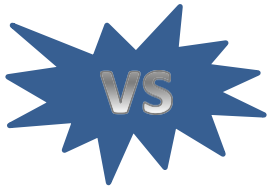
Object Oriented Programming

Lecture 01

Introduction to Object Oriented Programming

Computing Paradigms

Procedural Programming  **Object Oriented Programming**

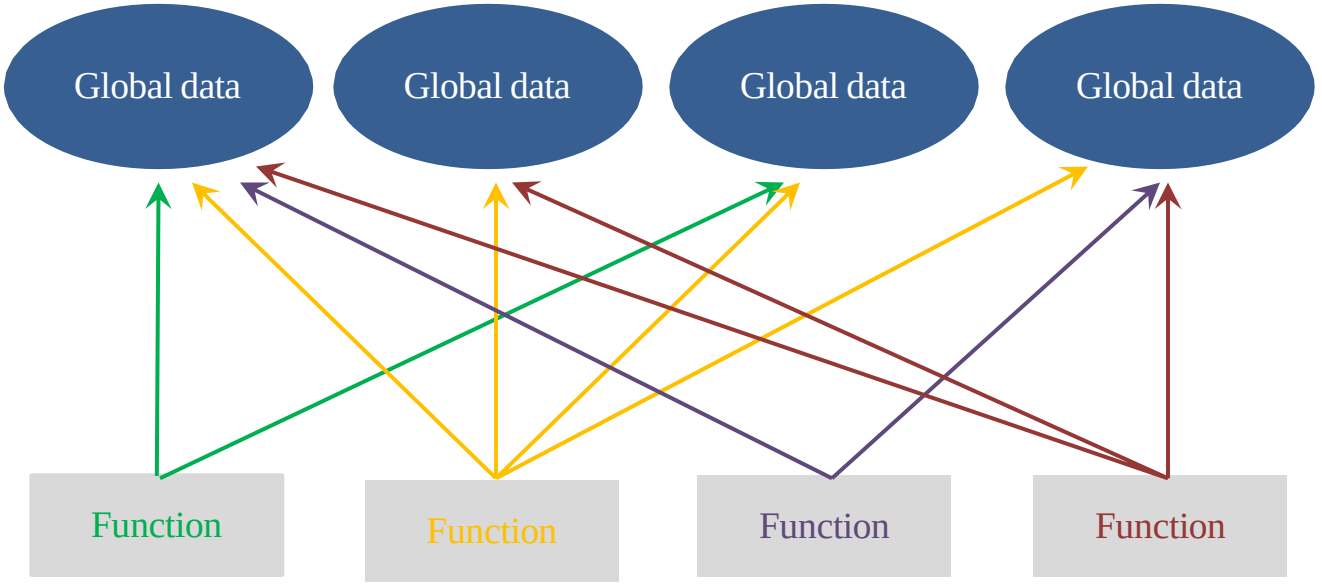
Procedural Programming

- A program is a list of instructions
- Consists of data stored in some suitable structures and functions that operate on data
- Top-down approach - focuses on the main goal of a program first, and then identifies the smaller components that will solve the main goal
- Procedures/functions dominate over data
- Programming languages examples: Fortran, BASIC, Pascal, C

- Easy for small programs
  - simple list of instructions carried out by the computer
- For larger programs
  - divide your code into functions to make it more readable and easy to handle

Computing Paradigms – **Procedural Programming**

- However for very large programs
  - the code becomes difficult to conceptualize and handle, even with functions
  - unrestricted access to global data and functional interdependencies make the code difficult to modify, often results in unintended consequences



- real world objects have both data and functionalities, procedural code fails to keep both attributes together

Computing Paradigms – Procedural Programming

Example:
My Bake Shop

Functions:

- Add a number of items to stock
- Show number of items in stock
- Take order, calculate bill and adjust stock
- Set employee wage rates
- Add hours worked
- Calculate payment
- Calculate bonus
- Some boutique functions ….

Variables:

- samosas_in_stock
- rolls_in_stock
- samosa_unit_price
- roll_unit_price
- emp_A_wage_rate
- emp_B_wage_rate
- emp_A_hours
- emp_B_hours
- total_sales
- Some boutique variables …

Computing Paradigms

Object Oriented Programming

- All computations are carried out using objects
- An object is defined by two components: data in the form of attributes/properties and behaviors specified by methods.

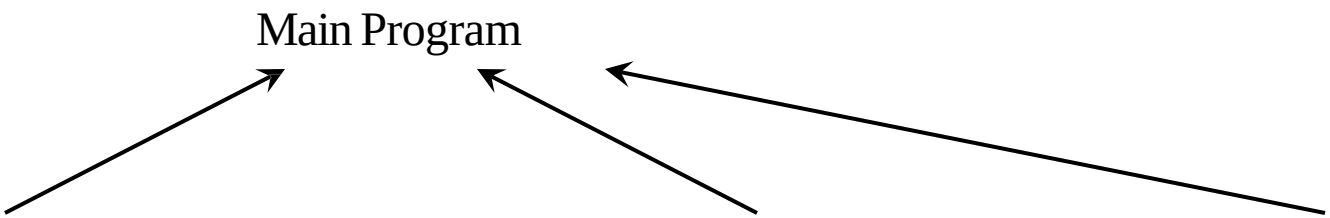
  A person is an object having attributes, such as eye color, age, height, and so on.

  A person also has behaviors, such as walking, talking, breathing, and so on.
- Bottom-up approach – focuses on selection and design of the objects first, which are then integrated with each other and the main program.
- Programming languages examples:

  Java, C++, C#, Python, PHP, Javascript, Ruby, Perl, Swift, Scala
- The concept of classes and objects help represent real world problems more easily
- Object oriented code is easy to debug and scale
- Objects are well suited for use on networks; internet and mobile web

Computing Paradigms – Object Oriented Programming

Example:
My Bake Shop



Object 1: Food

Attributes:
- samosas_in_stock
- rolls_in_stock
- samosa_unit_price
- roll_unit_price
- total_sales

Methods:
- Add a number of items to stock
- Show number of items in stock
- Take order, calculate bill …..

Object 2: Employee

Attributes:
- emp_A_wage_rate
- emp_B_wage_rate
- emp_A_hours
- emp_B_hours

Methods:
- Set employee wage rates
- Add hours worked
- Calculate payment
- Calculate bonus

Object 3: Boutique

Attributes: …..

Methods: …..

Computing Paradigms – Key Differences

Procedural Programming

- A program is divided into small parts called functions
- The attributes and behaviors are normally separated
- Follows top down approach
- Does not have any proper way for hiding data so it is less secure
- Good when you have a fixed set of things, and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone

Object Oriented Programming

- A program is divided into small parts called objects
- The attributes and behaviors are contained within a single object
- Follows bottom up approach
- Provides data abstraction which hides the background details, so it is more secure
- Good when you have a fixed set of operations on things, and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone

- It is also possible to use both the programming paradigms according to our own need
- Languages like python and java support both object oriented concept and are also functional by supporting various inbuilt functions

Objects

- Objects are building blocks of an object oriented program
- Each object is composed of two components: data and behaviours

- **Object data:**
  - represents the state of an object, the values that describe an object
  - also called attributes, fields, properties or simply variables

- **Object behaviour:**
  - represents what an object can do
  - also called methods, operations or functions

Examples

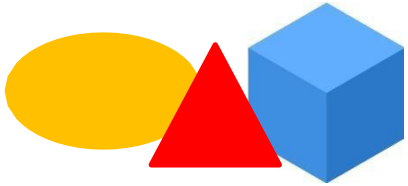
Object name: Mammals

Attributes:	Methods:
no_of_limbs	eat
eye_color	talk
habitat	walk

Objects – More Examples

1. Object name: Employee
   Attributes: employee_ID, gender, designation, date_of_birth
   Methods: getter/setter methods for attributes, calculate_salary, find_retirement_date

2. Object name: Drawing_shape
   Attributes: color, dimensions(2d or 3d)
   Methods: find_area, find volume



3. Object name: Vehicle
   Attributes: color, no_of_wheels, model_no
   Methods: find_mileage, cost_depreciations

Objects – Getters and Setters

- **Getter** is a method that retrieves value of an object attribute
  - also called accessor methods
- **Setter** is a methods that assigns/sets value of an object attribute
  - also called mutator methods

- Support data hiding
- Provide controlled access to an object's data

- **Example**

  Object name: Vehicle

  Attributes: color, no_of_wheels, model_no

  Methods: find_milaeage, cost_depreciations,

  get_color, set_color,

  get_no_of_wheels, set_no_of_wheels

  get_model_no, set_model_no

Class

- A class is a blueprint for an object
- Defines the type of an object
- Similar objects can use the same class/blueprint
- **Example**

  Class name: Pen

  Attributes: pen_type, case_color, ink_color, ink_type

  Methods: write

Packer_brushed_metal
pen_type: rollerball
case_color: silver
int_color: black
ink_type: gel

Piano_crystal
pen_type: ballpoint
case_color: blue
int_color: blue
ink_type: oil_based

Piano_black
pen_type: ballpoint
case_color: black
int_color: black
ink_type: oil_based

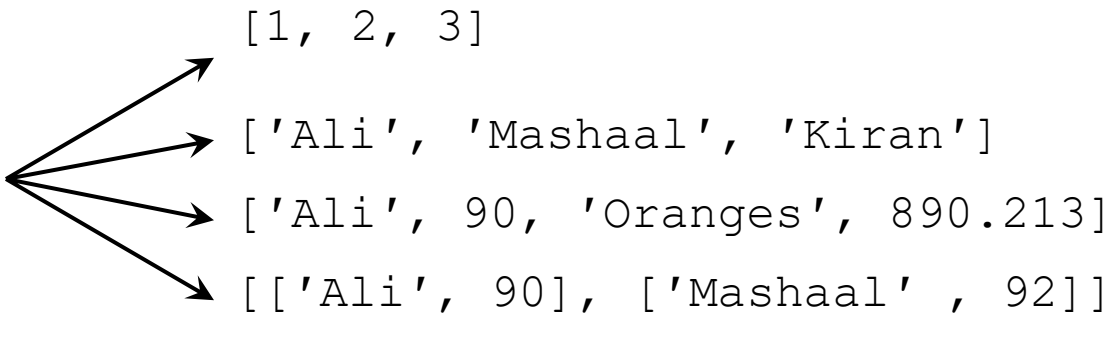
Class - Python Example

- Almost everything in Python is an object
- Python supports many different kinds of data:

```
1234                  ──────────►  integer
3.1419                ──────────►  float
'Hello World'         ──────────►  string
[1, 5, 7, 'abc']      ──────────►  list
{'CA' : 'California'} ──────────►  dictionary
```

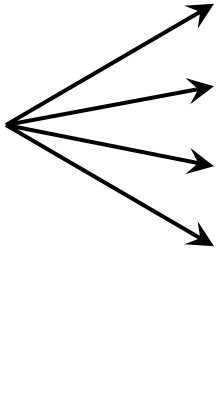
- Python has a class for each of these types
- A class is just a piece of code, an object when instantiated gets memory

- Class list



Class - Python Example

- Class list



```
[1, 2, 3]
['Ali', 'Mashaal', 'Kiran']
['Ali', 90, 'Oranges', 890.213]
[['Ali', 90], ['Mashaal' , 92 ]]
```

Objects

Methods:

```
insert(…)
remove(…)
sort(…)
reverse(…)
```

```
[1,2,3].sort()
Or
l=[1,2,3]
l.sort()
```

Similarly,

```
name=['Ali', 'Mashaal', 'Kiran']
name.sort()
```

- Each instantiated object has its own data and behaviours/methods

Features of Object Oriented Programming

- Following are the main features of object oriented programming:

  - Encapsulation and data hiding
  - Inheritance
  - Polymorphism
  - Association

- Every object oriented language provides syntax to implement these features

Encapsulation and Data Hiding

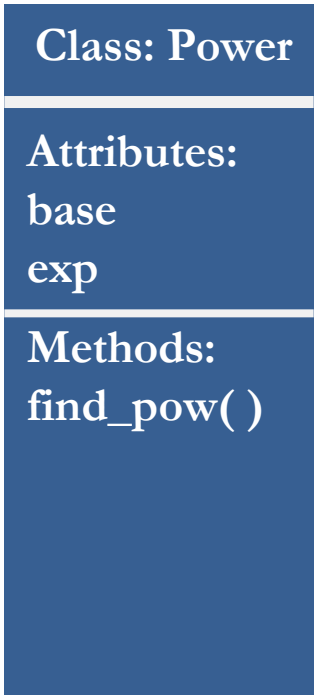
- Encapsulation – putting data and behaviours in a single object
- Data hiding – an object can reveal its details on need-to-know basis through its interface
- Interface – provides a way of communication between objects
- Access Specifiers
  - set the accessibility of attributes and methods
  - also called access modifiers
- Types of access specifiers:
  - public: accessible from everywhere
  - private: accessible from within the class only
  - protected: accessible to only the current class, sub-class and sometimes package classes; provides controlled access to other objects
- Usually attributes are private or protected whereas methods are public
- Interface consists of public attributes and methods only

Encapsulation and Data Hiding - Example

Let's define a class for finding power of a number

- The base and exponent values are to be provided by the user
- The method find_pow( ) returns base raised to the power exp

- Let's define and object of this class: P1
- The object P1 can access the attributes base and exp and the method find_pow( )




```
P1.base = 2
P1.exp = 3
result = P1.find_pow()
```

Encapsulation?

- the data values (attributes) and the method for calculating power are bundled together in a single object P1
- multiple objects can be defined, each encapsulating its own set of attributes and method

Encapsulation and Data Hiding - Example

Data hiding?

- Case 1: define getter/setter methods for the attributes
  - set_base(x) and set_exp(y) set the values of base and exp to x and y respectively
  - get_base( ) and get_exp( ) return the values of base and exp respectively

  Are these attributes and methods accessible from outside?

  YES: all attributes and methods are public by default;

  P1 can access all of them

Class: Power
Attributes: base exp
Methods: find_pow( ) set_base(x) get_base( ) set_exp(y) get_exp( )

```
P1.set_base(2)
P1.set_exp(3)
result = P1.find_pow()
```

```
P1.base = 2
P1.exp = 3
result = P1.find_pow()
```

Encapsulation and Data Hiding - Example

Data hiding?

- Case 2: define getter/setter methods for the attributes and make the attributes private

  Are these attributes and methods accessible from outside?

  NO: for the private attributes. P1 accessing them will result in access error

  YES: for the public methods; can be accessed through P1

Class: Power
Attributes: base exp
Methods: find_pow( ) set_base(x) get_base( ) set_exp(y) get_exp( )

```
P1.set_base(2)
P1.set_exp(3)
result = P1.find_pow()
```

```
P1.base = 2
P1.exp = 3
result = P1.find_pow()
```

Encapsulation and Data Hiding - Example

What's the use?

