面向对象编程基础

面向对象编程

- 面向过程的方法将程序视为一系列命令集合，通过函数进行组织
- 面向对象的方法将程序视为一系列对象的集合，通过对象间的消息协作来组织

例子：面向过程打印成绩

```
student1 = { 'name': 'Michael', 'score': 98 }
student2 = { 'name': 'Bob', 'score': 81 }


def print_score(student):
    print("{}: {}".format(student['name'], student['score']))


print_score(student1)
print_score(student2)
```

```
>>>
Michael: 98
Bob: 81
```

例子：面向对象打印成绩

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print("{}: {}".format(self.name, self.score))

michael = Student("Michael", 98)
bob = Student("Bob", 81)

michael.print_score()
bob.print_score()
```

```
>>>
Michael: 98
Bob: 81
```

类的定义

```
class Student(object):
    pass
```

```
michael = Student()
```

```
>>> Student
<class '__main__.Student'>
>>> michael
<__main__.Student object at 0x03BA6E90>
```

类的定义

```
michael.name = ’Michael’
```

实例对象动态添加特性

```
>>> michael.name
'Michael'
```

```
bob = Student()
```

```
>>> bob.name
AttributeError: 'Student' object has no attribute 'name'
```

定义方法(method)

```
class Student(object):
    def print_status(self):
        print("Status: ", "Student")

michael = Student()

print(michael.print_status)
```

```
>>>
<bound method Student.print_status of <__main__.Student object at 0x03BF2410>>
```

```
michael.print_status()
```

```
>>> Status:  Student
```

__init__方法

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

bart = Student('Bart Simpson', 59)
```

```
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

方法 vs. 函数

- 方法是绑定到对象的函数

```
method ≈ (object, function)
```

- 方法调用的特殊语义

```
object.method(arguments) = function(object, arguments)
```

Example: Pizza

```
class Pizza:
    def __init__(self, radius, toppings, slices=8):
        self.radius = radius
        self.toppings = toppings
        self.slices_left = slices

    def eat_slice(self):
        if self.slices_left > 0:
            self.slices_left -= 1
        else:
            print("Oh no! Out of pizza")

    def __repr__(self):
        return '{}" pizza'.format(self.radius)
```

Example: Pizza

```
p = Pizza(14, ("Pepperoni", "Olives"), slices=12)
print(Pizza.eat_slice)
# => <function Pizza.eat_slice at 0x04594A50>


print(p.eat_slice)
# => <bound method Pizza.eat_slice of 14" Pizza>

method = p.eat_slice
method.__self__          # => 14" Pizza
method.__func__          # => <function Pizza.eat_slice>

p.eat_slice()            # Implicitly calls Pizza.eat_slice(p)
```

类变量 vs. 实例变量

```
class Dog:
    kind = 'Canine'   # class variable shared by all instances

    def __init__(self, name):
        self.name = name  # instance variable unique to each instance

a = Dog('Astro')
pb = Dog('Mr. Peanut Butter')


 a.kind     #  'Canine' (shared by all dogs)
 pb.kind    #  'Canine' (shared by all dogs)
 a.name     # 'Astro' (unique to a)
 pb.name    # 'Mr. Peanut Butter' (unique to pb)
```

类变量 vs. 实例变量

```
class Student(object):
    name = 'Student'

michael = Student()
print("instance: ", michael.name)
print("class: ", Student.name)
print()
michael.name = 'Michael'
print("instance: ", michael.name)
print("class: ", Student.name)
print()
del michael.name
print("instance: ", michael.name)
```

```
>>>
instance: Student
class: Student

instance: Michael
class: Student

instance: Student
```

Warning

```
class Dog:
    tricks = []

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Warning

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks

# => ['roll over', 'play dead'] (shared value)
```

用参数初始化？

```
class Dog:
    # Let’s try a default argument!
    def __init__(self, name=’’, tricks=[]):
        self.name = name
        self.tricks = tricks

    def add_trick(self, trick):
        self.tricks.append(trick)
```

用参数初始化？

```
d = Dog(’Fido’)
e = Dog(’Buddy’)
d.add_trick(’roll over’)
e.add_trick(’play dead’)
d.tricks

