# 面向对象

# 面向过程(Procedure-oriented)

可以使用在函数中调用其他函数的方式设计我们的程序。这叫做面向过程的编程方式。它的特点是把程序分成多个步骤，用函数把这些步骤一步一步实现，使用的时候串行依次调用

# 面向对象编程

## (Object Oriented Programming - OOP)

面向对象编程是一种程序设计思想，OOP把对象作为程序的基本单元，一个对象可能包含了数据、属性和操作数据的方法。

In Python everything is an object

# 对象和类

1. 类。定义了一件事物的抽象特点，如共有的属性和方法
2. 对象。是类的实例

# 使用类有什么优势呢?

1. 继承（inheritance）。子类可以继承父类通用类型的属性和方法。也就是在父类或者说基类里面实现一次就能被子类重用
2. 封装(Encapsulation)。对外部隐藏有关对象工作原理的细节
3. 多态（polymorphism）。也就是同一个方法，不同的行为，指由继承而产生的相关但不同的类，其对象对同一消息会做出不同的响应

# 属性和方法

1. 属性(attribute)。 对象可以使用属于它的普通变量来存储数据，这种从属于对象或类的变量就是变量，它描述了对象的特征。

2. 方法(method)。也就是类中的函数，能通过它对对象做操作。

# 条目基类

```
class Subject:
    kind = None

    def __init__(self, id, category_id, title):
        self.id = id
        self.category_id = category_id
        self.title = title

    def show_title(self):
        return self.title

    def update_title(self, title):
        self.title = title
```

# 经典类和新式类的的区别

1. 继承搜索顺序。新式类的MRO(基类搜索顺序)算法采用C3广度优先算法，经典类采用深度优先。多重继承中搜索结果可能不同
2. 类和类型合并。新式类统一了类（class）和类型（type）
3. 新的高级工具。新式类有更多的高级工具，如slot、特性、描述符等

# 使用类

```
# 实例化，subject这个变量被赋值为一个对象
In : subject = Subject(1, 1001, '条目1')

# 获得对象属性，也就是对象的特征
In : subject.kind, subject.id, subject.category_id, subject.title
Out: (None, 1, 1001, '条目1')

# 调用对象方法
In : subject.show_title()
Out: '条目1'

# 方法内会更新对象属性
In : subject.update_title('新条目')

In : subject.show_title(), subject.title
Out: ('新条目', '新条目')
```

```
# 等于 subject.show_title()
In : Subject.show_title(subject)
Out: '条目1'

# 等于 subject.update_title('新条目')
In : Subject.update_title(subject, '新条目')
```

注意subject的大小写：首字母大写的是类，
全小写的是类的实例，也就是一个对象

# 创建不同的对象

```
In : subject2 = Subject(2, 1002, '条目2')

In : subject2.id, subject2.category_id, subject2.title, subject2.show_title()
Out: (2, 1002, '条目2', '条目2')

In : subject2.kind is None
Out: True
```

# 继承

```
class Movie(Subject):
    kind = 'movie'
```

```
In : movie = Movie(3, 1002, '电影1')

In : movie.id, movie.category_id, movie.title, movie.kind
Out: (3, 1002, '电影1', 'movie')

In : movie.show_title()
Out: '电影1'
```

```
class Movie(Subject):
    kind = 'movie'

    def __init__(self, id, category_id, title, directors=[]):
        super().__init__(id, category_id, title)
        self.directors = directors

    def show_directors(self):
        return self.directors

    def show_title(self):
        return f'Movie: {self.title}'
```

# 覆盖（override）

如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖，也称为方法的重写。在子类定义父类同名方法之后，父类方法就被覆盖了。

```
super().__init__(id, category_id, title) # Python 3

super(Movie, self).__init__(id, category_id, title) # Python 2
```

# 方法解析顺序
# (Method Resolution Order -- MRO)

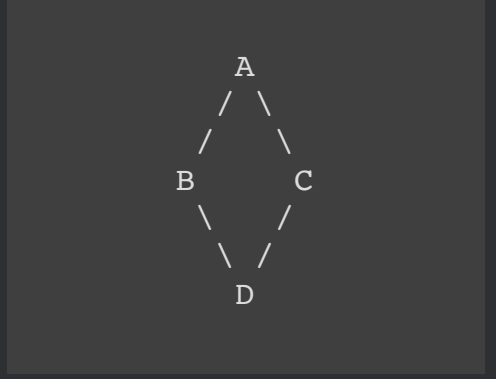
```
class A:
    def run(self):
        print('A.run')


class B(A):
    pass


class C(A):
    def run(self):
        print('C.run')


class D(B, C):
    pass
```

# 关系图(菱形继承)




## 经典类顺序👇

```
In : import inspect

In : inspect.getmro(D)
Out:
(<class classic_mro.D at 0x1080b67a0>,
 <class classic_mro.B at 0x1080b6668>,
 <class classic_mro.A at 0x1080b6738>,
 <class classic_mro.C at 0x1080b66d0>)

In : d = D()

In : d.run()
A.run
```

