Python
WEEK 5
OOP Continued

AI Academy

COMPUTER PROGRAMMING WITH PYTHON

Instructor - James E. Robinson, III

Teaching Assistant - Travis Martin

LIGHTNING REVIEW

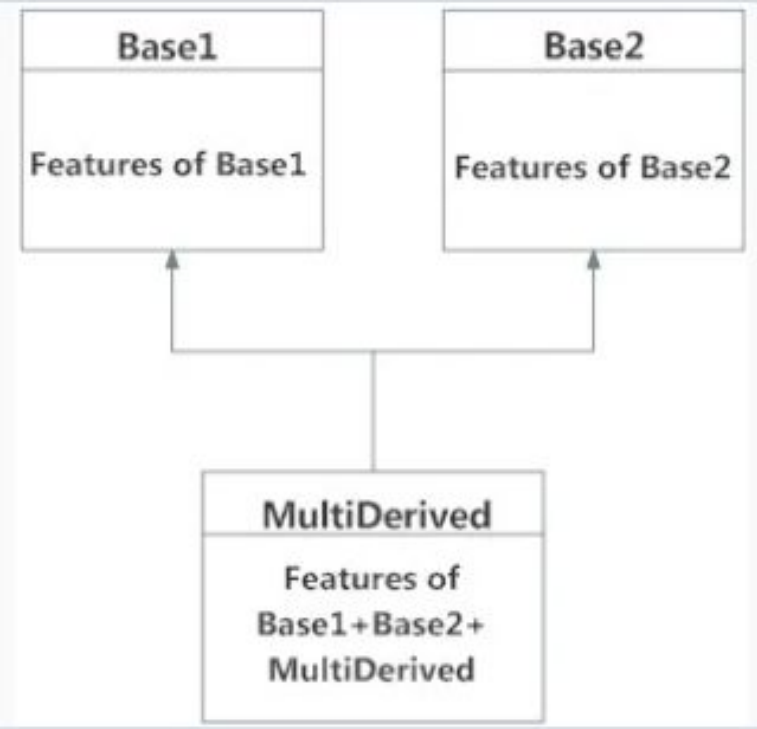
- Variables
- Input / Output
- Expressions
- Functions
- Conditional Control
- Looping
- Data Types
- Logging
- Functions
- Scope
- Decorators
- Recursion
- Dynamic Prg
- Exceptions
- Classes
- Objects
- Encapsulation
- Public v/s Private
- Dunder Methods
- Instances
- Inheritance

TOPICS COVERED

- Types of Inheritance
  - Multiple Inheritance
  - Multilevel Inheritance
  - Method Resolution Order (MRO)
- Polymorphism
  - Method Overriding
- Advanced Object Composition
- Collections of Objects
  - Queue
  - Stack

MULTIPLE INHERITANCE

A CLASS CAN HAVE MORE THAN ONE PARENT



Syntax

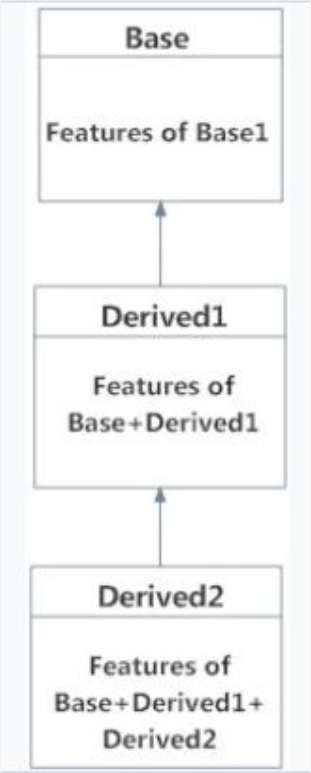
```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

MULTILEVEL INHERITANCE

A PARENT CLASS CAN HAVE PARENTS TOO



Syntax

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

INHERITANCE TYPES

EXAMPLE CODE

```
class Animal():
    def isAnimal(self):
        print("Is an animal")

class Transport():
    def canTravel(self):
        print("Can be used for transport")

class Horse(Animal,Transport):
    def describe(self):
        print("This is a Horse")

class GroundVehicle(Transport):
    def terrain(self):
        print("Travels on ground")
```

```
class Car(GroundVehicle):
    def describe(self):
        print("This is a Car")

h = Horse()
c = Car()

h.describe()
h.isAnimal()
h.canTravel()

