

Case Study: Python

Object Oriented Programming

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

08, 2020

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

- Class is a user-defined prototype for an object

```
class <Clsname>:  
    'Optional_class_documentation_string'  
    <class_suite>
```

- Class is a user-defined prototype for an object

```
class <Clsname>: => Clsname: name of the class  
    'Optional_class_documentation_string'  
    <class_suite>
```

- 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 is a user-defined prototype for an object

```
class <Clsname>: => Clsname: name of the class  
    'Optional_class_documentation_string'  
<class_suite>    => fields, methods
```

- 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 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 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 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 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 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 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 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 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 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 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)
```

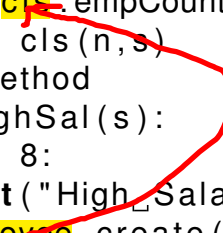
```
class Employee:
    @classmethod
    def create(cls, n, s):
        print(cls.empCount)
        return cls(n, s)
    @staticmethod
    def isHighSal(s):
        if s > 8:
            print("High_Salary")
obj = Employee.create("Nam", 30)
Employee.isHighSal(30)
```

=>to define class method

```
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)
```

=>to define class method
=>first parameter for class

pass



```
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)
```

=>to define class method
=>first parameter for class
=>access to static fields

```
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 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)
```

=>to define class method

=>first parameter for class

=>access to static fields

=>to define static method

=>no parameter for class

```
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)
```

=>to define class method
=>first parameter for class
=>access to static fields

=>to define static method
=>no parameter for class
=>unable to access any fields

```
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)
```

=>to define class method

=>first parameter for class

=>access to static fields

=>to define static method

=>no parameter for class

=>unable to access any fields

=>Employee is passed to cls


