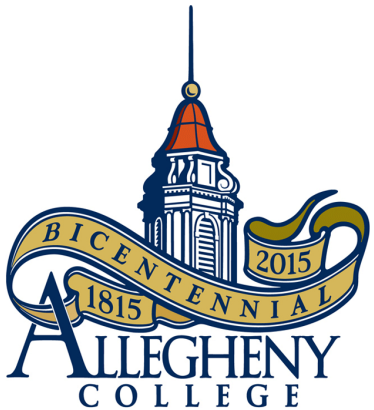
CS101 - Data Abstraction

OOPS - Module 2

Aravind Mohan

Allegheny College

March 16, 2021



Last Week

- Software Goals.
- Classes and Objects.
- Importing external class files.
- Constructor
- Lists

Refer Week3 slides, video, and notes ...

Homework Follow up

Find min, max, and average in a list of exam scores!

```
exam = [80,90,86,79,92,82,94,85,91,92]
total,low,high = 0,0,100
for i in range(len(exam)):
    if (exam[i] < high):
        high = exam[i]
    if (exam[i] > low):
        low = exam[i]
    total += exam[i]

min,max = high,low
avg = total/len(exam)
print(f"min:{min}\tmax:{max}\tavg:{avg}")
```

As a next step, modularize this code into different methods!

Discussion Based On ...

- GT (Goodrich Textbook) Chapter 02 [2.1,2.2,2.4]

An Implementation Of Exception Handling

Divide user provided numbers!

```
first = int(input("Enter first:"))
second = int(input("Enter second:"))
try:
    divide = first/second
    print(divide)
except ZeroDivisionError:
    print ("WARNING: Invalid Equation")
```

PS divide.py in the repo

Inheritance

- Definition: A programming technique or mechanism for creating a hierarchy of classes.
- Automatically parent class methods are available in child class.
- Code redundancy is always a big problem.
- Single, Multilevel, and Multiple inheritance.

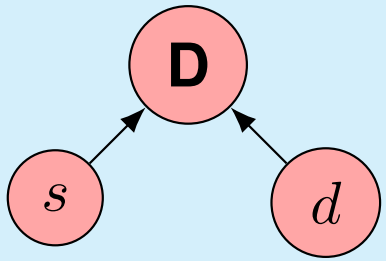
Inheritance Implementation

Single Inheritance

```
class dad():
  d_fname = "Peter"
  d_lname = "Smith"
  d_age = 50

from dad import dad
class daughter(dad):
  dg_fname = "Diana"
  dg_age = 18

from dad import dad
class son(dad):
  s_fname = "Bob"
  s_age = 20
```

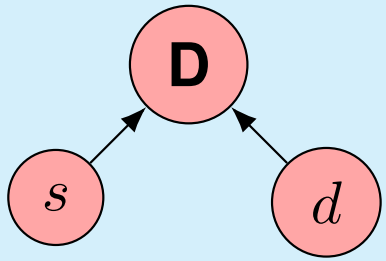


PS oops/single folder in repo.

Inheritance Implementation

Single Inheritance

```
from son import son
from daughter import daughter
s1 = son()
d1 = daughter()
print("Dad: " + s1.d_fname + " " +
            s1.d_lname + " is " +
            str(s1.d_age) + " years old.")
print("Son: " + s1.s_fname + " " +
            s1.d_lname + " is " +
            str(s1.s_age) + " years old.")
print("Daughter: " + d1.dg_fname
            + " " + d1.d_lname + " is " +
            str(d1.dg_age) + " years old.")
```



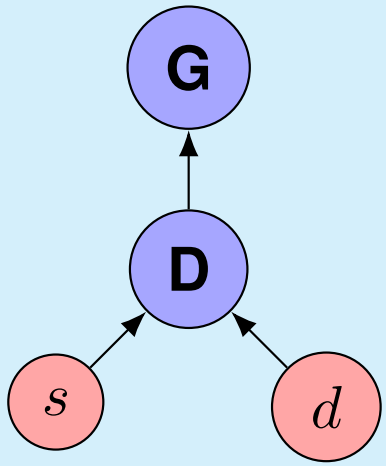
PS oops/single folder in repo.

Inheritance Implementation

Multilevel Inheritance

```
class grandpa():
    g_fname = "Charles"
    g_lname = "Smith"
    g_age = 80

from grandpa import grandpa
class dad(grandpa):
    d_fname = "Peter"
    d_age = 50
```



PS oops/multilevel folder in repo.

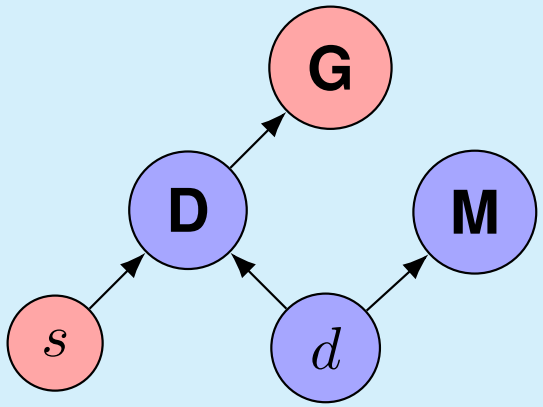
Inheritance Implementation

Multiple Inheritance

```
from grandpa import grandpa
class dad(grandpa):
  d_fname = "Peter"
  d_age = 50

class mom():
  m_fname = "Alice"
  m_lname = "Nicholas"
  m_age = 45

from dad import dad
from mom import mom
class daughter(dad,mom):
  dg_fname = "Diana"
  dg_age = 18
```



PS oops/multiple folder in repo.

