STATS 507
Data Analysis in Python

Week4-1: Python classes (OOP)

Dr. Xian Zhang

Adapted from slides by Professor Jeffrey Regier

Intro: Objects are everywhere in python

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**append**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**extend**(*L*)
Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.`**insert**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**remove**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**pop**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

Image credit:
Charles
Severence

https://docs.python.org/3/tutorial/datastructures.html

Intro: Data scientists frequently use objects

12.6. sqlite3 — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')
```

Image credit:
Charles
Severence

Intro: Object in Python

Python supports many kinds of data…

- **Primitive**: 1, 2,0, True, None…
- Strings, lists, tuple, dictionary…
  - “We can also create our own data type in Python”
  - [1, 2, 3]
  - (1, 2, 3)
  - {‘MI’: ‘Michigan’, ‘CA’: ‘California’}

Each is an object instance, and each object has:

- An internal **data representation**
- A set of **procedures** (function/methods) for interaction with the object.

https://docs.python.org/3/tutorial/classes.html

Intro: objects as data abstraction

Objects are a data abstraction that captures;

- An **internal representation**
  - Through data attributes

- An **interface** for interacting with objects
  - Though methods (aka procedures/functions)
  - Defines behaviors but hides implementations

An object is an **instance** of a **type**

- 507 is an instance of an `int`
- {‘MI’: ‘Michigan’, ‘CA’: ‘California’} is an instance of a `dict`.

https://docs.python.org/3/tutorial/classes.html

1. Creating/ using a class

2. Inheritance

Class: programmer-defined types

Sometimes we use a collection of variables to represent a specific object

- **Example:** we used a list of list to represent a matrix
- **Example:** representing state of a board game list of players, piece positions, etc.
- **Example:** representing a statistical model
  - Want to support methods for estimation, data generation, etc.

Important point: these data structures quickly become very complicated, and we want a way to **encapsulate** them. This is a core motivation (but hardly the only one) for **object-oriented programming**.

https://docs.python.org/3/tutorial/classes.html

First thing: creating v.s using an instance of a class

Creating the class:

- Define the class **name**
- Define class data attributes
- Define procedural attributes
  - Example: implement the Python list class

Using the class:

- Create new **instances** of the class
- Doing operations on the instances
  - `my_list = [1,2 3], my_list [0], len(my_list)`

Creating class: a parallel with functions.

Defining a class is like defining functions.

- With functions, we tell python this procedure exists using `def`
- With classes, we tell Python about a **blueprint** for this new data type
  - **Data** attributes
  - **Procedural** attributes

```python
class Point:
    '''Represents a 2-d point.'''
```

Class header declares a new class, called `Point`.

Docstring provides explanation of what the class represents, and a bit about what it does. This is an ideal place to document your class.

```python
print(Point)
```

```
<class '__main__.Point'>
```

Credit: Running example adapted from A. B. Downey, *Think Python*

Creating a new object

Creating instances of objects is like calling the functions.

- With functions, we call functions with different parameters
- With classes, we create new object instances in memory of this type

```
class Point:
    '''Represents a 2-d point.'''

p = Point()
p
```

```
<__main__.Point at 0x10669b940>
```

Creating a new object is called **instantiation**. Here we are creating an **instance** `p` of the class `Point`.

Indeed, `p` is of type `Point`.

Note: An **instance** is an individual object from a given class.
In general, the terms **object** and **instance** are interchangeable: an object is an instantiation of a class.

Adding attributes: 1) data attribute

1. We can add data attributes that are **directly** inside the class, but outside of any method, also called class attributes.
2. Using the __init__() method: The most common and recommended way to initialize instance attributes.

```
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
print(Point)
```

```
<class '__main__.Point'>
```

```
p1 = Point(1.0, 2.5)
print(p1.name)
print(p1.x, p1.y)
print(type(p))
```

```
I am a Point class data attibute
1.0 2.5
<class '__main__.Point'>
```

The `__init__()` method

A special method to **initialize** some data attributes or perform initialize operations.

Defines how to create an instance of the class.