# => [’roll over’, ’play dead’] (shared value)
```

为每个实例创建列表

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # New list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

为每个实例创建列表

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')

d.tricks  # => ['roll over']
e.tricks  # => ['play dead']
```

方法的类型

- 以self作为第一个参数的方法是实例方法 (instance method)
- 用前缀装饰器@classmethod指定的方法是类方法 (class method)
- 用@staticmethod修饰的方法是静态方法(static method)，既不会影响类也不会影响类的对象

方法的类型

```
class A():
    count = 0
    def __init__(self):
        A.count += 1
    def exclaim(self):
        print(”I’m an A!”)
    @classmethod
    def kids(cls):
        print(”A has”, cls.count, ”little objects.”)

easy_a = A()
breezy_a = A()
wheezy_a = A()
A.kids()                # => A has 3 little objects.
```

方法的类型

```
class CoyoteWeapon():
    @staticmethod
    def commercial():
        print(”This CoyoteWeapon has been brought to you by Acme”)


CoyoteWeapon.commercial()


# => This CoyoteWeapon has been brought to you by Acme
```

```
class A():
    def f(*args):
        return args
class B():
    @classmethod
    def f(*args):
        return args
class C():
    @staticmethod
    def f(*args):
        return args

a = A()
print(a.f('an arg'))
print(a.f('an arg')[0] is a)
b = B()
print(b.f('an arg'))
print(b.f('an arg')[0] is B)
c = C()
print(c.f('an arg'))
```

```
>>>
(<__main__.A object at 0x02BDCF70>,
'an arg')
True

(<class '__main__.B'>, 'an arg')
True

('an arg',)
```

数据封装

```
def print_score(student):
    print("{}: {}".format(student.name, student.score))

print_score(bart)
```

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print("{}: {}".format(self.name, self.score))
```

```
bart.print_score()
```

访问限制

- 外部代码可以直接调用实例变量来操作数据

```
bart = Student("Bart Simpson", 59)
bart.score = 99
```

- 实例的变量名以__开头，声明为“私有变量”

访问限制

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print("{}: {}".format(self.__name, self.__score))

bart = Student("Bart Simpson", 59)
bart.__name
```

```
>>>
Traceback (most recent call last):
  File "<input>", line 11, in <module>
AttributeError: 'Student' object has no attribute '__name'
```

访问限制

- 添加函数以获取__name和__score

```
class Student(object):
    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score

    def set_score(self, score):
        self.__score = score
```

访问限制

- 直接获取__name？

```
>>> bart._Student__name
'Bart Simpson'
```

- 避免直接给__name赋值

```
>>> bart.__name = "Jack"
>>> bart.__name
'Jack'
>>> bart.get_name()
'Bart Simpson'
```

继承

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    pass

class Cat(Animal):
    pass

dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

```
>>>
Animal is running...
Animal is running...
```

继承

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

继承

- 使用super()

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

class EmailPerson(Person):
    def __init__(self, name, email):
        super().__init__(name)
        self.email = email

bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

多态

- isinstance()函数

```
def run_twice(animal):
    animal.run()
    animal.run()
```

新增Animal的子类，无需对run_twice()做任何修改

```
run_twice(Animal())
```

```
>>>
Animal is running...
Animal is running...
```

```
run_twice(Dog())
```

```
>>>
Dog is running...
Dog is running...
```

多态

- 动态语言的“鸭子类型”
  - 一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子

```
class Timer(object):
    def run(self):
        print('TickTick...')

run_twice(Timer())
```

```
>>>
TickTick...
TickTick...
```

获取对象信息

- type()函数
- isinstance()函数
- dir()函数 – 获得一个对象的所有属性

```
>>> dir(dog)
['__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__', 'eat', 'run']
```

获取对象信息

- 特殊方法
  - __len__方法返回长度

```
>>> len('ABC')
3

>>> 'ABC'.__len__()
3
```

```
class Student(object):
    ……
    def __len__(self):
        return len(self.__name)

bart = Student("Bart Simpson", 59)
```

```
>>> len(bart)
12
```

