CptS 111, Spring 2023
Lect. #22, Apr. 12, 2023
Class Notes

Today's Agenda:

1. Demo of PA 7
2. A few more list methods (plus a function)
3. More on simultaneous assignment
4. List slicing
5. Loops modifying lists

Ch. 9

Lists and Dictionaries

2. (More) List Methods (and Another List Function)

Earlier we learned the most useful list functions and some of the more useful list methods. Today we'll cover the following:

- `.insert()`
- `.extend()`
- `.reverse()`
- `.sort()` with more functionality
- `sorted()`

In [1]:

```python
# .insert(): for (index, a), insert a before index

twos = [2, 22, 222]
twos.insert(0,0.2)  # .insert(index, value)
twos
```

Out[1]:

```
[0.2, 2, 22, 222]
```

If we want to add an element to a list, we use the `.append()` method, but suppose we want to add a collection of items to a list. We can do so if the collection is an iterable, e.g., a string, a tuple, a set, or another list using the `.extend()` method.

```
In [2]: # Use .extend() to add values of a tuple to a list

        tuple_of_twos = (2222, 22222)
        twos.extend(tuple_of_twos)  # Add elements of an iterable
        twos
```

```
Out[2]: [0.2, 2, 22, 222, 2222, 22222]
```

```
In [3]: # Use .extend() to add string characters to a list
        # I wonder if there's some practical application for this!

        twos.extend('twos') # add elements of iterable to list
        twos
```

```
Out[3]: [0.2, 2, 22, 222, 2222, 22222, 't', 'w', 'o', 's']
```

We've discussed the `.sort()` method previously, but I wanted to add that this method and others modify lists in place; this is known as in-place modification.

```
In [4]: # .sort(): sort list from smallest to largest

        primes = [19, 7, 23, 11, 13, 5, 17]
        primes.sort()
        primes
```

```
Out[4]: [5, 7, 11, 13, 17, 19, 23]
```

```
In [5]: # .reverse(): reverses element order

        primes.reverse()
        primes
```

```
Out[5]: [23, 19, 17, 13, 11, 7, 5]
```

We can actually combine these two examples using a kwag!

```
In [6]: # Sort list and reverse the elements by changing the value of the kwarg
        # 'reverse' to True

        primes = [19, 7, 23, 11, 13, 5, 17]
        primes.sort(reverse=True)
        primes
```

```
Out[6]: [23, 19, 17, 13, 11, 7, 5]
```

In addition to the **void method** `.sort()` , Python has the built-in **non-void function** `sorted()` for lists. Both the ***void*** `.sort()` method and the ***non-void*** `sorted()` function perform the same operation. Python has both because sorting is so useful. The difference between the two is that `.sort()` sorts a list in place, i.e., the original list is permanently changed; with `sorted()` , the original list is retained, and the sorted list is assigned to an lvalue.

You can use the keyword `key` with either of these if you want to sort the list in a particular

In [7]:

```
# Void method .sort() sorts list in place

animals = ['monkey', 'Ostrich', 'Zebra', 'alligator', 'cow']
animals.sort()
print(animals)
```

```
['Ostrich', 'Zebra', 'alligator', 'cow', 'monkey']
```

In [8]:

```
# We can use key=str.lower so case doesn't matter

animals.sort(key=str.lower)
print(animals)
```

```
['alligator', 'cow', 'monkey', 'Ostrich', 'Zebra']
```

In [9]:

```
# Non-void function sorted() returns sorted list; can use key=str.lower
# here as well

animals = ['monkey', 'Ostrich', 'Zebra', 'alligator', 'cow']
sorted_animals = sorted(animals, key=str.lower)
print('Sorted list:', sorted_animals)
print('Original list:', animals)
```

```
Sorted list: ['alligator', 'cow', 'monkey', 'Ostrich', 'Zebra']
Original list: ['monkey', 'Ostrich', 'Zebra', 'alligator', 'cow']
```

In [10]:

```
# reverse=True works for sorted() function as well

primes = [19, 7, 23, 11, 13, 5, 17]
rev_primes = sorted(primes, reverse=True)
print('Reversed sorted list:', rev_primes)
print('Original list:', primes)
```

```
Reversed sorted list: [23, 19, 17, 13, 11, 7, 5]
Original list: [19, 7, 23, 11, 13, 5, 17]
```

3. Simultaneous Assignment Redux

We've used simultaneous assignment many times before, but now that we've learned loops, I want to show you a few other ways to use it.

In [11]:

```python
# Use simultaneous assignment with nested lists
# author is first element in list, novels is second element in list

author, novels = ['J. Austen', ['Pride and Prejudice', 'Persuasion',
                'Sense and Sensibility','Emma', 'Northanger Abbey',
                'Mansfield Park']]
print(f'{author}')
for novel in novels:
    print(f'  - {novel}')
```

```
J. Austen
  - Pride and Prejudice
  - Persuasion
  - Sense and Sensibility
  - Emma
  - Northanger Abbey
  - Mansfield Park
```

The basic rule for simultaneous assignment is that the number of lvalues to the left of the assignment operator must equal the number of elements to the right.