```
class Point():
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
print(Point)
```

```
<class '__main__.Point'>
```

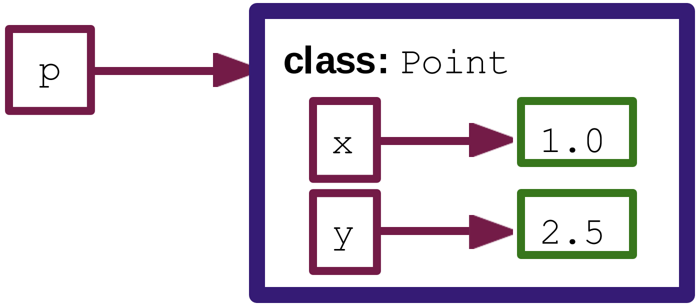
```
p = Point(1.0, 2.5)
print(p.x, p.y)
print(type(p))
```

```
1.0 2.5
<class '__main__.Point'>
```

`__init__` is a special method that gets called when we instantiate an object. This one takes 3 arguments.

`self` allows you to create variables that belong to this object. Every object will have those data attributes.

Without `self`, you are just creating regular variables



The `__str__()` method

Another special method for Python, also know as a “dunder” (double underscore) methods

```
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval

    def __str__(self):
        return f"Point({self.x}, {self.y})"
```

```
p1 = Point(1.0, 2.5)
print(p1)
print(type(p))
```

```
Point(1.0, 2.5)
<class '__main__.Point'>
```

Defines the string representation of an object

Should return a string when

- `print(object)`
- `str(object)`
- `f’’{object}’’`

What is `self`

`self` represents an instance of a class, It is a parameter to refer to an instance of the class without creating one yet. **Always** going to be the first parameter of a method.

```
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

We can refer to object attributes without creating an instance first

```
print(Point)
```

```
<class '__main__.Point'>
```

Do **NOT** need to provide argument for `self`, Python does this automatically

```
def __init__(self, xval, yval):
    self.x = xval
    self.y = yval
```

```
p2 = Point(0, 0)
print(p2.name)
print(p2.x, p2.y)
print(type(p))
```

```
I am a Point class data attibute
0 0
<class '__main__.Point'>
```

Adding attributes: 2) procedural attribute

Also use `self` as the first parameter

```python
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def distance(self, other_point):
        x_diff_square = (self.x - other_point.x) ** 2
        y_diff_square = (self.y - other_point.y) ** 2
        return (x_diff_square + y_diff_square) ** 0.5
```

```python
p1 = Point(1.0, 2.5)
p2 = Point(0, 0)
print(p1.distance(p2))
```

```
2.692582403567252
```

Other parameter to method

Access class data attribute using dot notation

In-class practice

Getters and setters in class

Getters and setters should be used outside of class to access data attributes

```python
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval

    def get_x(self):
        return self.x

    def set_x(self, value):
        if not isinstance(value, (int, float)):
            raise ValueError("x must be a number")
        self.x = value

    def get_y(self):
        return self.y

    def set_y(self, value):
        if not isinstance(value, (int, float)):
            raise ValueError("y must be a number")
        self.y = value
```

```python
p1 = Point(3,4)
# Get values
print(p1.get_x(), p1.get_y())  # Output: 3 4

# Set new values
p1.set_x(5)
p1.set_y(6)
print(p1.get_x(), p1.get_y())  # Output: 5 6
```

Which one is better?

- p1.x
- p1.get_x()

Ans: p1.get_x()