c.describe()
c.canTravel()
c.terrain()
```

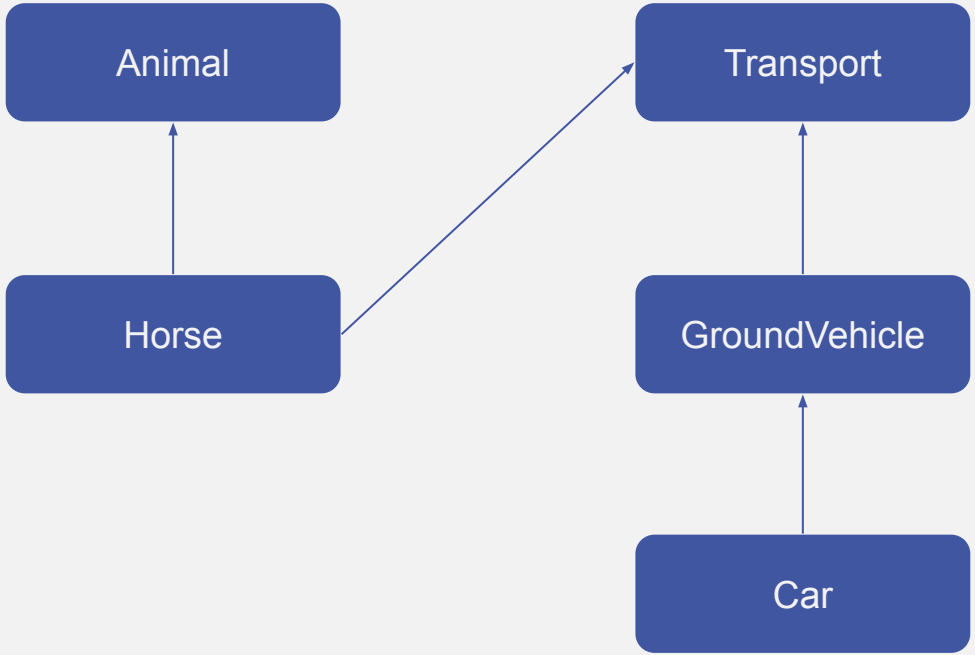
OUTPUT:

```
This is a Horse
Is an animal
Can be used for transport

This is a Car
Can be used for transport
Travels on ground
```

INHERITANCE TYPES

EXAMPLE INHERITANCE DIAGRAM



METHOD RESOLUTION ORDER (MRO)

LINEAR REPRESENTATION OF CLASS HIERARCHY

Any specified entity is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.

Syntax

ClassName.mro()

Note:

It ensures that a class always appears before its parents

In case of multiple parents, the order is the same as tuples of base classes

All base classes are objects of class 'object'

Example

Horse.mro()

OUTPUT:

[__main__.Horse, __main__.Animal, __main__.Transport, object]

Car.mro()

OUTPUT:

[__main__.Car, __main__.Ground, __main__.Transport, object]

POLYMORPHISM

THE ABILITY TO TAKE DIFFERENT FORMS

Example

```
class Rectangle():
    def area(self,h,w):
        return h*w

class Circle():
    def area(self,r):
        return r*r*3.1415

r = Rectangle()
print(r.area(2,50))
c = Circle()
print(c.area(50))
```

OUTPUT:

7853.75

100

Note:

Polymorphism refers to methods with the same name that have different functionalities based on the object that calls the method

Polymorphic methods can have different parameters and return values

POLYMORPHISM - METHOD OVERRIDING

POLYMORPHISM WITH INHERITANCE IN CLASSES

Example

```
class Bird():
    def __init__(self,name):
        self.name = name
    def flight(self):
        print("It Can Fly")

class Eagle(Bird):
    def describe(self):
        print(f"{self.name} is an Eagle")

class Penguin(Bird):
    def describe(self):
        print(f"{self.name} is a Penguin")
    def flight(self):
        print("It Can Not Fly")
```

```
e = Eagle("Bob")
p = Penguin("Sam")
for b in (e,p):
    b.describe()
    b.flight()
```

OUTPUT:

Bob is an Eagle

It Can Fly

Sam is a Penguin

It Can Not Fly

Note:

Polymorphism finds correct version of overridden method depending on the type of object

Using super() is another form of polymorphism

ADVANCED OBJECT COMPOSITION

OBJECTS MAY ALSO HAVE ATTRIBUTES THAT ARE OBJECTS

Instead of multiple inheritance, child objects can also be used as attributes

Example

```
class MyAnimal():
    def __init__(self,given_name):
        self.animal = Tiger(given_name)
        self.name = self.animal.name
    def isTiger(self):
        print(f"{self.name} Is a Tiger")
class Tiger():
    def __init__(self,name):
        self.name = name
    def activity(self):
        print(f"{self.name} Hunts many Preys")
```

