Case Study: Python
Object Oriented Programming

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2020

Class

- Class vs. Object
- Instance vs. static fields
- Instance, Class, Static methods
- Encapsulation
- Inheritance
- Polymorphism

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:
    'Optional class documentation string'
    <class_suite>
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:   => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>    => fields, methods
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:   => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>    => fields, methods
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>   => fields, methods
```

- Object: an instance of the class

```
class Employee:
    'Common base class for all employees'
    empCount = 0     => empCount: static field
    def __init__(self,n,s):
        self.name = n
        self.salary = s
        Employee.empCount += 1
obj = Employee("Nam",30)
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>   => fields, methods
```

- Object: an instance of the class

```
class Employee:
    'Common base class for all employees'
    empCount = 0     => empCount: static field
    def __init__(self,n,s):=>constructor
        self.name = n
        self.salary = s
        Employee.empCount += 1
obj = Employee("Nam",30)
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>   => fields, methods
```

- Object: an instance of the class

```
class Employee:
    'Common base class for all employees'
    empCount = 0     => empCount: static field
    def __init__(self,n,s):=>constructor
        self.name = n              =>name: instance field
        self.salary = s
        Employee.empCount += 1
obj = Employee("Nam",30)
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>   => fields, methods
```

- Object: an instance of the class

```
class Employee:
    'Common base class for all employees'
    empCount = 0     => empCount: static field
    def __init__(self,n,s):=>constructor
        self.name = n              =>name: instance field
        self.salary = s            =>salary: instance field
        Employee.empCount += 1
obj = Employee("Nam",30)
```

Class vs. Object

- Class is a user-defined prototype for an object

```
class <Clsname>:  => Clsname: name of the class
    'Optional class documentation string'
    <class_suite>   => fields, methods
```

- Object: an instance of the class

```
class Employee:
    'Common base class for all employees'
    empCount = 0     => empCount: static field
    def __init__(self,n,s):=>constructor
        self.name = n              =>name: instance field
        self.salary = s            =>salary: instance field
        Employee.empCount += 1
obj = Employee("Nam",30)  => create object
```

Instance Methods

- Instance Method: method belongs to the instance

```
class Employee:
   def displayEmployee(self):
      print( "Name : ", self.name,
                  ", Salary: ", self.salary)
obj = Employee("Nam",30)
obj.displayEmployee()
```

Instance Methods

- Instance Method: method belongs to the instance

```
class Employee:
   def displayEmployee(self):
```

=> first parameter for instance

```
      print( "Name : ", self.name,
                  ", Salary: ", self.salary)
obj = Employee("Nam",30)
obj.displayEmployee()
```

Instance Methods

- Instance Method: method belongs to the instance

```
class Employee:
   def displayEmployee(self):  => first parameter for instance
      print( "Name : ", self.name,
                  ", Salary: ", self.salary)
obj = Employee("Nam",30)
obj.displayEmployee()  =>obj is passed to self
```

Instance Methods

- Instance Method: method belongs to the instance

```
class Employee:
   def displayEmployee(self):  => first parameter for instance
      print( "Name : ", self.name,
                  ", Salary: ", self.salary)
obj = Employee("Nam",30)
obj.displayEmployee()  =>obj is passed to self
```

Instance Methods

- Instance Method: method belongs to the instance

```
class Employee:
   def displayEmployee(self):  => first parameter for instance
      print( "Name : ", self.name,
                  ", Salary: ", self.salary)
obj = Employee("Nam",30)
obj.displayEmployee()  =>obj is passed to self
```

- able to access to instance fields through first parameter and '.' => self.name
- able to access to static field through class name and '.' => Employee.empCount

Class and Static Methods

