Python-01

Intro to OOP

Sergey M. Kabanov

National Research University Higher School of Economics

September 16, 2018

OOP

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

Key concepts

- Classes and objects
- Inheritance
- Polymorphysm
- Encapsulation

Class and object

Languages that support object-oriented programming typically use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts:

- **Classes** – the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contain the data members and member functions
- **Objects** – instances of classes

Built-in types (classes)

```
>>> i = int(1)
>>> type(int) # class int (built-in type int)
<class 'type'>
>>> type(i) # instance of class (type) int
<class 'int'>
>>> s = slice(2, -1, 1)
>>> type(slice) # class slice (built-in type slice)
<class 'type'>
>>> type(s) # instance of class (type) slice
<class 'slice'>
```

Built-in types (classes)

```
>>> i = int(1)
>>> type(int) # class int (built-in type int)
<class 'type'>
>>> type(i) # instance of class (type) int
<class 'int'>
>>> s = slice(2, -1, 1)
>>> type(slice) # class slice (built-in type slice)
<class 'type'>
>>> type(s) # instance of class (type) slice
<class 'slice'>
```

Built-in types (classes)

```
>>> s.start # attribute start
2
>>> s.stop # attribute stop
-1
>>> s.step # attribute step
1
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[s] == l[2:-1:1]
True
>>> hasattr(s, 'start')
True
```

Built-in types (classes)

```
>>> s.start # attribute start
2
>>> s.stop # attribute stop
-1
>>> s.step # attribute step
1
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[s] == l[2:-1:1]
True
>>> hasattr(s, 'start')
True
```

Built-in types (classes)

```
>>> s = "abcd"
>>> s.split() # string method split
['abcd']
>>> s.upper() # string method upper
'ABCD'
>>> s.islower() # string method islower
True
>>> s[0].isupper() # what is s[0] method?
False
```

User-defined type (class)

```
>>> class Test:
...     pass
...
>>> type(Test)
<class 'type'>
>>> t = Test()
>>> type(t)
<class '__console__.Test'>
>>> isinstance(t, Test)
True
```

Dynamic attributes

```
>>> hasattr(t, "test")
False
>>> t.test = 11
>>> hasattr(t, "test")
True
>>> t.test
11
>>> hasattr(t, "name")
False
>>> t.name = "Test"
>>> hasattr(t, "name")
True
```

Initialization

```
>>> class Test:
...     def __init__(self, attr):
...         self.attr = attr
...
>>> t = Test("test")
>>> hasattr(t, "attr")
True
>>> t.attr
'test'
```

Methods

```
>>> class Test:
...     def __init__(self, attr):
...         self.attr = attr
...     def print_attr(self):
...         print(self.attr)
...
>>> t = Test("some attr")
>>> t.print_attr()
some attr
>>> Test.print_attr(t)
some attr
```

Class attributes

```
>>> class Test:
...     class_attr = []
...     def __init__(self, attr):
...         self.attr = attr
...
>>> t = Test("instance attr")
>>> t.attr
'instance attr'
>>> t.class_attr
[]
>>> Test.class_attr
[]
```

Class attributes

```
>>> other = Test("attr")
>>> other.attr
'attr'
>>> other.class_attr
[]
>>> other.class_attr.append("class_attr")
>>> t.class_attr
['class_attr']
>>> Test.class_attr
['class_attr']
```

Class Customer

```
>>> class Customer:
...     def __init__(self, name):
...         self.name = name
...         self.balance = 0
...     def withdraw(self, amount):
...         if amount < self.balance:
...             self.balance -= amount
...             return self.balance
...     def deposit(self, amount):
...         self.balance += amount
...         return self.balance
...
```

Class Customer

```
>>> john = Customer('John')
>>> john.deposit(1000)
1000
>>> john.withdraw(255)
745
>>> john.balance
745
>>> john.name
'John'
```

```
>>> john.deposit(200)
945
>>> john.withdraw(1200)
>>> john.deposit(3000)
3945
>>> john.withdraw(1200)
2745
>>> john.withdraw(1300)
1445
```

Inheritance

In object-oriented programming, **inheritance** is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object. Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a directed graph.