Keeping the interface same, the code using the class will not be effected if:

- an attribute's name is altered within the class
- a methods implementation is changed

Consider two implementations for the method find_pow()

Implementation 1

```
result=1
for i in range(exp):
    res=res*base
return result
```

Implementation 2

```
return base**exp
```

Class: Power
Attributes: base exp
Methods: find_pow( ) set_base(x) get_base( ) set_exp(y) get_exp( )

No one needs to know how the class is implemented at the backend, as long as the interface remains same

Inheritance

- Inheritance enables reusability of classes – allows a programmer to extend behaviour of an existing class

- It enables a class to inherit attributes and methods of another class

- The inheriting class is called child class / sub-class / derived class
- The class from which the child has inherited is called parent class / super-class / base class

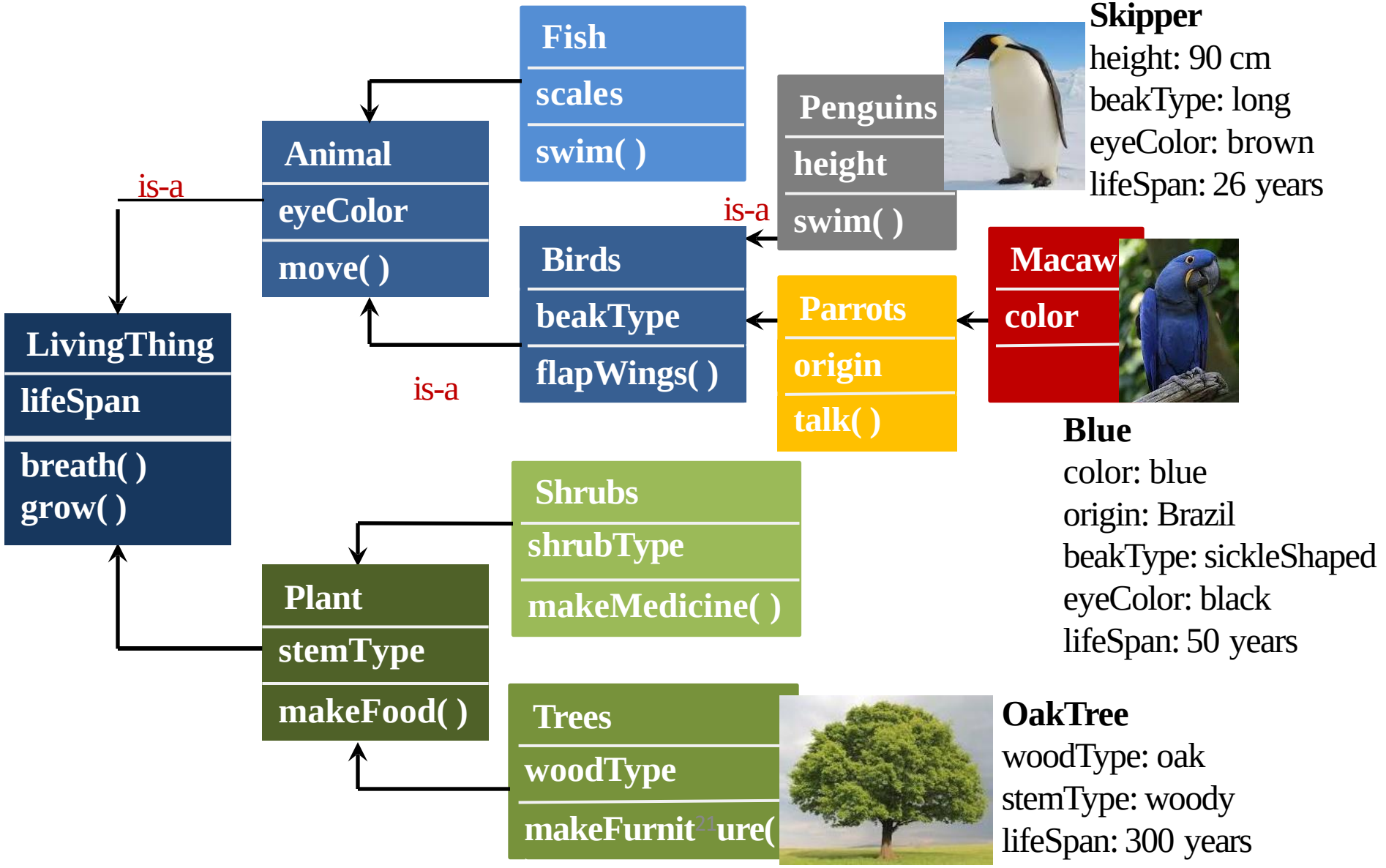
- A child class contains everything in its own class, as well as the public and protected attributes and methods of its parent class

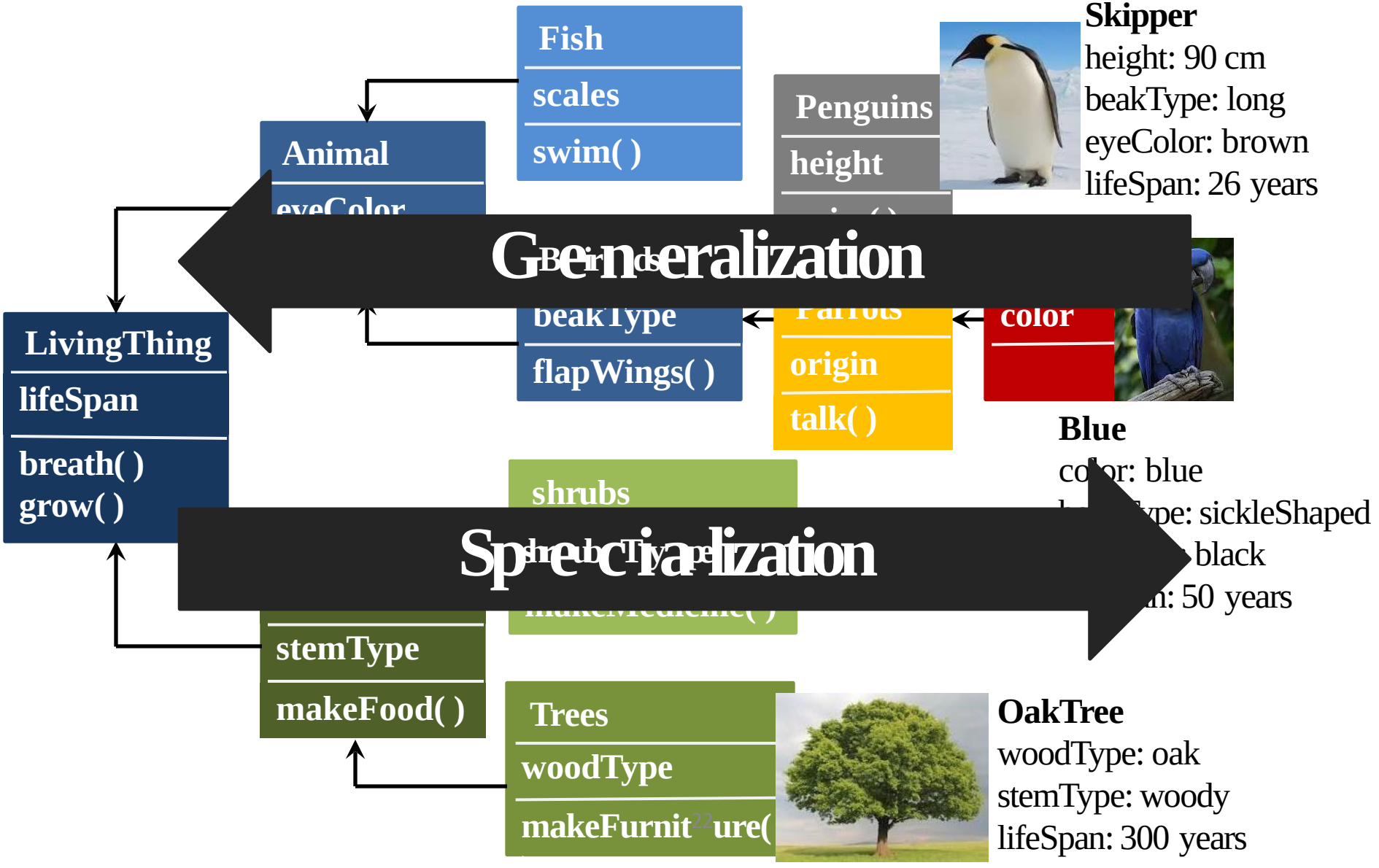
	Parent class – where the attribute / method is defined	Objects of the parent class instantiated in any child class	Objects of parent class instantiated in the main program
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

- The power of inheritance lies in its abstraction (layering) and organization techniques

- Implements is-a relationship

Inheritance - Example 1: The Living Things Hierarchy



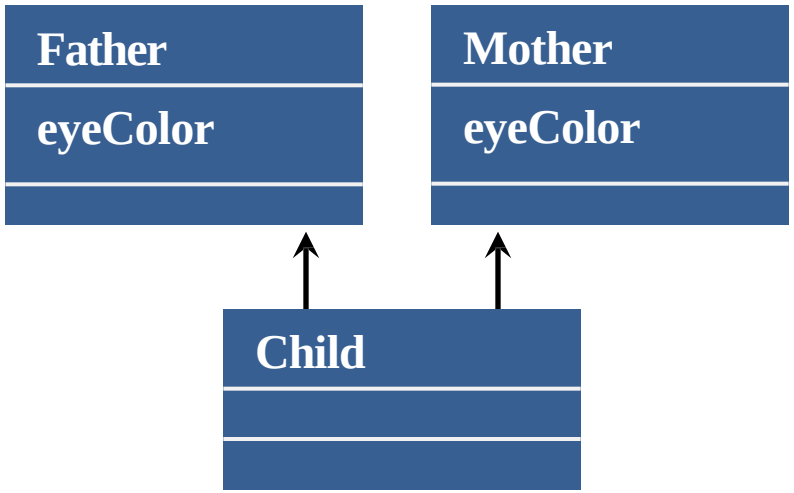

Fish
scales
swim( )
Animal
eyeColor
Penguins
height
Skipper
height: 90 cm
beakType: long
eyeColor: brown
lifeSpan: 26 years
Generalization
LivingThing
lifeSpan
breath( )
grow( )
beakType
flapWings( )
origin
talk( )
color
Blue
color: blue
sickleShaped
black
50 years
shrubs
Specialization
stemType
makeFood( )
Trees
woodType
OakTree
woodType: oak
stemType: woody
lifeSpan: 300 years

Inheritance

Multiple Inheritance

- Occurs when a sub-class is allowed to inherit from more than one super-classes
- The child class inherits features of all its parent classes

- **Example**

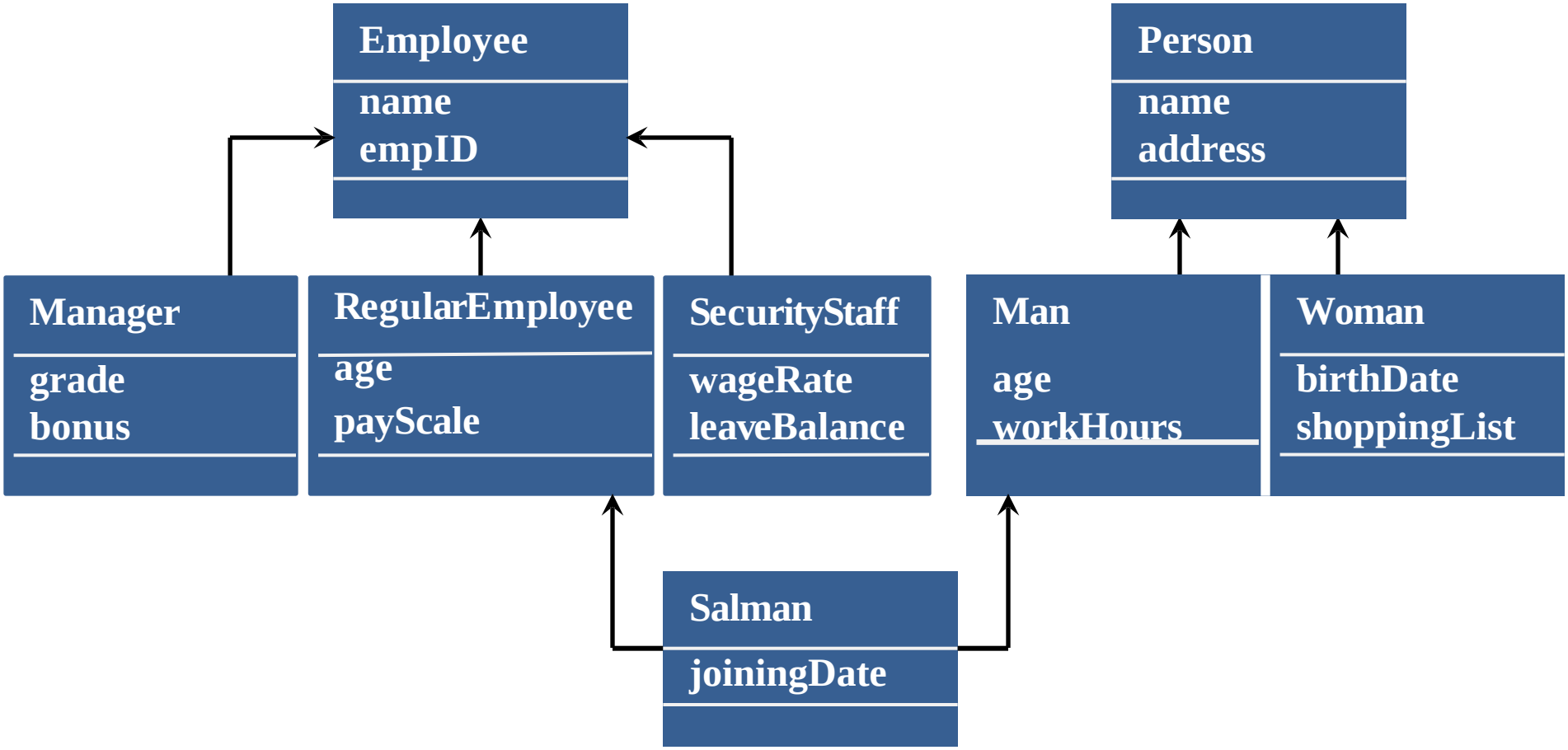


- Which eyeColor will the child inherit?

- If all the super-classes have an attribute or a method with same name, a pre-decided order determines from which super-class will the child inherit

Inheritance

Multiple Inheritance – Another Example



- Try listing attributes for Salman

Polymorphism

- Polymorphism is a Greek word meaning occurring in different / many forms

Example

While in quarantine I am missing my friend …
I can use my cell phone to reach my friend …
- I can send an email
- I can write a text message
- I can make a voice call
- I can make a video call

All these are different forms of communication
The goal is common but approach is different – This is polymorphism

- In computer programming, polymorphism refers to processing of objects differently depending on their data types and class
- Methods and operators can be defined and redefined in in different ways
- Common forms of polymorphism in computer programming:
  - Operator overloading
  - Method overloading
  - Method overriding

Polymorphism

Operator Overloading

- Operator overloading enables a programmer to change the meaning of an operator

Example from Python

`8 + 3` → **11** Arithmetic addition

`'Hello' + '  ' + 'World'` → **Hello World** String concatenation

`[ 1 , 2 ] + [44]` → **[1, 2 , 44]** List concatenation

`'Year' + 2020` → **?** Operator overloading will allow you to implement this

Polymorphism

Method Overloading

- Method overloading is defining two or methods with same method name but different parameters/arguments
- It adds to or extend a method's behavior.
- Overloaded methods belong to the same class / program

Example 1

C++ code

```
int multiply(int a, int b)
{
    return a*b;
}

float multiply(float a, float b)
{
    return a*b;
}
```

Python code

```
def multiply(a, b)
    return a*b


float multiply(float a)
{
    return a*10;
}
```

Polymorphism - Method Overloading

Example 2

```
def multiply(a,b)
    return a*b

def multiply(a)
    return a*10
```

- Will this work in Python??

Polymorphism

Method Overriding

- Overriding means having two methods with the same method name and same parameters / arguments, implementation is different
- The concept of method overriding is tightly coupled to inheritance - A method defines in parent class can be redefined in a child class
- It changes the existing behaviour of a method
- Overrided methods belong to the different classes

Polymorphism –

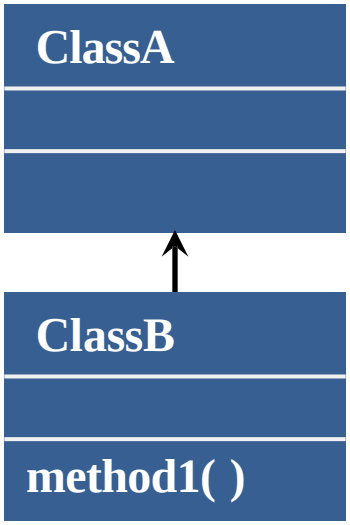
MethodOverriding

Example

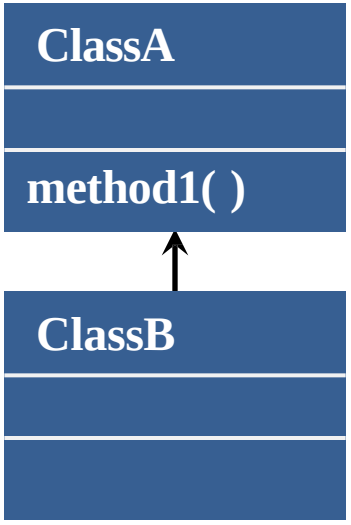
Consider a super-class called ClassA and its sub-class called ClassB

