(Python) OOP Workshop

Object Oriented Programming

Mason Smith

10/25/24

RISE Lab
Ira A. Fulton Schools of Engineering,
Arizona State University

Learning Objectives

1. Understand the core principles of OOP and why its useful
2. Describe when and when not to use OOP for different implementations
3. Define different components of OOP and their use
4. Be able to implement magic methods for more efficient code
5. Be able to implement class inheritance for safer and cleaner code
6. Know what's out there!

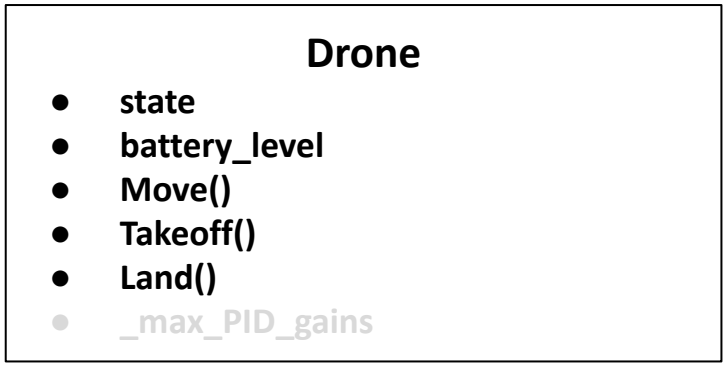
Agenda

Object Oriented Programming

1. Learning objectives
2. Basics of OOP
   - 2.1. Core Principles
   - 2.2. OOP vs FP
3. OOP Structure
   - 3.1. Component Definitions
   - 3.2. Instance vs Class Variables
4. Magic Methods
5. Decorators
6. Inheritance
7. Review & Discussion

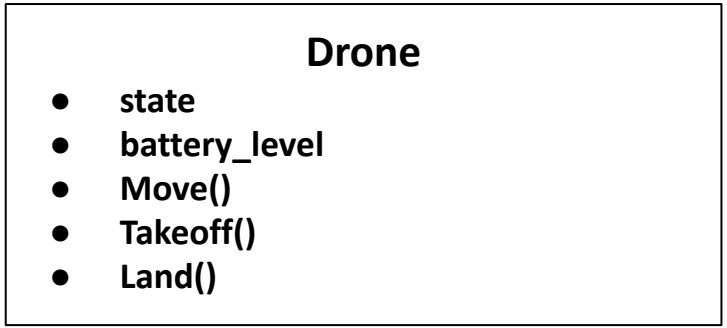
Core Principles of OOP

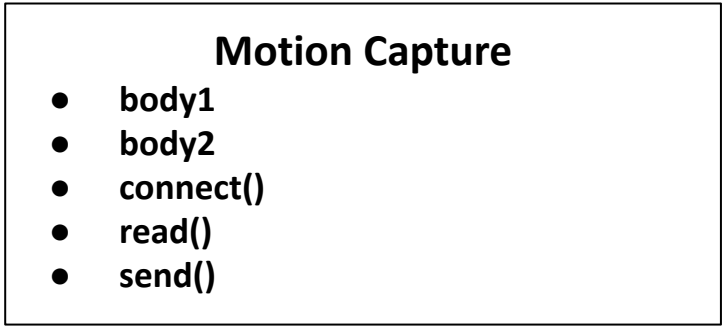
- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")



Core Principles of OOP

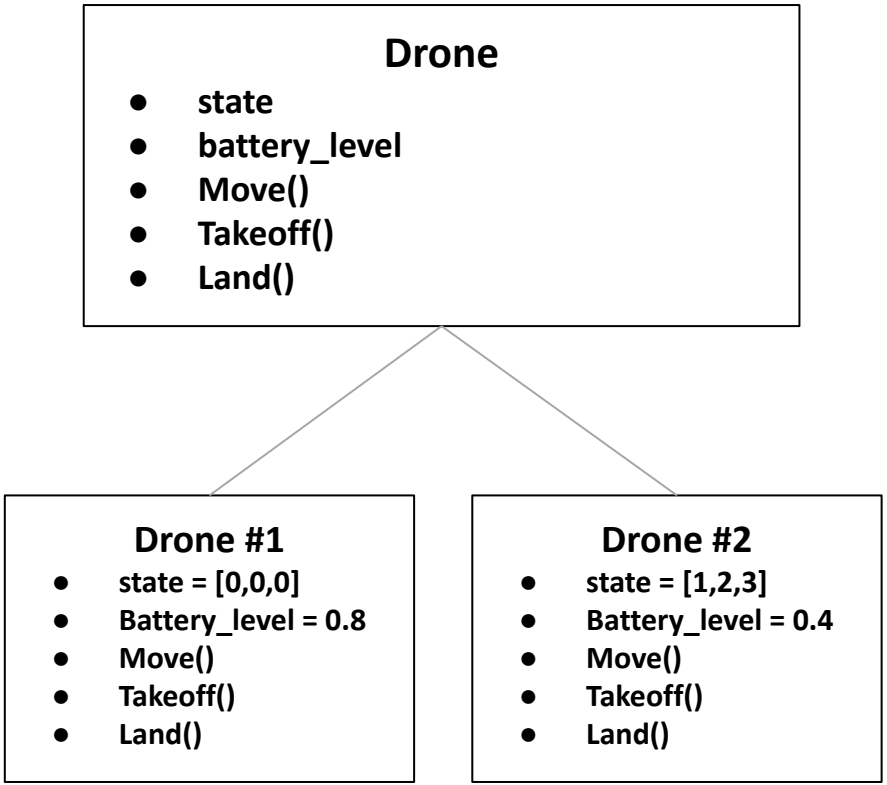
- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")





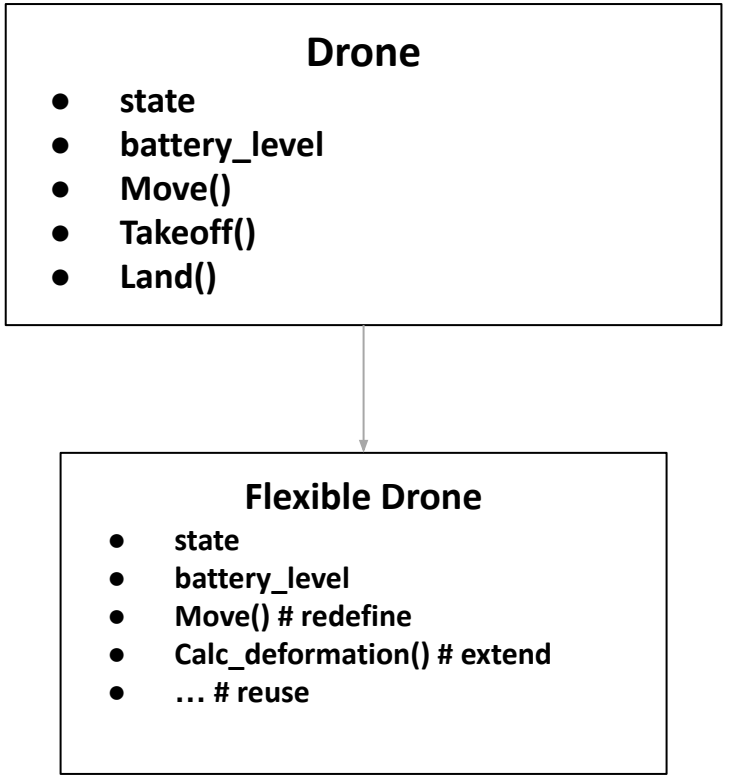
Core Principles of OOP

- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")