Class CreditCustomer

```
>>> class CreditCustomer(Customer):
...     def __init__(self, limit):
...         self.limit = limit
...     def withdraw(self, amount):
...         if amount < self.balance + self.limit:
...             self.balance -= amount
...             return self.balance
...
```

Class CreditCustomer

```
>>> cc = CreditCustomer(limit=1000)
>>> cc.withdraw(900)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in withdraw
AttributeError: 'CreditCustomer' object has no attribute 'balance'
>>> cc.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'CreditCustomer' object has no attribute 'name'
```

Class CreditCustomer

```
>>> class CreditCustomer(Customer):
...     def __init__(self, name, limit):
...         Customer.__init__(self, name) # bad practice
...         self.limit = limit
...     def withdraw(self, amount):
...         if amount < self.balance + self.limit:
...             self.balance -= amount
...             return self.balance
...
```

Class CreditCustomer

```
>>> cc = CreditCustomer(name='Ivan', limit=1000)
>>> cc.withdraw(900)
-900
>>> cc.name
'Ivan'
>>> cc.deposit(300)
-600
>>> cc.withdraw(400)
>>> cc.withdraw(350)
-950
```

Encapsulation

In object oriented programming languages, **encapsulation** is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting direct access to some of the object's components
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data

Encapsulation

```
>>> john.deposit(500)
1945
>>> john.withdraw(945)
1000
>>> john.balance = -1000 # somewhere in the program
>>> john.withdraw(500)
>>> john.balance
-1000
```

Private in C++

```
class Customer {
public:
        Customer(const std::string& name):
                name(name),
                balance(0.0) { }

        std::string name;
private:
        double balance;
};
```

Private Python convention

```
>>> class Customer:
...     def __init__(self, name):
...         self.name = name
...         self._balance = 0
...     def withdraw(self, amount):
...         if amount < self._balance:
...             self._balance -= amount
...             return self._balance
...     def deposit(self, amount):
...         self._balance += amount
...         return self._balance
...
```

Private Python convention

```
>>> john = Customer('John')
>>> john.deposit(1000)
1000
>>> john.withdraw(200)
800
>>> john._balance -= 1000
>>> john.withdraw(200)
>>> john._balance
-200
```

Real Private in Python?

```
>>> class Customer:
...     def __init__(self, name):
...         self._name = name
...         self.__balance = 0
...
>>> john = Customer('John')
>>> john._name
'John'
>>> john.__balance
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Customer' object has no attribute '__balance'
>>> john._Customer__balance
0
```

Getter-Setter

```
>>> class Customer:
...     def __init__(self, name):
...         self.__name = name # better to use _name
...         self.__balance = 0 # better to use _balance
...     def get_name(self):
...         return self.__name
...     def set_name(self, name):
...         self.__name = name
...
```

Getter-Setter

```
>>> john = Customer("John")
>>> john.get_name()
'John'
>>> john.set_name("Ivan")
>>> john.get_name()
'Ivan'
>>> john._Customer__name = "Olga" # well, use _name
>>> john.get_name()
'Olga'
```

Private convention

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Magic attributes

```
>>> help(dir) # buit-in function
Help on built-in function dir in module builtins:

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
    Else, return an alphabetized list of names comprising (some of) the attribute
    of the given object, and of attributes reachable from it.
    If the object supplies a method named __dir__, it will be used; otherwise
    the default dir() logic is used and returns:
      for a module object: the module's attributes.
      for a class object:  its attributes, and recursively the attributes
        of its bases.
      for any other object: its attributes, its class's attributes, and
        recursively the attributes of its class's base classes.
```

Magic attributes

```
>>> for i, attr in enumerate(dir(john)):
...     print(attr, end=', ') if i % 4 else print(attr)
...
_Customer__balance
_Customer__name, __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__, get_name
set_name,
```

Magic attributes

```
>>> john.__module__ # module where defined
'__console__'
>>> john.__class__ # type(john)
<class '__console__.Customer'>
>>> john.__doc__ # docstring
>>> john.__dict__ # not magic attributes
{'_Customer__name': 'Olga', '_Customer__balance': 0}
```