## 新式类顺序👇

```
In : import inspect

In : inspect.getmro(D)
Out: (classic_mro.D, classic_mro.B,
      classic_mro.C, classic_mro.A, obje

In : d = D()

In : d.run()
C.run
```

# property

```
class Movie(Subject):
    kind = 'movie'

    def __init__(self, id, category_id, title, directors=[]):
        super().__init__(id, category_id, title)
        self._directors = directors

    @property
    def directors(self):
        return self._directors
```

```
In : from movie_property import Movie

In : movie = Movie(4, 1002, '电影2', ['导演1'])

In : movie.directors
Out: ['导演1']

In : movie.directors = ['导演2']
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-3dc277f50810> in <module>()
----> 1 movie.directors = ['导演2']

AttributeError: can't set attribute

In : del movie.directors
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-fb6b6720629d> in <module>()
----> 1 del movie.directors

AttributeError: can't delete attribute
```

```python
class Movie(Subject):
    kind = 'movie'

    def __init__(self, id, category_id, title, directors=[]):
        super().__init__(id, category_id, title)
        self._directors = directors

    @property
    def directors(self):
        return self._directors

    @directors.setter
    def directors(self, value):
        if not isinstance(value, list):
            raise ValueError('invalid type')
        self._directors = value

    @directors.deleter
    def directors(self):
        print('del')
```

```
In : movie = Movie(4, 1002, '电影2', ['导演1'])

In : movie.directors = '导演2'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-7b52e2eb7690> in <module>()
----> 1 movie.directors = '导演2'

/Users/dongweiming/sansa/introduction-python/2.Python知识/17/movie_property.py i
     27     def directors(self, value):
     28         if not isinstance(value, list):
---> 29             raise ValueError('invalid type')
     30         self._directors = value
     31

ValueError: invalid type

In : movie.directors = ['导演2']

In : movie.directors
Out: ['导演2']

In : del movie.directors
del
```

```
class Movie(Subject):
    kind = 'movie'

    def __init__(self, id, category_id, title, directors=[]):
        super().__init__(id, category_id, title)
        self._directors = directors

    def get_directors(self):
        return self._directors

    def set_directors(self, value):
        if not isinstance(value, list):
            raise ValueError('invalid type')
        self._directors = value

    def del_directors(self):
        print('del')

    directors = property(get_directors, set_directors, del_directors)
```

# 静态方法/类方法

```
class A(object):
    count = 0

    def incr_count(self):
        self.count += 1

    @classmethod
    def incr_count2(cls):
        cls.count += 1

    @staticmethod
    def incr_count3():
        A.count += 1

    @staticmethod
    def avg(*items):
        return sum(items) / len(items)
```

```
In : a = A()

In : a.count, A.count
Out: (0, 0)

In : a.incr_count()

In : a.count, A.count
Out: (1, 0)

In : a = A()

In : a.count, A.count
Out: (0, 0)

In : a.incr_count2()

In : a.count, A.count
Out: (1, 1)
```

```
In : A.count = 0  # 重置

In : a = A()

In : A.incr_count()  # 对象方法不能直接用 类.方法 的方式调用
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-a30ae2a86247> in <module>()
----> 1 A.incr_count()

TypeError: incr_count() missing 1 required positional argument: 'self'

In : A.incr_count(a)  # 需要绑定对象到self上

In : a.count, A.count
Out: (1, 0)  # 对象方法依然不影响类变量的值

In : A.incr_count2()

In : a.count, A.count
Out: (1, 1)

In : A.incr_count2()

In : a.count, A.count
Out: (1, 2)  # 调用2次类方法，只影响了类变量，没有影响到对象变量
```

```
In : A.count = 0

In : a = A()

In : a.incr_count3()

In : a.count, A.count
Out: (1, 1)
```

# 静态方法/类方法总结

静态方法和类方法都访问不到对象变量，因为没有self，静态方法也访问不到cls，只能把类名写进去才能访问，incr_count3方法事实上这样用已经违背了静态方法不能访问类本身的原则，要访问当前类就应该用类方法，avg才是一个正确静态方法用法，方法内的逻辑和类A完全无关。

# 私有变量

```
class Employee:
    _kind = 'employee'
    def __init__(self, name):
        self.__name = name
```

```
In : e = Employee('em1')

In : e._kind
Out: 'employee'

In : e.__name
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent
<ipython-input-47-933d96f9c2ce> in <module>()
----> 1 e.__name

AttributeError: 'Employee' object has no attribute '__name'

In : e._Employee__name
Out: 'em1'
```

# 常用"魔法"方法

# 构造方法

```
class ExampleClass:
    def __new__(cls, *args, **kwargs):
        print('Creating new instance...')
        instance = super().__new__(cls)
        instance.PAYLOAD = (args, kwargs)
        return instance

    def __init__(self, payload):
        print('Initialising instance...')
        self.payload = payload
```

```
In : ec = ExampleClass({'a': 1})
Creating new instance...
Initialising instance...

In : ec.PAYLOAD
Out: (({'a': 1},), {})

In : ec.payload
Out: {'a': 1}
```