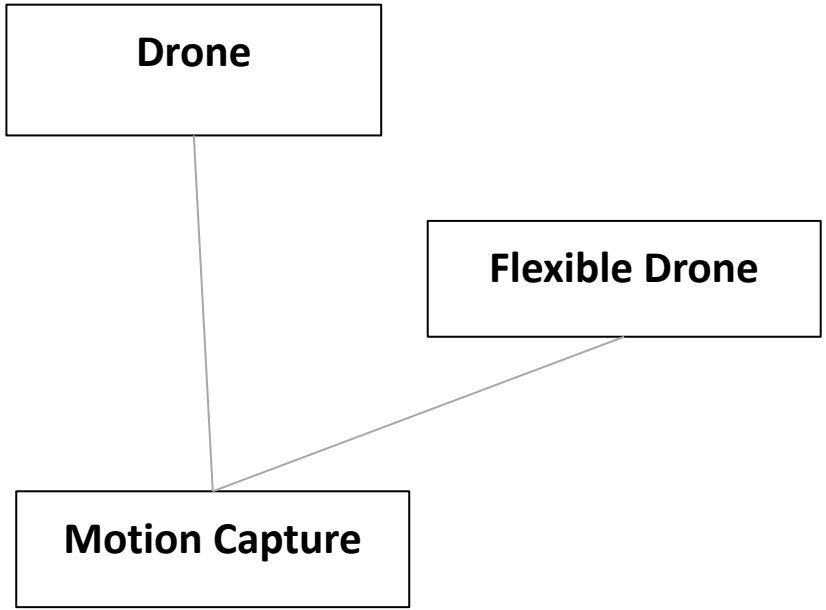
Core Principles of OOP

- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")



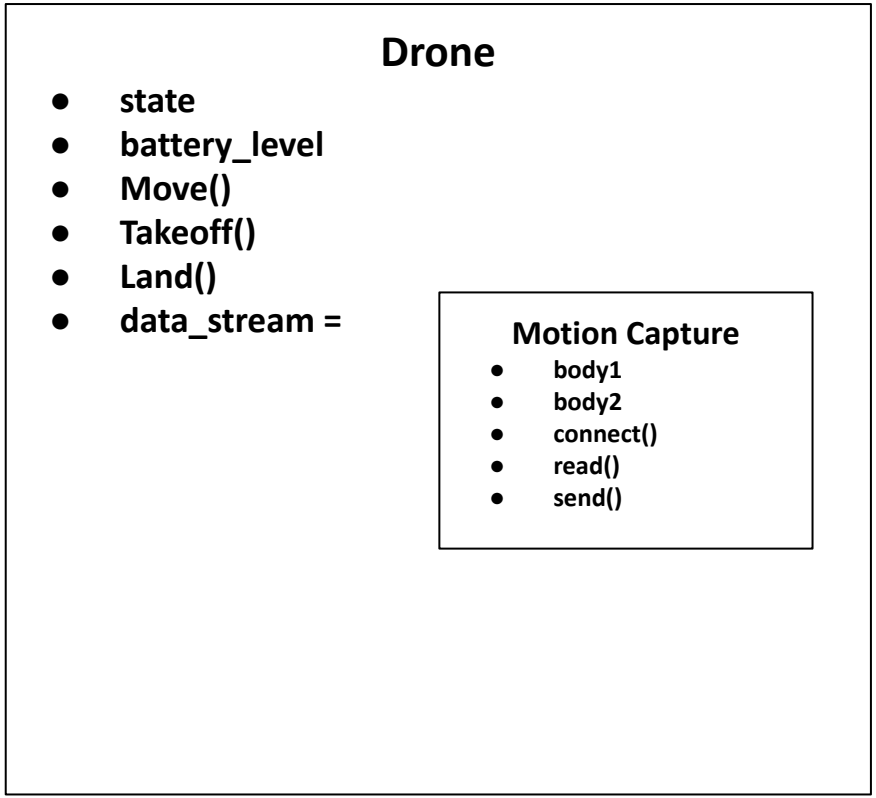
Core Principles of OOP

- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")



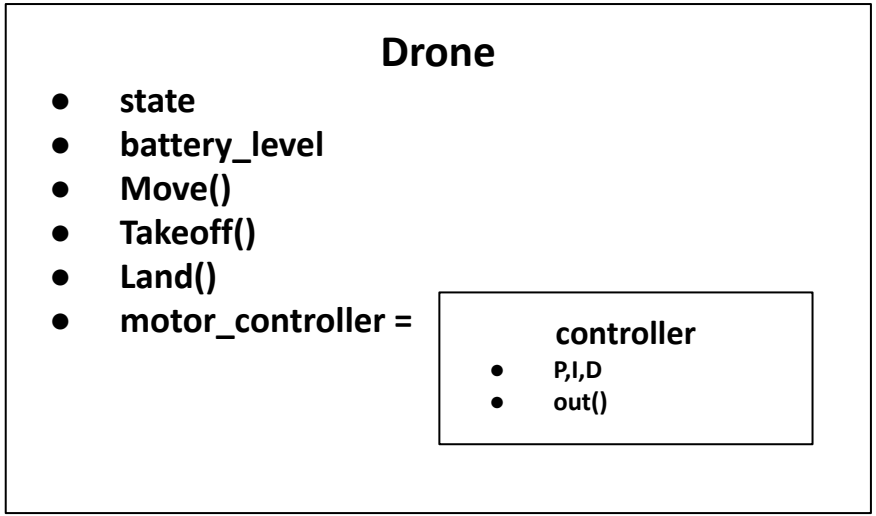
Core Principles of OOP

- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes ("is a …")
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures ("contains a…")
- **Composition**: allows nested dependent object structures ("composed of…")



Core Principles of OOP

- **Abstraction**: hide unnecessary information from user
- **Encapsulation**: grouping data and methods in containers
- **Polymorphism**: access unique objects with same interface (e.g., state)
- **Inheritance**: allows properties and behaviors of other classes (“is a …”)
- **Association**: manage complex peer2peer relationships between objects
- **Aggregation**: allows nested independent object structures (“contains a…”)
- **Composition**: allows nested dependent object structures (“composed of…”)



Signs you should be using OOP

Wrapper

- Large hyper-parameter functions
- Complex functions relating to single theme
- Deploying many of same type of entity

Interpretability & Modularity

- Multiple programmers on same project
- Frequently updated code
- Need code reusability

Signs you should be using OOP

Wrapper

- Large hyper-parameter functions
- Complex functions relating to single theme
- Deploying many of same type of entity

Interpretability & Modularity

- Multiple programmers on same project
- Frequently updated code
- Need code reusability

My rule of thumb:

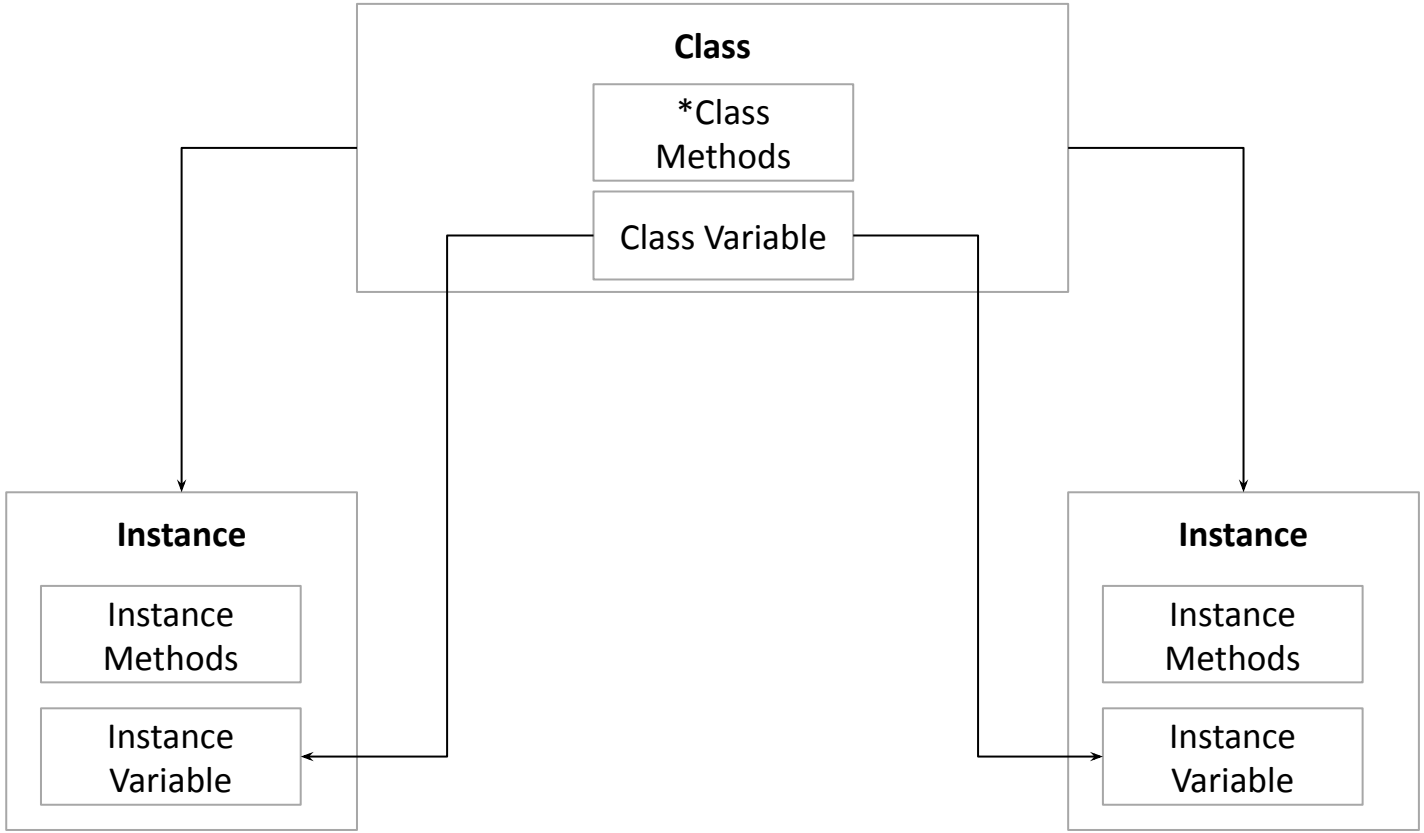
- OOP: When you have fixed set of operations and want to add new things
- FP: When you have fixed set of things and want to add new operations

