INTRODUCING AI-DS-ML

PYTHON PROGRAMMING

CHAPTER 4
LISTS AND SEQUENCES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

4. LISTS AND SEQUENCES

BY FRED ANNEXSTEIN

CHAPTER 4 OVERVIEW

In this chapter we will be covering how data sequences are created using lists and tuples and processed in python. We abstract away the details of these data operations using the sequence interface. We define and implement several functional operations on sequences, and design their code implementations using both iterative and recursive problem solving and programming techniques.

CHAPTER 4 OBJECTIVES

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

- Understand and apply the role, syntax, and abstractions of lists, tuples, and sequences.
- Apply recursive problem solving when processing lists and sequences.
- Apply the concept of a programming interface to programming with sequences.

- Practice the use of generalized for loops, list comprehension, and list slicing.
- Implement the functional operations of map, filter, and frequency to sequences.

SEQUENCES

It is important when processing data to maintain and utilize ordered collections. The term sequence refers to an ordered collection of values. The sequence is a powerful, fundamental abstraction in computer science. Sequences are not instances of a particular built-in type or abstract data representation, but instead a collection of behaviors that are shared among several different types of data. That is, there are many kinds of sequences, but they all share common behavior. In particular, a sequence has a finite length. An empty sequence has length 0. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

LISTS

Python includes several native data types that are sequences, the most important of which is the list. A list is a sequence that can have arbitrary, but finite length.

Lists have a large set of built-in behaviors, along with specific syntax to express those behaviors. We have already seen the list literal, which evaluates to a list instance, as well as an element selection expression that evaluates to a value in the list. The built-in `len` function returns the length of a sequence. Below, digits is a list with four elements. The elements in a sequence can be accessed with 0-based indexing. The element at index 3 of digits is -1.

```
>>> digits = [1, 2, 8, -1]
>>> len(digits)
4
>>> digits[3]
-1
```

LISTS ARE MUTABLE

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> digits[1] = 5
>>> digits
[ 1, 5, 8, -1]
```

The element in position 1 of digits, which used to be 2, is now updated to 5.

FOR AND *IN* CONSTRUCTS

Python's *for* and *in* constructs are extremely useful when working with lists and sequences. The *for* construct -- for var in list -- is an easy way to look at each element in a list (or other iterable collection). Do not add or remove from the list during iteration.

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
    sum += num
print(sum)  ## 30
```

The *in* construct is used as an easy way to test if an element appears in a list (or other iterable collection) by returning True/False.

```
list = ['larry', 'curly', 'moe']
if 'curly' in list:
      print(‘NaNaNaNa’)
```

The for/in constructs are very commonly used in Python code and so you should just memorize their syntax. You can also use for/in to work on a string. The string acts like a list of its chars, so

```
# prints all the chars in a string.
for ch in s:
    print(ch)
```

LIST SLICES

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

```
>>> list = ['a', 'b', 'c', 'd']
>>> list[1:-1]
```

```
['b', 'c']
>>> list[0:2] = ‘z'
>>> list
['z', 'c', ‘d']
```

APPENDING TO A LIST WITH +=

We will start with an empty list, and then use a for statement and += operator to append the values 1 to 10 —note the list grows dynamically to accommodate each item.

```
a_list=[]
for i in range(1,11):
    a_list += [10*i]

>>> a_list
[10, 20, 30, 40, 50, 60, 70, 80, 90,
100]
```

TRAVERSING A LIST

The most common way to traverse the elements of a list is with a for loop. Here will will use a for loop over a range value to access list indices and list values.

```
a_list=[]
for i in range(1,11):
    a_list += [10*i]

for i in range(len(a_list)):
    print(i, a_list[i])
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n−1, where n is the length of the list. Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value. A for loop over an empty list never runs the body.