```
class Employee:
   @classmethod
   def create(cls,n,s):
      print(cls.empCount)
      return cls(n,s)
   @staticmethod
   def isHighSal(s):
      if s > 8:
         print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):
        print(cls.empCount)
        return cls(n,s)
    @staticmethod
    def isHighSal(s):
        if s > 8:
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)
        return cls(n,s)
    @staticmethod
    def isHighSal(s):
        if s > 8:
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod
    def isHighSal(s):
        if s > 8:
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod                   =>to define static method
    def isHighSal(s):
        if s > 8:
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod                   =>to define static method
    def isHighSal(s):               =>no parameter for class
        if s > 8:
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod                   =>to define static method
    def isHighSal(s):               =>no parameter for class
        if s > 8:                   =>unable to access any fields
            print("High Salary")
obj = Employee.create("Nam",30)
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod                   =>to define static method
    def isHighSal(s):               =>no parameter for class
        if s > 8:                   =>unable to access any fields
            print("High Salary")
obj = Employee.create("Nam",30)  =>Employee is passed to cls
Employee.isHighSal(30)
```

Class and Static Methods

```
class Employee:
    @classmethod                    =>to define class method
    def create(cls,n,s):            =>first parameter for class
        print(cls.empCount)         =>access to static fields
        return cls(n,s)
    @staticmethod                   =>to define static method
    def isHighSal(s):               =>no parameter for class
        if s > 8:                   =>unable to access any fields
            print("High Salary")
obj = Employee.create("Nam",30)  =>Employee is passed to cls
Employee.isHighSal(30)  =>Employee is used to resolve
```

Encapsulation

- to hide fields and methods
- based on name of fields and methods
  - Protected: prefix by a single underscore (_example)
  - Private: prefix by a double underscores (__example)
  - Public: begin with a letter

Class Hierarchy

- Python 3: root is **object**
- Multiple Inheritance

  class <clsname>(<parent>(,<parent>)*)?:

- For example,

  class A: => superclass is **object**
  class Rectangle(Parallelogram):

  class Square(Rhombus, Rectangle):

Class Hierarchy