- Good style
- Easy to maintain
- Prevent bugs

Nesting Objects

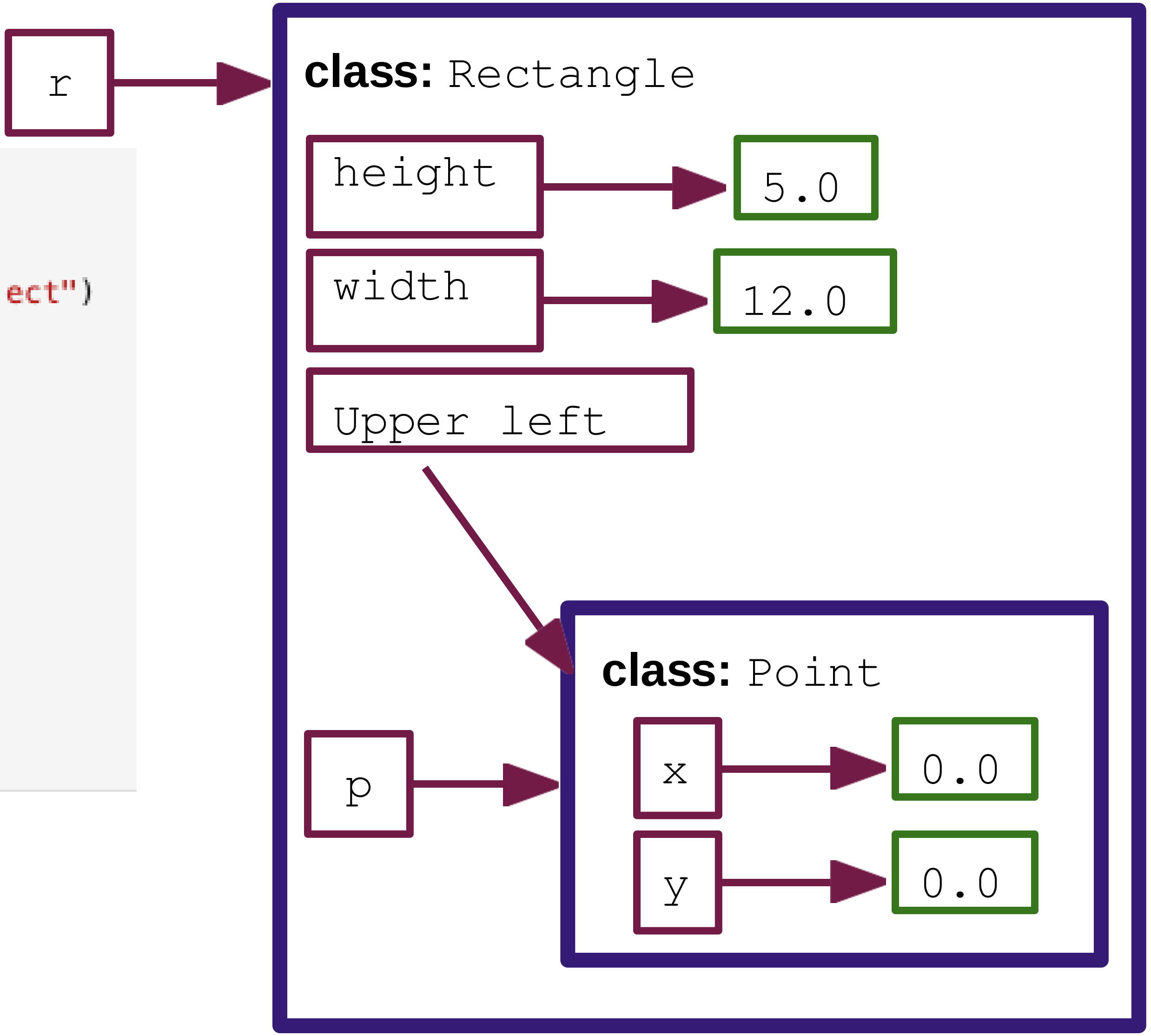
Objects can have other objects as their attributes.
We often call the attribute object **embedded**.

```python
class Rectangle:
    def __init__(self, upper_left, height, width):
        if not isinstance(upper_left, Point):
            raise TypeError("upper_left must be a Point object")
        self.upper_left = upper_left
        self.height = height
        self.width = width

    def area(self):
        return self.height * self.width

upper_left_corner = Point(0,0)
my_rectangle = Rectangle(upper_left_corner, 2, 4)
print(my_rectangle)
print(my_rectangle.area())
```

```
Rectangle(upper_left=Point(0, 0), height=2, width=4)
8
```



Objects are mutable

```
class Point():
    name = "I am a Point class data attibute"
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
p1 = Point(1.0, 2.5)
print(p1.name)
print(p1.x, p1.y)
print(type(p))

p1.x = p1.x + 3
print(p1.x, p1.y)
```

```
I am a Point class data attibute
1.0 2.5
<class '__main__.Point'>
4.0 2.5
```

If my `Point` object were immutable, this line would be an error, because I'm making an assignment.

Since objects are mutable, I can change attributes of an object inside a function and those changes remain in the object in the `__main__` namespace.