Let Obj1 be an object of ClassB

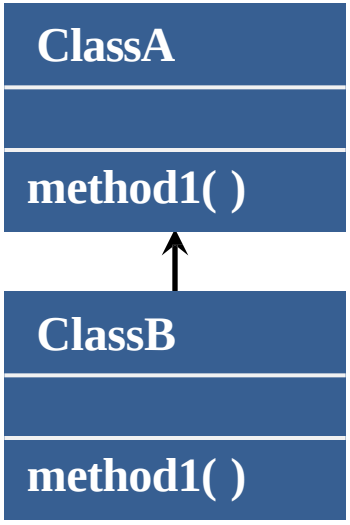
Obj1 calls method1( )



method1( ) of ClassB will be executed



method1( ) of ClassB will be executed

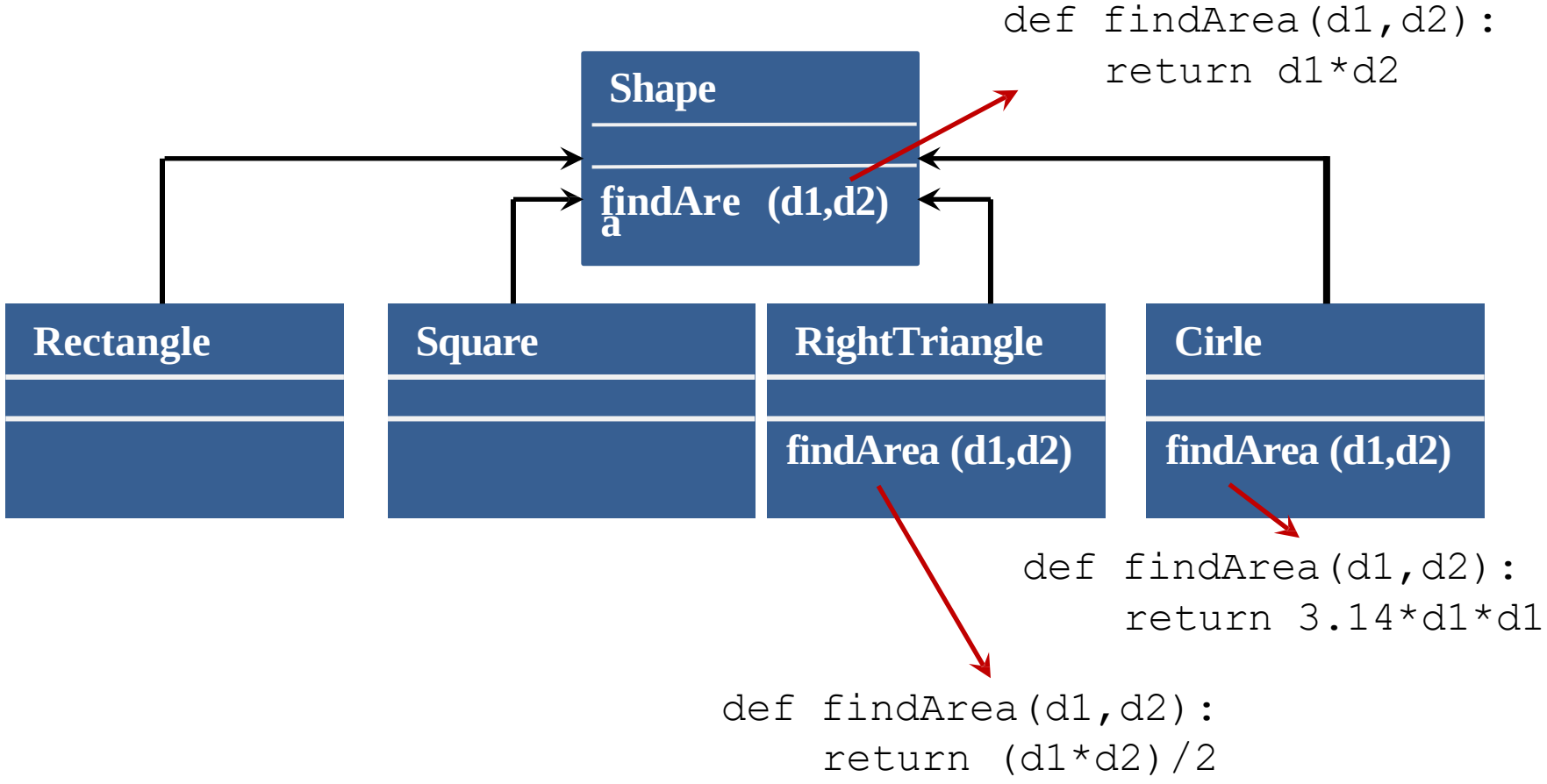


method1( ) of ClassB will override method1( ) of ClassA and will be executed

Polymorphism –

MethodOverriding

Example



Polymorphism –

MethodOverriding

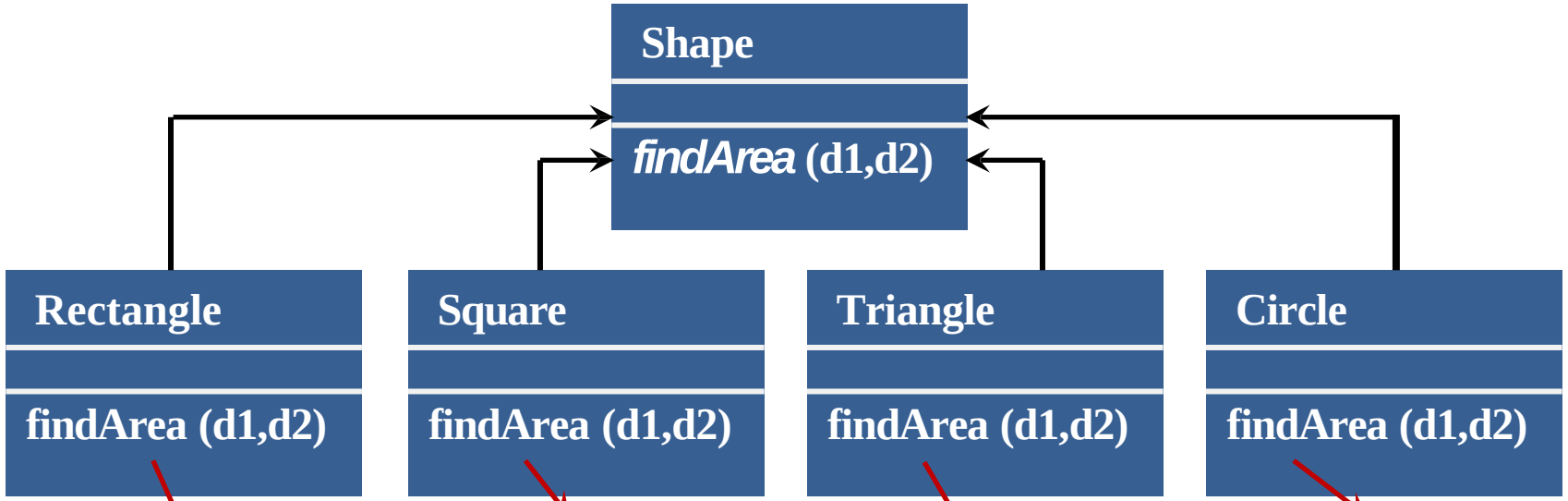
Abstract Methods

- An abstract method is declared in a class, but contains no implementation
- A class containing an abstract method becomes an abstract class
- Abstract classes cannot be instantiated, and require subclasses to provide implementations for the abstract methods
- If a sub-class inherits an abstract method from a super-class, it must provide a concrete implementation of that method or else it will be an abstract class itself
- Abstract methods cannot be private
- By defining an abstract method, we can enforce all the subclasses to implement that abstract method, but in their own ways

Polymorphism –

MethodOverriding

Example




```
def findArea(d1,d2):
    return d1*d1
```

```
def findArea(d1,d2):
    return 3.14*d1*d1
```

```
def findArea(d1,d2):
    return d1*d2
```

```
def findArea(d1,d2):
    return (d1*d2)/2
```

Defining Classes in Python -

Syntax

```
class <class_name>:
        <attribute_1> = <value>
        .
        .
        .
        <attribute_n> = <value>

        def <method_1> (self, arg_1, arg_2, ... , arg_n):
                <implementation>
        .
        .
        .
        def <method_n> (self, arg_1, arg_2, ... , arg_n):
                <implementation>
```

- Represents an object / instance of the class
- Could be any name, but self is used conventionally

Defining Classes in Python -

Example

Define a class Animal with two attributes specie and language and following method:

- speak ( ) – prints a message from the animal [e.g.: I am a cat and I meow]

```
class Animal:
    specie = 'cat'
    language = 'meow'

    def speak (self):
        print('I am a'+self.specie+' and I can '+self.language)

a1=Animal()
print ('Attribute 1 of Animal: ' + a1.specie)
print('Attribute 2 of Animal: ' + a1.language)
a1.speak()
```

Output

Attribute 1of Animal: cat
Attribute 2 of Animal: meow
I am a cat and I can meow

Defining Classes in Python -

Example

Changing values of the attributes

Method 1: Accessing class attributes from the main code

```
class Animal:
    specie = 'cat'
    language = 'meow'

    def speak (self):
        print('I am a'+self.specie+' and I can '+self.language)


a1=Animal()
a1.specie='lion'
a1.language='roar'
print ('Attribute 1 of Animal: ' + a1.specie)
print('Attribute 2 of Animal: ' + a1.language)
a1.speak()
```

Output

Attribute 1 of Animal: lion
Attribute 2 of Animal: roar
I am a lion and I can roar

Defining Classes in Python -

Example

Changing values of the attributes

Method 2: Defining setter methods

```
class Animal:
    s
    p
    e
    c
    i
    e
    =
    '
    c
    a
    t
    '
    l
    g
    u
    a
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()
```

Output

I am a mouse and I can squeak

Defining Classes in Python -

Example

Changing values of the attributes

Method 3: Defining setter methods without defining class attributes explicitly

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()
```

```
class Animal:
    def setSpecie(self,
```

```
a1=Animal()
a1.specie
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()
```

→ **error**

Defining Classes in Python -

Example

Changing values of the attributes

Let's define getter methods too

```
class Animal:
    def setSpecie(self,specie):
        self.specie=specie
    def setLanguage(self,language):
        self.language=language
    def getSpecie(self):
        return self.specie
    def getLanguage(self):
        return self.language
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
print(a1.getSpecie())
```

```
a1=Animal()
print(a1.getSpecie())
a1.setSpecie('mouse')
a1.setLanguage('squeak')
```

→ **error**

Documenting a Class

- IDLE displays some default documentation for a class through the function help( )
- docstrings can be used to add custom documentation to a class and its methods

Example: User Documentation

```
class Animal:
    '''Class Animal defines an animal object having. . . .
       It has two attributes and three methods'''
    specie = 'cat'
    language = 'meow'

    def setSpecie (self,s):
        '''This is setter method for the attribute specie'''
        self.specie = s
    .
    .
    .
    .
```

Practice Problems

Practice Problem 1

Create a class Point having two attributes x and y (the two coordinates of a point) and the following methods:

- setx(xcoord): sets the x coordinate of the point to xcoord
- sety(ycoord): sets the y coordinate of the point to ycoord
- get( ): returns the x and y coordinates of the Point type object as tuple (x , y)
- move(dx, dy): changes the coordinates of the Point type object from the current position (x , y) to (x+dx , y+dy)

```
class Point:
    x=0
    y=0

    def setx(self,xcoord):
        self.x=xcoord
    def sety(self,ycoord):
        self.y=ycoord
    def get(self):
        return self.x, self.y
    def move(self, dx, dy):
        self.x+=dx
        self.y+=dy

p1=Point( )
print(p1.get( ))
p1.setx(4)
p1.sety(7)
print(p1.get( ))
p1.move(1,1)
print(p1.get( ))
```

Output

(0, 0)
(4, 7)
(5, 8)

Object Oriented Programming

Lecture 05

Classes as Namespaces

Namespaces

Revisiting the Animal class:

```
class Animal:
    specie = 'cat'
    language = 'meow'

    def setSpecie (self, s):
        self.specie = s
    def setLanguage (self, l):
        self.language = l
    def speak (self):
        print(...)
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()

a2=Animal()
a2.setSpecie('lion')
```

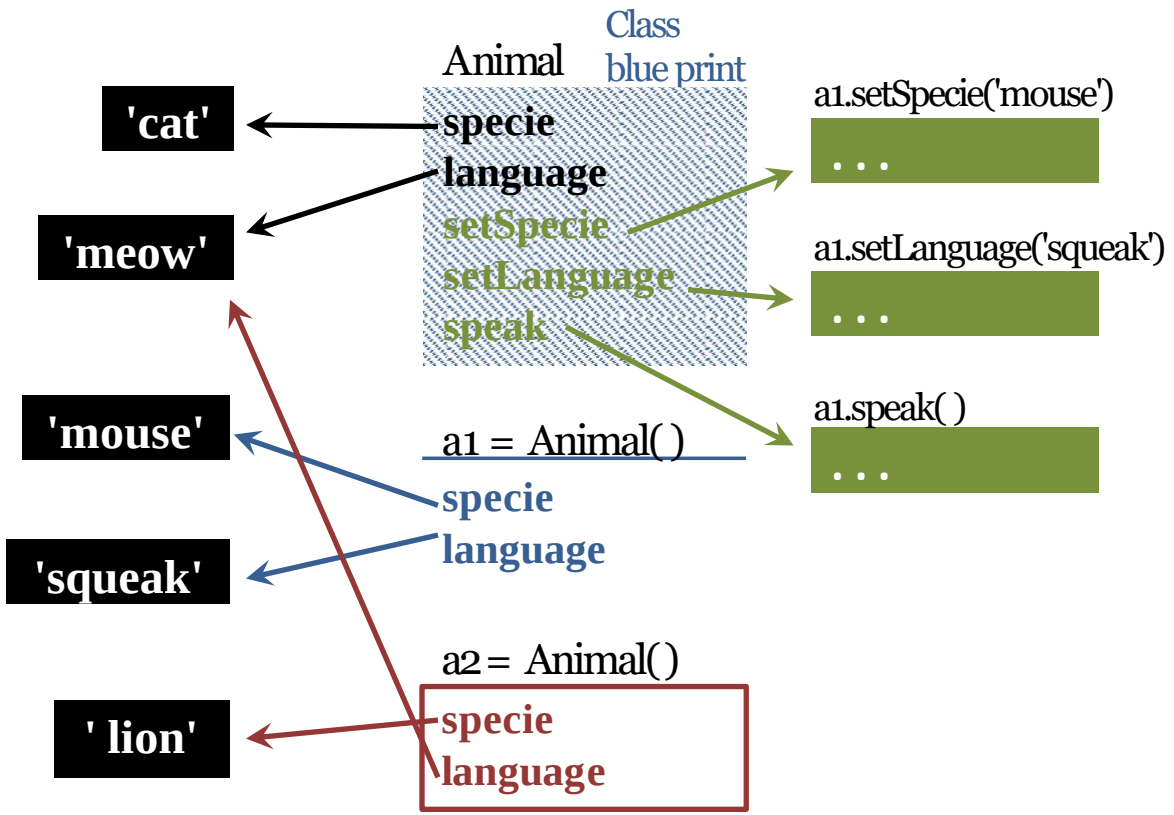
Output

I am a mouse and I can squeak

Classes, Objects and Namespaces

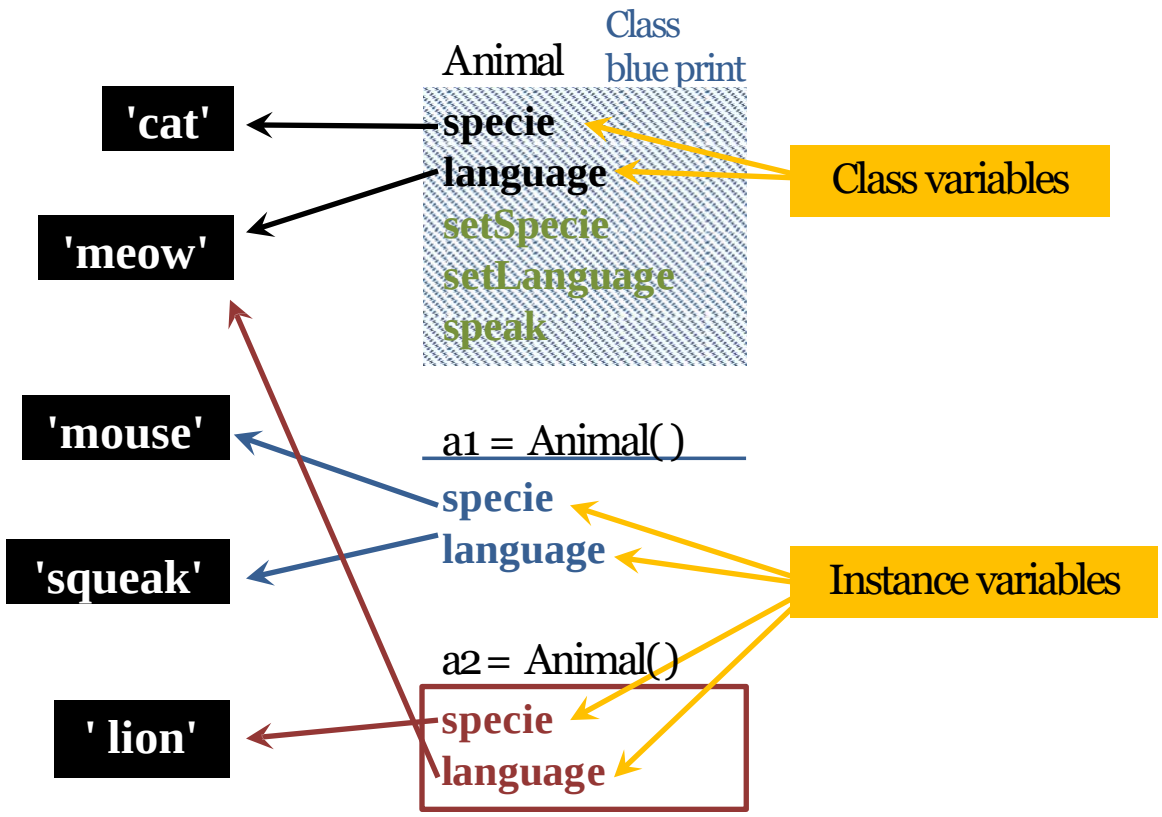
```
class Animal:
    specie = 'cat'
    language = 'meow'
       .
       .
       .
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()

a2=Animal()
a2.setSpecie('lion')
```



Classes, Objects and Namespaces

```
class Animal:
    specie = 'cat'
    language = 'meow'
       .
       .
       .

a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speak()

a2=Animal()
a2.setSpecie('lion')
```



Classes, Objects and Namespaces

Adding another method speakAlot( ):