Magic attributes

```
>>> john
<__console__.Customer object at 0x7fb1a43de588>
>>> print(john)
<__console__.Customer object at 0x7fb1a43de588>
>>> [1, 2, 3]
[1, 2, 3]
>>> print([1, 2, 3])
[1, 2, 3]
>>> {1, 2, 3}
{1, 2, 3}
>>> print({1, 2, 3})
{1, 2, 3}
```

Magic attributes

```
>>> john
<__console__.Customer object at 0x7fb1a43de588>
>>> print(john)
<__console__.Customer object at 0x7fb1a43de588>
>>> [1, 2, 3]
[1, 2, 3]
>>> print([1, 2, 3])
[1, 2, 3]
>>> {1, 2, 3}
{1, 2, 3}
>>> print({1, 2, 3})
{1, 2, 3}
```

Magic __str__()

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return f'Hi, my name is {self.name}!'
...
>>> print(Person('Galina'))
Hi, my name is Galina!
>>> str(Person('Galina'))
'Hi, my name is Galina!'
>>> Person('Galina')
<__console__.Person object at 0x7fb1a4438e48>
```

Magic __repr__()

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return f'Hi, my name is {self.name}!'
...     __repr__ = __str__
...
>>> print(Person('Galina'))
Hi, my name is Galina!
>>> Person('Galina')
Hi, my name is Galina!
```

Magic __eq__()

```
>>> class Person:
...     def __init__(self, name, bday):
...         self.name = name
...         self.bday = bday
...     def __eq__(self, other):
...         print('call __eq__') # track print
...         name_bool = self.name == other.name
...         bday_bool = self.bday == other.bday
...         return name_bool and bday_bool
...
```

Magic __eq__()

```
>>> g1 = Person('Galina', '01/01/1999')
>>> a1 = Person('Andrey', '01/01/1999')
>>> a2 = Person('Andrey', '01/01/2000')
>>> a1 == g1
call __eq__
False
>>> a2 == a1
call __eq__
False
>>> a2 != a1
call __eq__
True
```

Magic __ne__()

```
>>> a2 != a1
call __eq__
True
```

By default, __ne__() delegates to __eq__() and inverts the result unless it is NotImplemented. There are no other implied relationships among the comparison operators, for example, the truth of (x<y or x==y) does not imply x<=y. To automatically generate ordering operations from a single root operation, see functools.total_ordering().

More magic comparisons

- object.__lt__(self, other), <
- object.__le__(self, other), <=
- object.__eq__(self, other), ==
- object.__ne__(self, other), !=
- object.__gt__(self, other), >
- object.__ge__(self, other), >=

```
>>> a1.__eq__(a1)
call __eq__
True
>>> a1 == a1
call __eq__
True
```

If and magic __bool__()

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...     def __bool__(self):
...         print('call __bool__')
...         return self.name == 'Galina'
...
>>> print("Yes") if Person("Galina") else print("No")
call __bool__
Yes
>>> bool(Person("Andrey")) # built-in bool() function
call __bool__
False
```

Magic arithemic

- object.__add__(self, other), $+$
- object.__sub__(self, other), $-$
- object.__mul__(self, other), $*$
- object.__matmul__(self, other), @
- object.__truediv__(self, other), $/$
- object.__floordiv__(self, other), $//$
- object.__mod__(self, other), %
- object.__divmod__(self, other), divmod()
- object.__pow__(self, other[, modulo]), pow(), **
- object.__lshift__(self, other), $<<$
- object.__rshift__(self, other), $>>$
- object.__and__(self, other), &

Magic arithemic

- object.__neg__(self), $-$
- object.__pos__(self), $+$
- object.__abs__(self), abs
- object.__invert__(self, other),
- object.__complex__(self, other), complex()
- object.__int__(self, other), int()
- object.__float__(self, other), float()

More magic operators (see: Python Data Model)

Magic slicing