Agenda

Object Oriented Programming

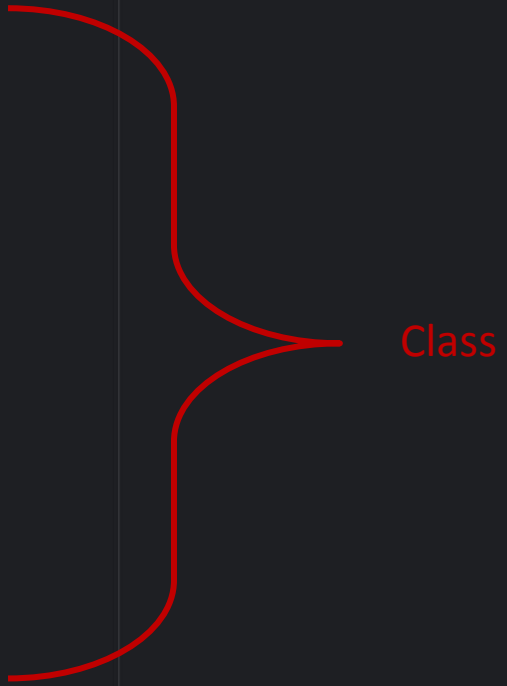
1. Learning objectives
2. Basics of OOP
   - 2.1. Core Principles
   - 2.2. OOP vs FP
3. OOP Structure
   - 3.1. Component Definitions
   - 3.2. Instance vs Class Variables
4. Magic Methods
5. Decorators
6. Inheritance
7. Review & Discussion

Class vs Instance



OOP Structure

```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                        # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)
    print(f'DroneObject.instance_cnt = {DroneObject.instance_cnt}')
    print(f'drone1.instance_cnt      = {drone1.instance_cnt}')
    print(f'drone2.instance_cnt      = {drone2.instance_cnt}')
    print(f'DroneObject.disp_overview()... {drone2.instance_cnt}')
```



OOP Structure

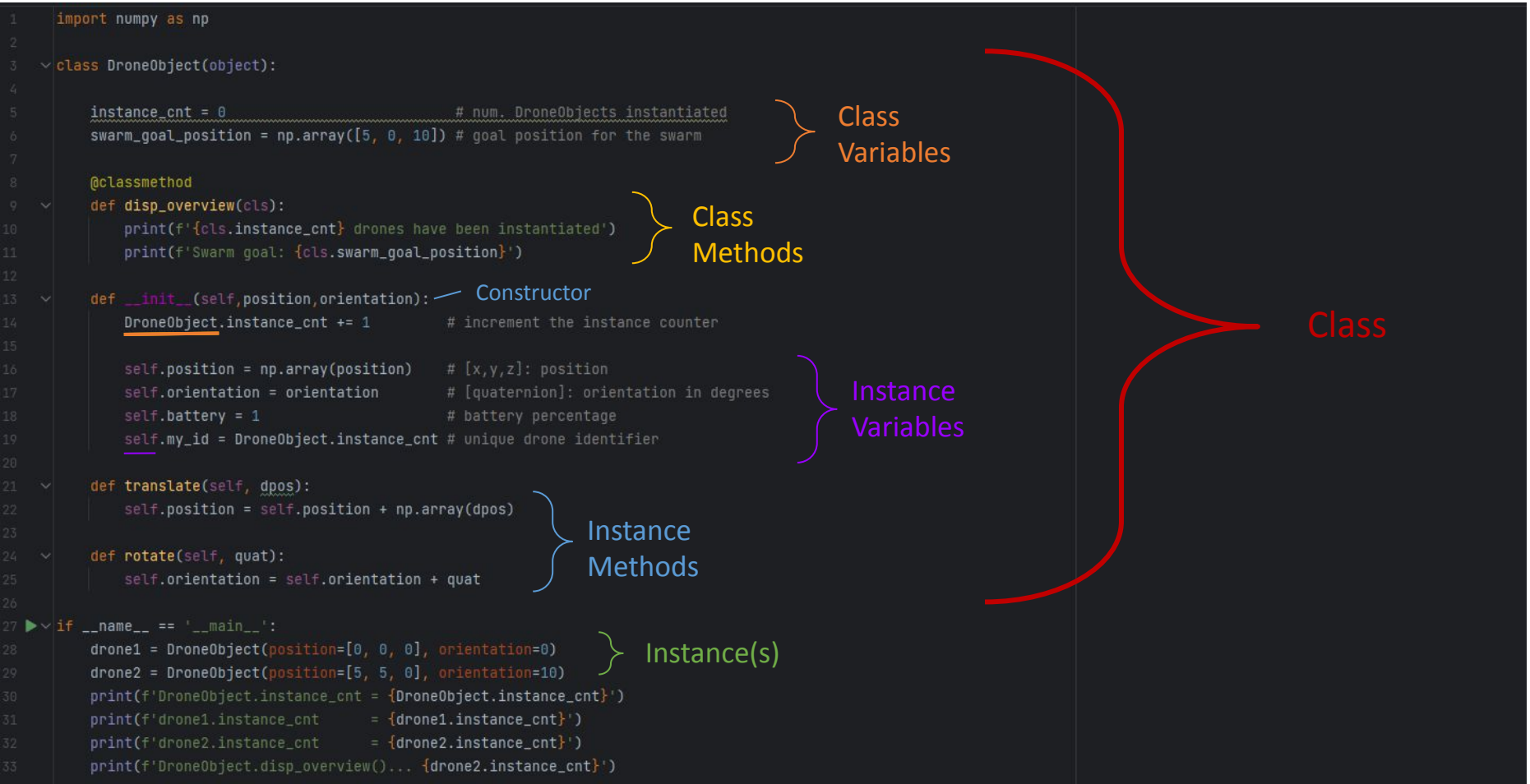
```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                          # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)
    print(f'DroneObject.instance_cnt = {DroneObject.instance_cnt}')
    print(f'drone1.instance_cnt      = {drone1.instance_cnt}')
    print(f'drone2.instance_cnt      = {drone2.instance_cnt}')
    print(f'DroneObject.disp_overview()... {drone2.instance_cnt}')
```