获取对象信息

```
class MyObject(object):
    def __init__(self):
        self.x = 10

    def power(self):
        return self.x * self.x

obj = MyObject()

hasattr(obj, ’x’)                   >>> True
hasattr(obj, ’y’)                   >>> False
setattr(obj, ’y’, 20)
hasattr(obj, ’y’)                   >>> True
getattr(obj, ’y’)                   >>> 20
getattr(obj, ’z’)         >>> AttributeError: 'MyObject' object has no attribute 'z'
getattr(obj, ’z’, 404)              >>> 404
```

获取对象信息

```
hasattr(obj, 'power')

>>> True

getattr(obj, 'power')
>>> <bound method MyObject.power of <MyObject object at 0x048307D0>>

fn = getattr(obj, 'power')

>>> <bound method MyObject.power of <MyObject object at 0x048307D0>>


>>> fn()
100
```

获取对象信息

- 在不知道对象信息的情况下才需要去获取对象信息

```
sum = obj.x + obj.y

sum = getattr(obj, 'x') + getattr(obj, 'y')
```

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

使用__slots__

- 创建实例后可动态绑定属性和方法

```
class Student(object):
    pass


s = Student()
s.name = 'Michael'
print(s.name)
```

```
>>>
Michael
```

使用__slots__

```
def set_age(self, age):
    self.age = age

from types import MethodType
s.set_age = MethodType(set_age, s)
s.set_age(25)
s.age                  # => 25


s2 = Student()
s2.set_age
```

# => AttributeError: 'Student' object has no attribute 'set_age'

使用__slots__

```
def set_score(self,  score):
    self.score = score

Student.set_score = set_score

s.set_score(100)
s.score                    # =>100


s2.set_score(59)
s2.score                   # => 59
```

使用__slots__

- 限制实例的属性添加

```
class Student(object):
    __slots__ = (’name’, ’age’)

s = Student()
s.name = ’Michael’
s.age = 25
s.score = 90
```

