty):
    return [first, rest]
```

This function constructs a linked list made up of two objects - a first element and a next element. The next element is also a linked list or defaults to the empty value indicating the empty linked list.

SELECTORS

We design two function to select elements from a linked link as follows.

```
def first(s):
    return s[0]

def rest(s):
    return s[1]
```

Linked lists have recursive structure: the rest of a linked list is a linked list or 'empty'. We can define an abstract data representation to validate, construct, and select the components of linked lists.

In the following code we define a linked list called alpha constructed from a pair containing the first element of the sequence (in this case ‘A’) and the rest of the sequence (in this case a representation of a linked list containing three elements: ’B’,’C’,’D’.). The second element of alpha is a linked list.

```
>>> alpha = link(‘A’, link(‘B’,
link(‘C’, link(‘D’))))

>>> first(alpha)
'A'

>>> rest(alpha)
['B', ['C', ['D', 'empty']]]

>>> first(rest(alpha))
'B'
```

LINKED LIST SEQUENCE INTERFACE

The linked list can store a sequence of values in order, but we have not yet shown that it satisfies the sequence abstraction. Using the abstract data representation we have defined, we can implement the two behaviors that characterize a sequence: a length function and an element selection function called getitem which takes an index value as an argument.

```
def len_link(s):
    length = 0
    while s != empty:
       s, length = rest(s), length 1
    return length
```

```
def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

```
>>> len_link(alpha)
4

>>> getitem_link(alpha, 3)
'D'
```

Now, we can manipulate a linked list just as we can with any sequence using these interface functions.

RECURSIVE IMPLEMENTATION

Both len_link and getitem_link are iterative. They move through each layer of nested pair until the end of the list (in len_link) or the desired element (in getitem_link) is reached. We can also implement length and element selection using recursion, as follows.

```
def len_recursive(s):
    if s == empty: return 0
    return 1 + len_recursive(rest(s))

def getitem_recursive(s, i):
    if i == 0: return first(s)
    return getitem_recursive(rest(s),
                             i - 1)
>>> len_recursive(alpha)
4
>>> getitem_recursive(alpha,3)
'D'
```

CHARACTER CODES AND THE CHR AND ORD FUNCTIONS

Python has a builtin ord() function which returns the Unicode code from a given character. This function accepts a string of unit length as an argument and returns the Unicode equivalence of the passed argument. In other words, given a string of length 1, the ord() function returns an integer representing the Unicode code point of the character when an argument is a Unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord(‘A’) returns the integer 65, ord(‘a’) returns the integer 97, ord(‘€’) (Euro sign) returns 8364.

The ord function is the inverse of the chr() functions for 8-bit characters and of unichr() for Unicode objects. Character’s code point must be in the range [0..65535] inclusive. We can print Unicode math symbols since they reside starting at 8704 code point.

```
>>> for i in range(8704,8854):
    print(chr(i), end='  ‘)
∀   ∁   ∂   ∃   ∄   ∅   ∆   ∇   ∈   ∉   ∊   ∋   ∌
∍   ∎   ∏   ∐   ∑   −   ∓   ∔   ∕   ∖   ∗   ∘   ∙
√   ∛   ∜   ∝   ∞   ∟   ∠   ∡   ∢   ∣   ∤   ∥   ∦
∧   ∨   ∩   ∪   ∫   ∬   ∭   ∮   ∯   ∰   ∱   ∲
∳   ∴   ∵   ∶   ∷   ∸   ∹   ∺   ∻   ∼   ∽   ∾   ∿
≀   ≁   ≂   ≃   ≄   ≅   ≆   ≇   ≈   ≉   ≊   ≋   ≌
≍   ≎   ≏   ≐   ≑   ≒   ≓   ≔   ≕   ≖   ≗   ≘   ≙
```

```
≚   ≛   ≜   ≝   ≞   ≟   ≠   ≡   ≢   ≣   ≤   ≥   ≦
≧   ≨   ≩   ≪   ≫   ≬   ≭   ≮   ≯   ≰   ≱   ≲   ≳
≴   ≵   ≶   ≷   ≸   ≹   ≺   ≻   ≼   ≽   ≾   ≿   ⊀
⊁   ⊂   ⊃   ⊄   ⊅   ⊆   ⊇   ⊈   ⊉   ⊊   ⊋   ⊌   ⊍
⊎   ⊏   ⊐   ⊑   ⊒   ⊓   ⊔   ⊕

>>> [chr(i) for i in range(8704,8714)]
['∀', '∁', '∂', '∃', '∄', '∅', '∆',
'∇', '∈', '∉']
```

