NYU

Special Topics: Python

Class #3

Class #2 and Lab #2 Review

- Let's take some time to review concepts from Class #2 and Lab #2
- Any remaining questions?

Lists within Lists

- A list can have **any kind** of object within it
- Everything in Python is an object
- Can we have a List within a List?

2-D Lists

- A "list of lists" is often called a 2-Dimensional List

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

- Each element of `my_list` is itself a list of integers

```
my_list[0]  # [1, 2, 3]
my_list[1]  # [4, 5, 6]
my_list[2]  # [7, 8, 9]
len(my_list)  # 3
```

Visualizing 2D Lists

- Print each inner list on its own "row"

```
my_list = [[1,2,3],[4,5,6],[7,8,9]]
for row in my_list:
    print(row)
```

- Result:

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

Indexing 2D Lists

- What value would the following resolve to?

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

```
my_list[0][2]
```

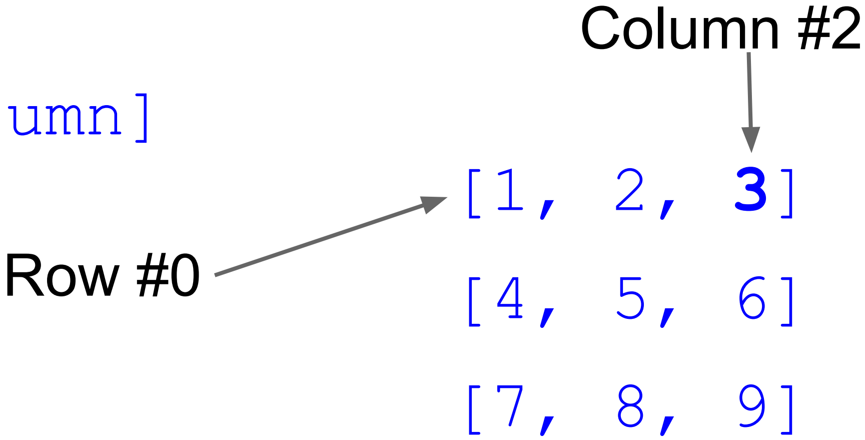
- Consider it one statement at a time
  - `my_list[0][2]`
  - `[1,2,3][2]`
  - `3`

Indexing 2D Lists

- Or if you like thinking of them as rows and columns:

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

```
my_list[row][column]
```



Column #2

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

Row #0

Nested Lists

- Of course, nested lists don't have to be perfectly rectangular

```
my_list = [[1],[1,2],[1,2,3]]
```

- Nor do they have to be of the same type

```
my_list = ['a', [1,'a'], [1,[1],1], []]
```

- But just because you *can*, doesn't mean you *should*.
- Nested lists can be very convenient, but avoid situations that are overly complex or confusing

Sets

- **Set**: An unordered collection of distinct values
  - Unordered: unlike a List or Tuple, there is no concept of "first"
  - Distinct: Every object in a set is unique. No duplicates
- Why no duplicates?
  - Sets in mathematics are used to represent things like "the set of whole numbers".
  - It's meaningless to say "how many times does 5 exist in the set of whole numbers"
  - You only ask "is 5 in the set of whole numbers, or not"?
- Sets in computer science work the same way

Defining a set

- A set is defined by putting a comma-separated list of values inside curly braces:

```
my_set = {'foo', 10, False}
```

- As with other iterables, you can mix types.
- Proof of the new type:

```
type({'foo', 'bar'})
```

Proving Set Qualities

- Proof that sets have **distinct** values

```
{1, 2, 2} == {1, 2}  # True
print({1, 2, 2})  # Prints "{1, 2}"
{1, 2, 2, 2, 2} == {1, 2} # True
```

- Creating a set with duplicate values will resolve to its **proper form**, where all values are distinct
- There's no way to force a set to have duplicates, since that would make it no longer a set!

Proving Set Qualities

- Proof that sets are also **unordered**

```
{1, 2} == {2, 1}   # True
[1, 2] == [2, 1]   # False, Lists are ordered
{1, 2, 2, 1, 1} == {2, 1}   # True
```

- Because sets are unordered, you cannot index them like you would a list.

```
{'foo', 'bar'}[0]   # Error!
```

- Neither one is the 0th because neither one comes "first"!

