HSS 611 - Week 3: Lists, Tuples, and More

2023-09-11

Agenda

- Lists
- Tuples
- String methods
- Mutability
- Aliasing and cloning
- List comprehensions

Lists

- Lists are one of four major built-in data types to store collections of data
- The others are tuples, dictionaries, and sets
- Used to store “things” in a single variable

```
my_list = ['apple', 'orange', 'banana', 'cherry']
```

```
type(my_list)
```

```
list
```

Lists

- Items in a list don't need to be of the same type

```python
miscellaneous_list = ['a', 3, False, None, [3.2, ' ']]
print(miscellaneous_list)
```

```
['a', 3, False, None, [3.2, ' ']]
```

Lists

- Can initialize an empty list with just two squared brackets

```
empty_list = []
print(empty_list)
```

```
[]
```

Lists are ordered

- Lists are ordered
- So you can access an item in a specific index
- This is called indexing/slicing

```
print(my_list)
my_list[0]
my_list[2]
```

```
['apple', 'orange', 'banana', 'cherry']
```

```
'banana'
```

Lists are ordered

- Sets are not ordred/indexed (this can be good!)

```
my_set = set(my_list)
print(my_set)
```

```
{'cherry', 'apple', 'orange', 'banana'}
```

Lists are ordered

- The index can be negative too

```
print(my_list)
print(my_list[-1])
print(my_list[-4])
```

```
['apple', 'orange', 'banana', 'cherry']
cherry
apple
```

Lists are ordered

- An item can occur more than once (unlike sets)

```
fruit = ['apple', 'orange', 'apple']
print(fruit)
fruit_set = set(fruit)
print(fruit_set)
```

```
['apple', 'orange', 'apple']
{'apple', 'orange'}
```

Lists are mutable

- Lists are **mutable** objects
- That means they can be modified
- For example

```
print(fruit)
fruit[2] = 'cherry'
print(fruit)
```

```
['apple', 'orange', 'apple']
['apple', 'orange', 'cherry']
```

Length of a list

- The `len()` function returns the number of items in (or **length** of) a list

```
print(len(fruit))
print(len('fruit!')) # string
```

```
3
6
```

Methods

- In Python, “methods” are functions that belong to an object
- They only work with that object
- They are used with dot notation: `object.method()`

List Methods

- There are many useful methods that come with list objects
- We'll be able to look at some of them

append()

- append() adds an object to the end of the list

```
cars = ['Ford', 'BMW', 'Hyundai']
cars.append('Mazda')
print(cars)
```

```
['Ford', 'BMW', 'Hyundai', 'Mazda']
```

- Note that no re-assignment is necessary
- Once append() is run, the list is modified in memory
- Often used inside for loops

append()

- Often used inside for loops

```
empty_list = []
for i in range(0, 10):
  empty_list.append(i)
print(empty_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

insert()

- insert() adds an element at the specified position

```
cars.insert(1, 'Toyota')
print(cars)
```

```
['Ford', 'Toyota', 'BMW', 'Hyundai', 'Mazda']
```

reverse()

- `reverse()` reverses the order of the list
- Again, no need to reassign

```
print(cars)
```

```
['Ford', 'Toyota', 'BMW', 'Hyundai', 'Mazda']
```

```
cars.reverse()
print(cars)
```

```
['Mazda', 'Hyundai', 'BMW', 'Toyota', 'Ford']
```

sort()

- sort() sorts the list (no assignemnt necessary too)

```
print(cars)
```

```
['Mazda', 'Hyundai', 'BMW', 'Toyota', 'Ford']
```

```
cars.sort()
print(cars)
```

```
['BMW', 'Ford', 'Hyundai', 'Mazda', 'Toyota']
```

sort()

- Elements are not always comparable to each other, but see the following

```
num_bool = [1.2, 5, 2.3, False, True]
num_bool.sort()
print(num_bool)
```

```
[False, True, 1.2, 2.3, 5]
```

sorted() function

- `sorted()` is a function but **not** a method
- Returns a sorted version of the list, but original stays intact

```
cars = ['Mazda', 'BMW', 'Toyota', 'Ford']
print(sorted(cars))
```

```
['BMW', 'Ford', 'Mazda', 'Toyota']
```

```
print(cars)
```

```
['Mazda', 'BMW', 'Toyota', 'Ford']
```

index()

- `index()` returns the index of the **first** element matching the specified value

```
print(cars)
cars.index('Toyota')
```

```
['Mazda', 'BMW', 'Toyota', 'Ford']
```

```
2
```

`extend()`

- `extend()` adds the elements of another list (or any iterable), to the end of the current list

```python
other_cars = ['Audi', 'Kia']
cars.extend(other_cars)
print(cars)

my_list = [1, 2, 3]
my_set = {4, 5, 6}
my_list.extend(my_set)
print(my_list)
```

```
['Mazda', 'BMW', 'Toyota', 'Ford', 'Audi', 'Kia']
[1, 2, 3, 4, 5, 6]
```

extend()

- What will it return?