# => AttributeError: 'Student' object has no attribute 'score'

使用@property

```
s = Student()
s.score = 9999
```

访问属性方便，
但无参数检查

```
class Student(object):

    def get_score(self):
         return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

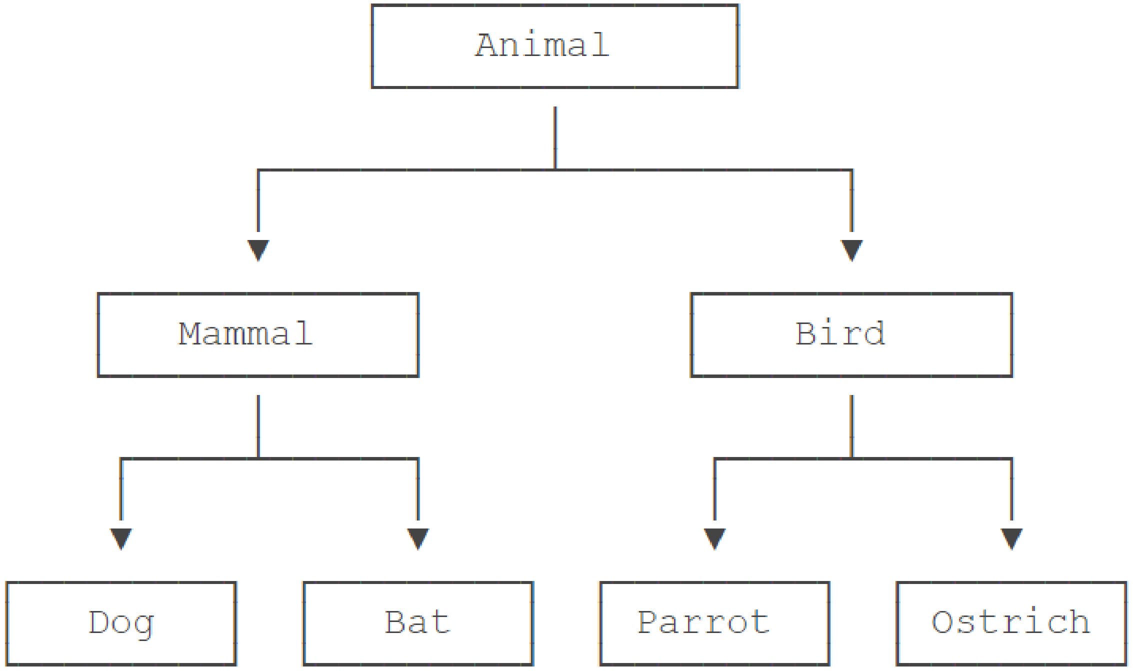
```
>>> s = Student()
>>> s.set_score(60)
>>> s.get_score()
60
>>> s.set_score(9999)
ValueError: score must between 0 ~ 100!
```

有参数检查，但调用较麻烦

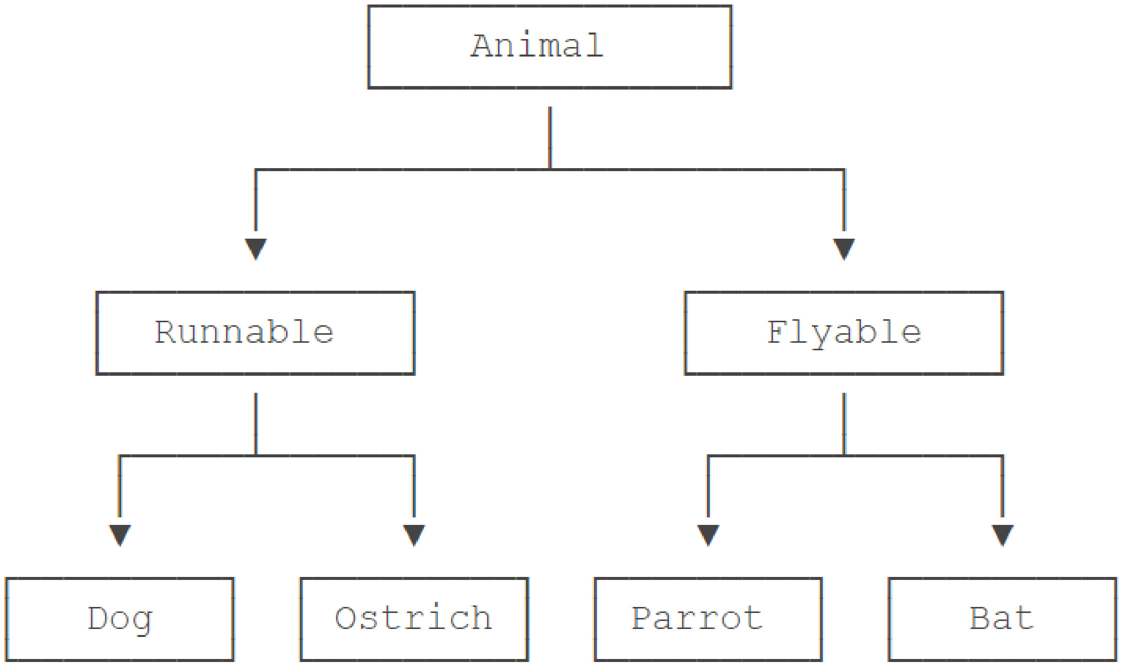
```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value


