Show Jupyter Notebook Demo for pickle and joblib
Objects and Classes
Outline

- Objects, classes, and object-oriented programming
  - relationship between classes and objects
  - abstraction
- Anatomy of a class
  - instance variables
  - instance methods
  - constructors
Objects and classes

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Writing programs that perform most of their behavior as interactions between objects.

- **class**: 1. A program. or,
  2. A blueprint of an object.

  - classes you may have used so far:
    - `str`, `list`, `dict`, etc

- We will write classes to define new types of objects.
Abstraction

- **abstraction**: A distancing between ideas and details.
  - Objects in Python provide abstraction:
    - We can use them without knowing how they work.

- You use abstraction every day.
  - Example: Your smart phone.
    - You understand its external behavior (home button, screen, etc.)
    - You don't understand its inner details (and you don't need to).
Encapsulation

- **encapsulation**: Hiding implementation details of an object from clients.
- Encapsulation provides *abstraction*; we can use objects without knowing how they work.

The object has:
- an **external view** (its behavior)
- an **internal view** (the state and methods that accomplish the behavior)
Music player blueprint

**state:**
- current song
- volume
- battery life

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

Music player #1

**state:**
- song = "Let it snow"
- volume = 17
- battery life = 2.5 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

Music player #2

**state:**
- song = "Galaxy song"
- volume = 9
- battery life = 3.41 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

Music player #3

**state:**
- song = "Code Monkey"
- volume = 24
- battery life = 1.8 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song
def scopes():
    def localscope():
        s = 'local scope'

    def notlocalscope():
        nonlocal s
        s = 'nonlocal scope'

    def globalscope():
        global s
        s = 'global scope'

    s = 'scope'
    localscope()
    print('After local scope: ', s)
    notlocalscope()
    print('After notlocalscope: ', s)
    globalscope()
    print('After global scope: ', s)

print('Before calling scope')
scopes()
print('After calling scope ', s)

Before calling scope
After local scope: scope
After notlocalscope: nonlocal scope
After global scope: nonlocal scope
After calling scope global scope
class test():
    """ Example class"""
    x = 14
    def t(self):
        return 'test class'

r = test()
print(r.x)
a = r.t()
print(a)

14
test class
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3,4)
x.r, x.i

(3, 4)
class Cars():
    def __init__(self, model, color, year):
        self.model = model
        self.color = color
        self.year = year
        self.kind = 'Automobile'
        self.behavior = []
    def add_behavior(self, move):
        self.behavior.append(move)
class Cars():
    def __init__(self,model,color,year):
        self.model = model
        self.color = color
        self.year = year
        self.kind = 'Automobile'
        self.behavior = []
    def add_behavior(self,move):
        self.behavior.append(move)

c1 = Cars('Toyota','Silver','2009')
print(c1.model,c1.color,c1.year)
Toyota Silver 2009

c1.add_behavior('start')
print(c1.behavior)
['start']

c2 = Cars('Honda','white','2010')
print(c2.model,c2.color,c2.year)
Honda white 2010

c2.add_behavior('accelerate')
print(c2.behavior)
['accelerate']
How often would you expect to get snake eyes?

If you’re unsure on how to compute the probability then you write a program that simulates the process.
class SnakeEyes:
    def __init__(self, num_rolls):
        self.rolls = num_rolls
        self.count = 0
    def rollingDie(self):
        die1 = Die(6)
        die2 = Die(6)

        for i in range(self.rolls):
            face1Val = die1.roll()
            face2Val = die2.roll()
            # print(face1Val, ' ', face2Val)
            # print('============================')
            if face1Val == 1 and face2Val == 1:
                self.count += 1
        print("Num Snake Eyes: ", self.count)
        print("Num Rolls: ", self.rolls)
        print("Snake eyes probability: ", self.count/self.rolls)

def main():
    s = SnakeEyes(5000)
    s.rollingDie()

if __name__ == "__main__":
    main()
Die object

- State (data) of a Die object:

<table>
<thead>
<tr>
<th>Instance variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numFaces</td>
<td>the number of faces for a die</td>
</tr>
</tbody>
</table>

- Behavior (methods) of a Die object:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll()</td>
<td>roll the die</td>
</tr>
</tbody>
</table>
The Die class

- The class (blueprint) knows how to create objects.

Die class

state:
numFaces = 0
faceValue = 0
behavior:
roll()
getFaceValue()

Die object #1

state:
numFaces = 6
faceValue = 2
behavior:
roll()
getFaceValue()

die1 = Die(5,3)

Die object #2

state:
numFaces = 6
faceValue = 5
behavior:
roll()
getFaceValue()

Die object #3

state:
numFaces = 10
faceValue = 8
behavior:
roll()
getFaceValue()
Object state:
instance variables
The following code creates a new class named `Die`.

```python
class Die:
    faceValue = 0
    def __init__(self, faces):
        self.numFaces = faces
```

- Save this code into a file named `Die.py`.