Class Variables

Class Methods

Class

OOP Structure



OOP Structure

```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                          # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)
    print(f'DroneObject.instance_cnt = {DroneObject.instance_cnt}')
    print(f'drone1.instance_cnt      = {drone1.instance_cnt}')
    print(f'drone2.instance_cnt      = {drone2.instance_cnt}')
    print(f'DroneObject.disp_overview()... {drone2.instance_cnt}')
```

Class Variables

Class Methods

Constructor

Instance Variables

Instance Methods

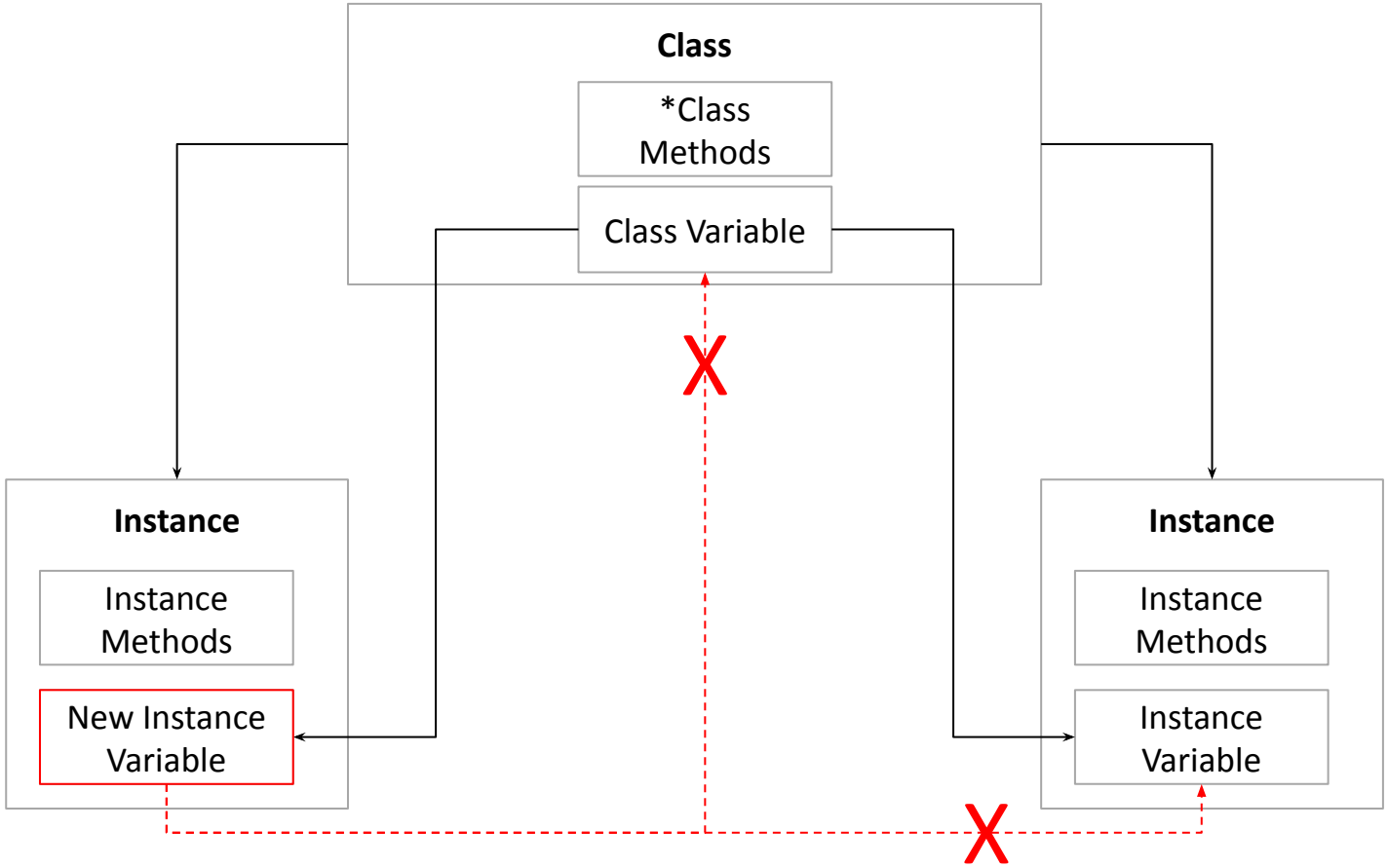
Instance(s)

Class

Result

```
DroneObject.instance_cnt = 2
drone1.instance_cnt      = 2
drone2.instance_cnt      = 2

DroneObject.disp_overview()...
2 drones have been instantiated
Swarm goal: [ 5  0 10]
```

Class vs Instance



Setting Class Variables…

```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                            # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    # Instantiate two drones
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)

    # Change the goal position of the swarm
    drone1.swarm_goal_position = np.array([10, 10, 10])
    print(f'DroneObject.instance_cnt = {DroneObject.swarm_goal_position}')
    print(f'drone1.instance_cnt      = {drone1.swarm_goal_position}')
    print(f'drone2.instance_cnt      = {drone2.swarm_goal_position}')
```

Set class var. by calling object

Result

```
DroneObject.instance_cnt =
drone1.instance_cnt      =
drone2.instance_cnt      =
```

?

Setting Class Variables…

```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                            # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    # Instantiate two drones
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)

    # Change the goal position of the swarm
    drone1.swarm_goal_position = np.array([10, 10, 10])
    print(f'DroneObject.instance_cnt = {DroneObject.swarm_goal_position}')
    print(f'drone1.instance_cnt      = {drone1.swarm_goal_position}')
    print(f'drone2.instance_cnt      = {drone2.swarm_goal_position}')
```

Set class var. by calling object…
ONLY SETS INSTANCE VAR!

Result

```
DroneObject.instance_cnt = [ 5  0 10]
drone1.instance_cnt      = [10 10 10]
drone2.instance_cnt      = [ 5  0 10]
```

Setting Class Variables…

```python
import numpy as np

class DroneObject(object):

    instance_cnt = 0                          # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm

    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')

    def __init__(self,position,orientation):
        DroneObject.instance_cnt += 1         # increment the instance counter
        self.instance_cnt += 1 # BAD!!!

        self.position = np.array(position)    # [x,y,z]: position
        self.orientation = orientation        # [quaternion]: orientation in degrees
        self.battery = 1                      # battery percentage
        self.my_id = DroneObject.instance_cnt # unique drone identifier

    def translate(self, dpos):
        self.position = self.position + np.array(dpos)

    def rotate(self, quat):
        self.orientation = self.orientation + quat

if __name__ == '__main__':
    # Instantiate two drones
    drone1 = DroneObject(position=[0, 0, 0], orientation=0)
    drone2 = DroneObject(position=[5, 5, 0], orientation=10)

    # Change the goal position of the swarm
    # drone1.swarm_goal_position = np.array([10, 10, 10])
    DroneObject.swarm_goal_position = np.array([10, 10, 10])
    print(f'DroneObject.instance_cnt = {DroneObject.swarm_goal_position}')
    print(f'drone1.instance_cnt      = {drone1.swarm_goal_position}')
    print(f'drone2.instance_cnt      = {drone2.swarm_goal_position}')
```

Also Bad

ALWAYS manage class vars with CLASS definition

Result

```
DroneObject.instance_cnt = [ 5  0 10]
drone1.instance_cnt      = [10 10 10]
drone2.instance_cnt      = [ 5  0 10]
```

Correct Result

```
DroneObject.instance_cnt = [10 10 10]
drone1.instance_cnt      = [10 10 10]
drone2.instance_cnt      = [10 10 10]
```

Concept:

Class variables are unidirectional

- Class → instance
- ~~Instance → class~~

Agenda

Object Oriented Programming

1. Learning objectives
2. Basics of OOP
   - 2.1. Core Principles
   - 2.2. OOP vs FP
3. OOP Structure
   - 3.1. Component Definitions
   - 3.2. Instance vs Class Variables
4. Magic Methods
5. Decorators
6. Inheritance
7. Review & Discussion