 s = Student()
 s.score = 60     # OK, 实际转化为s.set_score(60)
 s.score          # OK, 实际转化为s.get_score()  => 60
 s.score = 9999   # ValueError: score must between 0 ~ 100!
```

多重继承

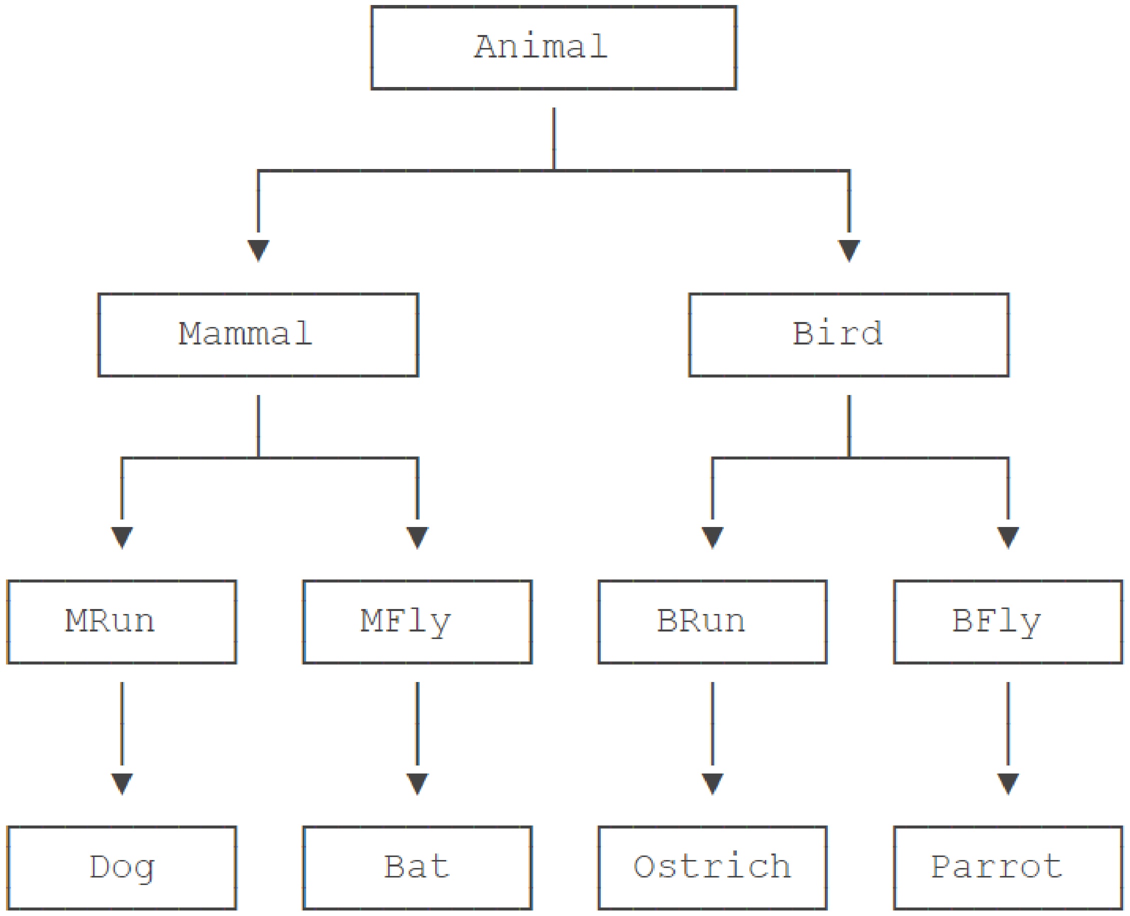
- 假设要实现以下4种动物：Dog, Bat, Parrot, Ostrich



多重继承



多重继承


Animal
Mammal
Bird
MRun
MFly
BRun
BFly
Dog
Bat
Ostrich
Parrot

```python
class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass
class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass
class Bat(Mammal):
    pass
class Parrot(Bird):
    pass
class Ostrich(Bird):
    pass
```

```python
class Runnable(object):
    def run(self):
        print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')


class Dog(Mammal, Runnable):
    pass

class Bat(Mammal, Flyable):
    pass
```

多重继承

```
class A():
    def ping(self):
        print('ping:', self)

class B(A):
    def pong(self):
        print('pong:', self)

class C(A):
    def pong(self):
        print('PONG:', self)
```

```
class D(B, C):
    def ping(self):
        super().ping()
        print('post-ping:', self)

    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

Python按照特定的顺序（方法解析顺序，Method Resolution Order, MRO）遍历继承图

多重继承

```
class A():
    def ping(self):
        print('ping:', self)

class B(A):
    def pong(self):
        print('pong:', self)

class C(A):
    def pong(self):
        print('PONG:', self)
```

```
class D(B, C):
    def ping(self):
        super().ping()
        print('post-ping:', self)

    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

```
>>> D.__mro__
(<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>)
```

多重继承

```
d = D()
print('d.ping():')
d.ping()
print()
print('d.pingpong():')
d.pingpong()
```

```
>>>
d.ping():
ping: <D object at 0x03A9A090>
post-ping: <D object at 0x03A9A090>

d.pingpong():
ping: <D object at 0x03A9A090>
post-ping: <D object at 0x03A9A090>
ping: <D object at 0x03A9A090>
pong: <D object at 0x03A9A090>
pong: <D object at 0x03A9A090>
PONG: <D object at 0x03A9A090>
```

定制类