```
class Animal:
    specie = 'cat'
    language = 'meow'

    def setSpecie (self, s):
        self.specie = s
    def setLanguage (self, l):
        self.language = l
    def speak (self):
        print(...)
    def speakAlot(self):
        for i in range(5):
            print(self.language)
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speakAlot()
```

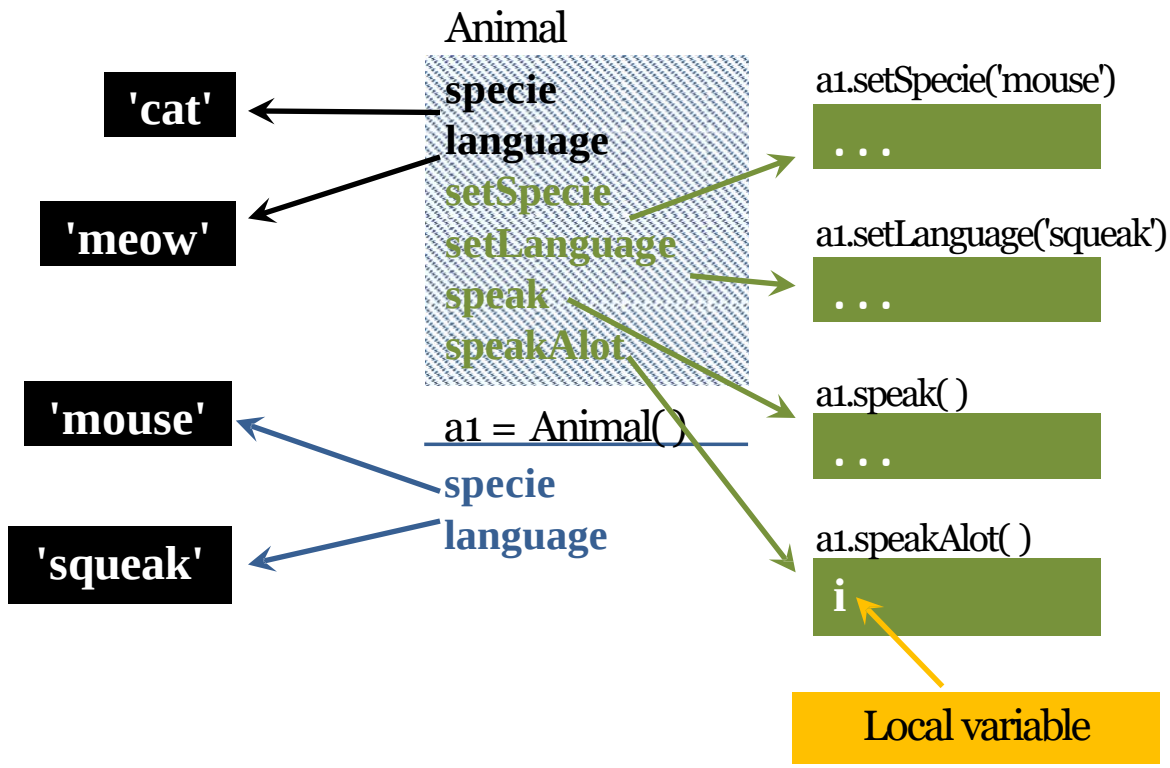
Output

squeak
squeak
squeak
squeak
squeak

Classes, Objects and Namespaces

```
class Animal:
    specie = 'cat'
    language = 'meow'
       .
       .
       .
```

```
a1=Animal()
a1.setSpecie('mouse')
a1.setLanguage('squeak')
a1.speakAlot()
```



Variables: Class, Instance and Local

Adding one more method countAnimal( )

```
class Animal:
    specie = 'cat'
    language = 'meow'
    count = 0

    #setter, getters, speak methods
    def speakAlot (self):
        for i in range(5):
            print(self.language)
    def countAnimal(self):
        Animal.count+=1
        return Animal.count
```

```
a1=Animal()
a1.speak()
print('Animal count:', \\
          a1.countAnimal())
a2=Animal()
a2.setSpecie('dog')
a2.setLanguage('bark')
a2.speak()
print('Animal count:', \\
       a2.countAnimal())
```

Output:
I am a cat and I can meow
Animal count: 1
I am a dog and I can bark
Animal count: 2

Variables: Class, Instance and Local

- A class and each of its objects own separate namespaces where their attributes live
- Each method in a class creates its separate namespace when invoked on an object
- Instance attributes / variables are defined in each object's namespace
  - accessed by object's name
  - live as long as that object lives
  - shared by all methods invoked on that object
- Class attributes / variables are defined in each class's namespace
  - accessed by class's name
  - live as long as the program lives
  - shared by all objects of that class instantiated in that program
- Local variables are defined in a method's namespace
  - accessed by its own name, without any handler
  - live as long as the method is executing
  - exclusive property of one instance of a method

Variables: Class, Instance and Local

Example

```
class Animal:
    specie = 'cat'
    language = 'meow'
    count=0

    #setter, getters, speak methods
    def speakAlot (self):
        for i in range(10):
            print(self.language)
    def countAnimal (self):
        Animal.count+=1
        self.assignID(Animal.count)
        return Animal.count
    def assignID (self, id):
        self.ID=id
    def getID (self):
        return self.ID
```

Class Variables:
specie
language
count

Instance Variables:
specie
language
ID

Local Variable:
i

Variables: Class, Instance and Local

Example

```
a1=Animal()
a1.countAnimal()
a2=Animal()
a2.countAnimal()
a3=Animal()
a3.countAnimal()
print('Count:', Animal.count)
print('ID of a1:', a1.getID())
print('Count:', Animal.count)
print('ID of a2:', a2.getID())
print('Count:', Animal.count)
print('ID of a3:', a3.getID())
print('Total count:', Animal.count)
```

Class Variables:
specie = 'cat'
language = 'meow'
count = 3

Instance Variables: a1
specie = 'cat'
language = 'meow'
ID = 1

Instance Variables: a2
specie = 'cat'
language = 'meow'
ID = 2

Instance Variables: a3
specie = 'cat'
language = 'meow'
ID = 3

Output:
Count: 3
ID of a1: 1
Count: 3
ID of a2: 2
Count: 3
ID of a3:3
Total count: 3

Variables: Class, Instance and Local

Determine Output

```
a1=Animal()
a1.countAnimal()
print('Count:', Animal.count)
print('ID of a1:', a1.getID())
a2=Animal()
a2.countAnimal()
print('Count:', Animal.count)
print('ID of a2:', a2.getID())
a3=Animal()
a3.countAnimal()
print('Count:', Animal.count)
print('ID of a3:', a3.getID())
print('Total count:', Animal.count)
print(Animal.ID)
```

Object Oriented Programming

Access Modifiers

Underscores in Python

Pattern	Example	Meaning
Single Leading Underscore	_var	- Naming convention indicating a name is meant for internal use - Generally not enforced by the Python interpreter (except in wildcard imports) and meant as a hint to the programmer only
Single Trailing Underscore	var_	- Used by convention to avoid naming conflicts with Python keywords
Double Leading Underscore	__var	- Triggers name mangling when used in a class context - Enforced by the Python interpreter
Double Leading and Trailing Underscore	__var__	- Indicates special methods defined by the Python language called dunders
Single Underscore	_	- Sometimes used as a name for temporary or insignificant variables

Name Mangling

- A double underscore prefix used with an attribute's or method's name helps avoid naming conflicts in subclasses

- The interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later

- The Python interpreter rewrites such name concatenating class names
  For example:
  - If `___var` is the name of a variable, interpreter will replace it by `_classname__var`, where `classname` is the name of the current class
  - `__var` is not accessible outside the class unless expanded to incorporate class name

Access Modifiers in Python

- Recalling access modifiers….
  - public: accessible from everywhere
  - private: accessible from within the class only
  - protected: accessible to only the current class, sub-class and sometimes package classes; provides controlled access to other objects

- Classical object-oriented languages, such as C++ and Java, control the access to class resources by public, private and protected keywords

- Python doesn't have any mechanism that effectively restricts access to any instance variable or method

The Public Access Modifier

- All members in a Python class are **public** by default.
- Any member can be accessed from outside the class environment

Example - Testing public attributes and methods

```
class TestPublic:
    def setPublicAttr(self, x):
        self.publicAttr=x
    def getPublicAttr(self):
        return self.publicAttr
    def publicMethod(self):
        print('I am a public method')

p1=TestPublic()
p1.setPublicAttr(2)
print(p1.getPublicAttr())
p1.publicAttr=99
print(p1.publicAttr)
p1.publicMethod()
```

Output:
2
99
I am a public method

The Protected Access Modifier

- Python prescribes a convention of prefixing the name of the variable/method with single underscore to emulate the behaviour of **protected** access specifiers
- This however, doesn't prevent instance variables from accessing or modifying them
- It is a hint for the responsible programmer - refrain from accessing and modifying instance variables prefixed with _ from outside its class

Example – Testing protected attributes and methods

```
class TestProtected:
    def setProtectedAttr(self, x):
        self._protectedAttr=x
    def getProtectedAttr(self):
        return self._protectedAttr
    def _protectedMethod(self):
        print('I am a protected method')
```

```
p1=TestProtected()
p1.setProtectedAttr(2)
print(p1.getProtectedAttr())
p1._protectedAttr=99
print(p1._protectedAttr)
p1._protectedMethod()
```

Output:
2
99
I am a protected method

The Private Access Modifier

- Python prescribes a convention of prefixing the name of the variable/method with double underscore to emulate the behaviour of **private** access specifiers

- It gives a strong suggestion not to touch it from outside the class, any attempt to do so will result in an error

- However, they can still be accessed from outside the class after name mangling expansion

Example – Testing private attributes and methods

```
class TestPrivate:
    def setPrivateAttr(self, x):
        self.__privateAttr=x
    def getPrivatetAttr(self):
        return self.__privateAttr
    def__privateMethod(self):
        print('I am a private method')
p2=TestPrivate()
p2.setPrivateAttr(44)
print(p2.getPrivatetAttr())
print(p2._privatetAttr)
print(p2._TestPrivate__privateAttr)
p2.__privateMethod()
p2._TestPrivate__privateMethod()
```

error

error

Output:

44

44

I am a private method

Practice Problem

Write code to implement class `Worker` that supports two private attributes `hoursWorked` and `wageRate`, and the following public methods:

- `setHoursWorked(h)`: sets `hoursWorked` to `h`
- `changeRate(r)`: changes the worker's pay rate to the new hourly rate `r`
- `pay()`: returns the pay as product of `hoursWorked` and `wageRate`

```
class Worker:
    def setHoursWorked(self, h):
        self._hoursWorked = h
    def changeRate(self, r):
        self._wageRate = r
    def pay(self):
        return self._wageRate*self.__hoursWorked
```

- Write some test code for this class

Object Oriented Programming

Constructors

Creating a New Class Fraction

```
class Fraction:
    def setNumerator(self,x):
        self.numerator=x
    def setDenominator(self,y):
        if y!=0:
            self.denominator=y
        else:
            print('Invalid value, setting to 1 instead')
            self.denominator=1
    def getFraction(self):
        return self.numerator, self.denominator
    def convertDecimal(self):
        return self.numerator/self.denominator
```

Creating a New Class Fraction

```
f1=Fraction()
f1.setNumerator(4)
f1.setDenominator(10)
print(f1.getFraction())
print(f1.convertDecimal())

f2=Fraction()
f2.setNumerator(9)
f2.setDenominator(0)
print(f2.getFraction())
```

Output:
(4, 10)
0.4

Output:
Invalid value, setting to 1 instead
(9, 1)

Object Oriented Programming

Lecture 06

Constructors

Creating a New Class Fraction

```
class Fraction:
    def setNumerator(self,x):
        self.numerator=x
    def setDenominator(self,y):
        if y!=0:
            self.denominator=y
        else:
            print('Invalid value, setting to 1 instead')
            self.denominator=1
    def getFraction(self):
        return self.numerator, self.denominator
    def convertDecimal(self):
        return self.numerator/self.denominator
```

Creating a New Class Fraction

```
f1=Fraction()
f1.setNumerator(4)
f1.setDenominator(10)
print(f1.getFraction())
print(f1.convertDecimal())

f2=Fraction()
f2.setNumerator(9)
f2.setDenominator(0)
print(f2.getFraction())
```

Output:
(4, 10)
0.4

Output:
Invalid value, setting to 1 instead
(9, 1)

Listing Class Attributes and Methods in Python

- Running `dir( )` function on any class (built-in or user-defined) shows all class attributes and methods defined in a class

Example: Listing attributes and methods in class `Fraction`

```
dir(Fraction)
```

Output:

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'convertDecimal', 'getFraction', 'setDenominator', 'setNumerator']

Listing Class Attributes and Methods in Python

Example: Listing attributes and methods for the object `f1`

```
dir(f1)
```

Output:
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'convertDecimal', 'denominator', 'getFraction', 'numerator', 'setDenominator', setNumerator']

- An object inherits all the attributes and methods of its class

Listing Class Attributes and Methods in Python

Why `dir(Fraction)` lists so many methods?

- All classes inherit by default from a base class called `object`
- The `object` class holds built-in properties/attributes and methods which are default for all classes
- The double underscore methods are called dunder methods, inherited from class `object`, can be overridden in the inheriting/user-defined class

The Dunder Methods

- Dunder methods are methods with leading and trailing underscores
- The word dunder comes from **d**ouble **under**scores
- These are magic methods which are automatically executed as per requirement, decided by the Python interpreter
- Dunder methods are usually public, can be accessed from the interface
- They can be overrided in a user-defined class to provide custom functionality
- They are commonly used as constructors and for operator overloading

Constructors

- A constructor is a special type of method that initializes an object automatically when created
  - saves us from setting instance variables by calling respective setters explicitly
  - saves us from error too; when getters are accidently called before setters
- In conventional object oriented languages, like C++ and java, compiler/interpreter identifies a given method as constructor by its name and return type
  - a constructor has the same name as that of the class where it is defined
  - a constructor does not return anything
  - Example form C++

```
class MyClass ──────────────► class definition
{
    public: ──────────────► access specifier
      MyClass() ──────────► constructor
          { <code> }
};
```

The `__init__` Methods

- Python uses initializers – loosely called constructors
- `__init__` is a dunder method of class object which is run when an object is instantiated
- User class override it to provide their own custom initializations

The `__init__` Methods

Example: Creating a constructor using `__init__` method

```
class Fraction:
    def __init__(self, x, y):
        self.numerator=x
        self.denominator=y
     # setNumerator, setDenominator methods
     # getFraction, convertDecimal methods

f1=Fraction(33,100)
print('Autoinitializations:', f1.getFraction())
f1.setNumerator(4)
f1.setDenominator(10)
print('Using setters:', f1.getFraction())
f2=Fraction()
```

→ error

Output:
Autoinitializations: (33, 100)
Using setters: (4, 10)

The `__init__` Methods

Example: Creating a default constructor

```
class Fraction:
    def __init_(self, x=0, y=1):
        self.numerator=x
        self.denominator=y
     # setNumerator, setDenominator methods
     # getFraction, convertDecimal methods

f1=Fraction(33,100)
print('f1:', f1.getFraction())
f2=Fraction()
print('f2:', f2.getFraction())
```

Output:
f1: (33, 100)
f2: (0, 1)

Destructors

- A destructor is a special type of method that destroys an object to cleanup in the end
  - they de-allocate the memory that has been allocated to the object by the constructor
  - they release resources allocated to the object (e.g.: close files, database connections, etc)

- In some object oriented languages, like C++, destructors are explicitly called to destroy objects before exit
  - Example form C++

```
class MyClass
{
     MyClass()  ──────────────►  constructor
          { <code> }
     ~MyClass()  ─────────────►  destructor
          { <code> }
};
```

Garbage Collector

- A garbage collector is a program that runs to recover the memory by deleting the objects which are no longer in use or have finished their life-cycle
  - provides automatic memory management
  - does the job of destructor
  - does not need to be called explicitly

- Java and Pyhton use garbage collection mechanisms instead of destructors

- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero, to destroy the object
  - `del<object_name>` can be used to forcefully destroy an object
  - `__del__` is the dunder method called when object is destroyed

Garbage Collector

Example

```
class Fraction:
    def __init__(self, x=0, y=1):
        self.numerator=x
        self.denominator=y
    def __del__(self):
        print('Goodbye everyone:((')
    # setNumerator, setDenominator methods
    # getFraction, convertDecimal methods

f1=Fraction(33,100)
print('f1:', f1.getFraction())
del f1
print('f1:', f1.getFraction())
```

Output:
f1: (33, 100)
Goodbye everyone:((

→ error

Object Oriented Programming

Lecture 07

Constructors

Creating a New Class Fraction

```
class Fraction:
    def setNumerator(self,x):
        self.numerator=x
    def setDenominator(self,y):
        if y!=0:
            self.denominator=y
        else:
            print('Invalid value, setting to 1 instead')
            self.denominator=1
    def getFraction(self):
        return self.numerator, self.denominator
    def convertDecimal(self):
        return self.numerator/self.denominator
```

Creating a New Class Fraction

