Python for Data Scientists
L5: Object oriented programming

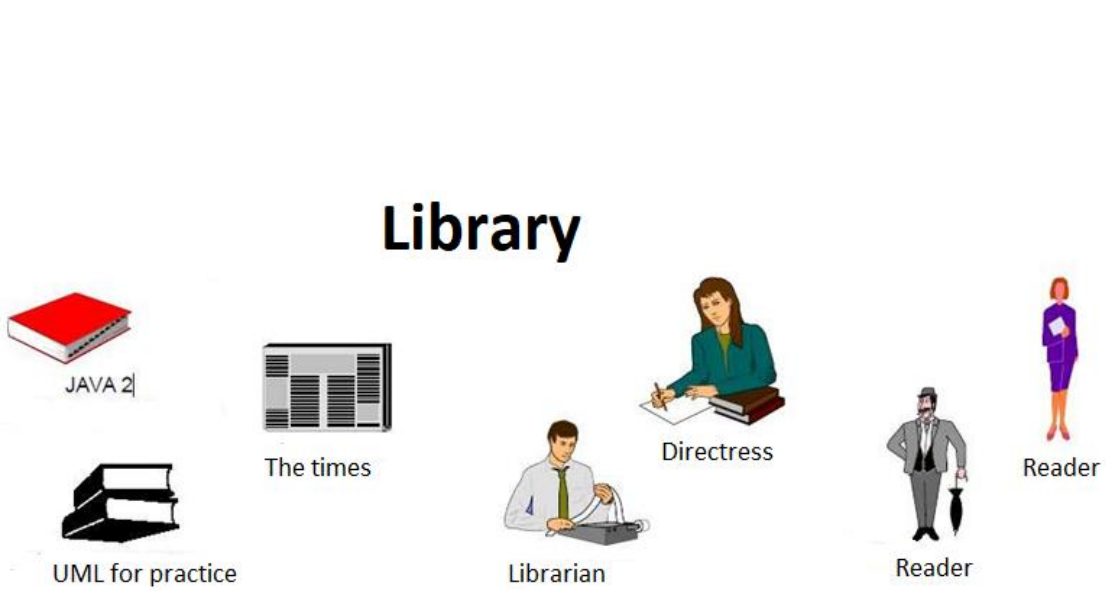
Object-oriented programming OOP

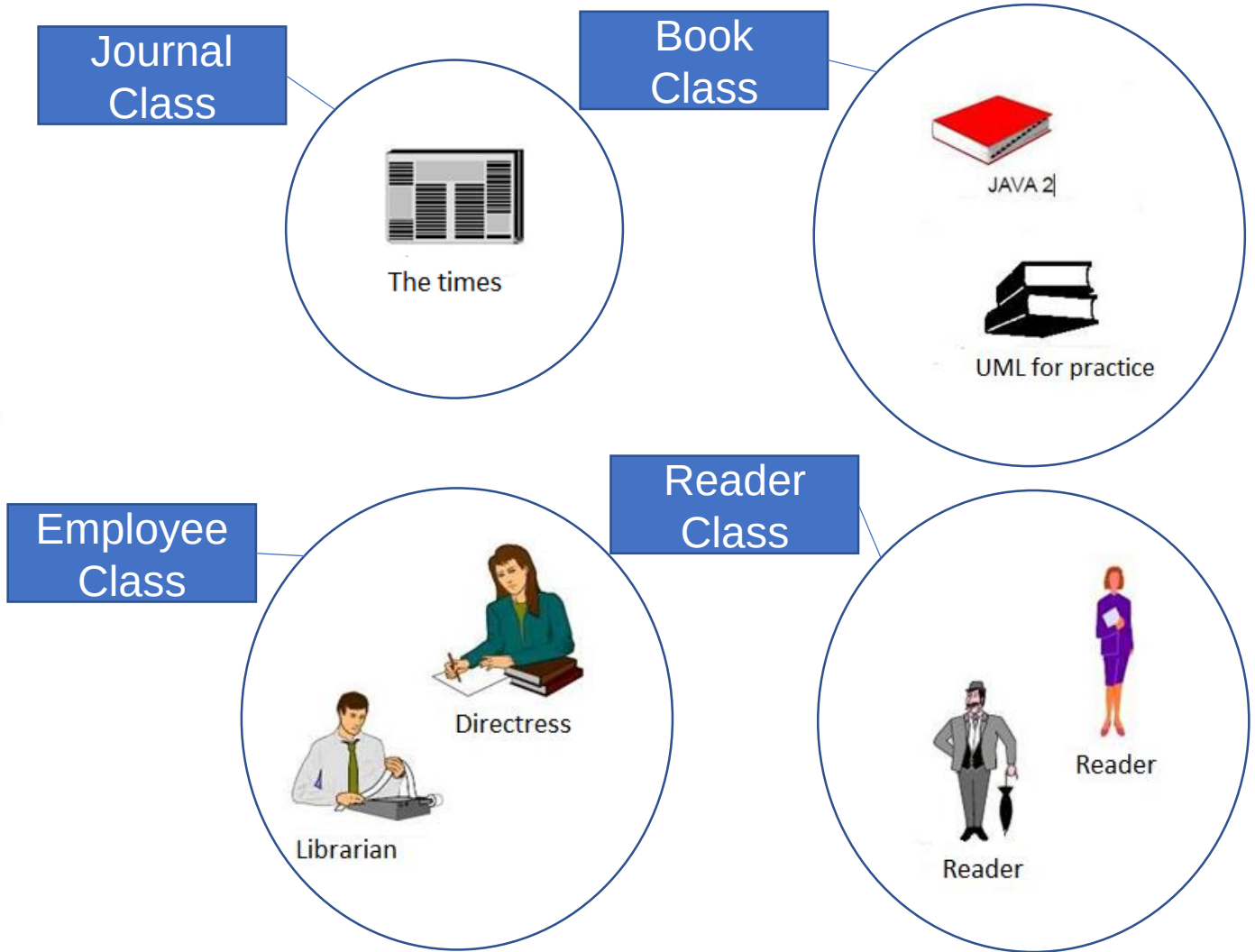
Object oriented programming is a methodology or paradigm to design a program using **Classes** and **Objects**.