- Python 3: root is **object**
- Multiple Inheritance

  class <clsname>(<parent>(,<parent>)*)?:

- For example,

  class A: => superclass is **object**
  class Rectangle(Parallelogram):
  => superclass is Parallelogram
  class Square(Rhombus, Rectangle):

Class Hierarchy

- Python 3: root is **object**
- Multiple Inheritance

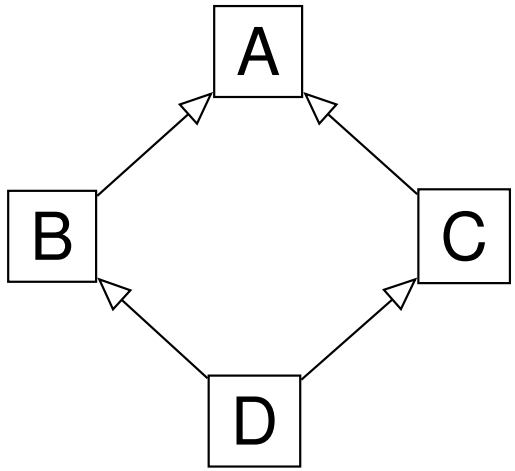
  class <clsname>(<parent>(,<parent>)*)?:

- For example,

  class A: => superclass is **object**
  class Rectangle(Parallelogram):
  => superclass is Parallelogram
  class Square(Rhombus, Rectangle):
  => superclasses are Rhombus and Rectangle

Inheritance

- Subclass inherits non-private fields and methods from super-classes
- Diamond problem



- Method Resolution Order: determine search sequence of a class

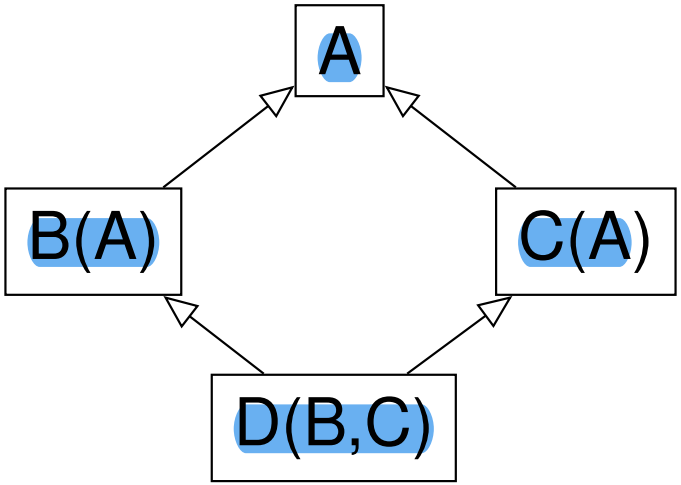
Method Resolution Order (MRO)

- MRO is used to determine search sequence L(M) of class M
  - L(object) = [object]
  - L(M(A,B,C)) = [M] + merge(L(A),L(B),L(C),[A,B,C])

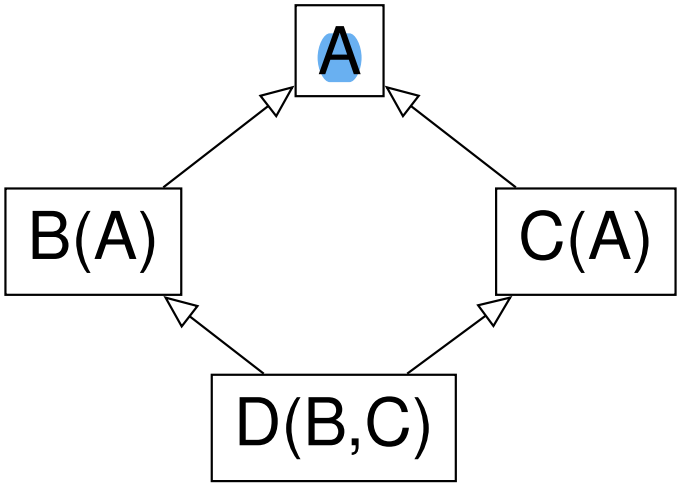
Method Resolution Order (MRO)

- MRO is used to determine search sequence L(M) of class M
  - L(object) = [object]
  - L(M(A,B,C)) = [M] + merge(L(A),L(B),L(C),[A,B,C])
- merge([$H_1|T_1$],[$H_2|T_2$],[$H_3|T_3$])