```
f1=Fraction()
f1.setNumerator(4)
f1.setDenominator(10)
print(f1.getFraction())
print(f1.convertDecimal())

f2=Fraction()
f2.setNumerator(9)
f2.setDenominator(0)
print(f2.getFraction())
```

Output:
(4, 10)
0.4

Output:
Invalid value, setting to 1 instead
(9, 1)

Object Oriented Programming

Lecture 08

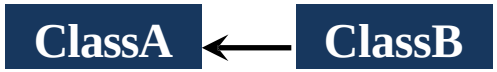
Basic Inheritance

Inheritance - Recalling from Lecture 3

- Inheritance enables reusability of classes – allows a programmer to extend behaviour of an existing class
- It enables a class to inherit attributes and methods of another class
- The inheriting class is called child class / sub-class / derived class
- The class from which the child has inherited is called parent class / super-class / base class
- A child class contains everything in its own class, as well as the public and protected attributes and methods of its parent class
- The power of inheritance lies in its abstraction (layering) and organization techniques
- Implements is-a relationship

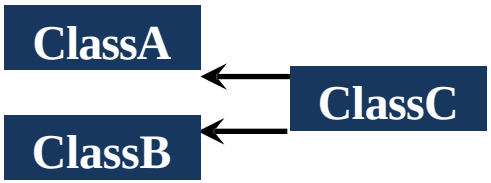
Types of Inheritance

- Basic / Single inheritance: when a child inherits from only one parent class



- Multilevel inheritance: when a child inherits from only one parent class and the parent itself inherits from another class

- Multiple inheritance: when a child inherits form multiple parent classes



Syntax

- Basic inheritance / Single inheritance

```
class <Class Name>(<Super Class>):
    <attributes and methods>
```

- Multiple inheritance

```
class <Class Name>(<Super Class1>, <Super Class2>, ...):
    <attributes and methods>
```

Basic Inheritance

- Technically, every class we create uses inheritance

- All Python classes inherit by default, from a special class called `object` which
  - provides very little in terms of data and behaviors
  - provides dunders intended for internal use only
  - allows Python to treat all objects in the same way

- If we don't explicitly inherit from a different class, our classes will automatically inherit from `object`

Basic Inheritance - Example

Example: Defining two classes: `Animal` and `Duck`, making class `Duck` child class of class `Animal`

```
class Animal:
  <attributes and methods>
```

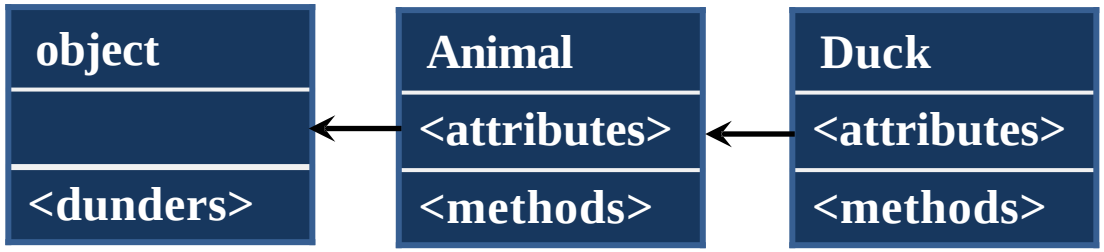
- Itself is a child class of class `object`
- Equivalent to:

```
class Animal (object):
    <attributes and methods>
```

```
class Duck (Animal):
  <attributes and methods>
```

- Child class of class `Animal`,
- Child class of `object`?

Yes. . . Multilevel inheritance



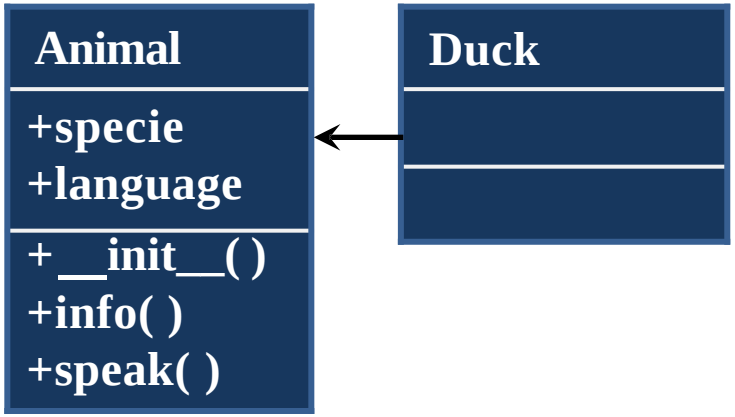
Defining Parent and Child Classes

A child class inherits all attributes and methods of the parent class

Example:

```
class Animal:
    def_init_(self, s='animal', l='talk'):
       self.specie=s
       self.language=l
    def info(self):
       print('Specie:',self.specie,'\nLanguage:',self.language)
    def speak(self):
       print('I am a', self.specie,'and I can',self.language)

class Duck(Animal):
    pass
```

Defining Parent and Child Classes - Example




```
tom = Animal('cat','meow')
tom.info()
tom.speak()
daffy = Duck('duck','quack')
daffy.info()
daffy.speak()
```

Output:
Specie: cat
Language: meow
I am a cat and I can meow
Specie: duck
Language: quack
I am a duck and I can quack

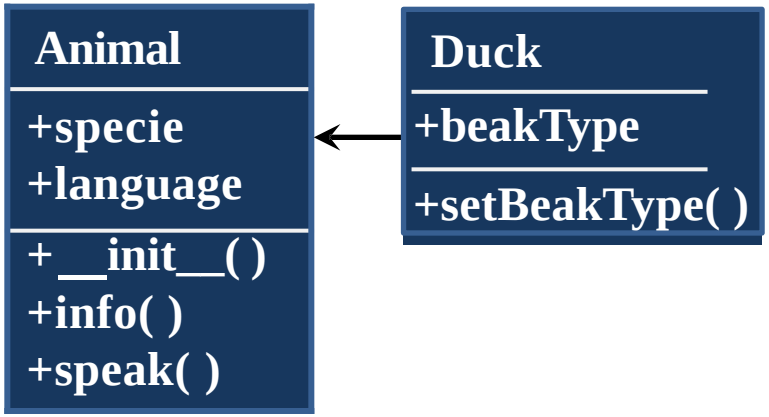
Extending the Child Class

New attributes and methods can be added to the child class

Example

```
class Animal:
    def_init_(self, s='animal', l='talk'):
       self.specie=s
       self.language=l
    def info(self):
       print('Specie:',self.specie,'\nLanguage:'self.language)
    def speak(self):
       print('I am a', self.specie,'and I can',self.language)

class Duck(Animal):
    def setBeakType(self, b='short'):
       self.beakType=b
```

Extending the Child Class - Example



Try executing:
`tom.setBeakType()`

```
tom=Animal('cat','meow')
tom.info()
tom.speak()
daffy=Duck('duck','quack')
daffy.setBeakType('long and curved')
daffy.info()
daffy.speak()
```

Output:
Specie: cat
Language: meow
I am a cat and I can meow
Specie: duck
Language: quack
I am a duck and I can quack

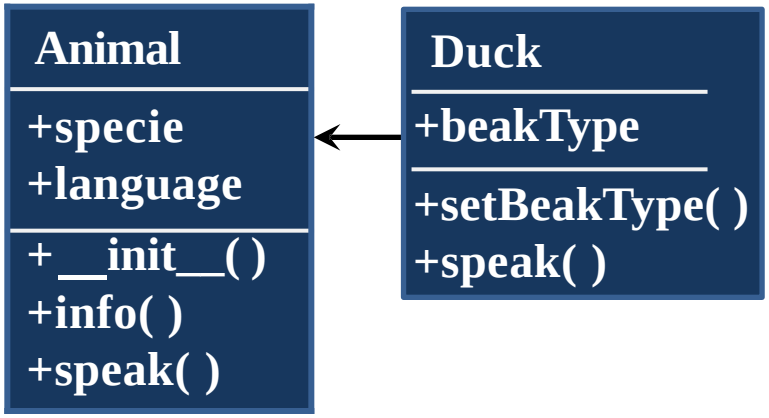
Overriding Methods of the Parent Class

A child class can override any method of the parent class

Example: 1. Overriding the `speak` method

```
class Animal:
    def_init_(self, s='animal', l='talk'):
    . . .
    def speak(self):
       print('I am a', self.specie,'and I can',self.language)

class Duck(Animal):
    def setBeakType(self, b='short'):
       self.beakType=b
    def speak(self):
       print('quack! quack!! quack!!!')
```

Overriding Methods of the Parent Class - Example




```
tom=Animal('cat','meow')
tom.info()
tom.speak()
daffy=Duck('duck','quack')
daffy.setBeakType('long and curved')
daffy.info()
daffy.speak()
```

Output:
Specie: cat
Language: meow
I am a cat and I can meow
Specie: duck
Language: quack
quack! quack!! quack!!!

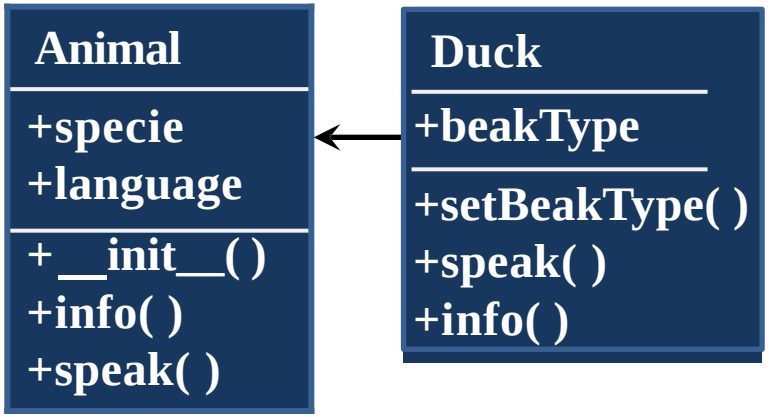
Overriding Methods of the Parent Class - Example

A child class can override any method of the parent class

Example: 1. Overriding the `info` method

```
class Animal:
     . . .
    def info(self):
       print('Specie:',self.specie,'\nLanguage:',self.language)
    . . .

class Duck(Animal):
     . . .
     def info(self):
       print('Specie:',self.specie,'\nLanguage:', \\
              self.language,\\ '\nBeak type:',self.beakType)
     . . .
```

Overriding Methods of the Parent Class - Example




```
tom=Animal('cat','meow')
tom.info()
tom.speak()
daffy=Duck('duck','quack')
daffy.setBeakType('long and curved')
daffy.info()
daffy.speak()
```

Output:
Specie: cat
Language: meow
I am a cat and I can meow
Specie: duck
Language: quack
Beak type: long and curved
quack! quack!! quack!!!

Accessing Methods of the Parent class

Two ways:

- Use parent class' name
- Use the function super( )

  - returns a temporary object that allows reference to a parent class

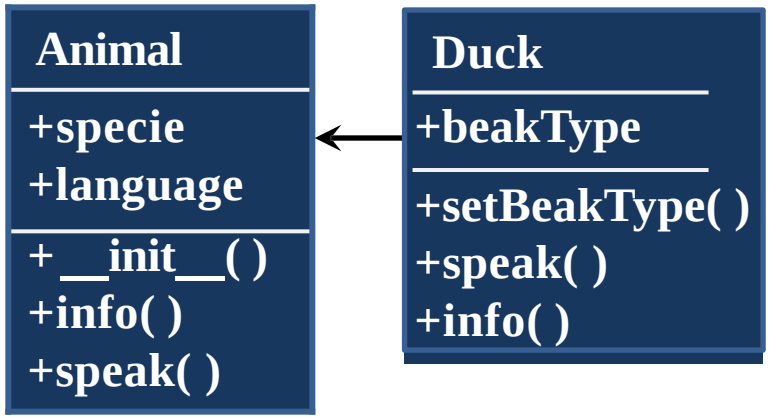
Example

```
class Parent:
    def methodParent(self):
        print('A parent class method')
class Child (Parent):
    def methodChild(self):
        Parent.methodParent(self)
        super().methodParent()
```

} **Use either of the two**

Overriding for Extension

Revisiting the example form lecture 10



Overriding for Extension

Example 1: Overriding the method `info`

Last time….

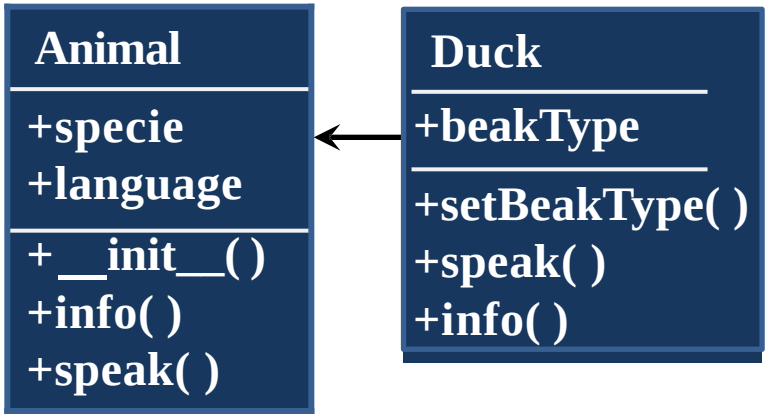
```
class Animal:
      . . .
    def info(self):
       print('Specie:',self.specie,'\nLanguage:',self.language)
    . . .

class Duck(Animal):
     . . .
     def info(self):
       print('Specie:',self.specie,'\nLanguage:', \
              self.language,'\nBeak type:',self.beakType)
     . . .
```

Overriding for Extension - **Example 1**

```
class Animal:
      . . .
    def info(self):
       print('Specie:',self.specie,'\nLanguage:',self.language)
    . . .

class Duck(Animal):
      . . .
     def info(self):
       Animal.info(self)
       print('Beak type:',self.beakType)
      . . .
```

→ or use: `super().info()`

Overriding for Extension - Example 1




```
tom=Animal('cat','meow')
tom.info()
daffy=Duck('duck','quack')
daffy.setBeakType('long and curved')
daffy.info()
```

Output:
Specie: cat
Language: meow
Specie: duck
Language: quack
Beak type: long and curved

Overriding for Extension

Example 2: Overriding the method `__init__`

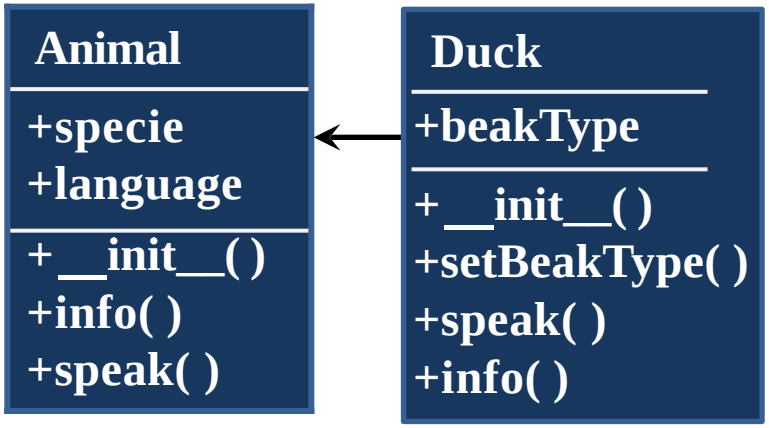
- Two approaches:
  - Initialize all three instance variables in `__init__` of the child class
  - Initialize only beakType in `__init__` of the child class, for the remaining two call `__init__` of the parent class – better approach as it avoids code redundancy

```
class Duck(Animal):
    def __init__(self, s, l, b='short'):
        Animal.__init__(self,s,l)
        self.beakType=b
    ...
```

→ or use: `super().__init__(s,l)`

- `__init__` is a public method, can be called explicitly as well
- Method `setBeakType` is no longer needed

Overriding for Extension - Example 1




```
daffy=Duck('duck','quack','long and curved')
daffy.info()
daffy.speak()
```

Output:
Specie: duck
Language: quack
Beak type: long and curved
quack! quack!! quack!!!

Print Representation of an Object

- Let us define a list object and print it:

```
myList=[1,2,3]
print(myList)
```

Output:
[1, 2, 3]

- Try printing objects of the Animal and Duck classes

```
tom=Animal('cat','meow')
print(tom)
daffy=Duck('duck','quack','long and curved')
print(daffy)
```

Output:
<__main__.Animal object at 0x04408BD0>
<__main__.Duck object at 0x04408BF0>

- Uninformative print representation
- Solution:
  Override the dunder `__str__`

Overriding `str` method

- Called automatically by Python when an object is printed
- Can be used to return an informal, ideally very readable, string representation of the object

Example 1

```
class Representation:
    def __str__(self):
        return 'Pretty string representation'

r=Representation()
print(r)
```

Output:
Pretty string representation

Overriding `__str__` method

Example 2: overriding `__str__` methods for classes Animal and Duck

Let us first choose how we want our object to be printed

```
daffy=Duck('duck','quack','long and curved')
print(daffy)
```

Output 1:
<duck, quack, long and curved>

Output 2:
Specie: duck
Language: quack
Beak: long and curved

Overriding `__str__` method

Example 2: overriding __str__ methods for Classes Animal and Duck

```
class Animal:
    . . .
    def__str__(self):
        return '<'+self.specie+', '+self.language+'>'
    . . .

class Duck(Animal):
    . . .
    def__str__(self):
        return '<'+self.specie+', '+self.language+', '\
                  +self.beakType+'>'
    . . .
```

Overriding `__str__` method

Example 2: overriding __str__ methods for Classes Animal and Duck

```
tom=Animal('cat','meow')
print(tom)
daffy=Duck('duck','quack','long and curved')
print(daffy)
```

Output 2:
<cat, meow>
<duck, quack, long and curved>

- Can you guess the output of this same test code if `__str__` is removed from class Duck?

Types and Classes

Run the following code for class Animal:

```
tom = Animal()
print(tom)
print(type(tom))

print(Animal)
print(type(Animal))

print(isinstance(tom, Animal))
```

→ printing the object

→ can ask for the type of an object instance: type of tom is Animal

→ Animal is a class

→ a class is a type

↓

to check if an object belongs to a class

Output:

```
<animal, talk>
<class '__main__.Animal'>
<class '__main__.Animal'>
<class 'type'>
True
```

Object Oriented Programming

Lecture 09

Operator Overloading

Polymorphism - **Recalling from Lecture 3**

- Polymorphism is a Greek word meaning occurring in different / many forms

- In computer programming, polymorphism refers to processing of objects differently depending on their data types and class
  - Methods and operators can be defined and redefined in in different ways

- Common forms of polymorphism in computer programming:
  - Operator overloading enables a programmer to change the meaning of an operator
  - Method overloading is defining two or more methods with same method name in the same class / program
  - Method overriding is defining two or more methods with same method name in different classes in a parent-child hierarchy

Operator Overloading

- Operator overloading lets classes intercept normal Python operations
- Classes can overload all Python expression operators
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc.
- Overloading makes class instances act more like built-in types
- Overloading is implemented by providing/overriding specially named methods in a class

Operator Overloading

- Operator overloading is done through method overriding – override dunders

- An overloaded operator is an operator that has been defined for multiple classes

Example: the + operator: Overloaded in many classes
- adds values when used with integers
- concatenates strings/lists when used with strings/lists

- Operators are class methods – operation evaluation is actually a method invocation
  Example:
  The algebraic expression `x+y` gets translated by Python Interpreter to `x.__add__(y)`

Overloading the + Operator

Example 1

`+` operator in string class concatenates two strings, but it does not concatenates any other object to string.

Define a new class `MyStr` having one instance variable `strg`, which can concatenate a string to any object through the `+` operator.

For instance:
Let `x` be an instance of `Mystr` class initialized to `'abc'`.
Then `x+3` should give `'abc3'`.

- `x+3` will be expanded into `x. add (3)`

Overloading the + Operator – Example 1

```
class MyStr():
    def_init_(self, s=''):
        self.strg=s
    def_add__(self,anyObject):
        return self.strg + str(anyObject)

x=MyStr('Python')
print(x+' Programming')
print(x+3)
print(x+3.7)
print(x+[2,3])
```

Output:
Python Programming
Python3
Python3.7
Python[2, 3]

Overloading the + Operator

Example 2:

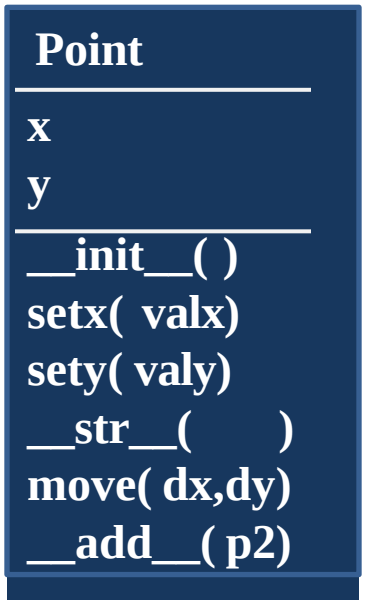
Define a class Point as follows:

Create a class Point having two attributes x and y (the two coordinates of a point) and the following methods:

- setx(xcoord): sets the x coordinate of the point to xcoord
- sety(ycoord): sets the y coordinate of the point to ycoord
- __str__( ): returns the x and y coordinates of the Point type object as (x , y)
- move(dx, dy): changes the coordinates of the Point type object from the current position (x , y) to (x+dx , y+dy)
- __add__(p2): add x and y coordinates of p2 to x and y coordinates of the current object respectively - overloading the + operator

Overloading the + Operator - Example 2

```
class Point:
    def_init__(self,xcoord=0,ycoord=0):
        self.x=xcoord
        self.y=ycoord
    def setx(self,xcoord):
        self.x=xcoord
    def sety(self,ycoord):
        self.y=ycoord
    def_str__(self):
        return '<'+str(self.x)+', '+str(self.y)+'>'
    def move(self, dx, dy):
        self.x+=dx
        self.y+=dy
```



Overloading the + Operator - **Example 2**

```
def_add_(self,p2):
        return self.x+p2.x,self.y+p2.y
        #return '<'+str(self.x+p2.x)+', '+str(self.y+p2.y)+'>'
        #return Point(self.x+p2.x,self.y+p2.y)

p1=Point(1,5)
p2=Point(2,2)
print(p1)
print(p2)
print(p1+p2)
```

Output:
<1, 5>
<2, 2>
(3, 7)

Arithmetic Operators and Corresponding Dunders

Operator	Dunder	Operation	Explanation
+	x.__add__(y)	x + y	Addition
-	x.__sub__(y)	x - y	Subtraction
*	x.__mul__(y)	x * y	Multiplication
/	x.__truediv__(y)	x / y	Division
//	x.__floordiv__(y)	x // y	Find quotient
%	x.__mod__(y)	x % y	Find remainder
**	x.__pow__(y)	x ** y	Find Power

Comparison Operators and Corresponding Dunders

Operator	Dunder	Operation	Explanation
<	x.__lt__(y)	x < y	Less than
>	x.__gt__(y)	x > y	Greater than
<=	x.__le__(y)	x <= y	Less than or equal to
>=	x.__ge__(y)	x >= y	Greater than or equal to
==	x.__eq__(y)	x == y	Equal to
!=	x.__ne__(y)	x != y	Not equal to

Assignment Operators and Corresponding Dunders

Operator	Dunder	Operation	Explanation
`+=`	`x.__iadd__(y)`	x = x + y	Add and assign
`-=`	`x.__isub__(y)`	x = x − y	Subtract and assign
`*=`	`x.__imul__(y)`	x = x * y	Multiply and assign
`/=`	`x.__idiv__(y)`	x = x / y	Divide and assign
`//=`	`x.__ifloordiv__(y)`	x = x // y	Find quotient and assign
`%=`	`x.__imod__(y)`	x = x % y	Find remainder and assign
`**=`	`x.__ipow__(y)`	x = x ** y	Find power and assign

Unary Operators and Corresponding Dunders

Operator	Dunder	Operation	Explanation
-	`x.__neg__(y)`	-x	Negative value
+	`x.__pos__(y)`	+x	Positive value
~	`x.__invert__(y)`	~x	Invert value

Other Commonly Overridden Dunders

Operator	Dunder	Explanation
`<type>(x)`	`<type>.__init__(x)`	Constructor
`str(x)`	`x.__str__()`	Informal string representation
`repr(x)`	`x.__repr__()`	Canonical string representation
`len(x)`	`x.__len__()`	Collection size (string, list, etc.)

Overloading the += Operator

Lets overload += operator for the `Point` class to implement `p1=p1+p2`

- the corresponding dunder method is `__iadd__`
- `p1+=p2` would change to `p1=p1.__iadd__(p2)`

```
class Point:
    ...
    def __iadd__(self,p2):
        self.x=self.x+p2.x
        self.y=self.y+p2.y
        return self
```

`return Point(self.x+p2.x,self.y+p2.y)`

```
p1=Point(1,5)
p2=Point(2,2)
p1+=p2
print(p1)
```

Output:
<3, 7>

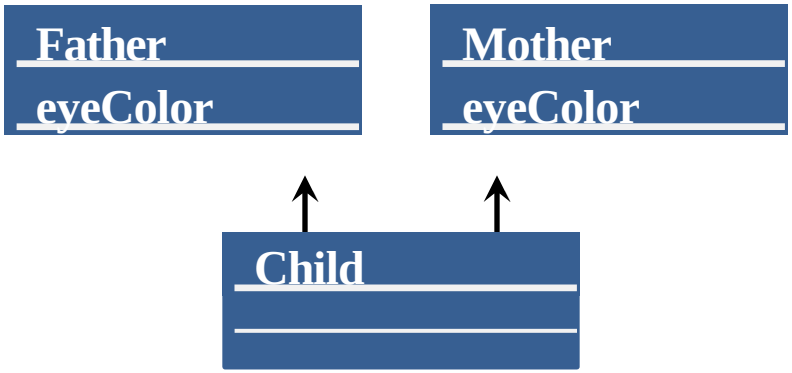
Object Oriented Programming

Lecture 10

Multiple Inheritance

Multiple Inheritance - **Recalling from Lecture 3**

- Occurs when a sub-class is allowed to inherit from more than one super-classes
- The child class inherits features of all its parent classes
- **Example**



- Which eyeColor will the child inherit?

- If all the super-classes have an attribute or a method with same name, a pre-decided order determines from which super-class will the child inherit

Multiple Inheritance

Syntax

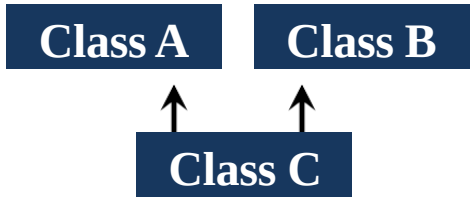
```
class Father:
    #attributes and methods

class Mother:
    #attributes and methods

class Child (Father, Mother):
    #attributes and methods
```

Multiple Inheritance

A child class inherits attributes and methods from all the parents

Example

```
class A:
    def methodA(self):
        print('In method A')
class B:
    def methodB(self):
        print('In method B')
class C(A, B):
    def methodC(self):
        print('In method C')
objectC=C()
objectC.methodA()
objectC.methodB()
objectC.methodC()
```



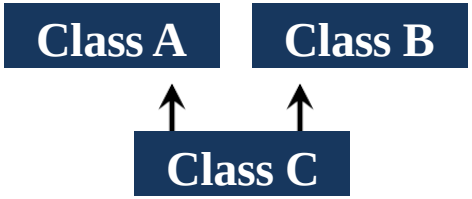
Output:
In method A
In method B
In method C

Multiple Inheritance

- How to decide the search order when a method with same name exists in more than one class in the hierarchy

Example 1

Consider the following inheritance hierarchy:



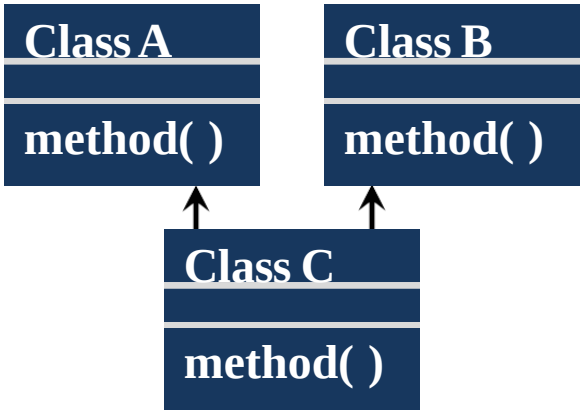
Search order:

- search in the current class
- search the left parent
- search the right parent

Multiple Inheritance - Example 1

Case 1: `method()` exists in all classes; A, B and C

```
class A:
    def method(self):
        print('In method A')
class B:
    def method(self):
        print('In method B')
class C(A, B):
    def method(self):
        print('In method C')

objectC=C()
objectC.method()
```

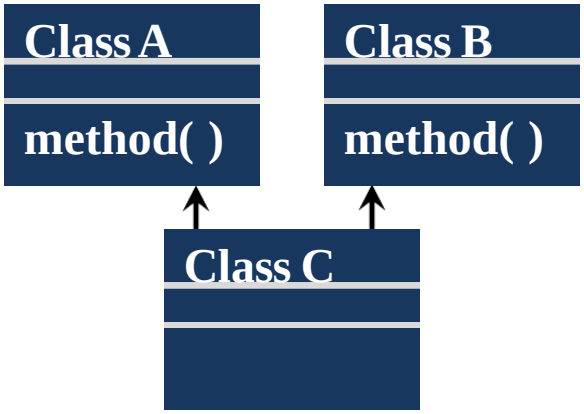


Output:
In method C

Multiple Inheritance - Example 1

Case 2: `method()` exists in parent classes only; A and B

```
class A:
    def method(self):
        print('In method A')
class B:
    def method(self):
        print('In method B')
class C(A, B):
    pass
objectC=C()
objectC.method()
```

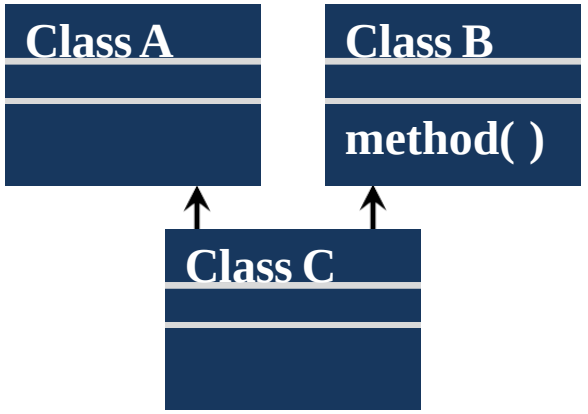


Output:
In method A

Multiple Inheritance - Example 1

Case 3: `method()` exists in class B only

```
class A:
    pass
class B:
    def method(self):
        print('In method B')
class C(A, B):
    pass
objectC=C()
objectC.method()
```

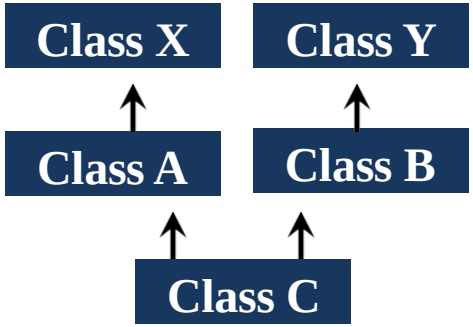


Output:
In method B

Multiple Inheritance and Multi-level Hierarchy

Example 2: Multiple Inheritance in a multi-level hierarchy

Consider the following inheritance hierarchy:



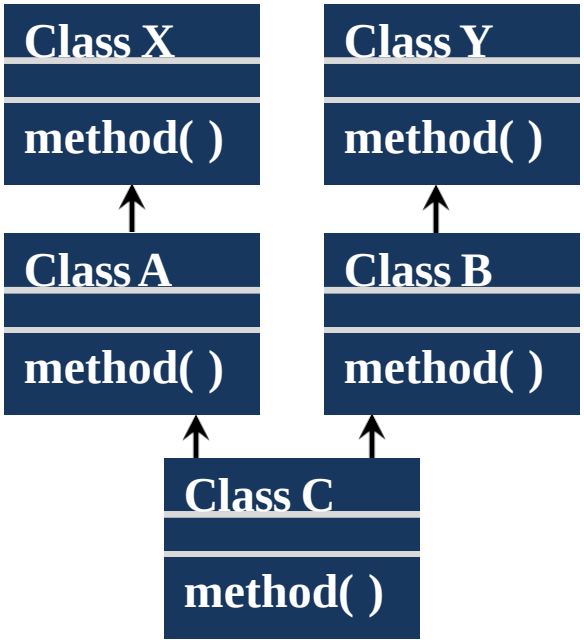
Search order:

- search in the current class
- search the left parent all the way up the hierarchy
- search the right parent all the way up the hierarchy

Multiple Inheritance and Multi-level Hierarchy - Example 2

Case 1: `method()` exists in all classes

```
class X:
    def method(self):
        print('In method X')
class Y:
    def method(self):
        print('In method Y')
class A(X):
    def method(self):
        print('In method A')
class B(Y):
    def method(self):
        print('In method B')
class C(A, B):
    def method(self):
        print('In method C')
objectC=C()
objectC.method()
```

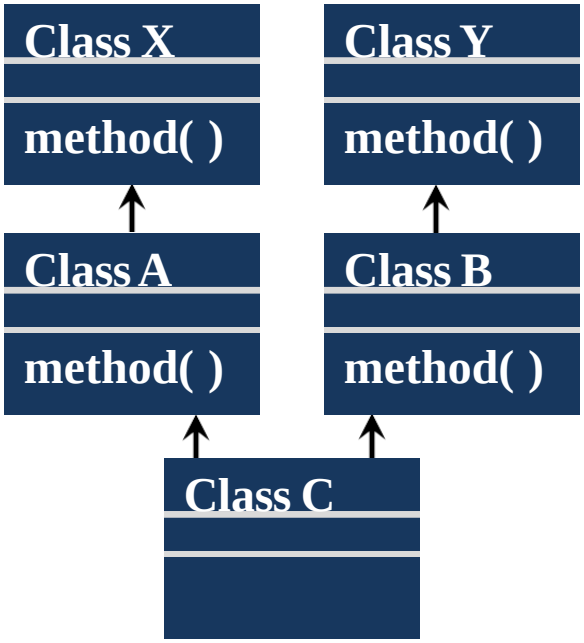


Output:
In method C

Multiple Inheritance and Multi-level Hierarchy - Example 2

Case 2: `method()` exists in classes A, B, X and Y