Working with Sets

- You can iterate over a Set, but the order you get the items in is **not** guaranteed

```
for item in my_set:
    do_something(item)
```

- The `in` and `not in` operators works as expected

```
5 in {1, 2, 3, 4, 5}  # True
0 not in {1, 2, 3, 4, 5}  # True
```

Math Operations on Sets

- All standard math operations on sets are supported:
  - Check for subsets / supersets
  - Calculate the union of two sets
  - Calculate the intersection of two sets
  - Calculate the difference of two sets
    - Read how here: https://docs.python.org/3.1/library/stdtypes.html#set-types-set-frozenset

Sets and Mutability

- Sets **are** mutable, and the following functions can be used on sets:
  - add(elem): Add element elem to the set.
  - remove(elem): Remove element elem from the set. Raises KeyError if elem is not contained in the set.
  - discard(elem): Remove element elem from the set if it is present.
  - pop(): Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.
  - clear(): Remove all elements from the set

Sets and Mutability

Example of mutating a set:

```
def toggle(item, set):
  if item in set:
    set.remove(item)
  else:
    set.add(item)
  return set
```

Examine the following:

```
my_set = {1,2,3,4,5}
my_set.toggle(3).toggle(3)
my_set.toggle(2)
my_set.toggle(7)
my_set.remove(1)
```

What is the value of `my_set`?

Set Contents

- If we can have a list within a list, can we have a set within a set?
- Actually, no.
  - Lists contain any Object.
  - Sets contain any **hashable**.
- What is a hashable?
  - We'll come back to this later.
  - Just note for now that *most* types are Hashable but not all
    - Mutable data structures are often not hashable, so Lists and Sets are not, but Tuples (immutable) are

Hashing and Hashable

[Explanation postponed to Class 4 for brevity]

The empty set

- You can create an empty list like so:

```
my_list = []
```

- So can you create an empty set like this?

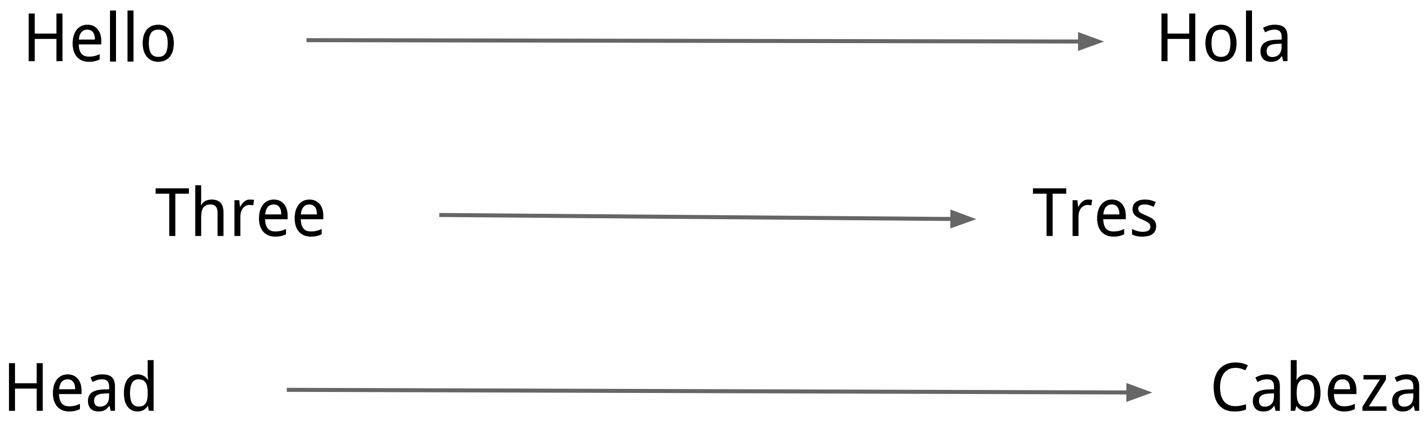
```
my_set = {}
```

- Try it. What do you think?
- Now try the following:

```
type(my_set)
```

Dictionaries

- **Dictionary**: A map of keys to values
- **Map**: Like in mathematics; represents a "mapping" of one set of values to another



Hello → Hola

Three → Tres

Head → Cabeza

Dictionary Mappings

- Dictionary mappings are **one-way**
    - You lookup a value by a key, never the other way around

- Keys must be unique, but values are unrestricted
    - i.e. a Dictionary has a **set** of keys
        - Thus, Dictionary keys must also be **hashable**

- Generally, your dictionaries *should* be one-to-one, with all of your keys of the same type.

Working with Dictionaries

- The syntax for creating a dictionary is:

```
my_dict = {'key1':'val1', 'key2':'val2'}
```

- This dictionary has two key-value pairs
  - The key 'key1' points to value 'val1'
  - The key 'key2' points to value 'val2'
- To access a value, index by the key:

```
my_dict['key1']  # 'val1'
my_dict['key2']  # 'val2'
```

Working with Dictionaries

- Dictionaries are mutable, so we can modify values

```
my_dict['key1'] = 100
my_dict['key1']  # 100
```

- We can even create new key-value pairs on the fly

```
dict2 = {}
dict2['foo'] = 'bar'
dict2  # {'foo:'bar'}
```

Working with Dictionaries

- The `in` and `not in` operators check if a **key** is in a dictionary, not a value

```
my_dict = {'foo':'bar'}
'foo' in my_dict  # True
'bar' in my_dict  # False
```

- This usually what you want to do anyway. Trying to access a non-existent key is an error, so it's good to check if a key is present first, if you're not sure.