Optional arguments for `__init__()`

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.h

t = Time(); t.print_time()
```

```
00:00:00
```

```
t = Time(10); t.print_time()
```

```
10:00:00
```

```
t = Time(10,20); t.print_time()
```

```
10:20:00
```

Important point: notice how much cleaner this is than creating an object and then assigning attributes like we did earlier. Defining an `__init__` method also lets us ensure that there are certain attributes that are **always** populated in an object. This avoids the risk of an `AttributeError` sneaking up on us later. **Best practice** is to create all of the attributes that an object is going to have **at initialization**. Once again, Python allows you to do something, but it's best never to do it!

Pure functions and modifiers.

A **pure function** is a function that returns an object

...and **does NOT** modify any of its arguments

A **modifier** is a function that changes attributes of one or more of its arguments

```
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dy
```

`double_sides` is a **pure function**. It creates a new object and returns it, without changing the attributes of its argument `r`.

`shift_rectangle` changes the attributes of its argument `rec`, so it is a **modifier**. We say that the function has **side effects**, in that it causes changes outside its scope.

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Pure functions and modifiers.

Why should one prefer one over the other?

Pure functions

Are often easier to debug and verify (i.e., check correctness)

https://en.wikipedia.org/wiki/Formal_verification

Common in **functional programming**

https://en.wikipedia.org/wiki/Functional_programming

Modifiers

Often faster and more efficient

Common in **object-oriented programmixwng**

Pure functions and modifiers.

A modifier is a **function** that changes attributes of its arguments

A **method** is *like* a function, but it is provided by an object.

Define a class representing a 24-hour time.

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time()
t.hours=12; t.mins=34; t.secs=56
t.print_time()
```

```
12:34:56
```

Class supports a **method** called `print_time`, which prints a string representation of the time.

Every method must include `self` as its first argument. The idea is that the object is, in some sense, the object on which the method is being called.

Credit: Running example adapted from A. B. Downey, *Think Python*

Pure functions and modifiers.

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)

def int_to_time(seconds):
    '''Convert a number of seconds to a Time object.'''
    t = Time()
    (minutes, t.secs) = divmod(seconds,60)
    (hrs, t.mins) = divmod(minutes,60)
    t.hours = hrs % 24 #military time!
    return t

t = int_to_time(1337)
t.time_to_int()
```

```
1337
```

`int_to_time` is a pure function that creates and returns a new `Time` object.

`Time.time_to_int` is a method, but it is still a pure function in that it has no side effects.

Pure functions and modifiers.

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

    def increment_modifier(self, seconds):
        '''Increment this time by the given
        number of seconds.'''
        (mins, self.secs) = divmod(self.secs+seconds, 60)
        (hours, self.mins) = divmod(self.mins+mins, 60)
        self.hours = (self.hours + hours)%24

t1 = int_to_time(1234)
t1.increment_modifier(1111)
t1.time_to_int()
```

```
2345
```

Two different versions of the same operation. One is a pure function (pure method?), that does not change attributes of the caller. The second method is a modifier.

The modifier method does indeed change the attributes of the caller.

Pure functions and modifiers.

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''
    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

t1.increment_pure(100, 200)
```

Here's an error you may encounter.
How the heck did `increment_pure` get 3 arguments?!

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-1d8fb5e5c628> in <module>()
     14         return t
     15 
---> 16 t1.increment_pure(100, 200)

TypeError: increment_pure() takes 2 positional arguments but 3 were given
```

Answer: the caller is considered an argument (because of `self`)!

1. Creating/ using a class

2. Inheritance

Inheritance hierarchies

Inheritance is perhaps the most useful feature of object-oriented programming

Inheritance allows us to <u>create new classes from old ones</u>

Parent class

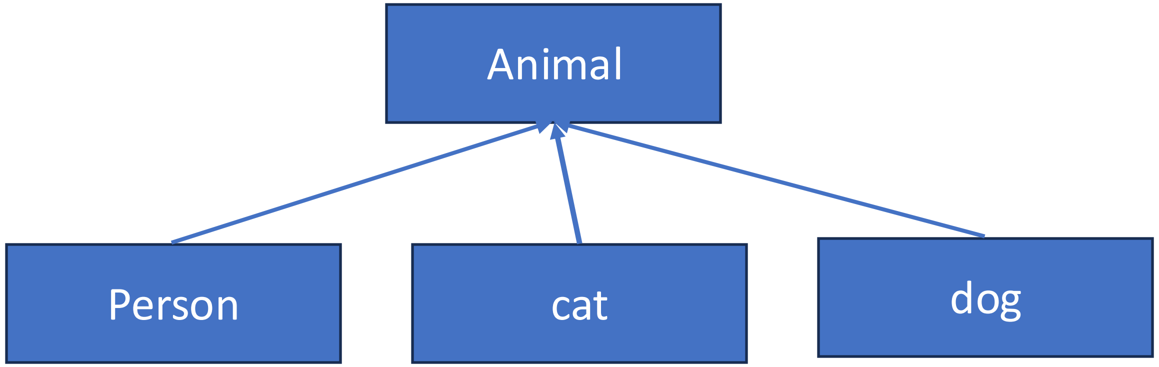
(base class/superclass)

```
class Animal():
    def __init__(self, age):
        self.age = age
        self.name = None
```