Objects :
Real world entities that has their own properties and behaviors

Classes :
Blueprint form which an objects properties and behaviors are decided

Object-oriented programming OOP





Why OOP?

- OOP provides a clear modular structure for programs
- OOP makes it easy to maintain and modify existing code

Python : OOP

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Objects

- Python supports many different kinds of data:

10 5,9 “Python” {“welcome”:2, “python”:4}

- each one of these examples is an **object** which has:
  - a type
  - an internal **data representation**
  - a set of procedures for **interaction** with the object

Objects

An object is an **instance** of a type

- 10 is an instance of an int
- “Python "is an instance of a string
- {“welcome”:2, “python”:4} is an instance of a dictionary

Objects

- **EVERYTHING IN PYTHON IS AN OBJECT**
  - can **create new objects** of some type
  - can **manipulate objects**
  - can **destroy objects**

- python system will reclaim destroyed or inaccessible objects : called “garbage collection”

Objects

Objects are **a data abstraction** that captures…

- an **internal representation** through data attributes
- an **interface** for interacting with object
  - through methods
  - defines behaviors but hides implementation

Creating your own classes

To create a class :

- define the class name
- define class attributes

Creating your own classes

To use a class :

- create new **instances** of objects
- do operations on the instances

Creating your own classes

To create a class, use the keyword **class**:

```
class Book:
    #define  attributes and methods here
```

- class : class definition
- Book : name of the class

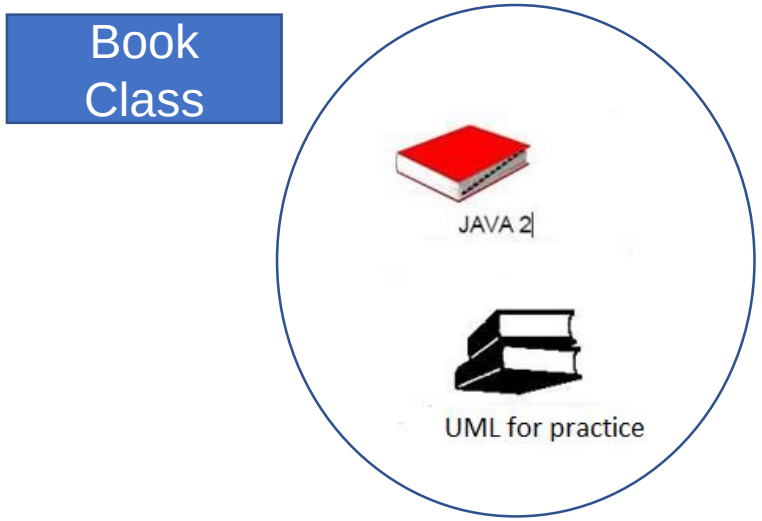
Creating your own classes

- Properties: data attributes
  - think of data as other objects that make up the class
- Behavior : methods
  - think of methods as functions that only work with this class
  - how to interact with the object