```
for x in []:
    print('This never runs.’)
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol
le Veq'], [1, 2, 3]]
```

TUPLES

Unlike lists, tuples are an immutable data type. However, similar to lists, tuples can store and access heterogeneous data. A tuple's length is its number of elements and cannot change during program execution.

To create an empty tuple, use empty parentheses: Recall that you can pack a tuple by separating its values with commas: When you output a tuple, Python always displays its contents in parentheses. You may surround a tuple's comma-separated list of values with optional parentheses:

ACCESSING TUPLE ELEMENTS

A tuple's elements, though related, are often of multiple types. Usually, you do not iterate over them. Rather, you access each individually. Like list indices, tuple indices

start at 0. The following code creates time_tuple representing an hour, minute and second, displays the tuple, then uses its elements to calculate the number of seconds since midnight—note that we perform a different operation with each value in the tuple:

```
>>> student_tuple = ()
>>> student_tuple
()
>>> len(student_tuple)
0
>>> student_tuple = 'John', 'Green',
3.3
>>> student_tuple
('John', 'Green', 3.3)
>>> len(student_tuple)
3
```

LIST METHODS- APPEND AND EXTEND

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
```

```
['a', 'b', 'c', 'd']
```
`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```
This example leaves `t2` unmodified.

SORT METHOD

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are not pure; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

MAP, FILTER, AND REDUCE

To add up all the numbers in a list t, you can use a loop like this:

```
def add_all(t):
  total = 0
  for x in t:
    total += x
return total
```

Note that total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable.

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an accumulator. Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is called a reduce function or a reduction.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings.

```
def capitalize_all(t):
  result = []
  for s in t:
    result.append(s.capitalize())
  return result
```

The variable result is initialized with an empty list; each time through the loop, we append the next element. So result is another kind of accumulator. An operation like capitalize_all is sometimes called a map because it “maps” a function (in this case the method capitalize) onto each of the elements in a sequence. Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings.

```
def only_upper(t):
  result = []
```

```
  for s in t:
        if s.isupper():
            result.append(s)
  return res
```

The function isupper is a builtin string method that returns True if the string contains only upper case letters.

An operation like only_upper is called a filter because it selects some of the elements and filters out the others. Most common list operations can be expressed as a combination of map, filter and reduce.

VARIABLE LENGTH ARGUMENT LISTS

We define an average function that can receive any number of arguments. The parameter name args is used by convention, but you may use any identifier. If the function has multiple parameters, the *args parameter must be the rightmost parameter. Now, let us call average several times with arbitrary argument lists of different lengths: To calculate the average, divide the sum of the args tuple's elements (returned by built-in function sum) by the tuple's number of elements (returned by built-in function len).

```
def average(*args):
  return sum(args) / len(args)
```

Note in our average function definition that if the length of args is 0, a ZeroDivisionError occurs.

The parameter name args is used by convention, but you may use any identifier. If the function has multiple parameters, the *args parameter must be the rightmost parameter. Now, let's call average several times with arbitrary argument lists of different lengths:

```
>>> average(5, 10)
7.5
>>> average(5, 10, 15)
10.0
>>> average(5, 10, 15, 20)
12.5
```

THE * OPERATOR

The * is a special operator in python. When the * operator is applied to an iterable argument in a function call, the effect is to unpack its elements. The following code

creates a five-element grades list, then uses the expression *grades to unpack its elements as average's arguments:

```
>>> grades = [88, 75, 96, 55, 83]
>>> average(*grades)
79.4
```

The call shown above is equivalent to average(88, 75, 96, 55, 83).

LIST COMPREHENSIONS

Many sequence processing operations can be expressed by evaluating a fixed expression for each element in a sequence and collecting the resulting values in a result sequence. In Python, a list comprehension is an expression that performs such a computation.

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x*10 for x in odds]
[10, 30, 50, 70, 90]
```

The `for` keyword above is not part of a `for` statement, but instead part of a list comprehension because it is contained within square brackets. The sub-expression `x+1` is evaluated with `x` bound to

each element of `odds` in turn, and the resulting values are collected into a list.

Another common sequence processing operation is to select a subset of values that satisfy some condition. List comprehensions can also express this pattern, for instance selecting all elements of `odds` that evenly divide `25`.