```
class MagicClass:
    def __init__(self): ...
    def __contains__(self, key): ...
    def __add__(self, other): ...
    def __iter__(self): ...
    def __next__(self): ...
    def __getitem__(self, key): ...
    def __len__(self): ...
    def __lt__(self, other): ...
    def __eq__(self, other): ...
    def __str__(self): ...
    def __repr__(self): ...   # And even more...
```

定制类

```
x = MagicClass()
y = MagicClass()

str(x)     # => x.__str__()

x == y     # => x.__eq__(y)

x < y      # => x.__lt__(y)
x + y      # => x.__add__(y)
iter(x)    # => x.__iter__()
next(x)    # => x.__next__()
len(x)     # => x.__len__()
el in x    # => x.__contains__(el)
```

例子

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def rotate_90_CC(self):
        self.x, self.y = -self.y, self.x

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return "Point({0}, {1})".format(self.x, self.y)
```

例子

```
o = Point()
print(o)   # Point(0, 0)

p1 = Point(3, 5)
p2 = Point(4, 6)
print(p1, p2)   # Point(3, 5) Point(4, 6)

p1.rotate_90_CC()
print(p1)   # Point(-5, 3)


print(p1 + p2)   # Point(-1, 9)
```

__str__

```
class Student(object):
    def __init__(self, name):
        self.name = name

print(Student('Michael'))
```

```
>>>
<Student object at 0x03A9F9F0>
```

__str__

```
class Student(object):

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'Student object (name: {})'.format(self.name)

    __repr__ = __str__

print(Student('Michael'))
```

```
>>>
Student object (name: Michael)
```

__iter__

```python
class Fib(object):

    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 10: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值

for n in Fib():
    print(n)
```

```
>>>
1
1
2
3
5
8
```

__getitem__

```
class slice(stop)
class slice(start, stop[, step])
```

```
class Fib(object):

    def __getitem__(self, n):
        if isinstance(n, int):
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice):
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

__getitem__

```
f = Fib()
print(f[0])            # => 1
print(f[5])            # => 8
print(f[100])          # => 573147844013817084101
print(f[0:5])          # => [1, 1, 2, 3, 5]
print(f[:10])          #=> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

__call__

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __call__(self, z):
        return self.x + self.y + z
```

```
>>> p = Point(1,2)
>>> p(7)
10
```

Collections模块

namedtuple

具备tuple的不变性，又可以根据属性来引用

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])  # Defining the namedtuple

p = Point(10, y=20)  # Creating an object

p                    # => Point(x=10, y=20)

p.x + p.y            # => 30

p[0] + p[1]          # => 30

x, y = p             # Unpacking the tuple

x                    # => 10
y                    # => 20
```

namedtuple

```
from collections import namedtuple

Student = namedtuple('Student', ['id', 'name', 'score'])

students = [(1, 'Wu', 90), (2, 'Xing', 89), (3, 'Yuan', 98)]

for s in students:
    stu = Student._make(s)
    print(stu)

# Output:
# Student(id=1, name='Wu', score=90)
# Student(id=2, name='Xing', score=89)
# Student(id=3, name='Yuan', score=98)
```

namedtuple

```
Point = namedtuple('Point', ['x', 'y'])

p = Point(11,22)
print(p._asdict())
# OrderedDict([('x', 11), ('y', 22)])

new_p = p._replace(x=33)
print(p)
print(new_p)
# Point(x=11, y=22)
# Point(x=33, y=22)
```

deque

高效实现插入和删除操作的双向列表，适合用于队列和栈

```
from collections import deque
q = deque([’a’, ’b’, ’c’])

q.append(’x’)
q.appendleft(’y’)

q    # => deque([’y’, ’a’, ’b’, ’c’, ’x’])
```

defaultdict

key不存在时，返回一个默认值

```
from collections import defaultdict

strings = (’puppy’, ’kitten’, ’puppy’, ’puppy’,
           ’weasel’, ’puppy’, ’kitten’, ’puppy’)

counts = defaultdict(lambda: 0)  # 使用lambda来定义简单的函数

for s in strings:
    counts[s] += 1
```

defaultdict

```
from collections import defaultdict

s=[(’yellow’,1),(’blue’, 2), (’yellow’, 3), (’blue’, 4),
(’red’, 1)]
d=defaultdict(list)

for k, v in s:
    d[k].append(v)
a=sorted(d.items())
print(a)
# [(’blue’, [2, 4]), (’red’, [1]), (’yellow’, [1, 3])]
```