Creating your own classes

- Properties: data attributes
  - id
  - title
  - price

- Behavior : methods
  - init() : void
  - borrow(Reader r): void
  - buy () : Boolean
  - description() : String



__init__() function

- All classes have a function called __init__()
- It executed when the class is being initiated
- It is used to initialize values to object properties, or other operations that are necessary to do when the object is being created

__init__() function

Syntax

```
class ClassName :
   def __init__(self, parameterName1, parameterName2, . . .) :
      constructor body
```

The special name __init__ is used to define a constructor.

A constructor defines and initializes the instance variables.

```
class BankAccount :
   def __init__(self) :
      self._balance = 0.0
      . . .
```

There can be only one constructor per class. But a constructor can contain default arguments to provide alternate forms for creating objects.

```
class BankAccount :
   def __init__(self, initialBalance = 0.0) :
      self._balance = initialBalance
      . . .
```

Python for everyone, Cay S. Horstmann, Rance D. Necaise

__init__() function

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price
```

- __init__: special method to create instance (__ is double underscore)
- self : parameter to refer to an instance of the class
- id, title, price : data to initialize a book object (three attribute for each book object

__init__() function

```
B1 = Book(100,"Harry Potter", 500)
print(Book)

print(B1)
print(type(B1))

print(isinstance(B1,Book))

print("Title : ", B1.title)
```

```
<class '__main__.Book’>

<__main__.Book object at 0x000001D1D8F33948>
<class '__main__.Book’>

True

Title :  Harry Potter
```

__init__() function (Poll)

What will be the output of this code:

```
class A:
    def __init__(self, id):
        self.id = id
        id = 10

val = A(1)
print (val.id)
```

A- SyntaxError

B- 1

C- 10

__init__() function

What will be the output of this code:

```
class A:
    def __init__(self, id):
        self.id = id
        id = 10
        print(id)

val = A(1)
print (val.id)
```

10

1

Object Methods

- Objects can contain methods.
- Methods in objects are functions that belong to the object.

Object Methods

Syntax

```
class ClassName :
   . . .
   def methodName(self, parameterName1, parameterName2, . . .) :
      method body
   . . .
```

```
class CashRegister :
   . . .
   def addItem(self, price) :
      self._itemCount = self._itemCount + 1
      self._totalPrice = self._totalPrice + price
   . . .
```

Every method must include the special `self` parameter variable. It is automatically assigned a value when the method is called.

Instance variables are referenced using the `self` parameter.

Local variable

Python for everyone, Cay S. Horstmann, Rance D. Necaise

Object Methods

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price


    def description(self):
        print("id : ", self.id, “, title : ", self.title, “, price : ", self.price)

B1 = Book(100,"Harry Potter", 500)
B1.description()
```

id : 100 , title : Harry Potter , price : 500

The self Parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

- It does not have to be named self, you can call it differently

- It has to be the first parameter of any function in the class

The self parameter

```
class Book:
    def __init__(myObject, id, title, price):
        myObject.id = id
        myObject.title = title
        myObject.price = price


    def description(mine):
        print("id : ", mine.id, ", title : ", mine.title, ", price : ", mine.price)

B1 = Book(100,"Harry Potter", 500)
print(B1.title)
B1.description()
```

```
Harry Potter
id :  100 , title :  Harry Potter , price :  500
```

Modify Object Properties

Object properties can be modified:

```
B1 = Book(100,"Harry Poter", 500)
B1.description()

B1.price = 600
B1.description()
```

dot notation used to access attributes (data and methods)

Delete Object Properties

Object properties can be deleted

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price


    def description(self):
        print("id : ", self.id, ", title : ",
self.title, ", price : ", self.price)