  - Step 1: if H_1 is a **good** head which is NOT in the tail of other lists, take H_1 out as an output, remove H_1 out of all lists, back to Step 1.
  - Step 2: if H_1 is not a good head, check if H_2 is a good head. If it is, apply Step1 for H_2. If it is not, check for H_3 and so on. If there is no good head, give an error.

Example

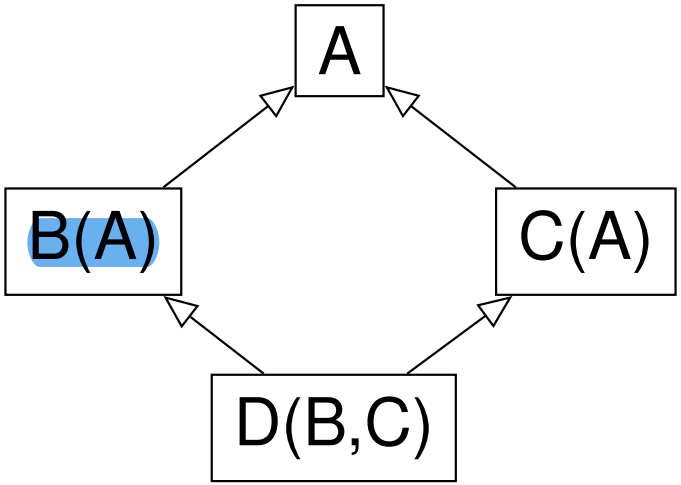


Example



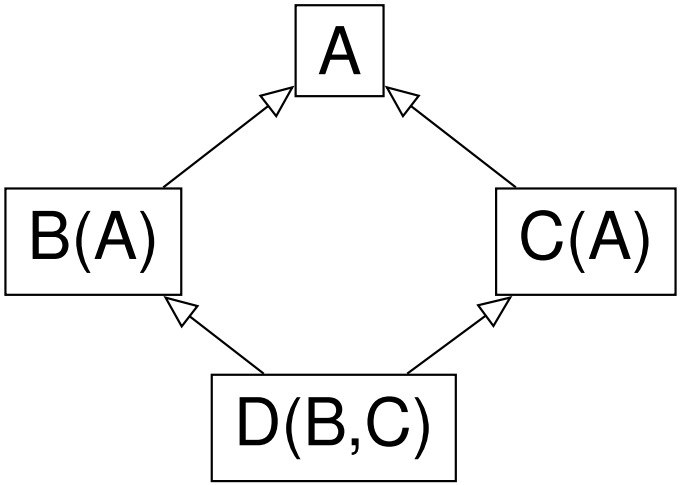
- L(A) = [A,object]

Example



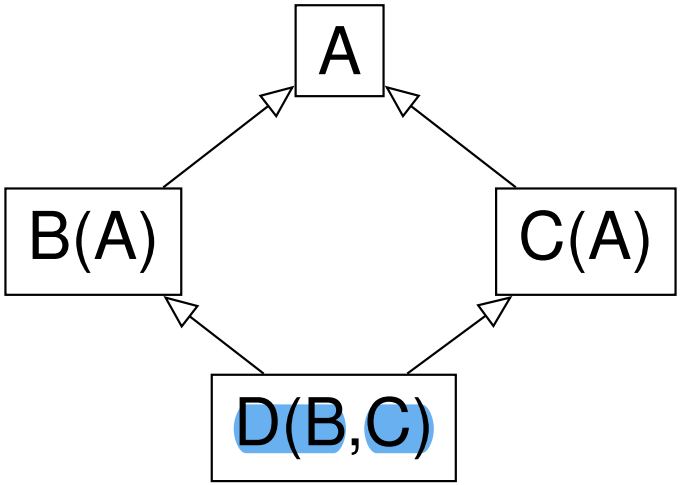
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]

Example



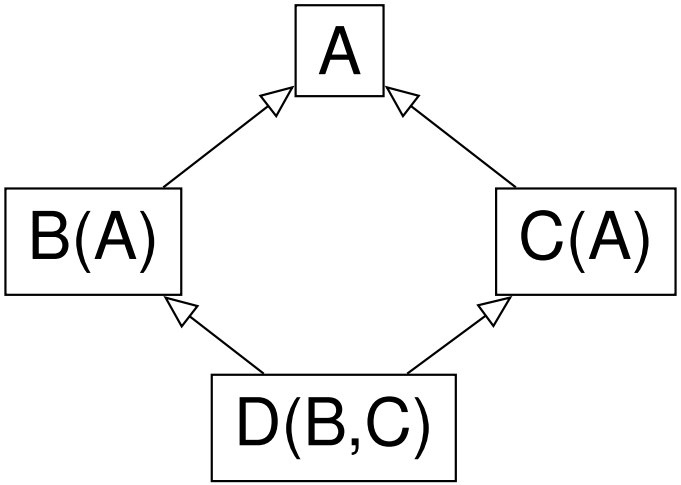
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]

Example



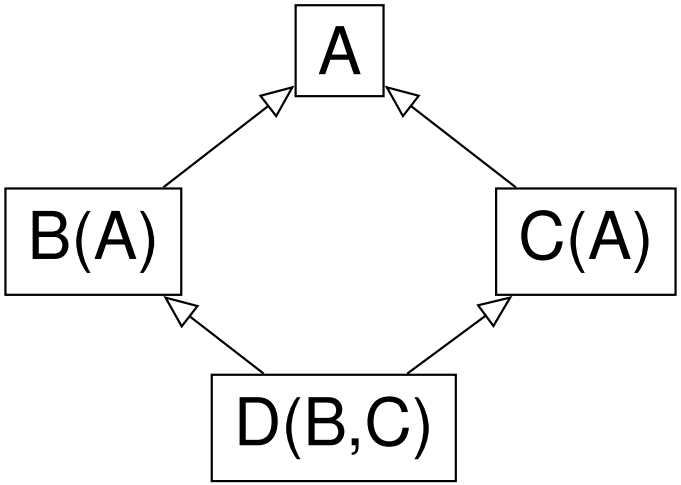
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])

Example



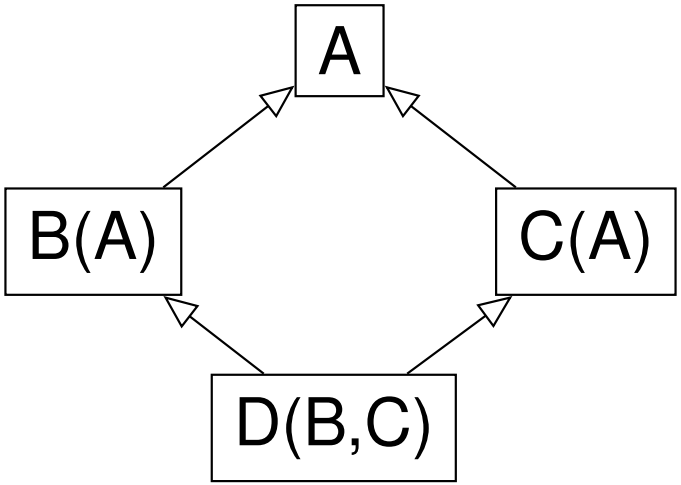
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])
- merge([B,A,o],[C,A,o],[B,C]) =>

Example



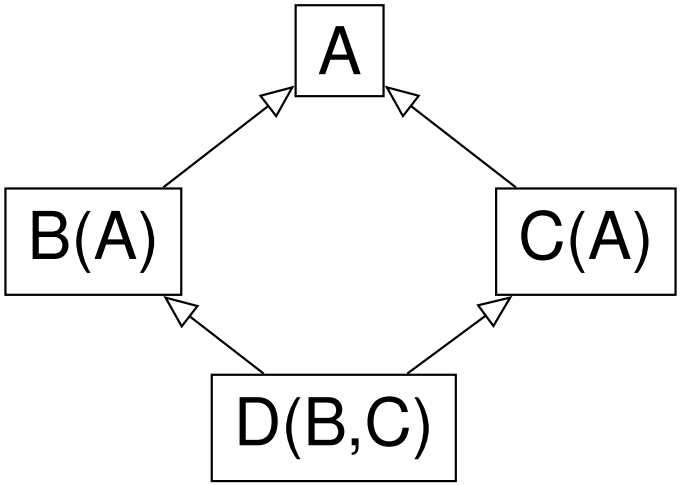
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])
- merge([B,A,o],[C,A,o],[B,C]) =>
  - B is a good head => [B]+merge([A,o],[C,A,o],[C])

Example



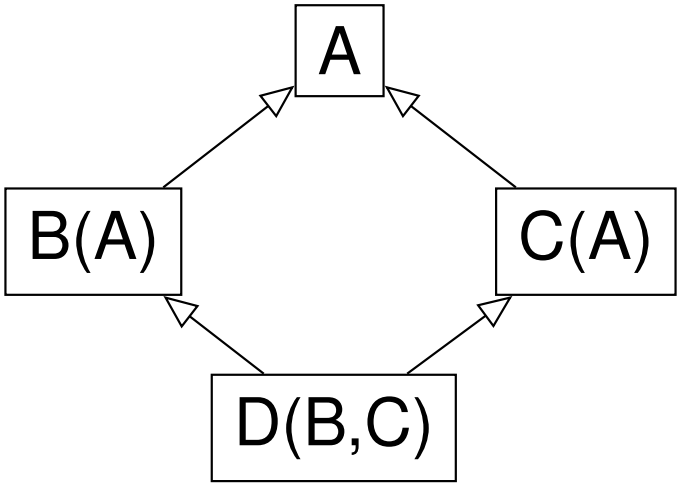
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])
- merge([B,A,o],[C,A,o],[B,C]) =>
  - B is a good head => [B]+merge([A,o],[C,A,o],[C])
  - A is NOT a good head, check C, and C is a good head => [B,C] + merge([A,o],[A,o],[])