OrderedDict

保持Key的顺序

```
from collections import OrderedDict
d = dict([('a', 1), ('b', 2), ('c', 3)])
d # dict的Key是无序的
# => {'a': 1, 'c': 3, 'b': 2}

od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
od # OrderedDict的Key是有序的
# => OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Counter

一个简单的计数器

```
from collections import Counter
c = Counter()
for ch in 'programming':
    c[ch] = c[ch] + 1
c   # => Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})

# Counter类的创建
c = Counter('gallahad')   # 从一个可iterable对象（list、tuple、dict、字符串等）创建
c = Counter({'a': 4, 'b': 2})   # 从一个字典对象创建
c = Counter(a=4, b=2)   # 从一组键值对创建


Counter('abracadabra').most_common(3)
# => [('a', 5), ('r', 2), ('b', 2)]
```

异常处理

异常

```
>>> 10 * (1/0)

Traceback (most recent call last):
 File "<stdin>", line 1
ZeroDivisionError: division by zero

>>> 4 + spam*3

Traceback (most recent call last):
 File "<stdin>", line 1
NameError: name 'spam' is not defined

>>> '2' + 2

Traceback (most recent call last):
 File "<stdin>", line 1
TypeError: Can't convert 'int' object to str
implicitly
```

处理异常

```
def read_int():
    """Reads an integer from the user (broken)"""
    return int(input("Please enter a number: "))
```

如果输入非数值会发生什么？

处理异常

```
def read_int():
    ″″″Reads an integer from the user (fixed)″″″
    while True:
        try:
            x = int(input(″Please enter a number: ″))
            break
        except ValueError:
            print(″Oops! Invalid input. Try again...″)
    return x
```

处理异常

```
try:
    distance = int(input(”How far? ”))
    time = distance / car.speed
    car.drive(time)

except ValueError as e:
    print(e)

except ZeroDivisionError:
    print(”Division by zero!”)

except (NameError, AttributeError):
    print(”Bad Car”)

except:
    print(”Car unexpectedly crashed!”)
```

Bind a name to the exception instance

Catch multiple exceptions

"Wildcard" catches everything

处理异常

```
def read_int():
    """Reads an integer from the user (fixed?)"""
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except:
            print("Oops! Invalid input. Try again...")
    return x
```

"I'll just catch 'em all!"

Oops! Now we can't CTRL+C to escape

抛出异常

```
>>> raise NameError('Why hello there!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Why hello there!

>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError
```

You can raise either instance objects or class objects

抛出异常

```
try:
    raise NotImplementedError("TODO")
except NotImplementedError:
    print('Looks like an exception to me!')
    raise
# Looks like an exception to me!
# Traceback (most recent call last):
# File "<stdin>", line 2, in <module>
# NotImplementedError: TODO
```

自定义异常

```
class MyException(BaseException):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return self.value

if 1 > 0:
    raise MyException("1确实大于0，'--索引错误'")
```

```
>>>
Traceback (most recent call last):
  File "<input>", line 7, in <module>
MyException: 1确实大于0，'--索引错误'
```

使用else

```
try:
    …
except …:
    …
else:
    do_something()
```

Code that executes if the try clause does not raise an exception

使用else

