THE UNIVERSITY OF CHICAGO

MASTERS IN COMPUTATIONAL SOCIAL SCIENCE
THE UNIVERSITY OF CHICAGO

MACS 30111

Classes and Objects

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- Object-oriented modeling
- Class composition
- Private attributes
- Dunder methods

Object-oriented paradigm

The *object-oriented paradigm* (OO or OOP) provides a way for us to define new data types that **encapsulate attributes and operations** that are allowed on the data type.

OO allows for a cleaner separation between:

- public attributes and operations that any user can access
- Private ones that are only accessible to certain developers

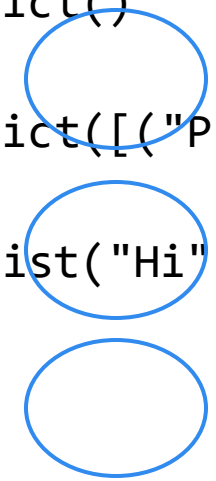
Topics:

- Function-based API VS OOP
- **Class construction, attributes, and methods**
- Object-oriented modeling
- Class composition
- Private attributes
- Dunder methods

Built-in classes

we've already been using classes in Python:

```
a = dict()                          # a = {}

b = dict([("P", 7), ("Q", 9)])      # b = {"P": 7, "Q": 9}

c = list("Hi")                      # c = ["H", "i"]
```



In the object-oriented paradigm:

- dict and list are called ***classes***
- a, b, and c are ***objects*** or ***instances of a class***

Object creation follows a pattern: object = class_name(...)

Define a class in Python

In the OO paradigm, a *class* is the definition of a new data type.

Define a class

class name

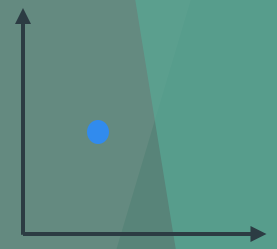
keyword

```
class Point:

    # Class definition here
    # Everything inside is indented
```

Create objects

```
p1 = Point(...)
```

Class constructor



An object is *constructed* or *initialized* with the special ***__init__*** function. The constructor initializes *attributes* of the class.

Define a class

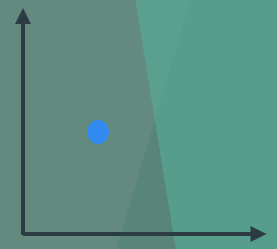
```
class Point:

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

Create objects

```
p1 = Point(2, 5)
```

```
Point.__init__(empty_object, 2, 5)
```

Class methods



A *method* is a function that belong to a class.
Methods are usually called on objects of that class using the dot (.) notation.

Define a class

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_to_origin(self):
        return (self.x**2 +
                self.y**2)**0.5
```

Create objects

```
p1 = Point(2, 5)

# Call class methods
p1.distance_to_origin()
```

```
Point.distance_to_origin(p1)
```

Coding practice: 2.4.1

Import classes

Similar to functional API, we can also put the definition of a class in a Python **file** and **import** it from IPython or other Python files.

myclass.py

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_to_origin(self):
        return (self.x**2 +
                self.y**2)**0.5
```

myprogram.py

```
import myclass

p1 = myclass.Point(2,5)

p1.distance_to_origin()
```

IPython

```
In [1]: import myclass

In [2]: p1 = myclass.Point(2,5)

In [3]: p1.distance_to_origin()
```

Import module from a different path:

```
import sys
sys.path.append("/path/to/my/modules/")
import my_module
```

Pause: WHY DO WE CARE?

- What would the advantages be of OOP?
- Why might we want classes and methods? Aren't they just extra complications?

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- **Object-oriented modeling**
- Class composition
- Private attributes
- Dunder methods

Modeling students

We could represent students with a dictionary of student data.

```
d1 = {"name": "Sam Student",
      "majors": ["Computer Science"],
      "year": 2}

d2 = {"name": "Ari Student",
      "majors": ["Computer Science", "Mathematics"],
      "year": 1}

d3 = {"name": "Janet Student",
      "majors": ["Economics"],
      "year": 4}
```

Modeling students

A better way to represent students is to create a Student class.

```
class Student:


    def __init__(self, name, majors, year):
        self.name = name
        self.majors = majors
        self.year = year


s1 = Student("Sam Student", ["Computer Science"], 2)
s2 = Student("Ari Student",
             ["Computer Science", "Mathematics"], 1)
s3 = Student("Janet Student", ["Economics"], 4)
```

What is a ‘student’?
What ‘properties’ do they ha

Modeling students

In particular, we can add methods that associate operations with the Student class.

```
class Student:

    def __init__(self, name, majors, year):
        self.name = name
        self.majors = majors
        self.year = year

    def num_majors(self):
        return len(self.majors)

    def __repr__(self):
        return "Student: {}".format(self.name)
```

Valid input

The assert statement can be used in a class constructor to check input.