```
empty_list = []
for i in range(0, 10):
  empty_list.extend(i)
```

extend()

- How about this?

```
empty_list = []
for i in ['a', 'b', 'c']:
  empty_list.extend(i)
print(empty_list)
```

extend()

- How about this?

```
empty_list = []
for i in ['a', 'b', 'c']:
  empty_list.extend(i)
print(empty_list)
```

```
['a', 'b', 'c']
```

append() vs. extend()

```
my_list = [1, 2, 3]
my_list.append(4)
my_list.append([5, 6])
my_list.append("89")
print(my_list)

my_list = [1, 2, 3]
my_list.extend([4, 5])
my_list.extend((6, 7))
my_list.extend("89")
print(my_list)
```

```
[1, 2, 3, 4, [5, 6], '89']
[1, 2, 3, 4, 5, 6, 7, '8', '9']
```

+

- To add lists together without modifying original lists, just +

```python
L1 = ['Mazda', 'BMW']
L2 = ['Ford', 'Nissan']
L = L1 + L2
print(L)
```

```
['Mazda', 'BMW', 'Ford', 'Nissan']
```

Removing Items

- `del()`: use a specific index
- Can also be used to remove the entire object

```
print(L)
del(L[2])
print(L)
```

```
['Mazda', 'BMW', 'Ford', 'Nissan']
['Mazda', 'BMW', 'Nissan']
```

- Return and then remove the last element

```
print(L)
L.pop()
print(L)
```

```
['Mazda', 'BMW', 'Nissan']
['Mazda', 'BMW']
```

Removing Items

- Remove a specific element

```
L.remove('BMW')
print(L)
```

```
['Mazda']
```

- It removes the first item that matches the specified value

```
new_L = [1, 3, 7, 5, 6, 7]
new_L.remove(7)
print(new_L)
```

```
[1, 3, 5, 6, 7]
```

Indexing & Slicing

Lists can be sliced with the following syntax (similar to `range()`):

`[start:stop:step]`

- **start** at start (default 0)
- **stop** one step before stop (default is length of list)
- **step** specifies how many indices to jump

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

- Get everything:

```
numbers[:]
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```
numbers[:3]
```

Slicing

```python
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```python
numbers[:3]
```

```
[1, 2, 3]
```

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```
numbers[1:3]
```

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```
numbers[1:3]
```

```
[2, 3]
```

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```
numbers[1:5:2]
```

Slicing

```python
numbers = [1, 2, 3, 4, 5, 6, 7]
```

What will this return?

```python
numbers[1:5:2]
```

```
[2, 4]
```

Slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7]
```

Look how easy it is to get first **n** elements, and then the rest

```
n = 4
print(numbers[:n])
print(numbers[n:])
```

```
[1, 2, 3, 4]
[5, 6, 7]
```

A few more examples

```
L = [3, 't', 10, [1, 2]]
```

```
len(L)
```

```
4
```

```
L[2] + 3
```

```
13
```

```
L[3] # another list
```

```
[1, 2]
```

A few more examples

```
L = [3, 't', 10, [1, 2]]
```

```
L[4] # would give error
```

Iterating over a list

```
L = [2, 3, 5]

total = 0

for i in L:
    total += i
print(total)
```

```
10
```

Tuples

- Tuples are also an ordered sequence of items
- They are created, typically, with parentheses ()
- Unlike lists, tuples are **immutable**

```
tpl  = ('a', 5, True) # different types
print(tpl)
```

```
('a', 5, True)
```

Tuples

But also like this:

```
new_tuple = 1, 2, 3
print(type(new_tuple))
print(new_tuple)
```

```
<class 'tuple'>
(1, 2, 3)
```

Tuples

- Can be ordered and be indexed

```
tpl  = ('a', 5, True)
print(tpl[0])
print(tpl[:])
print(tpl[:2])
print(tpl[-1])
```

```
a
('a', 5, True)
('a', 5)
True
```

Tuples

- Used to conveniently swap variable values

```python
a = 5 # initial values
b = 10

a, b = b, a # swapping using tuples

print("a:", a)  # output: a: 10
print("b:", b)  # output: b: 5
```

```
a: 10
b: 5
```

Tuples

- Used to return more than one value from a function, since it conveniently packages many values of different type into one object