We can use simultaneous assignment with lists of lists and nested loops:

In [14]:

```python
# Use simultaneous assignment with lists of lists and nested loops
# cars is nested list of length 3; each element in cars is a list
# which has a string element and a list element

cars = [
    ['Toyota', ['RAV4', 'Prius', 'Camry']],
    ['Ford', ['Explorer', 'F-150', 'Mustang']],
    ['Tesla', ['Model S', 'Model X', 'Model Y']]
       ]
# Length of cars:
print('Length of cars:', len(cars))
print('Length of cars[0]', len(cars[0]))

# car is list of length 2; make is first element and models is second
# models is also a list

for car in cars:         # car is a list of length 2
    make, models = car   # models is a list as well
    print(f'{make}:')
    for model in models:
        print(f'   - {model}')
```

```
Length of cars: 3
Length of cars[0] 2
Toyota:
   - RAV4
   - Prius
   - Camry
Ford:
   - Explorer
   - F-150
   - Mustang
Tesla:
   - Model S
   - Model X
   - Model Y
```

In [15]:

```
# Use simultaneous assignment in iterating for-loop header!
# shoes is nested list (list of lists)

shoes = [
    ['Christian Louboutin', 5995],
    ['Jimmy Choo', 950],
    ['Stuart Weitzman', 598],
    ['Miu Miu', 1200],
    ['Manolo Blahnik', 1795],
    ['Gucci', 950],
    ['Alexander McQueen', 690]
    ]
for designer, price in shoes:
    print(f'{designer}: ${price:,}') # Note use of comma
```

```
Christian Louboutin: $5,995
Jimmy Choo: $950
Stuart Weitzman: $598
Miu Miu: $1,200
Manolo Blahnik: $1,795
Gucci: $950
Alexander McQueen: $690
```

4. List Slicing

List slicing is analogous to string slicing. We can create a list from another list using list slicing.

Template for list slicing:

```
<list>[start : end : stride]
```

where:

```
start:  slice begins at start index
end:    slice ends one before end index (stop)
stride: default of 1, but other values can be used (increment)
```

Notes:

1. `[ : end]` will start at 0 and end one before end
2. `[start : ]` will start at start and include rest of list
3. `[ : : stride]` will start at 0, add stride to 0 and each successive index value, and end at end
4. `[ : ]` will create a deep copy of list

In [16]:

```
# Example 1:

letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters[ : 6]
```

Out[16]:

```
['a', 'b', 'c', 'd', 'e', 'f']
```

In [17]:

```
# Example 2:

letters[2 : ]
```

Out[17]:

```
['c', 'd', 'e', 'f', 'g']
```

In [18]:

```
# Example 3:

letters[ : : 2]
```

Out[18]:

```
['a', 'c', 'e', 'g']
```

Next, recall the following.

In [19]:

```
# We used this example in an earlier lecture

def negate(num_list):
    for i in range(len(num_list)):
        num_list[i] = -num_list[i]
    return num_list

nums1 = [1, 2, 3]
nums2 = nums1
neg_nums = negate(nums2)
print('Original list:', nums1)
print('Negated list returned by function:', neg_nums)
```

```
Original list: [-1, -2, -3]
Negated list returned by function: [-1, -2, -3]
```

We didn't want `nums1` to change which is why we made a copy of it, `nums2` , which we passed to the function. However, we didn't really create a copy; instead, we created an alias pointing to the same memory location. Thus, when `nums2` was changed, so was `nums1` .

We can use list slicing to create a deep copy of a list. Let's see what happens when we do.

In [20]:

```
# Example 4: Use a deep copy of nums to use as the argument

def negate(num_list):
    for i in range(len(num_list)):
        num_list[i] = -num_list[i]
    return num_list

nums1 = [1, 2, 3]
nums2 = nums1[ : ]          # This is only difference in code
neg_nums = negate(nums2)
print('Original list:', nums1)
print('Negated list returned by function:', neg_nums)
```

```
Original list: [1, 2, 3]
Negated list returned by function: [-1, -2, -3]
```

5. Modifying Lists in a Loop

Previously we learned that if you want to modify the values in a list within a loop, you must use a counting `for` -loop (see Jupyter lecture for 3.8.23), i.e., you can't use an iterating `for` -loop. Let's now consider a more complex example.

In [21]:

```
# Try to remove names from names1 that are in names2

names1 = ['Emi', 'Sun', 'Ann', 'Ali']
names2 = ['Ann', 'Ali', 'Sam', 'Tom']
for name in names1:
    if name in names2:
        names1.remove(name)
print('names1:', names1)
print('names2:', names2)
```

```
names1: ['Emi', 'Sun', 'Ali']
names2: ['Ann', 'Ali', 'Sam', 'Tom']
```

`Ann` was removed from `names1` , so why wasn't `Ali` ? To see why, consider the following:

In [22]:

```
# Same as previous example, but print loop variable

names1 = ['Emi', 'Sun', 'Ann', 'Ali']
names2 = ['Ann', 'Ali', 'Sam', 'Tom']
for name in names1:
    print('name in names1:', name)
    if name in names2:
        names1.remove(name)
print('names1:', names1)
print('names2:', names2)
```

```
name in names1: Emi
name in names1: Sun
name in names1: Ann
names1: ['Emi', 'Sun', 'Ali']
names2: ['Ann', 'Ali', 'Sam', 'Tom']
```

We see that iteration of the list stopped before the name `Ali` was reached. Because lists are mutable, `names1` is changed as the iterable in the `for` -loop! The Python interpreter sees that it has already looked at the first three values in `names1` so when the length of `names1` has been reduced to three values, it ends the loop. To circumvent this problem, we create a deep copy of `names1` .

In [23]:

```python
# Remove names from name1 that are in names2 by using deep copy of names1
# as the iterable in the for-loop.

names1 = ['Emi', 'Sun', 'Ann', 'Ali']
names2 = ['Ann', 'Ali', 'Sam', 'Tom']
for name in names1[ : ]:                # Use deep copy of names1; only di
    print('name in names1:', name)
    if name in names2:
        names1.remove(name)
print('names1:', names1)
print('names2:', names2)
```

```
name in names1: Emi
name in names1: Sun
name in names1: Ann
name in names1: Ali
names1: ['Emi', 'Sun']
names2: ['Ann', 'Ali', 'Sam', 'Tom']
```