Python Magic Methods Resource

Magic Methods: Aka __dunder__ methods

```python
class Quaternion:
    """ Quaternion state class handling operations between quaternions [add, subtract, multiply, print]"""

    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def catch_is_my_type(self, other):
        if not isinstance(other, type(self)):  # Quaternion
            raise TypeError(f"unsupported operand type(s) for +: 'Quaternion' and '{type(other).__name__}'")

    def __repr__(self):
        """String representation of the quaternion object (printing)"""
        return f"{self.a} + {self.b}i + {self.c}j + {self.d}k"

    def __round__(self, n):
        return Quaternion(round(self.a, n), round(self.b, n), round(self.c, n), round(self.d, n))

    def __add__(self, other):
        self.catch_is_my_type(other)
        new_quat = Quaternion(self.a + other.a, self.b + other.b, self.c + other.c, self.d + other.d)
        return new_quat

    def __sub__(self, other):
        self.catch_is_my_type(other)
        return Quaternion(self.a - other.a, self.b - other.b, self.c - other.c, self.d - other.d)

    def __mul__(self, other):
        """Hamilton product of two quaternions (multiplication)"""
        self.catch_is_my_type(other)
        a = self.a * other.a - self.b * other.b - self.c * other.c - self.d * other.d
        b = self.a * other.b + self.b * other.a + self.c * other.d - self.d * other.c
        c = self.a * other.c - self.b * other.d + self.b * other.a + self.d * other.b
        d = self.a * other.d + self.b * other.c - self.c * other.b + self.d * other.a
        return Quaternion(a, b, c, d)

    def __truediv__(self, other):
        self.catch_is_my_type(other)
        sq_norm = other.a ** 2 + other.b ** 2 + other.c ** 2 + other.d ** 2
        conj = Quaternion(other.a, -other.b, -other.c, -other.d)
        prod = self * conj
        return Quaternion(prod.a / sq_norm, prod.b / sq_norm, prod.c / sq_norm, prod.d / sq_norm)


    def __iadd__(self, other): return self + other # quat += other
    def __isub__(self, other): return self - other # quat-+= other
    def __imul__(self, other): return self * other # quat -= other
    def __itruediv__(self, other): return self / other # quat /= other
```

In Python…

- EVERYTHING = object
- **Dunder methods**: core methods and variables abstracted away from front-end developers

Uses: ⇒ Easier class interface

- Internal behaviors
  - Controlling Attribute Access
- Behavior with other classes
  - Arithmetic
  - Equating/comparing
  - Pickling
- Behavior with built-in python operations
  - Custom copy/deep copying
  - Context Managers (with Class(‘file’) as f: …)
  - Rounding, hashing, ect…

Magic Methods: Built-in Behaviors

```python
class Quaternion:
    """ Quaternion state class handling operations between quaternions [add, subtract, multiply, print]"""

    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def catch_is_my_type(self, other):
        if not isinstance(other, type(self)):  # Quaternion
            raise TypeError(f"unsupported operand type(s) for +: 'Quaternion' and '{type(other).__name__}'")

    def __repr__(self):
        """String representation of the quaternion object (printing)"""
        return f"{self.a} + {self.b}i + {self.c}j + {self.d}k"

    def __round__(self, n):
        return Quaternion(round(self.a, n), round(self.b, n), round(self.c, n), round(self.d, n))

    def __add__(self, other):
        self.catch_is_my_type(other)
        new_quat = Quaternion(self.a + other.a, self.b + other.b, self.c + other.c, self.d + other.d)
        return new_quat

    def __sub__(self, other):
        self.catch_is_my_type(other)
        return Quaternion(self.a - other.a, self.b - other.b, self.c - other.c, self.d - other.d)

    def __mul__(self, other):
        """Hamilton product of two quaternions (multiplication)"""
        self.catch_is_my_type(other)
        a = self.a * other.a - self.b * other.b - self.c * other.c - self.d * other.d
        b = self.a * other.b + self.b * other.a + self.c * other.d - self.d * other.c
        c = self.a * other.c - self.b * other.d + self.b * other.a + self.d * other.b
        d = self.a * other.d + self.b * other.c - self.c * other.b + self.d * other.a
        return Quaternion(a, b, c, d)

    def __truediv__(self, other):
        self.catch_is_my_type(other)
        sq_norm = other.a ** 2 + other.b ** 2 + other.c ** 2 + other.d ** 2
        conj = Quaternion(other.a, -other.b, -other.c, -other.d)
        prod = self * conj
        return Quaternion(prod.a / sq_norm, prod.b / sq_norm, prod.c / sq_norm, prod.d / sq_norm)


    def __iadd__(self, other): return self + other # quat += other
    def __isub__(self, other): return self - other # quat-+= other
    def __imul__(self, other): return self * other # quat -= other
    def __itruediv__(self, other): return self / other # quat /= other
```

```python
def __repr__(self):
```

```
q1 = Quaternion(1, 2, 3, 4)
print(q1)
Without __repr__: <__main__.Quaternion object at 0x00000264C2AA54D0>
With __repr__:    1 + 2i + 3j + 4k
```

```python
def __round__(self, n):
```

```
q1 = Quaternion(0.56894, 1e-10, 1500, 0)
print(round(q1,2))
Traceback (most recent call last):
  File "C:\PycharmProjects\OOP_workshop\magic_methods.py", line 64, in <module>
    print(f'Without __round__: {round(q1,2)}')
                                ^^^^^^^^^^
With __round__:    0.57 + 0.0i + 1500j + 0k
```

Other Built-in Behaviors:

- Indexing: obj[i]
- Hashing: hash(obj)
- Iter: for x in obj: ...

Magic Methods: Arithmetic Operators

```python
class Quaternion:
    """ Quaternion state class handling operations between quaternions [add, subtract, multiply, print]"""

    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def catch_is_my_type(self, other):
        if not isinstance(other, type(self)):  # Quaternion
            raise TypeError(f"unsupported operand type(s) for +: 'Quaternion' and '{type(other).__name__}'")

    def __repr__(self):
        """String representation of the quaternion object (printing)"""
        return f"{self.a} + {self.b}i + {self.c}j + {self.d}k"

    def __round__(self, n):
        return Quaternion(round(self.a, n), round(self.b, n), round(self.c, n), round(self.d, n))

    def __add__(self, other):
        self.catch_is_my_type(other)
        new_quat = Quaternion(self.a + other.a, self.b + other.b, self.c + other.c, self.d + other.d)
        return new_quat

    def __sub__(self, other):
        self.catch_is_my_type(other)
        return Quaternion(self.a - other.a, self.b - other.b, self.c - other.c, self.d - other.d)

    def __mul__(self, other):
        """Hamilton product of two quaternions (multiplication)"""
        self.catch_is_my_type(other)
        a = self.a * other.a - self.b * other.b - self.c * other.c - self.d * other.d
        b = self.a * other.b + self.b * other.a + self.c * other.d - self.d * other.c
        c = self.a * other.c - self.b * other.d + self.b * other.a + self.d * other.b
        d = self.a * other.d + self.b * other.c - self.c * other.b + self.d * other.a
        return Quaternion(a, b, c, d)

    def __truediv__(self, other):
        self.catch_is_my_type(other)
        sq_norm = other.a ** 2 + other.b ** 2 + other.c ** 2 + other.d ** 2
        conj = Quaternion(other.a, -other.b, -other.c, -other.d)
        prod = self * conj
        return Quaternion(prod.a / sq_norm, prod.b / sq_norm, prod.c / sq_norm, prod.d / sq_norm)


    def __iadd__(self, other): return self + other # quat += other
    def __isub__(self, other): return self - other # quat-+= other
    def __imul__(self, other): return self * other # quat -= other
    def __itruediv__(self, other): return self / other # quat /= other
```

```python
def __add__(self, other):
def __sub__(self, other):
def __mul__(self, other):
def __truediv__(self, other):
```

```python
q1 = Quaternion(1, 2, 3, 4)
q2 = Quaternion(5, 6, 7, 8)
print(f'q1 + q2 = {q1 + q2}') # q1 += q2
print(f'q1 - q2 = {q1 - q2}') # q1 -= q2
print(f'q1 * q2 = {q1 * q2}') # q1 *= q2
print(f'q1 / q2 = {round(q1 / q2,3)}') # q1 /= q2
```

```
q1 + q2 = 6 + 8i + 10j + 12k
q1 - q2 = -4 + -4i + -4j + -4k
q1 * q2 = -60 + 12i + 25j + 24k
q1 / q2 = 0.402 + 0.046i + -0.029j + 0.092k
```