```
def calculate_sum_and_product(a, b):
    _sum = a + b
    product = a * b
    return _sum, product
result_tuple = calculate_sum_and_product(5, 3)
print(result_tuple)
print(type(result_tuple))
```

```
(8, 15)
<class 'tuple'>
```

Tuples

- Tuples have just two methods: count() and index()

```
tpl = ('a', 'b', 'a')
print(tpl.count('a'))
```

```
2
```

```
print(tpl.index('b'))
```

```
1
```

Tuples

- Tuples are iterable
- It is possible to have tuples of length 1 (singleton)

```
# notice the comma, otherwise won't work
tpl1 = (5,)
```

```
len(tpl1)
```

```
1
```

String methods

- Python has really rich string methods
- See a full list here
- Let's explore some of them

startswith() and endswith()

- Return a Boolean value
- Use in operator to see if a string ‘just contains’ a substring

```
url = 'www.google.com'
url.startswith('www.g')
url.endswith('.co.kr')
```

```
False
```

```
print('www' in url and '.ac.kr' in url)
print('www' in url and '.com' in url)
```

```
False
True
```

capitalize(), title(), upper(), lower()

- capitalize() capitalizes the first character
- title() capitalizes the first character of every word
- upper() capitalizes everything
- lower() converts string to all lowercase

capitalize(), title(), upper(), lower()

```
paper = 'the New York times'
paper.capitalize()
```

```
'The new york times'
```

```
paper.title()
```

```
'The New York Times'
```

```
paper.upper()
```

```
'THE NEW YORK TIMES'
```

```
paper.lower()
```

```
'the new york times'
```

split()

- split() splits strings into a list of strings
- If no character is specified, it will split from space ' '

```
list_split = paper.split()
print(list_split)
```

```
['the', 'New', 'York', 'times']
```

```
print(url)
url.split('.')
```

```
www.google.com
```

```
['www', 'google', 'com']
```

```
'MICHIGAN'.split('I')
```

```
['M', 'CH', 'GAN']
```

strip, lstrip, rstrip

- Trim from right (`rstrip`), left (`lstrip`), or both sides (`strip`)
- Removes trailing white space characters such as `' '`, `\t`, `\n`

```
msg = '\n   Hello, world! Again!   \n'
print(msg)
```

```

   Hello, world! Again!
```

strip, lstrip, rstrip

```python
msg = '\n   Hello, world! Again!   \n'
print(msg.strip())
```

```
Hello, world! Again!
```

strip, lstrip, rstrip

```python
msg = '\n   Hello, world! Again!   \n'
print(msg.lstrip())
```

```
Hello, world! Again!
```

strip, lstrip, rstrip

```python
msg = '\n   Hello, world! Again!   \n'
print(msg.rstrip())
```

```

   Hello, world! Again!
```

String methods

- As you might have noticed, string methods will never modify in place

```
paper_mod = paper.upper()
print(paper_mod)
```

```
THE NEW YORK TIMES
```

Indexing and Slicing Strings

strings can be indexed and sliced just like lists

```python
word = 'slicing'
word[:3]
```

```
'sli'
```

- `[start:stop:step]` still works

```python
word[2:5:2]
```

```
'ii'
```

Indexing and Slicing Strings

```python
word = 'slicing'
```

```python
word[0]
```

```
's'
```

```python
word[-1]
```

```
'g'
```

```python
word[2]
```

```
'i'
```

Strings and for loops

- strings are immutable, but they are iterable

```
for char in 'hey you':
  if not char == ' ':
    print(char.upper() + '!')
  else:
    break
```

```
H!
E!
Y!
```

String length

- The length of a string is its number of characters (including white spaces)

```
len('Ann Arbor')
```

```
9
```

Strings and Lists

- Convert string to list—gives a list with every character in string

```
list('sci art')
```

```
['s', 'c', 'i', ' ', 'a', 'r', 't']
```

- We also saw this gives a list

```
'sci art'.split()
```

```
['sci', 'art']
```

Strings and Lists

- One can also join a list of elements into a string (e.g., when you want to lump multiple sentences)

```
letters = ['U', 'M', 'I', 'C', 'H']
''.join(letters)
```

```
'UMICH'
```

- Could join by another string too

```
'-'.join(letters)
```

```
'U-M-I-C-H'
```

Aliasing

- Variables do not hold the data themselves
- They are a reference to the memory location where the data is hold
- With aliasing, both variables point to the same object in memory

Aliasing

- **`w`** is an alias for **`warm`**
- It does **not** create a new object