```
a = MyAnimal('Carl')
a.isTiger()
a.animal.activity()
```

OUTPUT:

Carl Is a Tiger

Carl Hunts many Preys

Note:

Since Methods are not inherited in such a composition, method calls need to be navigated to the correct class.

Collection of Objects

AI Academy

COLLECTION OF OBJECTS

DATA STRUCTURES TO MAKE USE OF OBJECTS

To efficiently store objects, we require some unique data structures that help in using objects for OOP. Some of these structures are

Queues
A collection of entities that are maintained in a sequence and can be modified by the addition of objects at one end and the removal of objects from the other end of the sequence

Stacks
A collection of entities that are maintained in a sequence and can be modified by the addition and the removal of objects from the same end of the sequence.

QUEUES

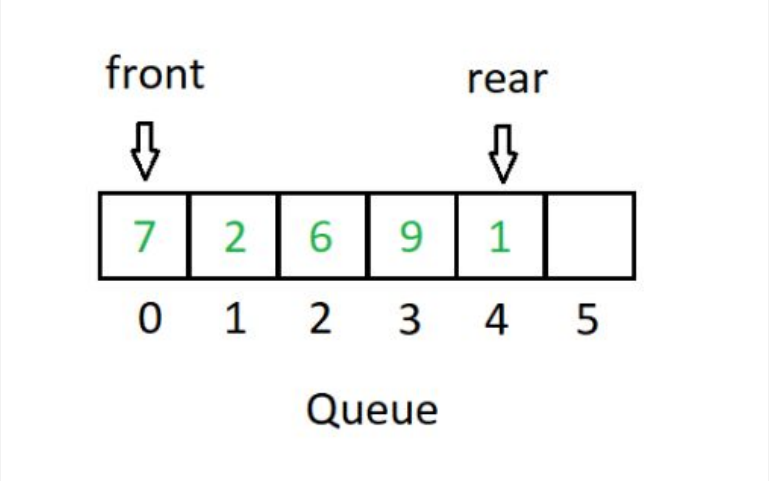
ADD FROM ONE END AND REMOVE FROM OTHER END

FIFO - First In First Out
The object that enters the queue first is the object that can be removed from the queue first.

Operations of Queues

- enqueue - add item
- dequeue - remove item
- front - examine first item, does not remove
- rear - examine last item, does not remove
- size - count of items

Queues are useful for scheduling tasks and buffering



QUEUES

IMPLEMENTATION USING LIST

```
class Queue():
    def __init__(self,elements):
        self.queue = elements
    def enqueue(self,element):
        self.queue.append(element)
    def dequeue(self):
        self.queue.pop(0)
    def size(self):
        print(len(self.queue)
    def front(self):
        print(self.queue[0])
    def rear(self):
        print(self.queue[-1])
```

```
q = Queue([ ])
q.enqueue(5)
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.queue)
q.size()
q.front()
q.rear()
q.dequeue()
print(q.queue)
```

OUTPUT:

[5, 1, 2, 3]

4

5

3

[1, 2, 3]

Note:

Here, class Queue is created to simulate the performance of a queue.

STACKS

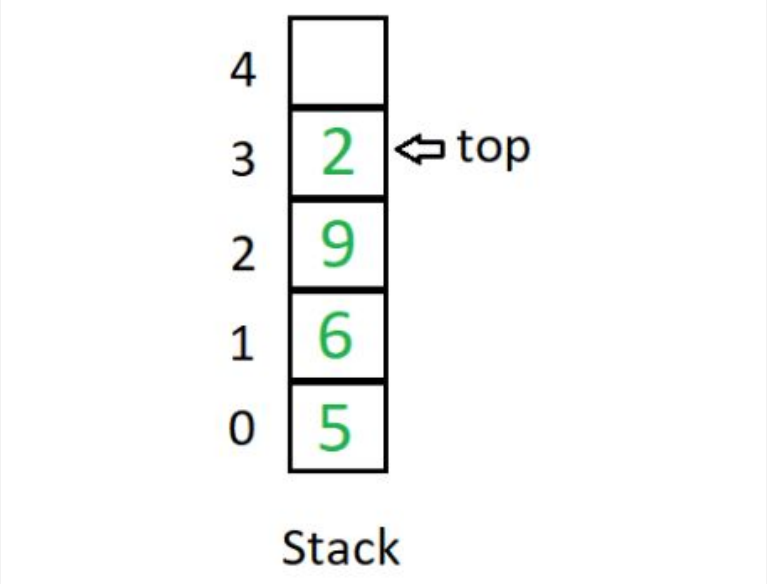
ADD AND REMOVE FROM ONLY ONE END

LIFO - Last In First Out
The object that enters the queue last is the object that can be removed from the queue first.

Operations of Stacks

- push() - add item to top
- pop() - retrieve top item
- peek() or top() - examine top item, does not remove
- size() - count of items

Queues are useful for Undo/Redo tasks and reversals



STACK

IMPLEMENTATION USING LIST

```
class Stack():
    def __init__(self,elements):
        self.stack = elements
    def push(self,element):
        self.stack.append(element)
    def pop(self):
        self.stack.pop()
    def size(self):
        print(len(self.stack))
    def top(self):
        print(self.stack[-1])
```

```
s = Stack([])
s.push(5)
s.push(1)
s.push(2)
s.push(3)
print(s.stack)
s.size()
s.top()
s.pop()
s.pop()
print(s.stack)
```

OUTPUT:

[5, 1, 2, 3]

4

3

[5, 1]

Note:

Here, class Stack is created to simulate the performance of a stack.

WEEK SUMMARY

- Learned more concepts of Object Oriented Programing
- Learned about multiple inheritance and polymorphism
- Learned the concept of MRO
- Learned new data structures - Stacks and Queues
- Learned how to implement a Queue
- Learned how to implement a Stack

THANK YOU

FOR ADDITIONAL QUERIES OR DOUBTS
CONTACT:
jerobins@ncsu.edu

AI Academy