```
>>> class List(list):
...     def __getitem__(self, key):
...         print(key)
...         print("call __getitem__")
...         return list.__getitem__(self, key)
...     def __setitem__(self, key, value):
...         print("call __setitem__")
...         return list.__setitem__(self, key, value)
...     def __delitem__(self, key):
...         print("call __delitem__")
...         return list.__delitem__(self, key)
...
```

Magic slicing

```
>>> l = List(range(10))
>>> l[0]
0
call __getitem__
0
>>> l[::-1] # intepreter converts [::-1] to slice object
slice(None, None, -1)
call __getitem__
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> l[1:4:2] # [1:4:2] == scile(1, 4, 2)
slice(1, 4, 2)
call __getitem__
[1, 3]
```

Magic slicing

```
>>> l[::-3]
slice(None, None, -3)
call __getitem__
[9, 6, 3, 0]
>>> l[0] = 10
call __setitem__
>>> l[3] = 143
call __setitem__
>>> del l[-1]
call __delitem__
>>> l
[10, 1, 2, 143, 4, 5, 6, 7, 8]
```

Magic slicing

```
>>> class DefaultDict(dict):
...     def __getitem__(self, key):
...         if key in self:
...             return dict.__getitem__(self, key)
...         return 0
...
>>>     # def __getitem__(self, key):
>>>     #     try:
>>>     #         return dict.__getitem__(self, key)
>>>     #     except KeyError:
>>>     #         return 0
```

Magic slicing

```
>>> d = {'a': 1, 'b': 2}
>>> d['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> d = DefaultDict(d)
>>> d['c']
0
>>> d
{'a': 1, 'b': 2}
```

Magic slicing

```
>>> class DefaultDict(dict):
...     def __init__(self, *args, value=0, **kwargs):
...         dict.__init__(self, *args, **kwargs)
...         self.value = value
...     def __missing__(self, key):
...         return self.value
...
```

Magic slicing

```
>>> d = {'a': 1, 'b': 2}
>>> d = DefaultDict(d, value='default')
>>> d['c']
'default'
>>> d
{'a': 1, 'b': 2}
>>> d['aaaaa']
'default'
```

Functions

```
>>> def add(a, b=1): return a + b
...
>>> type(add)
<class 'function'>
>>> for i, a in enumerate(set(dir(add)) - set(dir(int))):
...     print(a, end=' ') if i % 4 else print(a)
...
__closure__
__call__ __kwdefaults__ __name__ __code__
__dict__ __module__ __globals__ __get__
__annotations__ __defaults__ __qualname__
```

Functions

```
>>> add.__module__
'__console__'
>>> add.__name__
'add'
>>> add.__defaults__
(1,)
>>> add.attr = 'attr'
>>> add.__dict__
{'attr': 'attr'}
>>> add.attr
'attr'
```

```
>>> add(1, 2)
3
>>> add.__call__(1, 2)
3
>>> add.__call__(99)
100
>>> mul = lambda a, b: a * b
>>> add.__call__ = mul
>>> add.__call__(11, 11)
121
>>> add(11, 11) # ???
22
```

Callable objects

```
>>> class Parabola:
...     def __init__(self, a, b, c):
...         self.a, self.b, self.c = a, b, c
...     def __call__(self, x):
...         return self.a * x ** 2 + self.b * x + self.c
...     def __contains__(self, point):
...         return point.y == self(point.x)
...
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...
```

Callable objects

```
>>> parabola = Parabola(2, 5, 10)
>>> parabola(3)
43
>>> parabola.__call__(-1.33)
6.8878
>>> Point(3, 43) in parabola # uses __contains__
True
>>> Point(3, 33) in parabola
False
```

Multiple inheritance

```
>>> class Animal:
...     def __init__(self, name):
...         self.name = name
...     def sound(self):
...         print("Sound!")
...
```

Multiple inheritance

```
>>> class Cat(Animal):
...     def sound(self):
...         print('Meow-meow!')
...
>>> class Dog(Animal):
...     def sound(self):
...         print("Ruh-roh!")
...
```

Multiple inheritance