Working with Dictionaries

- If you iterate over a dictionary, you're really iterating over that dictionary's **keys**

```
for key in my_dict:
    print('Key: %s' % key)
    print('Value: %s' % my_dict[key])
```

Classes

- Think of a class as a "thing"
- Technically a class is a template for instances of that "thing"
- Classes describe:
  - What data / state each instance should have
  - What actions (methods) each instance can do
  - How instances interact with other objects

Class Example

- Think of a class that represents a player in a simple shooter game

Attributes

- Health
- Ammo
- Location (X, Y coordinates)

Functions:

- Shoot
- Reload
- Walk

- Every player has the same functions, and the same attributes, but different *values* for those attributes

Defining your own Class

```python
class Player:
    """Represents a player in the game."""
    def __init__(self):
        self.health = 100
        self.ammo = 50
        self.position = (0, 0)

player1 = Player()  # Creates a new player instance
player1.health = player1.health - 20
```

The __init__ method

- Remember during style we said to never name a method with leading and trailing underscores?
  - Because they're reserved for special Python features
- `__init__(self, …)` is a special function called an "initializer" (also known as a constructor)
- Whenever you create a new instance of your object, the `__init__` method is called and the new instance is passed into the `self` parameter.

The __init__ method

- **Instance Variable**: Technical name for our class' attributes
    - Each one is a variable, but they're independent across instances
        - i.e. changing player1's health doesn't affect player2
- We define instance variables in the `__init__` function just like we would define a variable, except we reference it on the `self` object

```
self.health = 100
```

- If the instance didn't have "health" before, it does now!

The __init__ method

- As you can see, we gave our instance variables a "default" value, since that's how we create a variable:

```
class Thing:
    def __init__(self):
        self.foo = 20
```

- In the above example, all newly-created instances of Thing will have a member called "foo" which is 20.
- If you don't want a default value, use `None.`

Attributes and Instance Vars

- Unlike other languages, you can give an instance a new attribute at any time, not just in the `__init__` function

```
x = player()
x.deaths = 2
```

- This is considered dangerous, and should rarely be done.
  - You always know what memebers your instances will have, because your class definition acts as a template
  - Creating new ones on the fly breaks that consistency

Instance Methods

- We've seen a lot of examples of objects that have methods you can call:

```
'foo'.swapcase()
```

```
[1,2,3].append(4)
```

- How can we create some for our own classes?
- Easy! Just define a method inside your class body
  - The first parameter **must** be `self` for all instance methods

Class Methods

```
class Player:
    def __init__(self):
        self.health = 100
    def take_damage(self, damage):
        self.health -= damage
        if self.health > 0:
            print('Alive with %s health' % self.health)
        else:
            print('Oh no, you're dead!')
```

Class Methods

```
player1 = Player()
player2 = Player()
player1.health  # 100
player1.take_damage(60)
    Alive with 40 health
player1.health  # 40
player2.health  # 100
player2.take_damage(200)
    Oh no, you're dead!
```

__init__ and Class Methods

- `__init__` is just a special class method - it too can take in additional parameters
  - The first **must** be self

```
class Player:
    def __init__(self, health, ammo, speed):
        self.health = health
        self.ammo = ammo
        self.speed = speed
```

Documentation

- Every class **must** have a Docstring.
  - In generally, it looks just like method docstrings only with an Attributes field instead of Args:

```
class SampleClass(object):
   """Summary of class here.

   Longer class information....
   Longer class information....

   Attributes:
       likes_spam: A boolean indicating if we like SPAM or not.
       eggs: An integer count of the eggs we have laid.
   """
```

Class Naming Style

- Classes are the first to break our mold in naming style
- The should be UpperCamelCase
  - No underscores separating words
  - First letter of each word is capitalized

Class member naming

- Often times, the members of a class should not be accessed outside that class
  - Instance variables will be for internal state - other people shouldn't be touching them
  - Methods might be convenience methods that are only for your other public methods to call
- This is when the privacy conventions through naming are extra important

Class Privacy Conventions

```
class Player:
    def __init__(self):
        self.__health = 100

    def take_damage(self, damage):
        self.__health = max(self.__health - damage, 0)

    def get_health(self):
        return self.__health
```

Class Privacy Conventions

```
class Player:
    ...
    def take_damage(self, damage):
        self.__health = self.__process_damage(damage)

    def __process_damage(self, damage):
        total_damage = damage / self.__armor
        armor_durability *= 0.85
        return self.__health - total_damage
```

Defining Classes - Practice

Let's take a few minutes to practice writing classes!