```
class Student:

    def __init__(self, name, majors, year):
        assert isinstance(name, str), "name must be a string"
        assert isinstance(majors, list) and \
               all([isinstance(major, str) for major in majors]),\
               "majors must be a list of strings"
        assert 1 <= year <= 4, "year must be between 1 and 4"

        self.name = name
        self.majors = majors
        self.year = year

    # Student methods
```

Learning check

Try:

```
for s in students:
    print(s)
```

```
class Student:

    def __init__(self, name, majors, year):
        assert isinstance(name, str), "name must be a string"
        assert isinstance(majors, list) and \
               all([isinstance(major, str) for major in majors]),
         "majors must be a list of strings"
        assert 1 <= year <= 4, "year must be between 1 and 4"

        self.name = name
        self.majors = majors
        self.year = year

s1 = Student("Sam Student", ["Computer Science"], 2)
s2 = Student("Ari Student",
             ["Computer Science", "Mathematics"], 1)
s3 = Student("Janet Student", ["Economics"], 4)

students = [s1, s2, s3]
```

Internal representation

The Student class with three attributes: name, major, and year.

```
class Student:

    def __init__(self, name, majors, year):
        self.name = name
        self.majors = majors
        self.year = year

    def num_majors(self):
        return len(self.majors)

    def __repr__(self):
        return "Student: {}".format(self.name)
```

```
# Create Student objects
students = [s1, s2, s3]

# Print number of majors
for s in students:
    print(s)
    print("Number of majors:",
          s.num_majors())
    print()
```

Internal representation

We can change the internals of Student without affecting the users of our class.

```
class Student:

    def __init__(self, name, majors, year):
        self.name = name
        self.primary_major = majors[0]
        self.secondary_majors = majors[1:]
        self.year = year

    def num_majors(self):
        return 1 + len(self.secondary_majors)

    def __repr__(self):
        return "Student: {}".format(self.name)
```

```
# Create Student objects
students = [s1, s2, s3]

# Print number of majors
for s in students:
    print(s)
    print("Number of majors:",
          s.num_majors())
    print()
```

Real-world example

The course registration website defines these classes:

- Student
- Major
- Instructor
- Quarter
- Course
- LectureSlot
- LectureSection
- WaitlistRequest
- …

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- Object-oriented modeling
- **Class composition**
- Private attributes
- Dunder methods

Divvy Data Challenge

What is the total duration and total distance of all the Divvy trips taken in 2013?

DIVVY

How It Works Pricing System Map Explore Chicago Help

Divvy Data

Historical trip data available to the public

Here you'll find Divvy's trip data for public use. So whether you're a policy maker, transportation professional, web developer, designer, or just plain curious, feel free to download it, map it, animate it, or bring it to life!

Note that we'll be releasing trip data twice a year: once following the end of calendar Q2 and once following the end of calendar Q4. This data is provided according to the Divvy Data License Agreement.

The Data

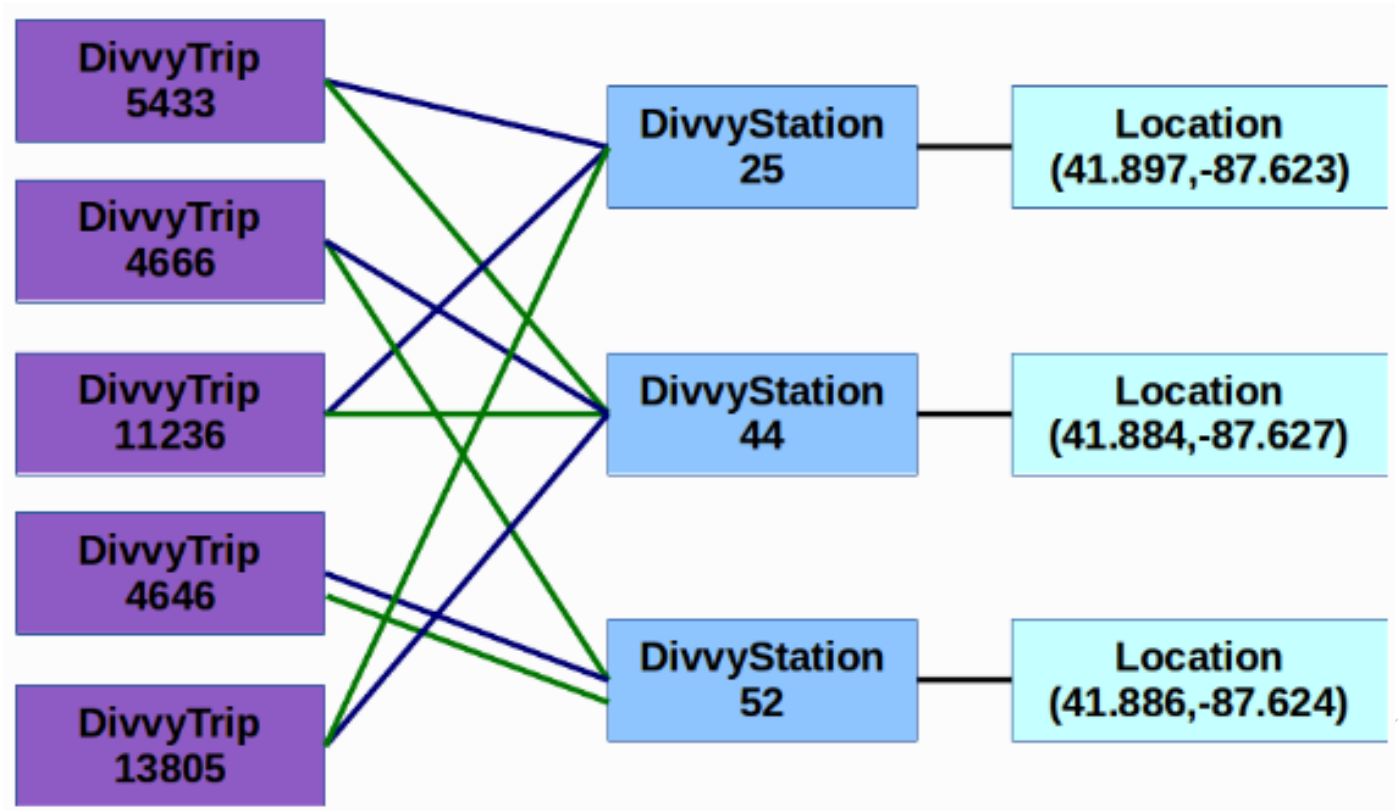
Each trip is anonymized and includes:

- Trip start day and time
- Trip end day and time
- Trip start station
- Trip end station
- Rider type (Member, Single Ride, and Day Pass)
- If a Member trip, it will also include Member's self-reported gender and year of birth

https://www.divvybikes.com/system-data

Class composition

Using DivvyStation **objects as attributes** of the DivvyTrip class is an example of *class composition*.



Coding practice: 2.4.2

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- Object-oriented modeling
- Class composition
- **Private attributes**
- Dunder methods

Invalid access