Each `Die` object contains two pieces of data:
- `numFaces`
- `faceValue`

No behavior (yet).

```python
dice = Die(5)
```
Instance variables

- **instance variable**: A variable inside an object that holds part of its state.
  - Each object has *its own copy*.
- Declaring an instance variable:
  
  `<name> = <value>`

```python
class Die():
    faceValue = 0
    def __init__(self,faces):
        self.numFaces = faces
```
Instance variables

Each Die object maintains its own `numFaces` and `faceValue` variable, and thus its own state.

```python
die1 = Die(5)
die2 = Die(6)
```
Accessing instance variables

- Code in other classes can access your object's instance variables.
  - Accessing an instance variable: dot operator
    
    `<variable name> . <instance variable>`

  - Modifying an instance variable:
    
    `<variable name> . <instance variable> = <value>`

- Examples:
  
  ```python
  print(“you rolled “, die.faceValue)
  die.faceValue = 20
  ```
Client code

- Die and snakeEyes can have a main ...
  - We will almost always do this.... WHY?
  - To test the class before it is used by other classes
- or can be used by other programs stored in separate .py files.

- **client code**: Code that uses a class

---

Roll.py (client code)

```python
def main():
    s = SnakeEyes(5000)
    s.rollingDie()

if __name__ == "__main__":
    main()
```

snakeEyes.py

```python
class SnakeEyes():
    def __init__(self,num_rolls):
        self.rolls = num_rolls
        self.count = 0
...
```
Object behavior: methods
**OO Instance methods**

- Classes combine **state** and **behavior**.
- **Instance variables**: define state
- **Instance methods**: define behavior for each object of a class. Methods are the way objects communicate with each other and with users.
- **Instance method declaration**, general syntax:

  `<name> ( <parameter(s)> ) : <statement(s)>`
class SnakeEyes():
    def __init__(self, num_rolls):
        self.rolls = num_rolls
        self.count = 0
    def rollingDie(self):
        die1 = Die(6)
        die2 = Die(6)

        for i in range(self.rolls):
            face1Val = die1.roll()
            face2Val = die2.roll()
            # print(face1Val, ' ', face2Val)
            # print('================================')
            if face1Val == 1 and face2Val == 1:
                self.count += 1
        print("Num Snake Eyes: ", self.count)
        print("Num Rolls: ", self.rolls)
        print("Snake eyes probability: ", self.count / self.rolls)

class Die():
    faceValue = 0
    def __init__(self, faces):
        self.numFaces = faces
        # self.faceValue = faceVal
    def roll(self):
        faceValue = random.randrange(1, self.numFaces + 1)
        return faceValue
Object initialization: constructors
Initializing objects

- When we create a new object, we can assign values to all, or some of, its instance variables:

```python
die1 = Die(6)
```
class Die():
    faceValue = 0
    def __init__(self, faces):
        self.numFaces = faces
        # self.faceValue = faceVal
    def roll(self):
        faceValue = random.randrange(1, self.numFaces + 1)
        return faceValue

die1 = Die(6)
**Constructors**

- **constructor**: creates and initializes a new object

  ```python
  def __init__(<parameter(s)>):
  <statement(s)>
  ```

  - For a constructor function name is `__init__`
  - A constructor runs when the client calls the `class`.
  - A constructor implicitly returns the newly created and initialized object.
  - You can create an object without calling on a constructor.
Multiple Constructors

- It is not supported by default.

- To define a class other than using `__init__()`, we can use a class method.

- A class method receives the class as the first argument.

- This class is used within the method to create and return the final instance.
No Constructor

- When we want to create an object for a class without calling the constructor, we should use `__new__`

```python
class noConstructorCall:
    def h(self):
        print("Hello")

t = noConstructorCall.__new__(noConstructorCall)
t.h()
```
When we want to create an object for a class without calling the constructor, we should use `__new__`

```python
class noConstructorCall:
    def h(self):
        print("Hello")

t = noConstructorCall.__new__(noConstructorCall)
t.h()
```
Magic methods

class magicMethods:
    def __new__(cls, *args, **kwargs):
        print("Magic Method runs first")
        print(cls)
        print(args)
        print(kwargs)
    def __init__(self, val):
        self.val = val
        print("Inside magic methods")

instance = magicMethods(5)
class magicMethods:
    def __new__(cls,*args,**kwargs):
        print("Magic Method runs first")
        print(cls)
        print(args)
        print(kwargs)
    def __init__(self,vall):
        self.val = vall
        print("Inside magic methods")

instance = magicMethods(5,1,x=3,y=4)
Magic Method runs first
<class '__main__.magicMethods'>
(5, 1)
{'x': 3, 'y': 4}