# 控制属性访问

1. __getattr__ 在属性被访问而对象没有这样的属性时自动调用
2. __setattr__ 试图给属性赋值时自动调用
3. __delattr__ 试图删除属性时自动调用
4. __getattribute__ 在属性被访问时自动调用(只适用于新式类)。 它和__getattr__的区别是无论属性是否存在，都要被调用

```python
class User:
    ...

class Proxy:
    title = '代理'
    _data = User()

    def show_title(self):
        return self.title

    def __getattr__(self, name):
        print('use __getattr__')
        return getattr(self._data, name)

    def __setattr__(self, name, value):
        print('use __setattr__')
        return object.__setattr__(self._data, name, value)

    def __delattr__(self, name):
        print('use __delattr__')
        return object.__delattr__(self._data, name)

    def __getattribute__(self, name):
        if name in ('_data', 'title', 'show_title'):
            return object.__getattribute__(self, name)
        print(f'use __getattribute__: {name}')
        if name.startswith('b'):
            raise AttributeError
        return object.__getattribute__(self._data, name)
```

```
In : p = Proxy()
use __getattribute__: __class__
use __getattribute__: __class__

In : p.title
Out: '代理'

In : p.show_title()
Out: '代理'

In : p.a = 1
use __setattr__

In : p.a
use __getattribute__: a
Out: 1

In : p.b = 2
use __setattr__

In : p.b
use __getattribute__: b
use __getattr__
Out: 2

In : p._data.b
Out: 2
```

```
In : del p.b
use __delattr__

In : p.b
use __getattribute__: b
use __getattr__
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-129-893b85f17d79> in <module>()
----> 1 p.b

/Users/dongweiming/sansa/introduction-python/2.Python知识/17/proxy.py in
    __getattr__(self, name)
     12     def __getattr__(self, name):
     13         print('use __getattr__')
---> 14         return getattr(self._data, name)
     15
     16     def __setattr__(self, name, value):

AttributeError: 'User' object has no attribute 'b'
```

# 类的表示

```
class MyClass:
    def __init__(self, id, name):
        self.id = id
        self.name = name
```

```
In : cls = MyClass(1, 'class1')

In : cls
Out: <print_cls.MyClass at 0x110d9cda0>
```

```
class MyClass:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __repr__(self):
        return f'{self.__class__.__name__} (id={self.id}, name={self.name})'
```

```
In : cls = MyClass(1, 'class1')

In : cls  # __repr__()
Out: MyClass (id=1, name=class1)

In : print(cls) # __str__()
MyClass (id=1, name=class1)

In : repr(cls) # __repr__()
Out: 'MyClass (id=1, name=class1)'

In : str(cls) # __str__()
Out: 'MyClass (id=1, name=class1)'
```

```
class MyClass:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __repr__(self):
        return f'{self.__class__.__name__} (id={self.id}, name={self.name})'

    def __str__(self):
        return f'{self.__class__.__name__} (id={self.id})'
```

```
In : cls = MyClass(1, 'class1')

In : cls
Out: MyClass (id=1, name=class1)

In : print(cls)
MyClass (id=1)

In : repr(cls)
Out: 'MyClass (id=1, name=class1)'

In : str(cls)
Out: 'MyClass (id=1)'
```

# 容器方法

1. __getitem__ 得到给定键(key)的值
2. __setitem__ 设置给定键(key)的值
3. __delitem__ 删除给定键(key)的值
4. __len__ 获得项的数目

# 例子(字典痛点):

```
In : d = {'a': 1}

In : d['a']
Out: 1

In : d.get('a', 0)
Out: 1

In : d.get('b', 0)
Out: 0

In : d.a
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-769f163885> in <module>()
----> 1 d.a

AttributeError: 'dict' object has no attribute 'a'
```

```python
class AttrDict:
    def __init__(self, **kwargs):
        self.__dict__.update(**kwargs)

    def __getitem__(self, key):
        return self.__getattribute__(key)

    def __setitem__(self, key, val):
        self.__setattr__(key, val)

    def __delitem__(self, key):
        self.__delattr__(key)

    def __len__(self):
        return len(self.__dict__)
```

```
In : d = AttrDict(a=1, b=2)

In : d.a
Out: 1

In : d['b']
Out: 2

In : d['c'] = 3

In : d.c
Out: 3

In : len(d)
Out: 3
```

```
In : d.d
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-64-94e3405c7951> in <module>()
----> 1 d.d

AttributeError: 'AttrDict' object has no attribute 'd'
```

# 最好的attrdict实现

```
class attrDict(dict):
    def __init__(self, *args, **kwargs):
        dict.__init__(self, *args, **kwargs)
        self.__dict__ = self
```

# 延伸阅读

1. http://dongweiming.github.io/Expert-Python/#31
2. http://zetcode.com/lang/python/oop/
3. https://docs.python.org/3/tutorial/classes.html
4. https://rszalski.github.io/magicmethods/
5. https://dbader.org/blog/python-dunder-methods