Abstract Class

- **Definition** - An abstract class is a template definition of methods and variables of a class.
- An abstract class cannot be instantiated. That is, an object for an abstract class cannot be created.
- It is not required to create abstract classes. But the code is cleaner and makes programming more efficient.
- Avoid looking at extraneous details frequently.
- Need to import **abc** module to use Abstract classes in Python.

Abstract Class

```
from abc import ABC, abstractmethod
class animal(ABC):
  @abstractmethod
  def sound(self):
    pass
class cat(animal):
  def sound(self):
    print("meow")
class lion(animal):
  def sound(self):
    print("roar")
class horse(animal):
  def sound(self):
    print("neigh")
class dog(animal):
  def sound(self):
    print("bark")
cat = cat()
cat.sound()
dog = dog()
dog.sound()
```

PS animals.py in repo

Access Modifiers

- Private - Only the class can access. External classes will get **Attribute Error**.

  Name Mangling is a bad practice.
- Protected - Everything can access except for external classes. (only within inheritance hierarchy). No error, just developer usage.
- Public - Everything can access. The class, any subclasses, any external classes. By default all members of a class are Public in Python.
- It is up to the Developer to use these conventions.

Access Modifiers are not enforced in Python like Java. ...

Encapsulation using Access Modifiers

```
class car:
  def __init__(self,model,odometer,fuel,mpg):
    self.__model = model
    self.__odometer = odometer
    self.__fuel = fuel
    self.__mpg = mpg
```

```
def fuel_tank(self,miles):
    fuel_required = miles/self.__mpg
    if (self.__fuel >= fuel_required):
      return True
    else:
      return False
def fillup(self):
    self.__fuel = 10
```

Encapsulation using Access Modifiers

```
def drive(self,miles):
    if (self.fuel_tank(miles)):
      print(f"trip start:\t{self.__odometer}")
      self.__odometer = self.__odometer + miles
      self.__fuel = self.__fuel - (miles / self.__mpg)
      print(f"trip end:\t{self.__odometer}")
    else:
      print("\tGAS STOP")
      self.fillup()
      self.drive(miles)
```

```
c1 = car("civic",10000,10.34,22.34)
c1.drive(100)
c1.drive(100)
c1.drive(100)
```

PS car.py in repo

Recap of OOPS

- Data hiding - Secure data from external world. Data exposed through a protected manner.
- Abstraction - Hiding internal implementation and highlighting the services.

Encapsulation is data hiding.

Encapsulation - Advantages and Disadvantages

Implemented using private variables in a class and by using **getters** and **setters**.

Advantages of encapsulation are:

1. Security
2. It becomes easier to Enhancement
3. Improves usability
4. Flexible maintainence

Disadvantages

1. Increases length of the code
2. Slows down execution

Polymorphism

- Definition: Polymorphism is ability of an object to appear and behave differently for the same invocation. ex: each car can give different mileage (when driving it)
- Enables ”programming in the general”
- The same invocation can produce ”many forms” of results

Polymorphism Case1 Operator Overloading

```
class patient:
    def __init__(self, inp, outp):
        self.inp = inp
        self.outp = outp
    def __str__(self):
        return str(self.inp) + " " + str(self.outp)
    def __add__(self, addend):
        inp = self.inp + addend.inp
        outp = self.outp + addend.outp
        return patient(inp,outp)
day1 = patient(20,30)
day2 = patient(10,20)
print(day1,day2)
print(day1+day2)
```

PS patient.py in repo

Polymorphism Case2

- Method Overloading: The notion of having two or more methods in the same class with the same name but different arguments.
- Not directly supported in **Python** but can be implemented using multiple options.

Polymorphism Case2

Option-1

```
def add(x,y,z=None):
    if (z == None):
      return x+y
    else:
      return x+y+z
print("add:", add(10,20))
print("add:", add(10,20,30))
```

30,60

Polymorphism Case2

Option-2

```
from multipledispatch import dispatch
@dispatch(int,int)
def add(x,y):
    return x+y
@dispatch(int,int,int)
def add(x,y,z):
    return x+y+z
print("add:", add(10,20))
print("add:", add(10,20,30))
```

30,60

Polymorphism Case3

- Method Overriding: Multiple methods with same signature (name and arguments), but different implementations.
- Parent and child class implementation.

Polymorphism Case3

```
class parent:
    def __init__(self,a,b):
      self.a = a
      self.b = b
    def compute(self):
      return self.a + self.b
```

```
class child(parent):
    def compute(self):
      return self.a - self.b
```

```
obj1 = parent(10,5)
obj2 = child(10,5)
print(obj1.compute(),obj2.compute())
```

15,5

Reading Assignment

- GT (Goodrich Textbook) Chapter 02 [2.1,2.2,2.4]

Questions?

Please ask if there are any Questions!