```
warm = ['red', 'yellow', 'orange']
w = warm
```

Aliasing

- When **`w`** changes, **`warm`** will also change (and vice versa)

```python
w.append('pink')
```

- append() also influenced **`hot`**

```python
print(warm)
```

```
['red', 'yellow', 'orange', 'pink']
```

Cloning

- **if** we want to avoid this, then we need to copy every element with `chill = cool[:]`

```
cool = ['blue', 'green', 'grey']
c = cool[:]
```

- Now, if we append something to **`c`**, it will not affect **`cool`**

Cloning

- Or use copy function

```python
import copy
cool = ['blue', 'green', 'grey']
c = copy.copy(cool)
print(c)
```

```
['blue', 'green', 'grey']
```

- Now, if we append something to `c`, it will not affect `cool`

Comprehensions

- `list`, `set`, and `dictionary` comprehensions
- It's possible to create a `list`, `set`, or `dictionary` using:
  - for / while loops
  - if / else statements in the loop
- **comprehensions** provide simple syntax to achieve it in a single line
- Better readability too

List comprehension

```python
import time
start_time = time.time() # for loop approach
squares_loop = []
for i in range(10**5):
    squares_loop.append(i * i)
end_time = time.time()
time_for_loop = end_time - start_time

start_time = time.time() # list comprehension
squares_comp = [i * i for i in range(10**5)]
end_time = time.time()
time_list_comprehension = end_time - start_time

time_for_loop, time_list_comprehension
```

```
(0.006700038909912109, 0.0026280879974365234)
```

List comprehension

- Simple example: create a copy of a list
- Let's do this with a for loop:

```
new_list = []

for number in range(10):
    new_list.append(number)

print(new_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension

- Create a copy of the list with list comprehension

```
new_list = [num for num in range(10)]
print(new_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension

- The more common scenario:
    - “Do something” to every element of a list

```
new_list = []
for number in range(10):
    new_list.append(number**2)
print(new_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension

- List comprehension: “square the number, for each number in numbers”

```python
squared = [num**2 for num in range(10)]
print(squared)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension

- Do something to element only if a condition is true
- New list with only even numbers squared
- Without list comprehension:

```python
some_squared = []
for num in numbers:
    if num % 2 == 0:
        some_squared.append(num**2)
print(some_squared)
```

```
[4, 16, 36]
```

List comprehension

- “square the number, for each number in numbers, if the number is even”

```
some_squared = [num**2 for num in numbers if num % 2 == 0]
print(some_squared)
```

```
[4, 16, 36]
```

- Notice, we can use any name instead of **`num`**

```
some_squared = [i**2 for i in numbers if i % 2 == 0]
print(some_squared)
```

```
[4, 16, 36]
```

List comprehension

- More advanced example
- Combine two digits to form a two-digit number, **if** the number is divisible by 3
- Need to iterate over two lists

```
L1 = [1, 2, 3]
L2 = [6, 7, 8]
L3 = [int(str(n1) + str(n2)) for n1 in L1 for n2 in L2 if (n1 + n2) % 3 == 0]
print(L3)
L4 = [int(str(n1) + str(n2)) # a bit tricky
      if (n1 + n2) % 3 == 0
      else None for n1 in L1 for n2 in L2]
print(L4)
```

```
[18, 27, 36]
[None, None, 18, None, 27, None, 36, None, None]
```

Reading and Writing Text Files

- Write a regular text file

```
file = open('mytext.txt', 'w')
file.write('Hi, there!.\nThis is my text file')
file.close()
```

Reading and Writing Text Files

- Read it back in

```
file = open('mytext.txt', 'r')
content = file.read()
file.close()
```

Reading and Writing Text Files

- Check content

```
print(content)
```

```
Hi, there!.
This is my text file
```

Reading and Writing Text Files

- `open()`, `read()` / `write()`, `close()` is a bit cumbersome
- The more preferred syntax: **`with()`**

```
with open('mytext.txt', 'w') as file:
    file.write('Comment 1\nComment 2\nComment 3')
```

Reading and Writing Text Files

- readlines()

```python
with open('mytext.txt', 'r') as file:
    content = file.readlines()
print(content)
type(content)
```

```
['Comment 1\n', 'Comment 2\n', 'Comment 3']
```

```
list
```

Reading and Writing Text Files

- read()

```
with open('mytext.txt', 'r') as file:
    content = file.read()
print(content)
type(content)
```

```
Comment 1
Comment 2
Comment 3
```

```
str
```