CREATING LINKED LISTS FROM LISTS

We now demonstrate a function that can construct a linked list from an ordinary list. We use a for loop to iterate through elements of the input list, and at each step we construct a new linked list by append the next element in turn as follows.

```
def list_to_link(lst):
    l_l = empty
    for x in lst:
        l_l = link(x,l_l)
    return l_l

>>> list_to_link(list(range(7)))
[[6, [5, [4, [3, [2, [1, [0,
‘empty’]]]]]]]

>>> list_to_link([chr(i) for i in
range(8704,8714)])
```

```
['∉', ['∈', ['∇', ['∆', ['∅', ['∄',
['∃', ['∂', ['∁', ['∀',
'empty']]]]]]]]]]
```

TRAVERSING A LINKED LIST

Since we now have a sequence interface, many problems that we would like to solve for linked lists become simpler by using the sequence interface. For example, if we want to print out the contents of a linked list we can use the get_item_link and len_link functions as follows:

```
def print_links(lst):
    for i in range(len_link(lst)):
        print(getitem_link(lst,i),
end= ' -> ' )
    print('empty')

>>> x= list_to_link([chr(i) for i in
range(8704,8714)])
>>>  print_links(x)
∉ -> ∈ -> ∇ -> ∆ -> ∅ -> ∄ -> ∃ -> ∂
-> ∁ -> ∀ -> empty
```

INSERTING INTO A LINKED LIST

Now we consider the implementation of an insert function that inserts an item at a specific index in the linked list. If the index is greater than the current length, then our default behavior is insert the item at the end of the list. This function will be much easier to implement using recursion, rather than using iteration!

Note that we are not mutating the list by inserting the item into the original linked list. Instead, we are creating a copy of the original linked list, but with the provided item added at the specified index. The original linked list stays the same, thus preserving the immutability of our data type.

```
def insert(lst, item, index):
    if lst == empty:
        return link(item, empty)
    elif index == 0:
        return link(item, lst)
    else:
        return link(first(lst),
insert(rest(lst), item, index-1))

 >>> lst = link(1, link(2, link(3)))
 >>> new = insert(lst, 9001, 1)
 >>> print_links(new)
 1 -> 9001 -> 2 -> 3 -> empty
```

SUMMING LINKED LIST ELEMENTS

Let us write a function that takes in a linked list `lst` and a function `fn` which is applied to each number in `lst` and returns the sum.

```
# Recursive Solution
def sum_linked_list_r(lst, fn):
    if lst == empty:
        return 0
    return fn(first(lst)) +
sum_linked_list(rest(lst), fn)

# Iterative Solution
def sum_linked_list_i(lst, fn):
    sum = 0
    while lst != empty:
        sum += fn(first(lst))
        lst = rest(lst)
    return sum

>>> alpha = link('A', link('B',
link('C', link('D'))))

>>> sum_linked_list_r(alpha, ord)
266

>>> sum_linked_list_r(alpha,
            lambda x: ord(x)-ord('A')
6
```

CHAPTER 6 EXERCISES

Question 1: Implement the is_sorted(lst) function, which returns True if the linked list lst is sorted in increasing from left to right. If two adjacent elements are equal, the linked list is still considered sorted.

Question 2: Implement a function interleave(s0, s1), which takes two linked lists and produces a new linked list with elements of s0 and s1 interleaved. In other words, the resulting list should have the first element of the s0, the first element of s1, the second element of s0, the second element of s1, and so on.

If the two lists are not the same length, then the leftover elements of the longer list should still appear at the end.

Question 3: A mad scientist has discovered a gene that compels people to enroll in CS2023. You may be afflicted!

A DNA sequence is represented as a linked list of elements `A`, `G`, `C` or `T`. This discovered gene has sequence `C A T C A T`. Write a function `has_2023_gene` that takes a DNA sequence and returns whether it contains the 2023 gene as a sub-sequence.

First, write a function `has_prefix` that takes two linked lists, `s` and `prefix`, and returns whether `s` starts with the elements of `prefix`. Note that `prefix` may be larger than `s`, in which case the function should return `False`.

Question 4: Count Change (with Linked Lists!)

A set of coins makes change for `n` if the sum of the values of the coins is `n`. For example, if you have 1-cent, 2-cent and 4-cent coins, the following sets make change for `7`:

- 7 1-cent coins
- 5 1-cent, 1 2-cent coins
- 3 1-cent, 2 2-cent coins
- 3 1-cent, 1 4-cent coins
- 1 1-cent, 3 2-cent coins
- 1 1-cent, 1 2-cent, 1 4-cent coins

Thus, there are 6 ways to make change for `7`. Write a function `count_change` that takes a positive integer `n` and a linked list of the coin denominations and returns the number of ways to make change for `n` using these coins.

Question 5: Assume there is a function cons(a, b) that constructs a pair, and car(pair) and cdr(pair) returns the first and last element of that pair. For example,

```
>>> car(cons(3, 4))
3
>>> cdr(cons(3, 4))
4
```

Here is an implementation of cons which uses a function as a return value for a pair:

```
def cons(a, b):
    def pair(f):
        return f(a, b)
    return pair
```

Provide an implementation for the functions car and cdr.