```
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)
```

=>to define class method
=>first parameter for class
=>access to static fields
=>to define static method
=>no parameter for class
=>unable to access any fields
=>Employee is passed to cls
=>Employee is used to resolve

- 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

- 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):

- 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):

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

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

- For example,

class A: => superclass is **object**

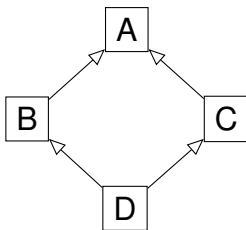
class Rectangle(Parallelogram):

=> superclass is **Parallelogram**

class Square(Rhombus, Rectangle):

=> superclasses are **Rhombus** and **Rectangle**

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

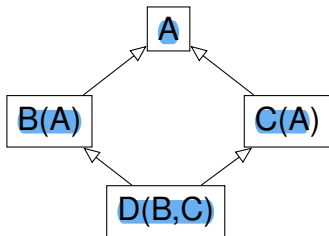


- Method Resolution Order: determine search sequence of a class

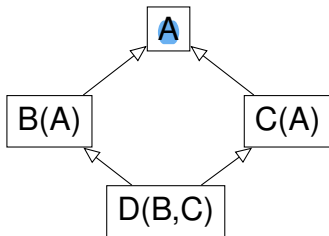
- MRO is used to determine search sequence $L(M)$ of class M
 - $L(\text{object}) = [\text{object}]$
 - $L(M(A,B,C)) = [M] + \text{merge}(L(A), L(B), L(C), [A, B, C])$

- MRO is used to determine search sequence $L(M)$ of class M
 - $L(\text{object}) = [\text{object}]$
 - $L(M(A,B,C)) = [M] + \text{merge}(L(A), L(B), L(C), [A,B,C])$
- $\text{merge}([H_1 | T_1], [H_2 | T_2], [H_3 | T_3])$
 - 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

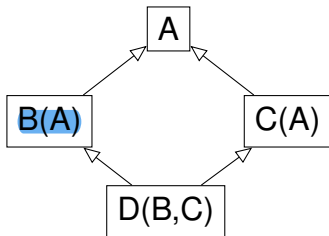


Example

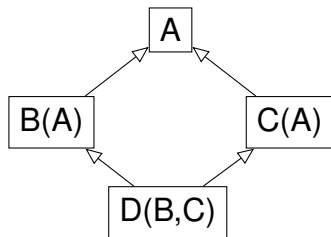


- $L(A) = [A, \text{object}]$

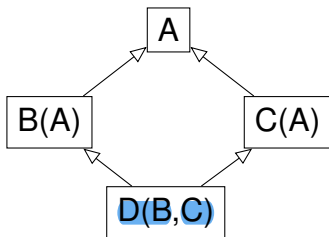
Example



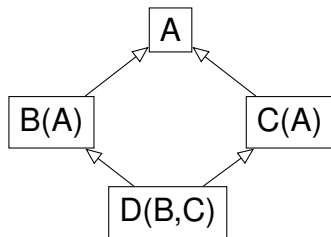
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$



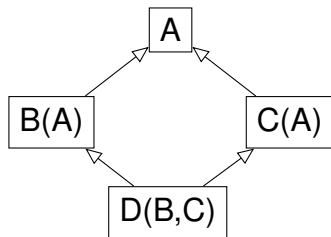
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$



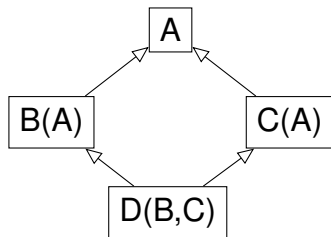
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$



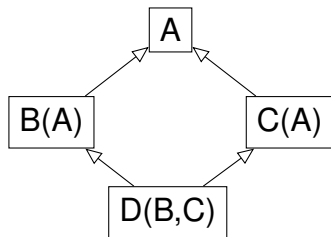
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$



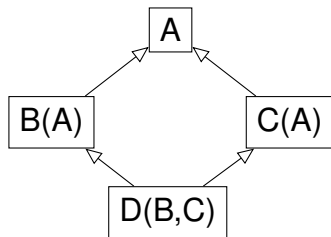
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$



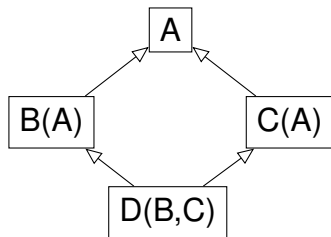
- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$
 - A is NOT a good head, check C, and C is a good head $\Rightarrow [B, C] + \text{merge}([A, o], [A, o], [])$



- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$
 - A is NOT a good head, check C, and C is a good head $\Rightarrow [B, C] + \text{merge}([A, o], [A, o], [])$
 - A is a good head $\Rightarrow [B, C, A] + \text{merge}([o], [o], [])$



- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C])$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$
 - A is NOT a good head, check C, and C is a good head $\Rightarrow [B, C] + \text{merge}([A, o], [A, o], [])$
 - A is a good head $\Rightarrow [B, C, A] + \text{merge}([o], [o], [])$
 - o is a good head $\Rightarrow [B, C, A, o]$



- $L(A) = [A, \text{object}]$
- $L(B) = [B] + \text{merge}(L(A), [A]) = [B, A, \text{object}]$
- $L(C) = [C] + \text{merge}(L(A), [A]) = [C, A, \text{object}]$
- $L(D) = [D] + \text{merge}(L(B), L(C), [B, C]) = [\text{D}, B, C, A, \text{object}]$
- $\text{merge}([B, A, o], [C, A, o], [B, C]) \Rightarrow$
 - B is a good head $\Rightarrow [B] + \text{merge}([A, o], [C, A, o], [C])$
 - A is NOT a good head, check C, and C is a good head $\Rightarrow [B, C] + \text{merge}([A, o], [A, o], [])$
 - A is a good head $\Rightarrow [B, C, A] + \text{merge}([o], [o], [])$
 - o is a good head $\Rightarrow [B, C, A, o]$

- **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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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()** => 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**

- **Overloading:**

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


- **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:**

- **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()
```

- **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()
```

- **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()
```

- **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()
```

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

```
x,y,z = 3,3,4
```

```
id(x)
```

```
id(y)
```

```
id(z)
```

- **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)

- **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)

- **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

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

x,y,z = 3,3,4

id(x)

id(y)

id(z)

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

- **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

- **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

- **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
```

- **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
```

- **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
```


- **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
```

- Context manager by **with** statement
with <expression> **as** <variable>:
 <stmt-list>

- Context manager by **with** statement
with <expression> **as** <variable>:
 <stmt-list>
 - <expression> must return an object which has
 __enter__ and __exit__ methods

- 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__()
```

- 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>

- 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
```

- 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
```

- [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.