Other Arithmetic Behaviors:

- Unary operators and functions (-q, !q)
- Pow and others
- Bitwise operations (or |, xor ^)
- Augmented assignment (q1 += q2)

An Implementation Example

```python
import numpy as np

class Quaternion:
    def __init__(self, a, b, c, d):...
    def catch_is_my_type(self, other):...
    def __repr__(self):...
    def __round__(self, n):...
    def __add__(self, other):...
    def __sub__(self, other):...
    def __mul__(self, other):...
    def __truediv__(self, other):...
    def __iadd__(self, other): return self + other  # quat += other
    def __isub__(self, other): return self - other  # quat-+= other
    def __imul__(self, other): return self * other  # quat -= other
    def __itruediv__(self, other): return self / other  # quat /= other


class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):...
    def __init__(self,position,orientation):...
    def translate(self, dpos):...
    def rotate(self, quat):
        self.orientation = self.orientation * quat

if __name__ == '__main__':
    q_start = Quaternion(0.7071081, 0, 0, 0.7071055)
    q_rx90 = Quaternion(-0.7071081, 0, 0, 0.7071055)
    drone1 = DroneObject(position=[0, 0, 0], orientation=q_start)
    drone1.rotate(q_rx90)
    print(f'drone1.orientation      = {round(drone1.orientation, 2)}')
```

- Never have to touch the quaternion code again!

Summary

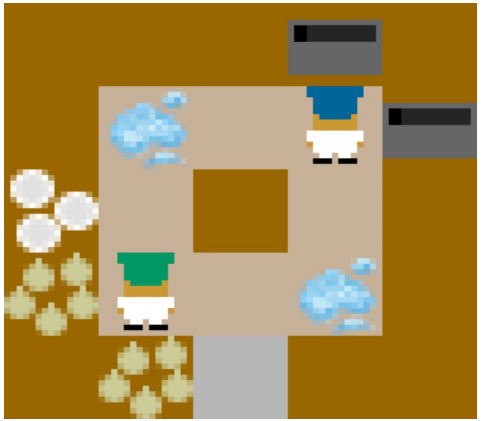
- Easy and intuitive interface
- Debugging to track down bad assignments/error checks
- Reduce errors

```
drone1.orientation      = -1.0 + 0.0i + 0.0j + 0.0k
```

… can get hairy with some methods

Another Example (equating)

```python
class OvercookedState(object):
    """A state in OvercookedGridworld."""

    ...

    def time_independent_equal(self, other):
        order_lists_equal = (
            self.all_orders == other.all_orders
            and self.bonus_orders == other.bonus_orders
        )

        return (
            isinstance(other, OvercookedState)
            and self.players == other.players
            and set(self.objects.items()) == set(other.objects.items())
            and order_lists_equal
        )

    def __eq__(self, other):
        return (
            self.time_independent_equal(other)

        )

    def __hash__(self):
        # NOTE: hash doesn't take into account timestep
        order_list_hash = hash(tuple(self.bonus_orders)) + hash(
            tuple(self.all_orders)
        )
        return hash(
            (self.players, tuple(self.objects.values()), order_list_hash)
        )
```



- if state == state:
  - state.players.pos
  - state.players.or
  - state.objects.onion[i].pos
  - state.objects.dish[i].pos
  - state.objects.soup[i].pos
  - state.objects.soup[i].n_ingred
  - state.objects.soup[i].cook_tic
  - state.orders
  - state.timestep

Use "`__eq__`"

Agenda

Object Oriented Programming

1. Learning objectives
2. Basics of OOP
   - 2.1. Core Principles
   - 2.2. OOP vs FP
3. OOP Structure
   - 3.1. Component Definitions
   - 3.2. Instance vs Class Variables
4. Magic Methods
5. Decorators
6. Inheritance

Python Decorator Resource

What's a Decorator?

- Already showed one?

What's a Decorator?

- Already showed one?

@Decorator

```
class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

What's a Decorator?

- Already showed one?
- What does it do?

@Decorator

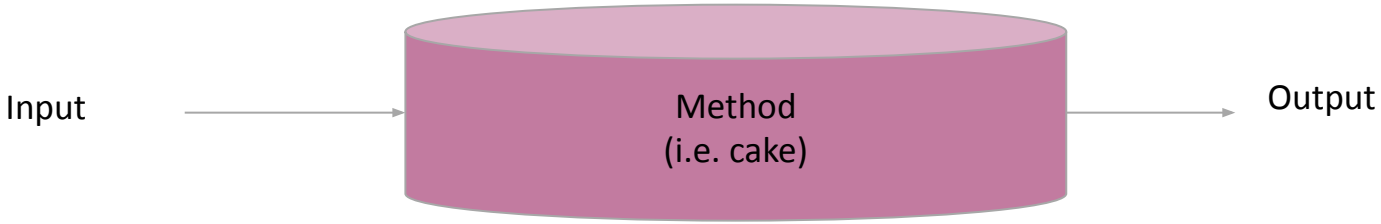
```
class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

What's a Decorator?

- Already showed one?
- What does it do?

@Decorator

```python
class DroneObject(object):
    instance_cnt = 0                            # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

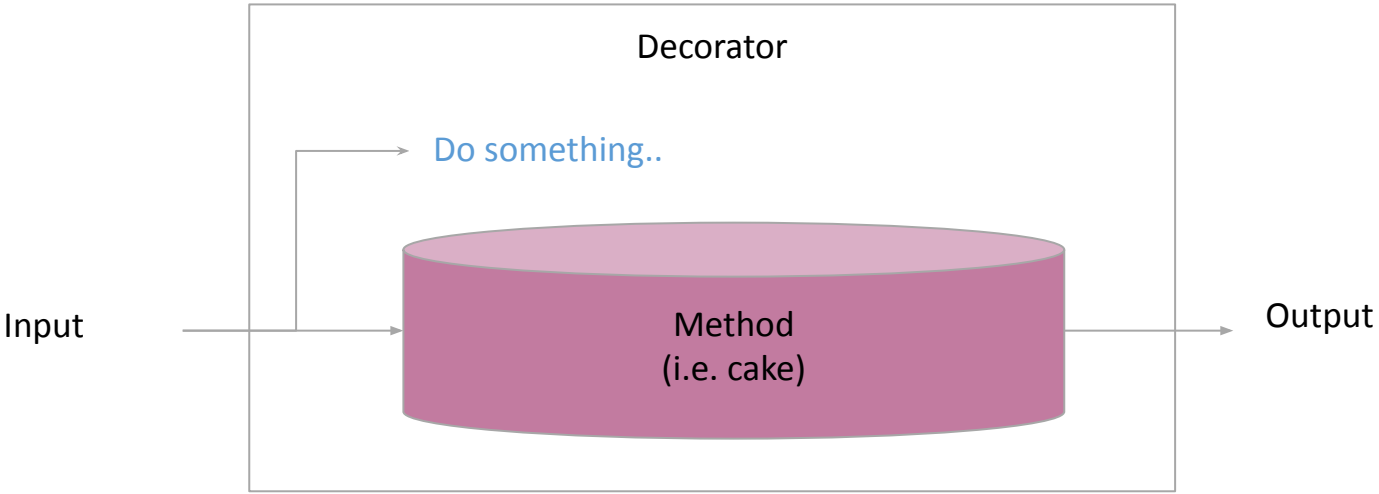


What's a Decorator?

- Already showed one?
- What does it do?

@Decorator