Example



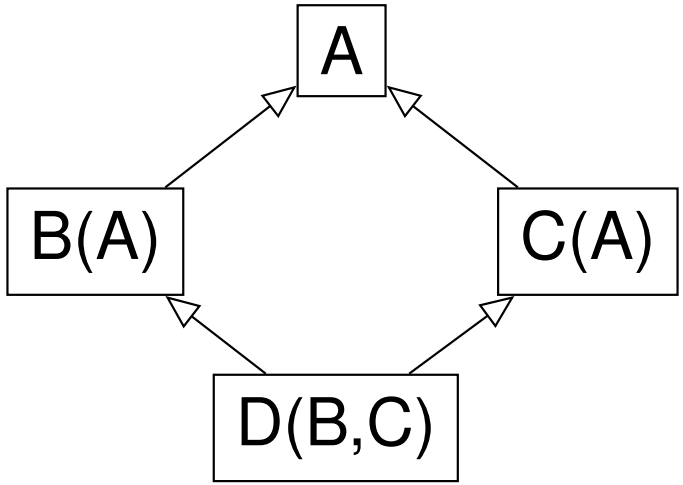
- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])
- merge([B,A,o],[C,A,o],[B,C]) =>
  - B is a good head => [B]+merge([A,o],[C,A,o],[C])
  - A is NOT a good head, check C, and C is a good head => [B,C] + merge([A,o],[A,o],[])
  - A is a good head => [B,C,A] + merge([o],[o],[])

Example



- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C])
- merge([B,A,o],[C,A,o],[B,C]) =>
  - B is a good head => [B]+merge([A,o],[C,A,o],[C])
  - A is NOT a good head, check C, and C is a good head => [B,C] + merge([A,o],[A,o],[])
  - A is a good head => [B,C,A] + merge([o],[o],[])
  - o is a good head => [B,C,A,o]

Example



- L(A) = [A,object]
- L(B) = [B] + merge(L(A),[A]) = [B,A,object]
- L(C) = [C] + merge(L(A),[A]) = [C,A,object]
- L(D) = [D]+merge(L(B),L(C),[B,C]) = [D,B,C,A,object]
- merge([B,A,o],[C,A,o],[B,C]) =>
  - B is a good head => [B]+merge([A,o],[C,A,o],[C])
  - A is NOT a good head, check C, and C is a good head => [B,C] + merge([A,o],[A,o],[])
  - A is a good head => [B,C,A] + merge([o],[o],[])
  - o is a good head => [B,C,A,o]

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)  => 3
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
isinstance(x,B)
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
```