B1 = Book(100,"Harry Potter", 500)
B1.description()

del B1.price
B1.description()
```

print("id : ", self.id, ", title : ", self.title, ", price : ", self.price)
AttributeError: 'Book' object has no attribute 'price'

Customizing attribute access

setattr() function sets the value of the specified attribute of the specified object.

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price :
{self.price}'.format(self=self)


B1 = Book(100,"Harry Potter", 500)
print(B1)
setattr(B1, "price", 600)
print(B1)
```

Customizing attribute access

- delattr(object, attr) function : remove an attribute
- getattr(object, attr) function : get the value of an attribute
- hasattr(object, attr) function : check if an attribute exist

dot notation used to access attributes (data and methods) though it is better to use getters and setters to access data attributes:

- good style
- easy to maintain code
- prevents bugs

Customizing attribute access

```
B1 = Book(100,"Harry Potter", 500)
print(B1)
print(getattr(B1, "title"))
delattr(B1,"price")
print(hasattr(B1,"price"))
```

```
id : 100, title : Harry Potter, price : 500
Harry Potter
False
```

Encapsulation

Encapsulation is seen as the bundling of data with the methods that operate on that data:

- Getter methods: for retrieving or accessing the values of attributes
- Setter: for changing the values of attributes

Encapsulation

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price :
{self.price}'.format(self=self)

    def set_price(self, price):
        self.price = price

    def get_price(self):
        return  self.price

B1 = Book(100,"Harry Potter", 500)
print(B1.get_price())
B1.set_price(1000)
print(B1.get_price()
```

```
500
1000
```

Pass statement

- Class definitions cannot be empty.

- To define a class with an empty content, use the pass statement to avoid errors

```
def buy(self):
    pass
```

Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances

```
class Animal:
    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes\
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return 'Type: {self.species}, Name : {self.name}, Age: {self.age}'.format(self=self)

cat = Animal('cat',2)
dog = Animal('dog',3)
print(cat)
print(dog)
```

```
Type: mammal, Name : cat, Age: 2
Type: mammal, Name : dog, Age: 3
```

Accessibility control of the attributes

- Private attributes : should only be used by the owner (inside of the class definition itself)
- Protected (restricted) Attributes: may be used, but at your own risk. Essentially, they should only be used under certain conditions.
- Public Attributes: can and should be freely used.

Accessibility control of the attributes

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside a class definition.
_name	Protected	Protected attributes should not be used outside the class definition, unless inside a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself.

Accessibility control of the attributes (Poll)

```
class A():

    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"


x = A()
print(x.pub)
print(x._prot)
print(x.__priv)
```

I am public

I am protected

print(x.__priv)
AttributeError: 'A' object has no attribute '__priv'

What will be the output of this code?

A- I am public I am protected I am private

B- I am public I am protected AttributeError

C- None of the above

Accessibility control of the attributes

```
class A():
    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
    def set_priv(self, value):
        self.__priv=value
    def get_priv(self):
        return self.__priv
    def set_prot(self, value):
        self._prot=value
    def get_prot(self):
        return self._prot
    def set_pub(self, value):
        self.pub=value
    def get_pub(self):
        return self.pub
x = A()
print(x.get_priv())
print(x.get_prot())
print(x.get_pub())
x.set_priv('new value for private')
x.set_prot('new value for protected')
x.set_pub('new value for public')
print(x.get_priv())
print(x.get_prot())
print(x.get_pub())
```

we should only be able to access private attributes via getters and setters.

```
I am private
I am protected
I am public
new value for private
new value for protected
new value for public
```

Accessibility control of the methods (Poll)

```
class A:
    def f1(self):
        print('f1')
    def __f2(self):
        print('f2')


d1 = A()
d1.f1()
d1.f2()
```

```
f1
AttributeError: 'A' object has no attribute 'f2'
```

What will be the output of this code?

A- f1 f2

B- f1 Syntax error

C- f1 AttributeError

Accessibility control of the methods

→ Private Access Modifier

```
class A:
    def f1(self):
        print('f1')
    def __f2(self):
        print('f2')
    def accessPrivateFunctionF2(self):
        self.__f2()


d1 = A()
d1.f1()
d1.accessPrivateFunctionF2()
```

Special method names

- A class can implement certain operations that are invoked by special syntax by defining methods with special names.

- This is Python’s approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators

Special method names

__**str**__(*self*)
Called by str(object) and the built-in functions format() and print() to compute the “informal” or nicely printable string representation of an object. The return value must be a string object.