```
class X:
    def method(self):
        print('In method X')
class Y:
    def method(self):
        print('In method Y')
class A(X):
    def method(self):
        print('In method A')
class B(Y):
    def method(self):
        print('In method B')
class C(A, B):
    pass
objectC=C()
objectC.method()
```

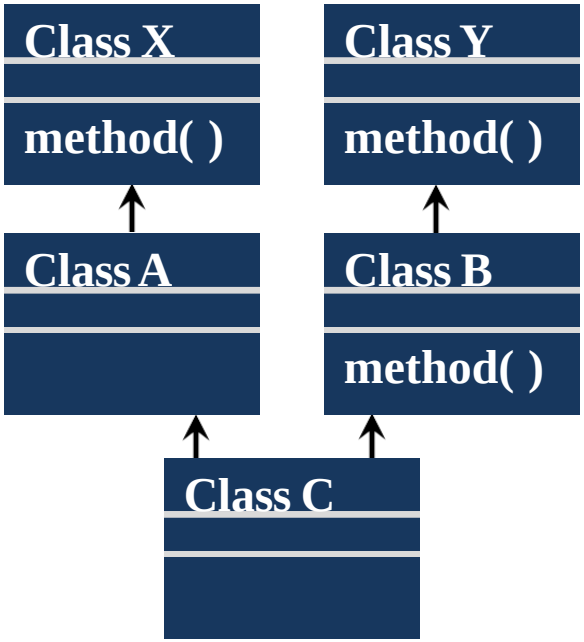


Output:
In method A

Multiple Inheritance and Multi-level Hierarchy - Example 2

Case 3: `method()` exists in classes B, X and Y

```
class X:
    def method(self):
        print('In method X')
class Y:
    def method(self):
        print('In method Y')
class A(X):
    pass
class B(Y):
    def method(self):
        print('In method B')
class C(A, B):
    pass
objectC=C()
objectC.method()
```

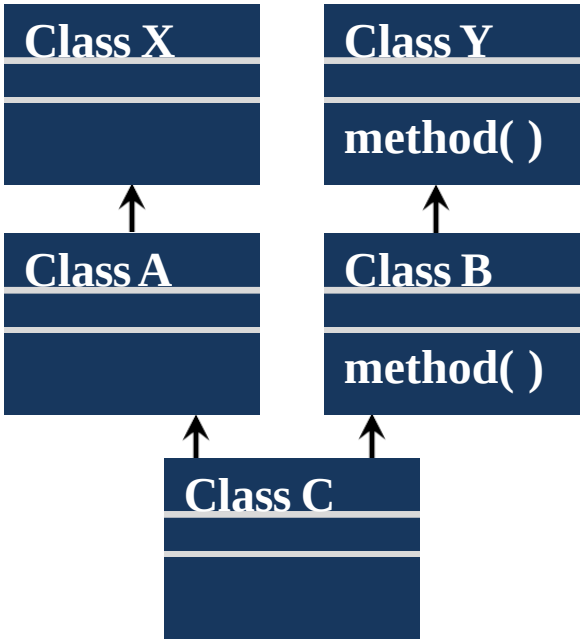


Output:
In method X

Multiple Inheritance and Multi-level Hierarchy - Example 2

Case 4: `method()` exists in classes B and Y

```
class X:
    pass
class Y:
    def method(self):
        print('In method Y')
class A(X):
    pass
class B(Y):
    def method(self):
        print('In method B')
class C(A, B):
    pass
objectC=C()
objectC.method()
```

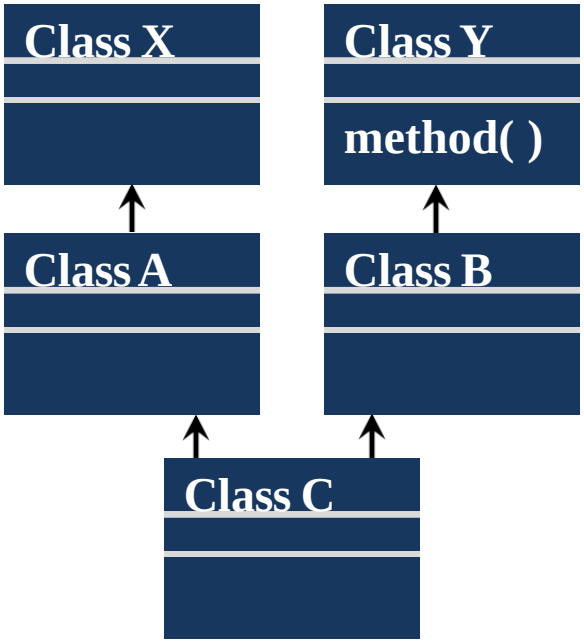


Output:
In method B

Multiple Inheritance and Multi-level Hierarchy - Example 2

Case 5: `method()` exists in class Y only

```
class X:
    pass
class Y:
    def method(self):
        print('In method Y')
class A(X):
    pass
class B(Y):
    pass
class C(A, B):
    pass
objectC=C()
objectC.method()
```

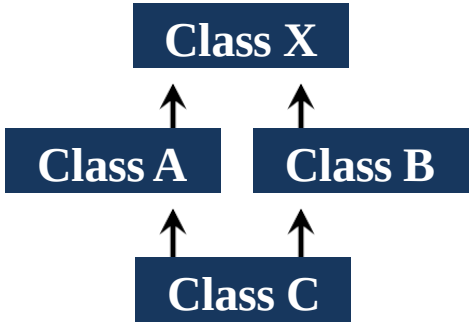


Output:
In method Y

The Diamond Pattern

Example 3: The diamond pattern

Consider the following inheritance hierarchy:



Search order:

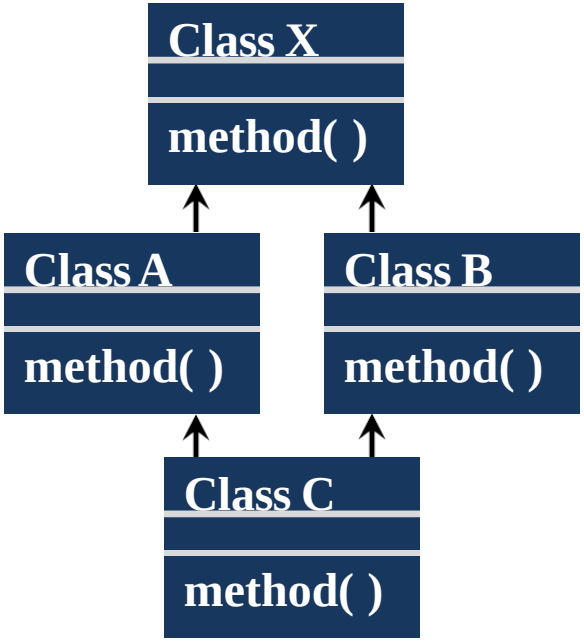
- search in the current class
- search the next level from left to right
- search the peak of the diamond

The Diamond Pattern

Case 1: `method()` exists in all classes

```
class X:
    def method(self):
        print('In method X')
class A(X):
    def method(self):
        print('In method A')
class B(X):
    def method(self):
        print('In method B')
class C(A, B):
    def method(self):
        print('In method C')

objectC=C()
objectC.method()
```

Output:
In method C

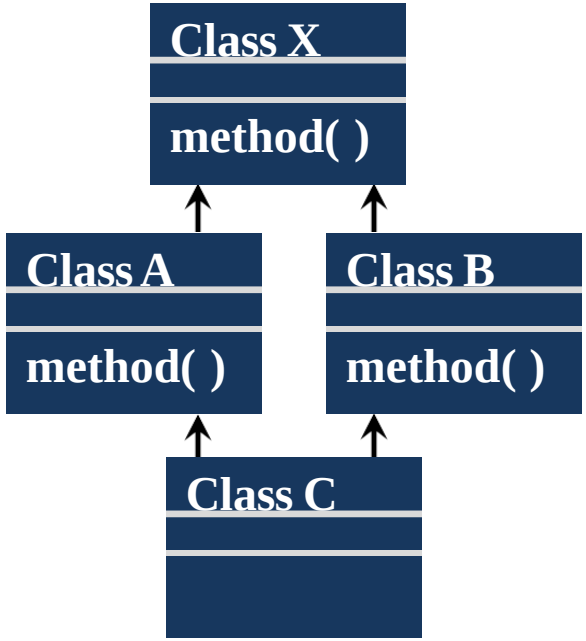


The Diamond Pattern

Case 2: `method()` exists in classes A, B and X

```
class X:
    def method(self):
        print('In method X')
class A(X):
    def method(self):
        print('In method A')
class B(X):
    def method(self):
        print('In method B')
class C(A, B):
    pass

objectC=C()
objectC.method()
```

Output:
In method A

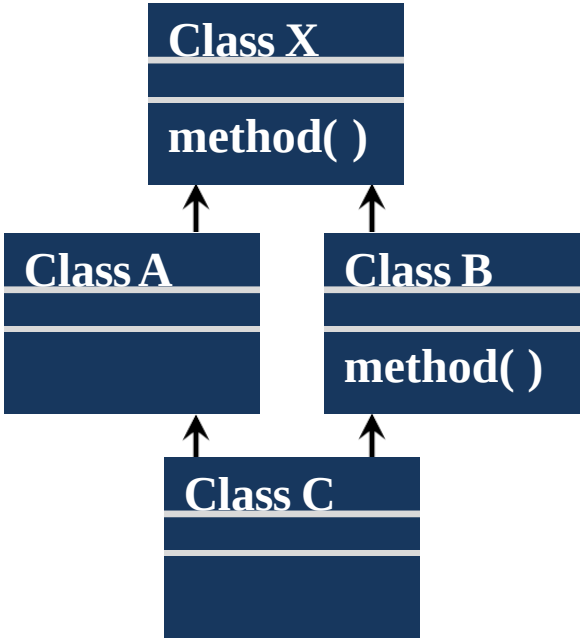


The Diamond Pattern

Case 3: `method()` exists in classes B and X

```
class X:
    def method(self):
        print('In method X')
class A(X):
    pass
class B(X):
    def method(self):
        print('In method B')
class C(A, B):
    pass

objectC=C()
objectC.method()
```

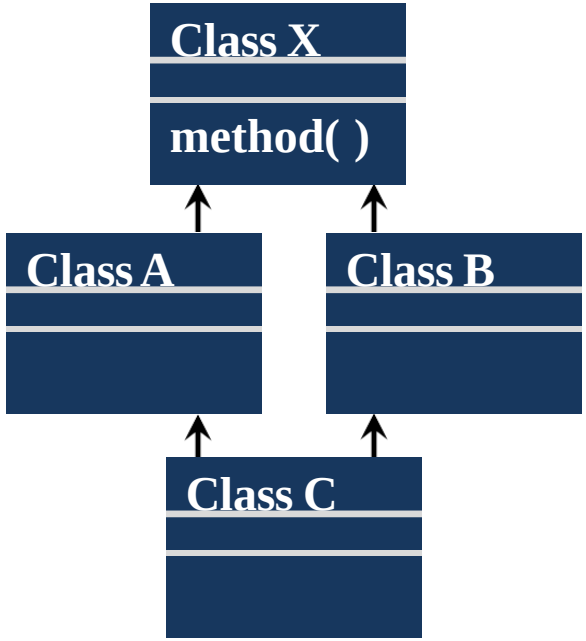
Output:
In method B



The Diamond Pattern

Case 4: `method()` exists in class X only

```
class X:
    def method(self):
        print('In method X')
class A(X):
    pass
class B(X):
    pass
class C(A, B):
    pass

objectC=C()
objectC.method()
```



Output:
In method X

Method Resolution Order (MRO)

- Method Resolution Order (MRO) is the order in which Python looks for a method in a hierarchy of classes
  - Especially in the context of multiple inheritance as single method may be found in multiple super classes

MRO

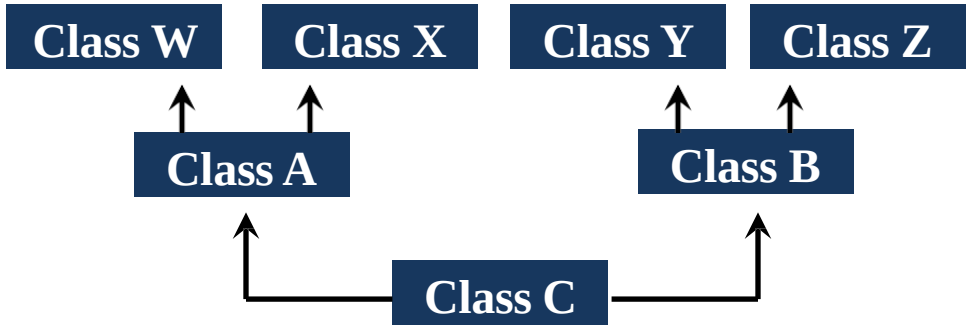
- When searching for an attribute, Python's inheritance search traverses all superclasses in the class header from left to right until a match is found
- Each parent is searched depth-first all the way to the top of the inheritance tree, and then from left to right
  - This order is usually called DFLR, for its depth-first, left-to-right path
- In diamond patterns, the search proceeds across by tree levels before moving up, in a more breadth-first fashion

Method Resolution Order (MRO)

- This order applies to attributes as well as methods
- A method from a superclass can still be called explicitly whenever needed by calling the method through the <class_name >
  Fro example: `X.method( )` can be called at any time in the code of slide 20
- The order in which methods can be called can be adapted on the fly by modifying the __mro__ (Method Resolution Order) attribute on the class

More Examples

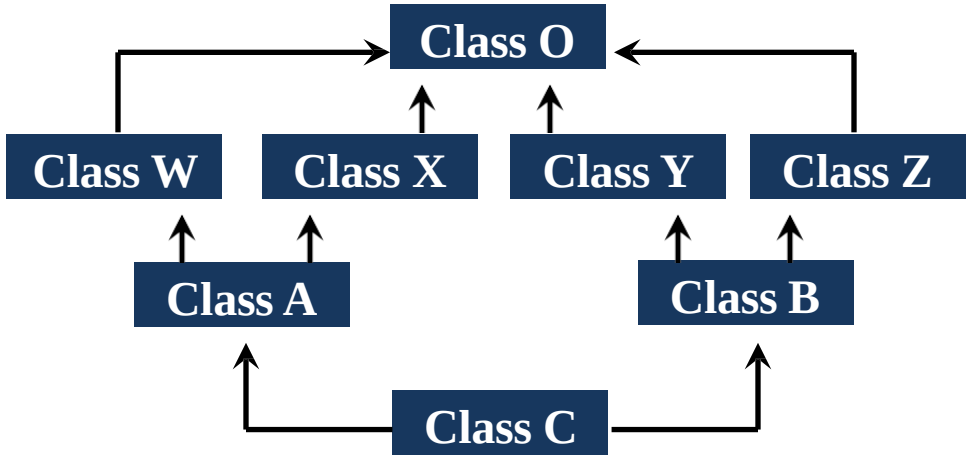
Example 1



Search Order: C, A, W, X, B, Y, Z

More Examples

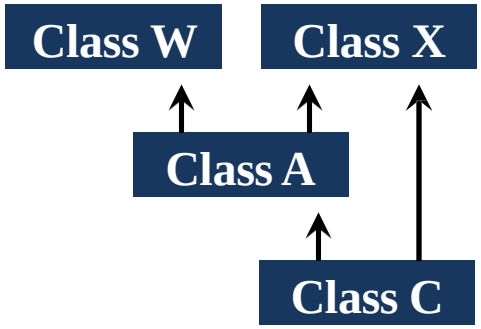
Example 2



Search Order: C, A, W, X, B, Y, Z, O

More Examples

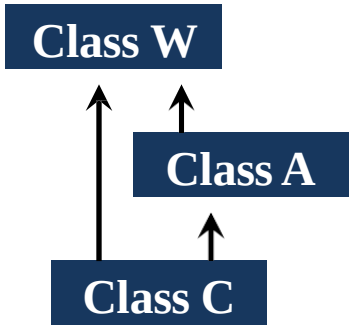
Example 3



Search Order: C, A, W, X

More Examples

Example 4



Expected Search Order: C, W,A, W

Error: cannot create a consistent method resolution order

- **A superclass cannot appear before a subclass**

Object Oriented Programming

Lecture 12 and 13

Decorators

Functions as Parameters

- Everything in Python is an object
- Functions are objects too
  - can be assigned to names, passed to functions, stored in data structures, and so on
  - a function can be defined within another function – nested function

Example 1: assigning a function to name

```
def funcObject():
    print('I am function object')
```

Output:
I am function object
I am function object
<function funcObject at 0x0381A618>

`m=funcObject()` ⟶ executed, `m` gets the return value, `None` in this case
`s=funcObject` ⟶ assigned to name `s`
`s()` ⟶ executed, equivalent to `funcObject()`
`print(s)` ⟶ printed a an object

Functions as Parameters

Example 2: passing function as parameter

```
def funcObject():
    print('I am function object')

def func_a(x):
    print(x)
    x()

func_a(funcObject)
s=funcObject
func_a(s)
```

`print(x)` → printed as an object

`x()` → executed

`func_a(funcObject)` → passed as parameter

Output:
<function funcObject at 0x039AA618>
I am function object

Functions as Parameters

Example 3: returning function from another function

```
def funcObject():
    print('I am function object')

def func_c():
    return funcObject
```

→ returned as an object

```
print(func_c())
```

→ func_c is executed, returns funcObject to be printed as an object