```
>>> class CatDog(Cat, Dog): # diamond inheritance
...     pass
...
>>> CatDog(name='Mark').sound() # which sound()?
Meow-meow!
>>> for c in CatDog.__mro__: # MRO C3 linearization
...     print(c)
...
<class '__console__.CatDog'>
<class '__console__.Cat'>
<class '__console__.Dog'>
<class '__console__.Animal'>
<class 'object'>
```

C3 linearization

In computing, the **C3 superclass linearization** is an algorithm used primarily to obtain the order in which methods should be inherited (the "linearization") in the presence of multiple inheritance, and is often termed **Method Resolution Order (MRO)**.

The name C3 refers to the three important properties of the resulting linearization:

- a consistent extended precedence graph,
- preservation of local precedence order, and
- fitting the monotonicity criterion.

C3 linearization

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class C(B): pass
...
>>> class X(A): pass
...
>>> class Y(B): pass
...
>>> class D(C, X, Y): pass
...
```

```
>>> class Final(D, X, Y):
...     pass
...
```

C3 linearization

```
>>> for c in Final.__mro__: print(c)
...
<class '__console__.Final'>
<class '__console__.D'>
<class '__console__.C'>
<class '__console__.X'>
<class '__console__.Y'>
<class '__console__.B'>
<class '__console__.A'>
<class 'object'>
```

super() and getattr()

super() is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by getattr() except that the type itself is skipped.
The __mro__ attribute of the type lists the method resolution search order used by both getattr() and super(). The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

super()

```
>>> class CreditCustomer(Customer):
...     def __init__(self, name, limit):
...         super().__init__(name=name) # good practice
...         self.limit = limit
...     def withdraw(self, amount):
...         if amount < self.balance + self.limit:
...             self.balance -= amount
...             return self.balance
...
```

super()

```
>>> class List(list):
...     def __getitem__(self, key):
...         return super().__getitem__(key)
...     def __setitem__(self, key, value):
...         return super().__setitem__(key, value)
...     def __delitem__(self, key):
...         return super(List, self).__delitem__(key)
...         # for compability super(Type, obj)
...
```

getattr()

```
>>> class A:
...     a = 'a'
...     def __init__(self, a1):
...         self.a1 = a1
...         self.a2 = 'a2'
...
>>> class B(A):
...     b = 'b'
...     def __init__(self, a1, b1):
...         super().__init__(a1)
...         self.b1 = b1
...
```

getattr()

```
>>> b = B('a1', 'b1')
>>> b.a1
'a1'
>>> b.b1
'b1'
>>> b.b
'b'
>>> b.a
'a'
>>> getattr(b, 'a')
'a'
>>> getattr(b, 'a2')
'a2'
```

getattr()

```
>>> b.__dict__
{'a1': 'a1', 'a2': 'a2', 'b1': 'b1'}
>>> B.__dict__['b']
'b'
>>> A.__dict__['a']
'a'
```

getattr()

```
>>> class B(A):
...     b = 'b'
...     def __init__(self, a1, b1):
...         super().__init__(a1)
...         self.b = "bbb"
...         self.b1 = b1
...
```

getattr()

```
>>> b = B('a1', 'b1')
>>> b.a1
'a1'
>>> b.b
'bbb'
>>> b.__dict__
{'a1': 'a1', 'a2': 'a2', 'b': 'bbb', 'b1': 'b1'}
```

Interpreter looks up an attribute in self.__dict__, if doesn't find then searches in __mro__ list classes, then raises AttributeError.

Polymorphysm

In programming languages and type theory, **polymorphism** is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

The most commonly recognised major classes of polymorphism are:

- **Ad hoc polymorphism**: defines a common interface for an arbitrary set of individually specified types.
- **Parametric polymorphism**: when one or more types are not specified by name but by abstract symbols that can represent any type.
- **Subtyping** (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass.

Learning resourses

- Video lectures
  Python OOP, RU
  Data model, testing, RU
- Testing
  unittest guide, RU
  unittest doc, ENG
- Python OOP
  Classes doc, ENG
  Data model doc, ENG
- MRO C3
  MRO C3, RU
  MRO C3, ENG