```
class Location(object):
    def __init__(self, latitude, longitude):
        self.latitude = latitude
        self.longitude = longitude
```

```
chicago_loc = Location( 41.8337329, -87.7321555 )
newyork_loc = Location( 40.7056308, 'foobar' )
```

- Need a controlled mechanism for accessing the class/values

Private attributes

A single underscore before an attribute name, makes the attribute private **by convention**:

- allows other classes to access the attributes
- Informs programmers not to use those attribute directly

```
class Point(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y
```

Controlled mechanism to read and modify the attributes: getter and setter method.

```
class Point(object):
    def __init__(self, x, y):
        self.set_x(x)
        self.set_y(y)

    def get_x(self):
        return self._x

    def set_x(self, x):
        if not isinstance(x, (int, float)):
            raise ValueError("Not a number")
        self._x = x
```

Two underscore before an attribute name, makes the attribute truly private: also prevent other classes from accessing those attributes.

Coding practice: 2.4.3

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- Object-oriented modeling
- Class composition
- Private attributes
- **Dunder methods**

Dunder methods

Dunder methods (sometimes called special or magic methods) are used to define the behavior with respect to the class.

- Initialization: `__init__`
- Representation: `__repr__`, `__str__`
- Comparison operations: `__eq__`, `__lt__`
- Arithmetic operations: `__add__`, `__sub__`

https://docs.python.org/3/reference/datamodel.html#special-method-names

__repr__

The *__repr__* method is a special method that returns a string representation of an object.

```
class Stack:

    def __init__(self):
        self.__lst = []

    # Stack methods

    def __repr__(self):
        return ("STACK: (bottom) "
                + ", ".join(str(x) for x in self.__lst)
                + " (top)")
```

__eq__

Implementing an *__eq__* method defines how Python should interpret the equality operator (==) for checking the equality of two objects.

```
class Stack:

    def __init__(self):
        self.__lst = []

    # Stack methods

    def __eq__(self, other):
        return self.__lst == other.__lst
```

Topics:

- Function-based API VS OOP
- Class construction, attributes, and methods
- Object-oriented modeling
- Class composition
- Private attributes
- Dunder methods

Function-based API

Earlier, we defined new data types by creating function-based APIs.

mystack.py

```
def stack_create():
    return []

def stack_push(stack, value):
    stack.append(value)

# stack operations
```

myprogram.py

```
import mystack

s = mystack.stack_create()

mystack.stack_push(s, 5)
```

Limitations:

- Nothing prevents us from manipulating it in non-stack ways
- Not all data types can be implemented like a list/dict (e.g., multiple attributes)