Child class

(derived class/ subclass)

- **Inherit** all data and behaviors of the parent class
- **Add** more **<u>info(data)</u>**
- **Add** more **<u>behavior</u>**
- **Override** behavior



Inheritance: parent class

Class definition Class name

```python
class Animal():
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animals: " + str(self.name) + ":" + str(self.age)
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, new_age):
        self.age = new_age
    def set_name(self, new_name = ''):
        self.name = new_name
```

Initialize all the data attributes/ instance variables

One instance

```python
cat1 = Animal(5)
type(cat1)
```

```
__main__.Animal
```

Inheritance: child class

Parent class

(base class/superclass)

Child class

(derived class/ subclass)

- **Inherit** all data and behaviors of the parent class
- **Add** more **info(data)**
- **Add** more **behavior**
- **Override** behavior

Class definition

Class name

Parent class

```
class Cat(Animal):
    def speak(self):
        print("meow")
```

```
my_cat = Cat(7)
my_cat.set_name("Fay")
print(my_cat.get_name())
print(my_cat.speak())
```

```
Fay
meow
None
```

Added behaviors are specific to subclass

Add new functionality (behaviors) with `speak()`

- Instance of type `Cat` can be called new methods
- Instance of type `Animal` throws error (AttributeError) if called with `Cat's` new method

```
my_cat2 = Animal(7)
my_cat2.set_name("Fay")
print(my_cat2.get_age(), my_cat2.get_name(), my_cat2.speak())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[150], line 3
      1 my_cat2 = Animal(7)
      2 my_cat2.set_name("Fay")
----> 3 print(my_cat2.get_age(), my_cat2.get_name(), my_cat2.speak())

AttributeError: 'Animal' object has no attribute 'speak'
```

A running example: Poker

https://en.wikipedia.org/wiki/Poker

Running example: Poker

We will encode suits and ranks by numbers, rather than strings. This will make comparison easier.

A card is specified by its suit and rank, so those will be the attributes of the card class. The default card will be the two of clubs.

```
class Card:
    '''Represents a playing card'''
    def __init__(suit=0,rank=2):
        self.suit = suit
        self.rank = rank
```

Suit encoding
1 : Clubs
2 : Diamonds
3 : Hearts
4 : Spades

Rank encoding
1 : None
2 : Ace
3 : 2
4 : 3
…
10 : 10
11 : Jack
12 : Queen
13 : King

This stage of choosing how you will represent objects (and **what objects to represent**) is often the most important part of the coding process. It's well worth your time to carefully plan and design your objects, how they will be represented and what methods they will support.

Creating `Card` class

```python
class Card:
    '''Represents a playing card'''

    suit_names = ['Spades','Hearts','Diamonds','Clubs']
    rank_names = [None,'Ace','2','3','4','5','6','7',
                  '8','9','10','Jack','Queen','King']

    def __init__(self,suit=0,rank=2):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rankstr = self.rank_names[self.rank]
        suitstr = self.suit_names[self.suit]
        return("%s of %s" % (rankstr, suitstr))

print(Card(0,1))
```

```
Ace of Spades
```

Variables defined in a class but outside any method are called **class attributes**. They are shared across all instances of the class.

Instance attributes are assigned to a specific object (e.g., `rank` and `suit`). Both class and instance attributes are accessed via dot notation.

Here we use instance attributes to index into class attributes.

Adding more operators using dunder method

```python
class Card:
    '''Represents a playing card'''
```

```python
    def __lt__(self, other):
        t1 = (self.rank, self.suit)
        t2 = (other.rank, other.suit)
        return t1 < t2

    def __gt__(self, other):
        return other < self

    def __eq__(self, other):
        return(self.rank==other.rank and self.suit==other.suit)
c1 = Card(2,11); c2 = Card(2,12)
c1 < c2
```

```
True
```

We've chosen to order cards based on rank and then suit, with aces low. So a jack is bigger than a ten, regardless of the suit of either one. Downey orders by suit first, then rank.

```python
c1 == Card(2,11)
```

```
True
```

Now that we've defined the `__eq__` operator, we can check for equivalence correctly.

