Comp 3005: Data Science Bridge Course: Computer Science Programming Basics

Class 5: Lists

Dalton Crutchfield
University of Denver

Data Structures Introduction

Data Structures

- All the variables we have seen so far have been names for single values.

- Oftentimes, we need to store more than one value, and naming each individual value is impractical.

- **Data Structures** allow us to conveniently manipulate large amounts of data.

Data Structures

- We will look at four types of data structures in Python:

1. **List**: ordered, changeable, and allows duplicate members

2. **Tuple**: ordered, unchangeable, and allows duplicate members

3. **Set**: unordered, changeable, no duplicates, and not indexed

4. **Dictionary**: unordered, changeable, and indexed

Lists
Indexing, Slicing, and Looping

Escape Sequences Examples

- Lists are **sequences** of ordered values of any type.
- You specify a list with comma-separated values surrounded by square braces.
- Syntax:

```
# a list of integers
nums = [3, 1, 4, 1, 5, 9, 2, 6]
# a list of strings
words = ['this', 'is', 'a', 'list', 'of', 'words']
# a list of mixed types, including another list!
stuff = [1, 2, 'hello', ['a', 'smaller', 'list']]
```

Lists

- A list can contain a mixture of types.
- A list that contains another list is called **nested**.
- We can use **indexing** to access individual values in the list.
- We can use functions defined in Python to manipulate lists.
- We can use loops to iterate over values of a list.

Indexing

- Every value in the list has an index.

- Since the values are ordered, the indices are integer values that indicate order of value i the list.

- The first element is indexed at 0, the second at 1, and so on.

- The last element can be indexed with -1, the second-to-last with -2, and so on.

- We index an element with square brackets.

Indexing Example

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
print(nums[2]) # prints 4
nums[0] = 7 # change first value
Print(nums[-1]) # prints 6
```

Slicing

- Slicing allows us to get a sublist from a list:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
```

- We can get a sublist from starting index (inclusive) to ending index (exclusive):

```
nums[1:4] #returns [1, 4, 1]
```

- If the end is not specified, then it slices all elements starting at index specified:

```
nums[3:] #returns [1, 5, 9, 2, 6]
```

- If the start is not specified, then it slices all elements up to index specified (exclusive):

```
nums[:4] #returns [3, 1, 4, 1]
```

Slicing

- Slicing allows us to get a sublist from a list:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
```

- We can skip values by specifying a step value list[start:stop:step]:

```
nums[0:7:3] #returns [3, 1, 2] skipping every third value
```

- We can slice the list backward:

```
nums[::-1] #returns [6, 2, 9, 5, 1, 4, 1, 3]
```

List Manipulation

List Manipulation

- There are Python functions that enable you to manipulate a list.
- Consider the list:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
```

- **len(nums)** returns the length of the list, which is 8 in this case.
- **sorted(nums)** returns a new list that is sorted in ascending order.
- **nums.index(5)** returns the index of the first value 5.
- **nums.index(1, 2)** returns the index of the first value 1 and starts the search from index 2.
    - Note: Both functions above will give an error if the value is not in the list.
- **nums.count(5)** returns the count of value 5 in the list.

List Manipulation

- **nums.append(10)** adds 10 to the end of the list nums.
- **nums.insert(12, 3)** inserts 12 at index 3 in nums.
- **nums.remove(9)** removes the first value 9 in the list.
- **nums.pop()** removes and returns the last value in the list (stack).
- **nums.sort()** will sort the list nums (modifies nums).
- **sorted(nums, reverse = True)** returns a new list that is sorted in descending order.
- **nums.sort(reverse = True)** will sort the list nums in descending order.
- **nums.extend([7, 2, 6])** appends the elements of the list [7, 2, 6] to nums.

List
Iteration

Iteration

- Iteration means going through a sequence in a specific order and performing a task on each of the values.
- Naturally, loops are used to iterate through lists.
- A while-loop can be used to iterate through the elements of a list by using a variable representing the index:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
i = 0
while index < len(nums):
    print(nums[i])
    i += 1
```

Iteration

- The for-loops are even more natural for iteration over a list since range and slicing can be used to determine the sequence:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
for i in range(len(nums)):
    print(nums[i])
```

- We can even use the list itself as the sequence we iterate over:

```
for value in nums:
    print(value)
```

Iteration

- And using slicing to obtain the sequence we can iterate over sublists:

```
#print values in indices 2 – 6 (exclusive)
for value in nums[2:6]:
    print(value)

#print list in reverse order
for value in nums[::-1]:
    print(value)
```

List
List Comprehension

List Comprehension

- Creating a list by specifying values by hand like we have been doing is not practical for very large lists.
- We need a procedural method for generating lists.
- Using range and mutability of lists, we can generate lists.
- The example below generates the first 10 squares:

```
squares = []
for i in range(1, 11):
    squares += [i ** 2]
```

- List comprehension allows us to write this same code with much shorter code:

```
squares = [i ** 2 for i in range(1,11)]
```

List Comprehension

- List comprehension in general:

```
List = [EXPRESSION for VAR in COLLECTION]
```

- The expression is applied to every variable VAR in the collection
- We can also restrict the values to only those that satisfy a condition:

```
List = [EXPRESSION for VAR in COLLECTION if CONDITION]
```

- Example:

```
#Generate a list of all squares between 0 and 100 but only  if they are
divisible by 4
squares = [i ** 2 for i in range(100) if i%4 == 0]
```

List
Mutability and Aliasing

List Mutability

- Lists are mutable, which means we can modify their values.
- We can use the square brackets to change the value of an individual element:

```
nums = [1, 2, 3]
nums[0] = 3
print(nums) # prints [3, 2, 3]
```