isinstance(x,B) => True

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
isinstance(x,A)
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
isinstance(x,A) => True
```

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
```

- **type(o)** => return the type of object **o**

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
type(x) is B
```

- **type(o)** => return the type of object **o**

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
```

type(x) is B => True

- **type(o)** => return the type of object **o**

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
type(x) is A
```

- **type(o)** => return the type of object **o**

super, isinstance, type

- **super()** => refer to the superclass

```
1 class A:
2     def foo(self,x):
3         print(x)
4 class B(A):
5     def foo(self,x):
6         super().foo(x)
7 B().foo(3)
```

- **isinstance(o,T)** => check if object **o** is of type **T**

```
class A: pass
class B(A): pass
x = B()
```

type(x) is A => False

- **type(o)** => return the type of object **o**

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

Parametric Polymorphism

```
class A:
  def func1(self):
    print("A")

class B:
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
```

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

Parametric Polymorphism

```
class A:
  def func1(self):
    print("A")

class B:
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
```

Subtyping Polymorphism

```
class A:
  def func1(self):
    print("A")

class B(A):
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
```

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

Parametric Polymorphism

```
class A:
  def func1(self):
    print("A")
  def func2(self): pass
class B:
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
  x.func2()
```

Subtyping Polymorphism

```
class A:
  def func1(self):
    print("A")

class B(A):
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
```

Polymorphism

- **Overloading**:

```
def func(param1,param2=0): pass
func(1,2)
func("asbc")
```

- **Universal Polymorphism**:

Parametric Polymorphism

```
class A:
  def func1(self):
    print("A")
  def func2(self): pass
class B:
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
  x.func2()
```

Subtyping Polymorphism

```
class A:
  def func1(self):
    print("A")
  def func2(self): pass
class B(A):
  def func1(self):
    print("B")
for x in [A(),B()]:
  x.func1()
  x.func2()
```

Some other issues

- **id()**: address of the specified object

```
x,y,z = 3,3,4
id(x)
id(y)
id(z)
```

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x) => address of object 3 which x points to
  id(y)
  id(z)

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x) => address of object 3 which x points to
  id(y) => same above address => x, y -> same object
  id(z)

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x) => address of object 3 which x points to
  id(y) => same above address => x, y -> same object
  id(z) => z points to different object

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object
  - **==**: True even when they are different objects but their attributes are equal

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object
  - **==**: True even when they are different objects but their attributes are equal

  x,y = [1,2,3],[1,2,3]
  x **is** y
  id(x) == **id**(y)
  x == y

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object
  - **==**: True even when they are different objects but their attributes are equal

  x,y = [1,2,3],[1,2,3]
  x **is** y => False
  id(x) == **id**(y)
  x == y

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object
  - **==**: True even when they are different objects but their attributes are equal

  x,y = [1,2,3],[1,2,3]
  x **is** y => False
  id(x) == **id**(y) => False
  x == y

Some other issues

- **id()**: address of the specified object

  x,y,z = 3,3,4
  id(x)
  id(y)
  id(z)

- **is** vs. **==**
  - **is**: True just when they are same object
  - **==**: True even when they are different objects but their attributes are equal

  x,y = [1,2,3],[1,2,3]
  x **is** y => False
  id(x) == **id**(y) => False
  x == y => True

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

  - **<expression>** must return an object which has **__enter__** and **__exit__** methods

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

  - **<expression>** must return an object which has __**enter**__ and __**exit**__ methods
  - The **with** statement is executed like:

    <variable> = <expression>
    <variable>.__enter__()
    <stmt_list>
    <variable>.__exit__()

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

  - **<expression>** must return an object which has __**enter**__ and __**exit**__ methods
  - The **with** statement is executed like:

    <variable> = <expression>
    <variable>.__enter__()
    <stmt_list>
    <variable>.__exit__()

  - __**exit**__ method is always executed before the control goes out of the **<stmt_list>**

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

  - **<expression>** must return an object which has **__enter__** and **__exit__** methods
  - The **with** statement is executed like:

    <variable> = <expression>
    <variable>.__enter__()
    <stmt_list>
    <variable>.__exit__()

  - **__exit__** method is always executed before the control goes out of the **<stmt_list>**
  - used for managing resources: file, database,...

    with open('abc.txt','r') **as** f:
        print(f.read())
    f.closed

Some other issues

- Context manager by **with** statement

  with <expression> **as** <variable>:
  <stmt−**list**>

  - **<expression>** must return an object which has **__enter__** and **__exit__** methods
  - The **with** statement is executed like:

    <variable> = <expression>
    <variable>.__enter__()
    <stmt_list>
    <variable>.__exit__()

  - **__exit__** method is always executed before the control goes out of the **<stmt_list>**
  - used for managing resources: file, database,...

    with open('abc.txt','r') **as** f:
        print(f.read())
    f.closed => True

References I

[1] Python Tutorial, http:w3schools.com/python, 10 08 2020.

[2] Python Programming Language, https://www.geeksforgeeks.org/python-programming-language/, 10 08 2020.

[3] Python Tutorial, https://www.tutorialspoint.com/python, 10 08 2020.

[4] Introduction to Python 3, https://realpython.com/python-introduction/, 10 08 2020.