```
class Book:
    def __init__(self, id, title, price):
        self.id = id
        self.title = title
        self.price = price

    def __str__(self):
        return 'id : {self.id}, title : {self.title}, price :
{self.price}'.format(self=self)


B1 = Book(100,"Harry Potter", 500)
print(B1)
```

id : 100, title : Harry Potter, price : 500

Special method names (Poll)

What will be the output of this code?

A- <__main__.A object at 0x000001C7452FD188>
<__main__.A object at 0x000001C7452FD188>

B- __rep__built in function is called
__rep__built in function is called

C- __rep__built in function is called
__str__built in function is called

```
class A:
    def __init__(self, id):
        self.id=id
    def __repr__(self):
        return '__rep__built in
function is called'


a = A(10)
print(repr(a))
print(a)
```

__rep__built in function is called
__rep__built in function is called

Special method names

Defined with double underscores before/after __X__

Expression	Method Name	Returns	Description
x + y	__add__(self, y)	object	Addition
x - y	__sub__(self, y)	object	Subtraction
x * y	__mul__(self, y)	object	Multiplication
x / y	__truediv__(self, y)	object	Real division
x // y	__floordiv__(self, y)	object	Floor division
x % y	__mod__(self, y)	object	Modulus
x ** y	__pow__(self, y)	object	Exponentiation
x == y	__eq__(self, y)	Boolean	Equal
x != y	__ne__(self, y)	Boolean	Not equal
x < y	__lt__(self, y)	Boolean	Less than
x <= y	__le__(self, y)	Boolean	Less than or equal

Link: https://docs.python.org/3/reference/datamodel.html#basic-customization

Special method names

```
class Day(object):
    def __init__(self, visits, contacts):
        self.visits = visits
        self.contacts = contacts

    def __add__(self, other):
        total_visits = self.visits + other.visits
        total_contacts = self.contacts +
other.contacts
        return Day(total_visits, total_contacts)

    def __str__(self):
        return 'Visits: %i, Contacts: %i' %
(self.visits, self.contacts)

day1 = Day(10, 1)
day2 = Day(20, 2)
day3 = day1 + day2
print(day3)
```

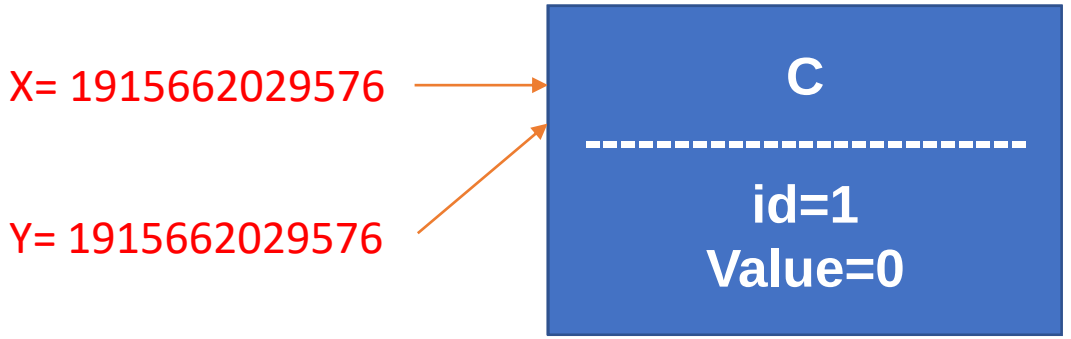
Visits: 30, Contacts: 3

http://www.marinamele.com/2014/04/modifying-add-method-of-python-class.html

Object references (Poll)

```
class C:
    def __init__(self, id, value=0):
        self.id = id

x = C(1)
y = x
print(id(x), id(y))
```



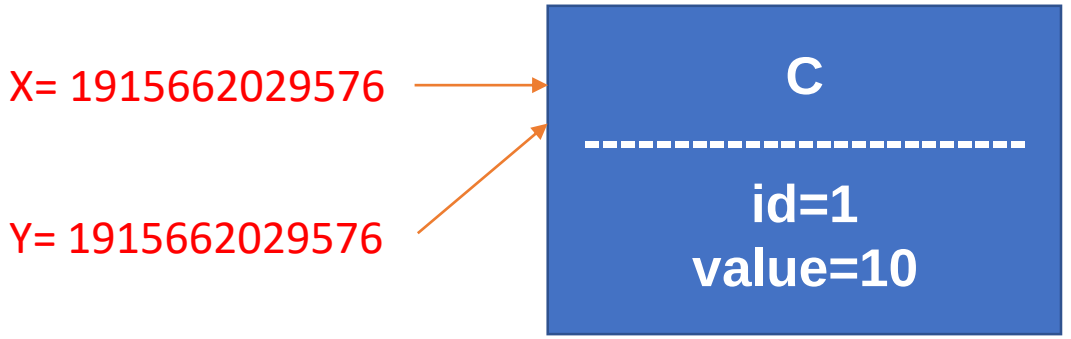
Will we have the same id for both x and y?

Object references