```
class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

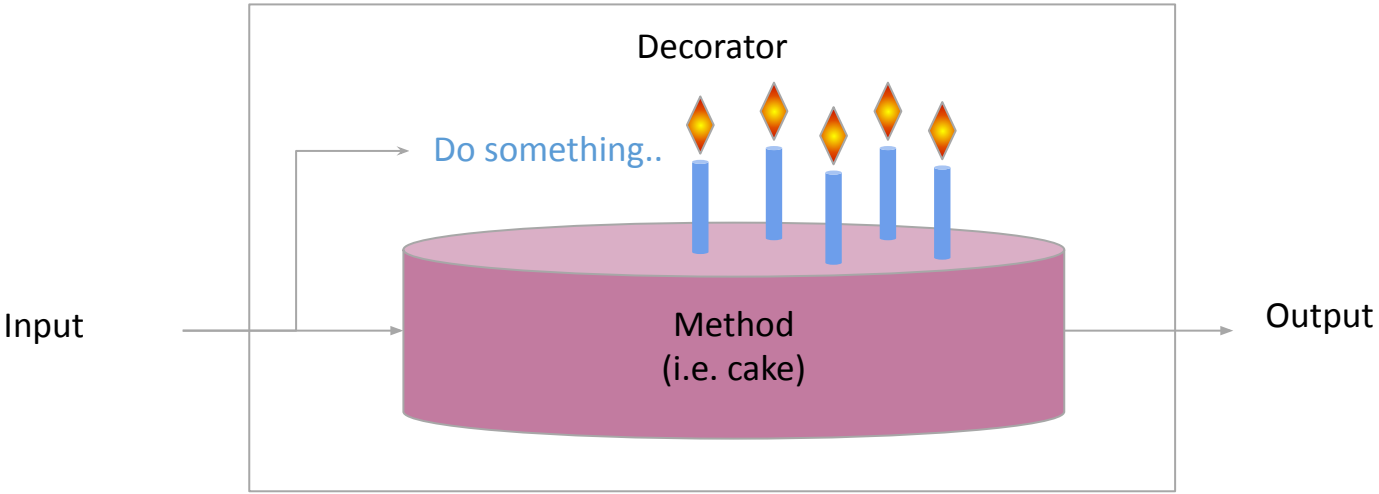


What's a Decorator?

- Already showed one?
- What does it do?

@Decorator

```
class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```



What's a Decorator?

@classmethod → redirects method object to parent class (not instance) properties

More commonly used for alternate constructors
`DroneObject.from_file('fname.npz')`

```python
class DroneObject(object):
    instance_cnt = 0                                # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

```python
class DroneObject(object):
    instance_cnt = 0                                # num. Droned
    swarm_goal_position = np.array([5, 0, 10]) # goal positi

    @classmethod
    def from_file(cls,filename):
        loaded_data = np.load(filename)
        return cls(loaded_data['position'], loaded_data['orientation'])
```

Tip:
use **`cls`** instead of class name (`DroneObject`) within `from_file()` for easy inheritance

What's a Decorator?

@classmethod → redirects method object to parent class (not instance) properties

More commonly used for alternate constructors
DroneObject.from_file('fname.npz')

```
class DroneObject(object):
    instance_cnt = 0                              # num. DroneObjects instantiated
    swarm_goal_position = np.array([5, 0, 10]) # goal position for the swarm
    @classmethod
    def disp_overview(cls):
        print(f'{cls.instance_cnt} drones have been instantiated')
        print(f'Swarm goal: {cls.swarm_goal_position}')
```

Useful decorators

- @classmethod: adds method to class properties instead of instance
- @dataclass: easier data management (adds magic methods/mutable)
- @property: makes method behave as variable (getter)
- @name.setter: defines method for setting private variables (cls._x) from a public @propery (cls.x ⇐ self.x())
- @typing.final: throws error when trying to override or inherit method
- @atexit.register: calls method when program shuts down (cleanup)

Example: @property

```
class Quaternion:
    """ Quaternion state class handling operations between quaternions [add, subtract, multiply, print]"""
    def __init__(self, a, b, c, d) -> object:...
    def catch_is_my_type(self, other):...
    def __repr__(self):...
    def __round__(self, n):...
    def __add__(self, other):...
    def __sub__(self, other):...
    def __mul__(self, other):...
    def __truediv__(self, other):...
    def __iadd__(self, other): return self + other # quat += other
    def __isub__(self, other): return self - other # quat-+= other
    def __imul__(self, other): return self * other # quat -= other
    def __itruediv__(self, other): return self / other # quat /= other

    @property
    def in_euler(self):
        t0 = +2.0 * (self.d * self.a + self.b * self.c)
        t1 = +1.0 - 2.0 * (self.a * self.a + self.b * self.b)
        roll_x = math.atan2(t0, t1)
        t2 = +2.0 * (self.d * self.b - self.c * self.a)
        t2 = +1.0 if t2 > +1.0 else t2
        t2 = -1.0 if t2 < -1.0 else t2
        pitch_y = math.asin(t2)
        t3 = +2.0 * (self.d * self.c + self.a * self.b)
        t4 = +1.0 - 2.0 * (self.b * self.b + self.c * self.c)
        yaw_z = math.atan2(t3, t4)

        return roll_x, pitch_y, yaw_z  # in radians

if __name__ == '__main__':
    q = Quaternion(1,0,0,0)
    print(f'Quaternion [{q}] = {q.in_euler}')
```

```
Quaternion [1 + 0i + 0j + 0k] = (3.141592653589793, 0.0, 0.0)
```

Other getter uses:

- “Insulate” code
- Guard mutable objects
- Hide variables

Example: @property + @name.setter

```
class Quaternion:
    """ Quaternion state class handling operations between quaternions [add, subtract, multiply, print
    def __init__(self, a, b, c, d):
        self._a = a # private instance variables
        self._b = b # private instance variables
        self._c = c # private instance variables
        self._d = d # private instance variables
    def catch_is_my_type(self, other):...
    def __repr__(self):...
    def __round__(self, n):...
    def __add__(self, other):...
    def __sub__(self, other):...
    def __mul__(self, other):...
    def __truediv__(self, other):...
    def __iadd__(self, other): return self + other # quat += other
    def __isub__(self, other): return self - other # quat-+= other
    def __imul__(self, other): return self * other # quat -= other
    def __itruediv__(self, other): return self / other # quat /= other


    @property
    def b(self):
        return copy.deepcopy(self._b) # if _b is mutable?
```

```
Quaternion = 1 + 1i + 0j + 0k
Traceback (most recent call last):
  File "C:\PycharmProjects\OOP_workshop\decorators.py", line 78, in <module>
    q.b = '1'; print(f'Quaternion = {q}')
    ^^^
  File "C:\PycharmProjects\OOP_workshop\decorators.py", line 70, in b
    assert isinstance(value, (int, float)), f"Expected int or float, got {type(value).__name__}"
AssertionError: Expected int or float, got str

Traceback (most recent call last):
  File "C:\PycharmProjects\OOP_workshop\decorators.py", line 79, in <module>
    q.b = 5;  print(f'Quaternion = {q}')
    ^^^
  File "C:\PycharmProjects\OOP_workshop\decorators.py", line 71, in b
    assert 0 <= value <= 1, f"Expected b in [0,1], got {value}"
           ^^^^^^^^^^^^^^^
AssertionError: Expected b in [0,1], got 5
```

Other setter uses:

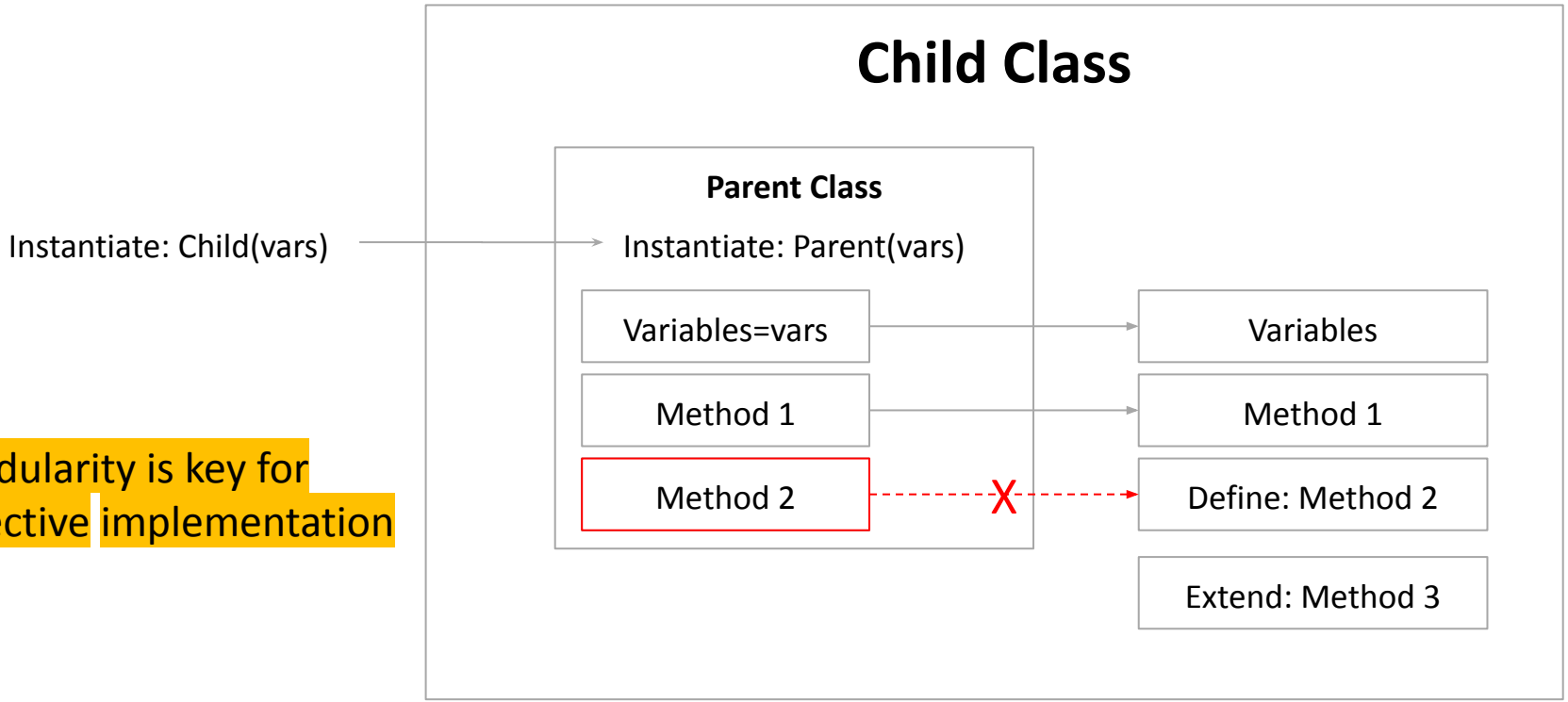
- "Insulate" code
- Intercept errors
- Formatting/type checking
- Hidden variable interface

Agenda

Object Oriented Programming

1. Learning objectives
2. Basics of OOP
   - 2.1. Core Principles
   - 2.2. OOP vs FP
3. OOP Structure
   - 3.1. Component Definitions
   - 3.2. Instance vs Class Variables
4. Magic Methods
5. Decorators
6. Inheritance
7. Review & Discussion

Inheritance Structure



Modularity is key for effective implementation

Example from my work

```
class Rational_SelfPlay_Agents(object):
    def __init__(self, obs_shape, n_actions, config, **kwargs):...

    def update_checkpoint(self):...

    def save_checkpoint(self,PATH):...
    ##################################################
    ## Memory ########################################
    ##################################################
    def memory_double_push(self, state, action, rewards, next_prospects, done):...

    def memory_sample(self):...

    @property
    def memory_len(self):...

    ##################################################
    # Self-Play Utils ###############################
    ##################################################

    def invert_prospect(self, prospects):...
    def invert_obs(self, obs_batch):...

    def invert_joint_action(self, action_batch):...

    ##################################################
    # Nash Utils ####################################
    ##################################################

    def get_normal_form_game(self, obs, use_target=False):...

    def level_k_qunatal(self,nf_games, sophistication=8, belief_trick=True,scaling=False):...

    def compute_EQ(self, NF_Games, update=False):...

    def choose_joint_action(self, obs, epsilon=0.0, feasible_JAs= None, debug=False):...

    ##################################################
    # Update Utils ##################################
    ##################################################
    def update(self):...

    def prospect_value_expectations(self,reward,done,prospect_masks,prospect_next_q_values,prospect_p_next_states):...

    def flatten_next_prospects(self,next_prospects):...

    def update_target(self):...
```

300+ lines

Must make minor modification to this method

Example: Copy and Paste Class?

300+ lines

Ctrl + C

300+ lines

Example: Copy and Paste Class?

One month of testing later…

Example: Copy and Paste Class?

```
class Rational_SelfPlay_Agents(object):
    def __init__(self, obs_shape, n_actions, config, **kwargs):...

    def update_checkpoint(self):...

    def save_checkpoint(self,PATH):...
    ##################################################
    ## Memory ########################################
    ##################################################
    def memory_double_push(self, state, action, rewards, next_prospects, do...

    def memory_sample(self):...

    @property
    def memory_len(self):...

    ##################################################
    # Self-Play Utils ###############################
    ##################################################

    def invert_prospect(self, prospects):...
    def invert_obs(self, obs_batch):...

    def invert_joint_action(self, action_batch):...

    ##################################################
    # Nash Utils ####################################
    ##################################################

    def get_normal_form_game(self, obs, use_target=False):...

    def level_k_qunatal(self,nf_games, sophistication=8, belief_trick=True,scaling=False):...

    def compute_EQ(self, NF_Games, update=False):...

    def choose_joint_action(self, obs, epsilon=0.0, feasible_JAs= None, debug=False):...

    ##################################################
    # Update Utils ##################################
    ##################################################
    def update(self):...

    def prospect_value_expectations(self,reward,done,prospect_masks,prospect_next_q_values,prospect_p_next_states):...

    def flatten_next_prospects(self,next_prospects):...

    def update_target(self):...
```

Opps,

Boom!

A hard to find error

```
class Biased_SelfPlay_Agents(object):
    def __init__(self, obs_shape, n_actions, config, **kwargs):...

    def update_checkpoint(self):...

    def save_checkpoint(self,PATH):...
    ##################################################
    ## Memory ########################################
    ##################################################
    def memory_double_push(self, state, action, rewards, next_prospects, done):...

    def memory_sample(self):...

    len(self):...

    ##################################################
    Utils ##########################################
    ##################################################

    elf, prospects):...
    f, obs_batch):...

    oint_action(self, action_batch):...

    ##################################################
    ##################################################
    ##################################################

    normal_form_game(self, obs, use_target=False):...

    k_qunatal(self,nf_games, sophistication=8, belief_trick=True,scaling=False):...

    _EQ(self, NF_Games, update=False):...

    def choo  oint_action(self, obs, epsilon=0.0, feasible_JAs= None, debug=False):...

    ##################################################
    # Update Utils ##################################
    ##################################################
    def update(self):...

    # Modified..
    def prospect_value_expectations(self,reward,done,prospect_masks,prospect_next_q_values,prospect_p_next_states):...

    def flatten_next_prospects(self,next_prospects):...

    def update_target(self):...
```

Gets excited that you finally found the issue and forgets to update other class…

Example: A better approach…

```
class Biased_SelfPlay_Agents(Rational_SelfPlay_Agents):
    def __init__(self, obs_shape, n_actions, config, **kwargs):
        super().__init__(obs_shape, n_actions, config, **kwargs)
        self.CPT = CumulativeProspectTheory(**config['cpt_params'])

    def prospect_value_expectations(self,reward,done,prospect_masks, prospect_next_q_values,prospect_p_next_states,  debug=True):...
```

Instead inherit!

15 lines

- Any update to parent → child
- Cleaner/easier to read code ⇒ avoids errors
- Strategy:
  - Build baseline algorithm with class
  - Inherit and make tweek

Inheritance

- Code reuse
- Extensibility
- Reliability
- Modularity

Thanks!

Mason Smith
mosmith3@asu.edu

Questions?