```
>>> [x for x in odds if 25 % x == 0]
[1, 5]
```

EXERCISES

Exercise 1. Write a function called nested_sum that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercise 2. Write a function called cumu that takes a list of numbers and returns the cumulative sum; that is, a new list where the ith element is the sum of the

first i+1 elements from the original list. For example:

```
>>> t = [1, 2, 3]
>>> cumu(t)
[1, 3, 6]
```

Exercise 3. Write a function called middle that takes a list and returns a new list that contains all but the first and last elements. For example:

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercise 4. Write a function called chop that takes a list, modifies it by removing the first and last elements, and returns None. For example:

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 5. Write a function called is_sorted that takes a list as a parameter and returns True if the list is

sorted in ascending order
and False otherwise. For example:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 6. Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called is_anagram that takes two strings and returns True if they are anagrams.

Exercise 7. Write a function called has_duplicates that takes a list and returns True if there is any element that appears more than once. It should not modify the original list.

Exercise 8. This exercise pertains to the so-called Birthday Paradox, which you can read about at http://en.wikipedia.org/wiki/Birthday_paradox.

If there are 23 students in your class, what are the chances that two of them have the same birthday? You can estimate this

probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the randint function in the random module.

PROGRAMMING CHALLENGE #2

Before attempting this challenge you should have some background knowledge of what the outcome of Newton's method does. We will implement the method here.

First consider the idea of an iterative numerical method. For example, here is an iterative methods to calculate pi. Let

$P(1) = 3.0$, and for $n>0$

$P(n + 1) = P(n) + \sin(P(n))$

...where P(n) is the approximation of pi at iteration n. Given a first estimate P(1) of pi as 3.0, this iteration converges to an approximation of pi correct to as many digits of precision that you are given when you call the sin function.

P(1) = 3.0

P(2) = P(1)+ sin(3.0) = 3.1411200...

P(3) = 3.14112 + sin(3.14112) = 3.141592654…

And so on.

After the second iteration, the error is very small (about 1.8*10^(-11)). And, on the next iteration, the error drops to about 10^(-32). Why does it this work so darn well?

This algorithm is a particular case of a method called fixed point iteration, and is used to find solutions to equations of the form:

x = f(x) [1]

In this particular case, we have f(x) = x + sin(x), and, as sin(n*pi) = 0 for any integer n, any multiple of pi is a solution of that equation.

An equation like [1] can sometimes (often?) be solved by iterating the formula:

x[n+1] = f(x[n])

where x[n] is the nth approximation of the solution. There is a famous folk theorem which states that the method will converge to the solution if the absolute value of the derivative f'(x) is less than some number L < 1 in some interval containing the solution, and if you start the iteration with a value in that interval. The smaller the derivative is, the faster the sequence will converge. In the case of f(x) = x + sin(x), we have: f'(x) = 1 + cos(x) and, if x is close to pi, cos(x) is close to -1, and f'(x) is close to 0, which explains why the method converges rapidly. We can also notice that f"(x), the second derivative, is -sin(x), which is 0 at the root. This means that you can start in a relatively large interval about pi, and still get a very fast convergence.

We can see this as a slight modification of Newton's method applied to the solution of the equation sin(x) = 0. Newton's method gives the formula: x[n+1] = x[n] - sin(x[n])/ cos(x[n] and, as cos(x[n]) is close to -1, you get your formula by replacing cos(x[n]) by -1 in the above formula.

Programming Challenge #2 Deliverables:

0. Source code that will run two functions fixed_point_iteration() and newton's method newton_find_zero(). See Sample Runs below.

1. Determine if Newton's method or the fixed point method is better for computing pi (up to 15 digits of accuracy) by counting how many iterations each needs to converge.

2. Determine if Newton's method or the fixed point method is better for computing the Dottie number (0.739085...), which is the unique, real number fixed point of the cos() function.

3. Include comments that discuss your successes and analysis of this problem.

You should write a function with an f-arg to compute an iterative fixed point iteration as follows.

```
def fixed_point_iteration(f, x=1.0):
    step = 0
    while not approx_fixed_point(f, x):
        x = f(x)
        step += 1
    return x, step

“””
```

```
“””
def approx_fixed_point(f, x) should
return True if and only if f(x) is
very close (distance < 1e-15) to x.
“””
```

Here is a set of sample runs which you should convert to doctests:

```
>>> from math import sin,cos
>>> fixed_point_iteration(lambda x:
sin(x) + x, 3.0)
(3.141592653589793, 3)

>>> newton_find_zero(lambda x:
sin(x) , lambda x: cos(x), 3.0)
(3.141592653589793, 3)

>>> print(fixed_point_iteration(lambda
x: cos(x), 1.0)
(0.7390851332151611, 86)
# Fixed point defining dottie number
>>> newton_find_zero(lambda x: cos(x)
- x , lambda x: -sin(x)-1, 1.0)
(0.7390851332151606, 7)
# Newtons's zero giving dottie
```