- Elements can be added to the list using augmented assignment:

```
nums = [1]
nums += [2, 3]
print(nums) ## prints [1, 2, 3]
```

Mutability

- An element at an index can be removed using the del operator:

```
nums = [1,2, 3, 4, 5]
del nums[2] #remove element at index 2
print(nums) # prints [1, 2, 4, 5]
del nums[1:3] #remove slice 1:3
print(nums) #prints [1, 5]
```

Function Call and Aliasing

- An alias is created when a mutable value is passed to a function as well:

```
def foo(a):
    a += [5] # a here is an alias to nums

nums = [1, 2, 3]
foo(nums) # pass nums to foo
print(nums) # prints [1, 2, 3, 5]
```

Reassignment

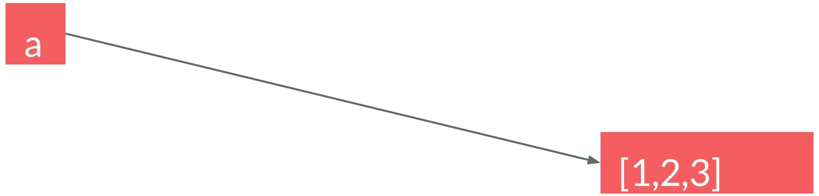
- What happens when a name is assigned a different value?

```
a = [1, 2, 3]
b = a
a = [4, 5, 6]
print(a)
print(b)
```

Reassignment

- What happens when a name is assigned a different value?

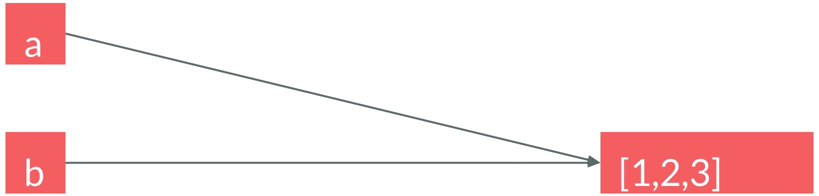
```
a = [1, 2, 3]
b = a
a = [4, 5, 6]
print(a)
print(b)
```



Reassignment

- What happens when a name is assigned a different value?

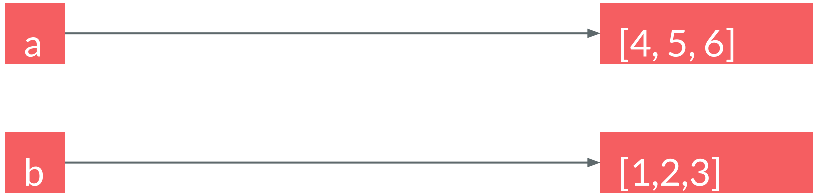
```
a = [1, 2, 3]
b = a
a = [4, 5, 6]
print(a)
print(b)
```



Reassignment

- What happens when a name is assigned a different value?

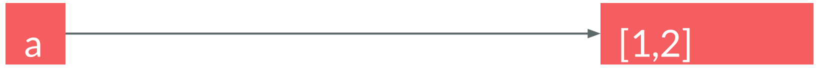
```
a = [1, 2, 3]
b = a
a = [4, 5, 6]
print(a)
print(b)
```



Nested Loops

- Since lists can contain other mutable values, like lists, aliases can cause unexpected behavior as well:

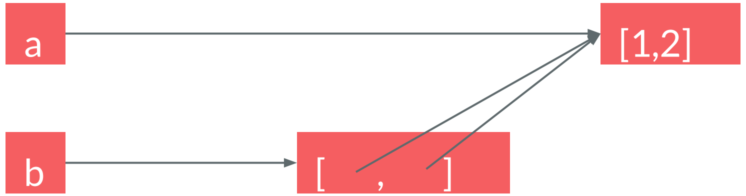
```
a = [1, 2]
b = [a, a] # therefore b is [[1, 2], [1,2]]
a += [3] # append 3 to list a
print(b) # prints [[1, 2, 3], [1, 2, 3]]
```



Nested Loops

- Since lists can contain other mutable values, like lists, aliases can cause unexpected behavior as well:

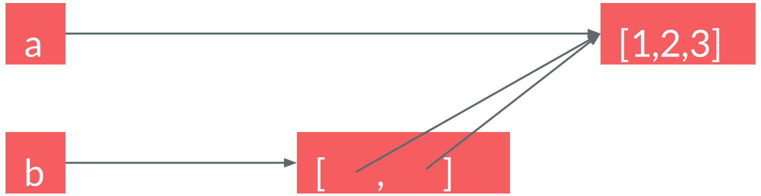
```
a = [1, 2]
b = [a, a] # therefore b is [[1, 2], [1,2]]
a += [3] # append 3 to list a
print(b) # prints [[1, 2, 3], [1, 2, 3]]
```



Nested Loops

- Since lists can contain other mutable values, like lists, aliases can cause unexpected behavior as well:

```
a = [1, 2]
b = [a, a] # therefore b is [[1, 2], [1,2]]
a += [3] # append 3 to list a
print(b) # prints [[1, 2, 3], [1, 2, 3]]
```



Today in Tech History

September 28, 1925

Supercomputer Pioneer Seymour Cray Born

Seymour Cray is born. Cray began his engineering career building cryptographic machinery for the US government and went on to co-found Control Data Corporation (CDC) in the late 1950s. For over three decades, first with CDC then with his own companies, Cray consistently built the fastest computers in the world, leading the industry with innovative architectures and packaging and allowing the solution of hundreds of difficult scientific, engineering, and military problems. Many of Cray's supercomputers are on exhibit at the Computer History Museum.