```
def func_c():
    return funcObject()
```

→ executed

```
print(func_c())
```

→ prints return value from funcObject

Output:
<function funcObject at 0x0396D660>
I am function object
None

Decorators

- A decorator is a tool for adding new functionality to an existing object without modifying its structure
- It comes in two flavors: Function decorators and Class decorators

- **Function decorators:**
  - augment function definitions
  - wrap a function in an extra layer of logic implemented as another function in order to extend the behavior of wrapped function, without permanently modifying it
  - act like metafunctions
- **Class decorators:**
  - augment class definitions
  - specify special operational modes for classes, adding support for management of whole objects and their interfaces
  - act like metaclasses

Function Decorators

- In decorators, functions are taken as argument into another function and then called inside a wrapper function
- Decorators are usually called before the definition of a function you want to decorate

Components:

- Define a decorator
- Use the decorator on a function
- Call the function

Function Decorators

Syntax to define a decorator:

```
def <decorator>(<function>):
    def <wrapper>:
        #add code
        #call the function
        #add more code
    return <wrapper>
```

Syntax to call the decorator:

```
@<decorator>
<function definition>
```

```
<function definition>
function=decorator(function)
```

Points to the object wrapper

Syntax to call the function:

```
function()
```

Function Decorators - Example

Another Example: A decorator to calculate run-time of a function

```
import time
def calcTime(func):
    def wrapper():
        begin=time.time()
        func()
        end=time.time()
        print('Run-time:',end-begin,'seconds')
    return wrapper
@calcTime
def long_loop():
    print('Take your time\nHit any alphanumeric key to exit!')
    input()
long_loop()
```

Object Oriented Programming

Static and Class Methods
Abstract Classes

Types of Methods

- Three types of methods can be defined in a class:
  - Instance methods
  - Class methods
  - Static methods

Types of Methods

Instance Methods

- Accessed via an instance
  - Also called bound methods, as they bound to an instance
- Take at least one argument – the self
- Different instances of a class have different values associated with them
- Can access and manipulate instance as well as class attributes/data
- Within the class: accessed using the self operator
- From outside the class: accessed using instance variables

Types of Methods

Class Methods

- Not bounded with any specific object of the class: instance-less methods
- Take at least one argument – the cls
- Accessed via class name
- Usually keep track of information that spans all instances
- Can only manipulate class attributes/data
- Defined using the decorator @classmethod
- Within the class: usually accessed using the class name
- From outside the class: accessed using the class name as well as instance variables

Types of Methods

Static Methods

- Not bounded with any specific object of the class: instance-less methods
  - Also known as unbound methods in Python 2.0
- Do not have an instance of the class or class itself as the first argument – no self or cls argument
- Used as utility function – perform common actions
- Can only manipulate class attributes/data
- Defined using the decorator @staticmethod - *optional*
- Within the class: usually accessed using the class name
- From outside the class: accessed using the class name as well as instance variables (only if @staticmethod is used)

Types of Methods

Example

```
class myClass:
    classAttribute='ClassAttribute'

    def__init__(self):
        self.instanceAttribute='InstanceAttribute'

    def instanceMethod(self):
        print(myClass.classAttribute)
        print(self.instanceAttribute)
        print('This is an instance method')
```

Types of Methods - **Example**

```
@classmethod
def classMethod(cls):
    print(cls.classAttribute)
    print(self.instanceAttribute)
    print(cls.instanceAttribute)
    print('This is a class method')

@staticmethod
def staticMethod():
    print(myClass.classAttribute)
    print(self.instanceAttribute)
    print(cls.instanceAttribute)
    print('This is a static method')
```

→ error

→ error

→ error

→ error

Types of Methods - **Example**

```
    def anotherMethod(self):
        self.instanceMethod()
        myClass.classMethod()
        myClass.staticMethod()
```

} can be accessed using self as well

```
a=myClass()
a.instanceMethod()
myClass.instanceMethod()
```

→ error

```
myClass.classMethod()
a.classMethod()
```

```
myClass.staticMethod()
a.staticMethod()
```

} usually not accessed from here

Abstract Classes – **Recap from Lecture 3**

Abstract Method

- An abstract method is declared in a class, but contains no implementation

Abstract Class

- A class containing an abstract method becomes an abstract class
- Objects of an abstract class cannot be instantiated
- If a sub-class inherits an abstract method from a super-class, it must provide a concrete implementation of that method or else it will be an abstract class itself
- An abstract class can have more than one abstract as well as concrete methods

The Need for Abstract Classes

Interface

- Interfaces form a contract between the class and the outside world
- An interface is the collection of attributes and methods that other objects can use to interact with that object
- Supports the idea of data hiding and encapsulation

- By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses
  - An abstract class serves as a blueprint for other classes

- This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins
- Also helps when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

How to Make an Abstract Base Class (ABC)

- Python comes with a module called `abc` which provides the base for defining Abstract Base classes(ABC)
- To make a class ABC:
  - import abc module
  - inherit your class from the built-in class ABC found in abc module
  - use @abstractmethod decorator with method you want to make abstract

Syntax

```
from abc import ABC, abstractmethod
class <my_ABC_name>(ABC):
    @abstractmethod
    def <method_name>():
        <implementation>
```

How to Make an Abstract Base Class (ABC)

Example 1: object of an abstract class cannot be instantiated

```
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noOfSides(self):
        pass
class Square(Polygon):
    def noOfSides(self):
        print('I have 4 sides')
class Triangle(Polygon):
    def noOfSides(self):
        print('I have 3 sides')
```

```
a=Polygon()
b=Square()
b.noOfSides()
c=Triangle()
c.noOfSides()
```

→ error

Output:
I have 4 sides
I have 3 sides

How to Make an Abstract Base Class (ABC)

Example 2: an abstract method can have implementation

```
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noOfSides(self):
        print('I am a type of polygon')
class Square(Polygon):
    def noOfSides(self):
        super().noOfSides()
        print('I have 4 sides')

b=Square()
b.noOfSides()
```

Output:
I am a type of polygon
I have 4 sides

How to Make an Abstract Base Class (ABC)

Example 3: an abstract method can have concrete methods as well

```
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noOfSides(self):
        print('I am a type of polygon')
    def shape(self):
        print('I am a 2D shape')
class Square(Polygon):
    def noOfSides(self):
        super().noOfSides()
        print('I have 4 sides')
```

```
b=Square()
b.noOfSides()
b.shape()
```

Output:
I am a type of polygon
I have 4 sides
I am a 2D shape

Object Oriented Programming

Exceptions

Exceptions

- Exceptions are special objects used to flag exceptional conditions, especially errors

- Examples of exceptional situations include:
  - attempting to read past the end of a file
  - evaluating the expression lst[i] where lst is a list, and i≥len(lst)
  - attempting to convert a nonnumeric string to a number
  - attempting to read a variable that has not been defined

  } cause run-time errors

- Exceptions represent a standard way to deal with run-time errors
- Python provides a comprehensive, uniform exception handling framework
- The proper use of Python's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors

Built-in Exceptions

- Python provides a number of built-in exception classes
- We can have user-defined exceptions too
- These exceptions all use a common syntax and are completely compatible with each other

Example of raising an exception:

```
>>> print 'Hello World')
SyntaxError: Missing parentheses in call to 'print'. Did you
mean print('Hello World'))?
```

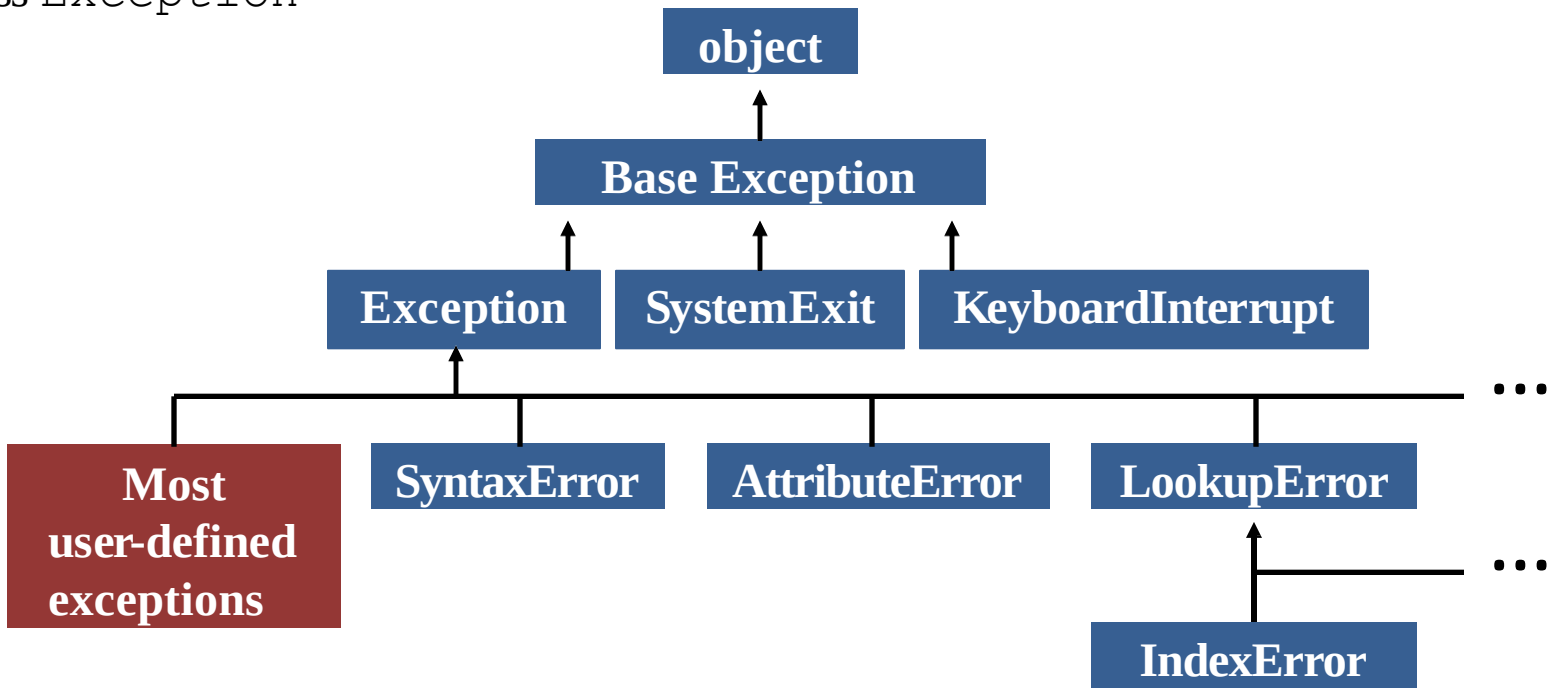
Exception class

Common Standard Exceptions

Class	Meaning
SyntaxError	Incorrect syntax
AttributeError	Object does not contain the specified instance variable or method
ImportError	The import statement fails to find a specified module or name in that module
IndexError	A sequence (list, string, tuple) index is out of range
KeyError	Specified key does not appear in a dictionary
NameError	Specified local or global name does not exist
TypeError	Operation or function applied to an inappropriate type
ValueError	Operation or function applied to correct type but inappropriate value
ZeroDivisionError	Second operand of division or modulus operation is zero

The Exception Hierarchy

- All exceptions inherit from a built-in class called `BaseException` or its built-in child class `Exception`



Exception Control Flow

- When an exception occurs
  - the normal execution flow of the program is interrupted
  - the execution switches to the so called exceptional control flow
- This can be done either by Python's default exception handler or by user-defined exception handler

Default exception handler

- Uses built-in exception classes
- Terminates the program and prints the error message contained in the exception object
  - also prints a traceback, which consists of all the function calls that got interrupted

User-defined exception handler

- try-except statements can be used in the code to decide what to do when an exception occurs

Catching and Handling Exceptions

Syntax for try-except statements

```
try:
      <code block 1>
except:
      <code block 2>
<code block 3>
```

Contains the code where exception might occur

The handler block which catches the exception

Flow

- <code block 1> is executed
- If exception occurs:
  - remaining statements in <code block 1> are skipped
  - <code block 2> is executed
- <code block 3> is executed

Catching and Handling Exceptions

Example: Write a program to input an amount of money and the number of its sharers; print the amount each sharer gets

Default exception handling

```
amount=int(input('Enter amount to be shared: '))
sharers=int(input('Enter number of sharers: '))
print('Each one will get Rs. ',amount/sharers)
print('Have a blessed day')
```

Output:

```
Enter amount to be shared: two hundred
Traceback (most recent call last):
  File "C:/Users/maria/AppData/Local/Programs/Python/Python37-32/CS-116-OOP-Lecture20-Code.py", line 3, in <module>
    amount=int(input('Enter amount to be shared: '))
ValueError: invalid literal for int() with base 10: 'two hundred'
```

Catching and Handling Exceptions - Example

User-defined exception handling

```
try:
    amount=int(input('Enter amount to be shared: '))
    sharers=int(input('Enter number of sharers: '))
    print('Each one will get Rs. ',amount/sharers)
except:
    print('Enter inputs in digits!')
print('Have a blessed day')
```

Output (when no exception occurs):
Enter amount to be shared: 12000
Enter number of sharers: 4
Each one will get Rs. 3000.0
Have a blessed day

Output (when exception occurs):
Enter amount to be shared: hundred
Enter inputs in digits!
Have a blessed day

Catching Multiple Exceptions

Example: Apart from `ValueError`, the code can also raise `ZeroDivisionError`

```
try:
    amount=int(input('Enter amount to be shared: '))
    sharers=int(input('Enter number of sharers: '))
    print('Each one will get Rs. ',amount/sharers)
except ValueError:
    print('Enter inputs in digits!')
except ZeroDivisionError:
    print('Number of sharers must be >=1')
print('Have a blessed day')
```

Output (when ValueError occurs):
Enter amount to be shared: e
Enter inputs in digits!
Have a blessed day

Output (when ZeroDivisionError occurs):
Enter amount to be shared: 12000
Enter number of sharers: 0
Number of sharers must be >=1
Have a blessed day

The Catch-all Exception Handler

Example: Catching an unexpected exception / all other exceptions

- If an exception is raised that does match the type specified in the except statement, then the except statement will not catch it; instead the default handler will handle it

```
try:
    amount=int(input('Enter amount to be shared: '))
    sharers=int(input('Enter number of sharers: '))
    print('Each one will get Rs. ',amount/sharers)
except ValueError:
    print('Enter inputs in digits!')
except ZeroDivisionError:
    print('Number of sharers must be >=1')
except:
    print('Something went wrong!')
print('Have a blessed day')
```

Catching Exception Objects

- **The `as` keyword captures an exception object as a variable**

Example

```
try:
    amount=int(input('Enter amount to be shared: '))
    sharers=int(input('Enter number of sharers: '))
    print('Each one will get Rs. ',amount/sharers)
except ValueError as e:
    print('Problem with value:',type(e),e)
except ZeroDivisionError as e:
    print('Problem with value:',type(e),e)
except:
    print('Cannot identify the problem')
print('Have a blessed day')
```

Raising an Exception

- The `raise` keyword raises an exception

must be an exception object

Argument(s) passed to the constructor

Syntax

```
raise <exception_object>(<custom_error_message>)
```

- Calls the constructor of the `<exception_object>`

Example

```
raise ValueError('Cannot enter a negative value')
```

- If raised and uncaught, the interpreter will print the message line at the end of the stack trace:
  ValueError: Cannot enter a negative value

- Alternatively, the raised exception can be caught in the except block

Raising an Exception

- If none of the Python's built-in exception types match your need, use the generic `Exception` class and provide a descriptive message to its constructor

- Sometimes it is appropriate for a function (or method) to catch an exception, take some action appropriate to its local context, and then re-raise the same exception so that the function's caller can take further action if necessary

The `else` and `finally` blocks

- The Python `try` statement supports optional `else` and `finally` blocks

- The `else` block is executed only if the statements in the `try` block don't raise an exception
  - If present, must appear after all of the `except` blocks

- Code within a `finally` block always executes whether the `try` block raises an exception or not
  - If present, must appear after all `except` blocks and after the `else` block
  - Usually contains clean-up code that must execute due to activity initiated in the `try` block

- The `try` keyword cannot appear without at least one of `except` or `finally`
  - This means the `except` blocks are also optional
- The `except` and `finally` blocks may not appear without an associated `try` block
- An `else` block must be used in the context of a `try` statement (or `while`, or `for` statement)

Creating Exception

- We can create and raise our own exceptions if we find that none of the built-in exceptions are suitable

- Exceptions are objects – creating an exception means defining a new class
  - must inherit from the `Exception` class; can extend `BaseException` directly, but then it will not be caught by generic except Exception clauses
  - the name of the class is usually designed to communicate what went wrong
  - any arbitrary number of arguments can be provided in the initializer to include additional information

Creating Exception

Example 1: A simple exception

```
class InvalidWithdrawal(Exception):
    pass

try:
    raise InvalidWithdrawal
except InvalidWithdrawal:
    print('I am sorry, but do not have enough balance')
```

Creating Exception

Example 2: Any number of arguments can be passed to the exception constructor

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__(f'Account doesn\'t have ${amount}')
        self.amount = amount
        self.balance = balance
    def overage(self):
        return self.amount - self.balance

try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print('I am sorry, but your withdrawal is more than your \
          balance by', e.overage())
```