```
class C:
    def __init__(self, id, value=0):
        self.id = id

x = C(1)
y = x
print(x is y)
setattr(y,'value', 10)
print(x is y)
```

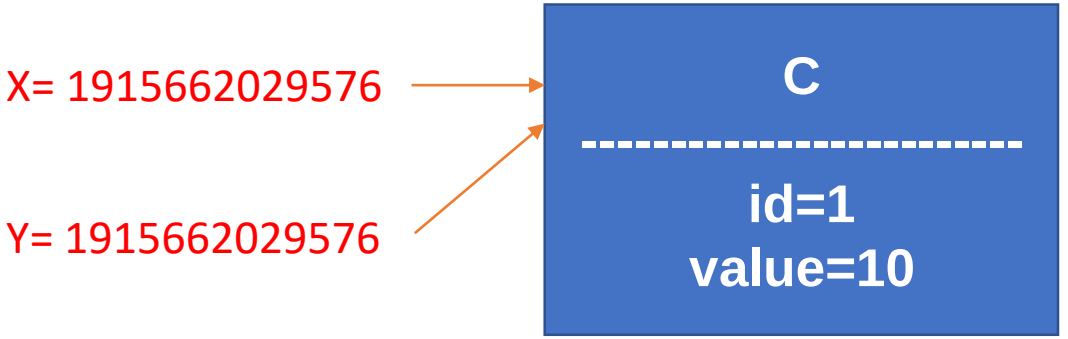
```
True
True
```



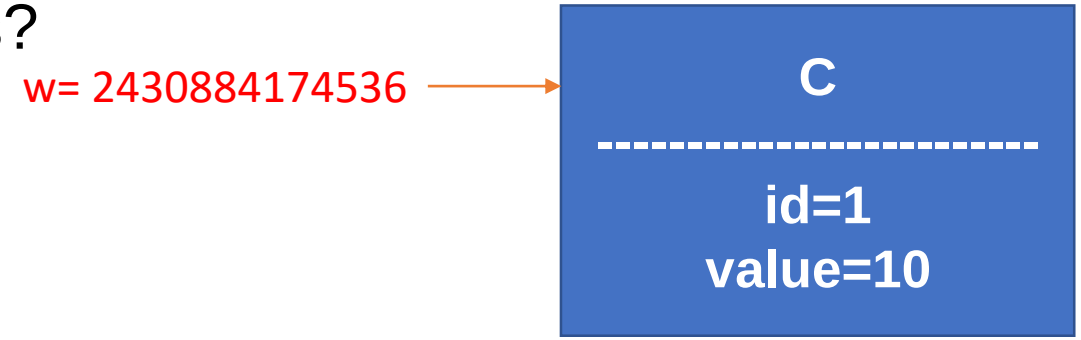
Object references (Poll)

```
class C:
    def __init__(self, id, value=0):
        self.id = id

x = C(1)
y = x
setattr(y,'value', 10)
print(x is y)
w = C(1,10)
print(x is w)
```



Is it correct to say that x and w are aliases?



True

False

OOPs vs Procedural programming

OOP

- Bottom up approach
- Divided into objects
- Has Access Modifiers
- Objects can move and communicate with each other through member function
- More secure
- Supports overloading

Procedural programming

- Top Down approach
- Divided into function
- Doesn’t have Access Modifiers
- Data can move freely from function to function in the system
- Less secure
- Do not support overloading

OOP advantages

- **bundle together objects** that share
  - common attributes
  - procedures that operate on those attributes

- use **abstraction** to make a distinction between how to implement an object vs how to use the object

- create your **own classes of objects** on top of Python's basic classes

- build **layers** of object abstractions that inherit behaviors from other classes of objects (Next lecture)