Creating `Deck` class

```python
class Deck:
    '''Represents a deck of cards'''
    def __init__(self):
        self.cards = list()
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit,rank)
                self.cards.append(card)

    def __str__(self):
        res = list()
        for c in self.cards:
            res.append(str(c))
        return('\n'.join(res))

d = Deck()
print(d)
```

```
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
```

Define a new object representing a deck of cards. A standard deck of playing cards is 52 cards, four suits, 13 ranks per suit, etc.

Represent cards in the deck via a **list**. To populate the list, just use a nested for-loop to iterate over suits and ranks.

String representation of a deck will just be the cards in the deck, in order, one per line. Note that this produces a **single string**, but it includes newline characters.

Providing additional methods for `Deck`

```python
import random
class Deck:
    '''Represents a deck of cards'''
```

```python
    def pop_card(self):
        return(self.cards.pop())
    def add_card(self,c):
        self.cards.append(c)
    def shuffle(self):
        random.shuffle(self.cards)
```

One method for dealing a card off the “top” of the deck, and one method for adding a card back to the “bottom” of the deck.

Note: methods like this that are really just wrappers around other existing methods are often called **veneer** or **thin methods**.

```python
d = Deck()
d.shuffle()
print(d)
```

```
2 of Hearts
9 of Clubs
Ace of Spades
3 of Clubs
6 of Spades
```

After shuffling, the cards are not in the same order as they were on initialization.

Now let’s take stock

We have:

a class that represents playing cards (and some basic methods) a class that represents a deck of cards (and some basic methods)

Now, the next logical thing we want is a **class** for representing a hand of cards So we can actually represent a game of poker, hearts, bridge, etc.

The naïve approach would be to create a new class Hand from scratch But a more graceful solution is to use **inheritance**

Key observation: a hand is a lot like a deck (it’s a collection of cards) ...of course, a hand is also different from a deck in some ways...

Inheritance

This syntax means that the class Hand**inherits** from the class Deck. Inheritance means that Handhas all the same methods and class attributes as Deckdoes.

```
class Hand(Deck):
    '''Represents a hand of cards'''

h = Hand()
h.shuffle()
print(h)
```

```
Ace of Clubs
Queen of Diamonds
9 of Hearts
King of Hearts
8 of Clubs
8 of Hearts
Queen of Clubs
3 of Diamonds
5 of Hearts
7 of Clubs
King of Diamonds
```

We say that the **child** class Hand inherits from the **parent** class Deck.

So, for example, `Hand` has `__init__` and `shuffle` methods, and they are identical to those in Deck. Of course, we quickly see that the `__init__` inherited from `Deck` isn't quite what we want for `Hand`. A hand of cards isn't usually the entire deck...

So we already see the ways in which inheritance can be useful, but we also see immediately that there's no free lunch here. We will have to **override** the `__init__` function inherited from Deck.

Inheritance: methods and overriding

```
class Hand(Deck):
    '''Represents a hand of cards'''

    def __init__(self, label=''):
        self.cards = list()
        self.label=label

h = Hand('new hand')
d = Deck(); d.shuffle()
h.add_card(d.pop_card())
print(h)
```

```
6 of Spades
```

Redefining the `__init__` method **overrides** the one inherited from `Deck`.

Simple way to **deal** a single card from the deck to the hand.

Inheritance: methods and overriding

```python
import random
class Deck:
    '''Represents a deck of cards'''

    def move_cards(self, hand, ncards):
        for i in range(ncards):
            hand.add_card(self.pop_card())
```

Encapsulate this pattern in a method supplied by `Deck`, and we have a method that deals cards to a hand.

```python
d = Deck(); d.shuffle()
h = Hand()
d.move_cards(h,5)
print(h)
```

```
2 of Spades
King of Spades
9 of Diamonds
2 of Diamonds
7 of Clubs
```

Note that this method is supplied by `Deck` but it modifies both the `caller` and the `Hand` object in the first argument.

Note: `Hand` also inherits the `move_cards` method from `Deck`, so we have a way to move cards from one hand to another (e.g., as at the beginning of a round of hearts)

Inheritance: pros and cons

Pros:

Makes for simple, fast program development

Enables code reuse

Can reflect some natural structure of the problem

Cons:

Can make debugging challenging (e.g., where did this method come from?)

Code gets spread across multiple classes

Can accidentally override (or forget to override) a method

Other things

HW3 due this week.

Coming next:

More class method, Exceptions, Iterators, and Generators