```
try:
    update_the_database()
except TransactionError:
    rollback()
    raise
else:
    commit()
```

If the commit raises an exception, we might actually *want* to crash

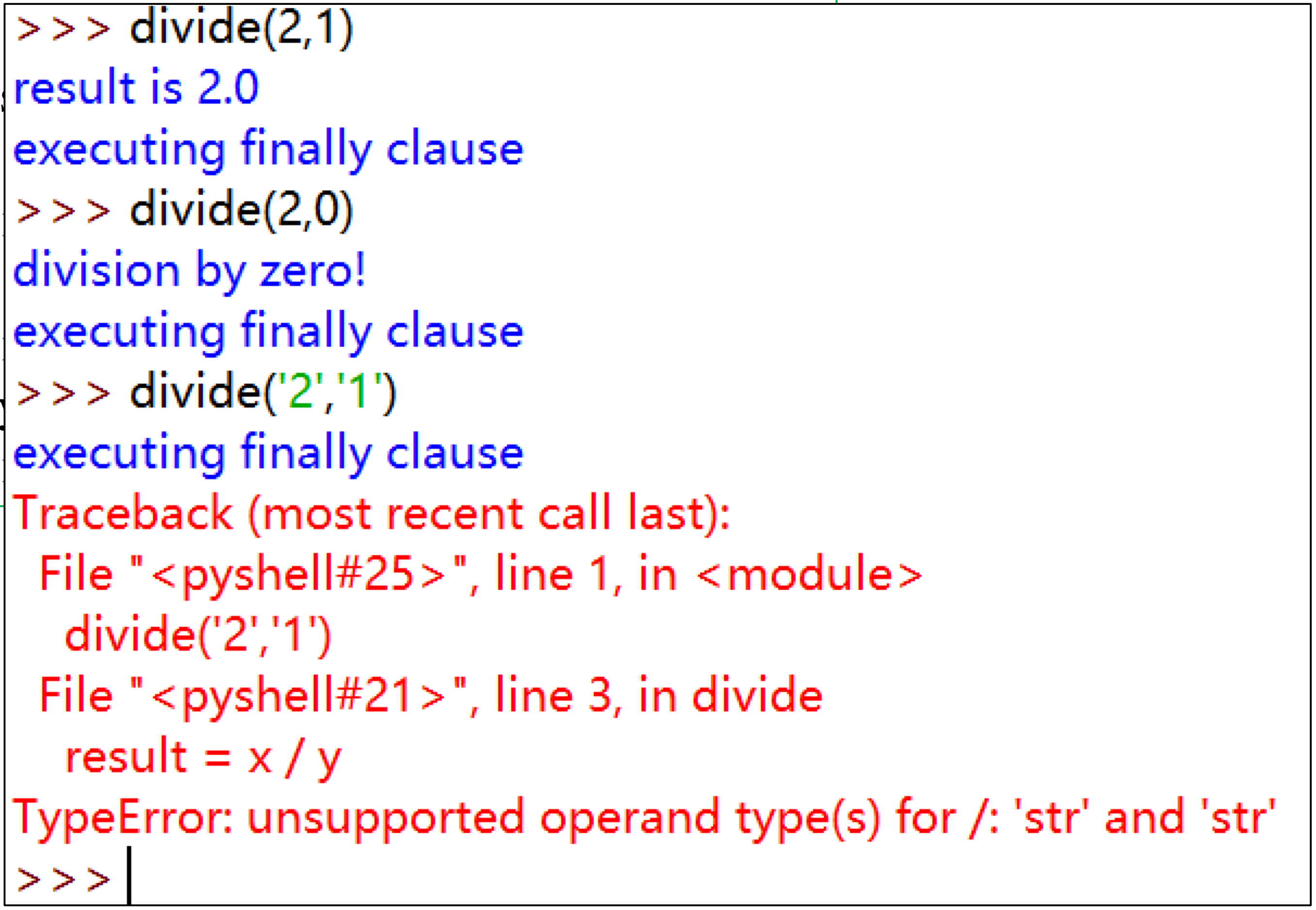
finally

```
try:
    raise NotImplementedError
finally:
    print('Goodbye, world!')

# Goodbye, world!
# Traceback (most recent call last):
# File "<stdin>", line 2, in <module>
# NotImplementedError
```

finally总是在离开try/except/else语句块前执行

finally

```
def divide(x,  y):
    try:
        re
    except
        pr
    else:
        pr
    finall
        pr
```



```
>>> divide(2,1)
result is 2.0
executing finally clause
>>> divide(2,0)
division by zero!
executing finally clause
>>> divide('2','1')
executing finally clause
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    divide('2','1')
  File "<pyshell#21>", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>> 
```

调用栈

```
# err.py
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

```
>>>
Traceback (most recent call last):
  File "<input>", line 11, in <module>
  File "<input>", line 9, in main
  File "<input>", line 6, in bar
  File "<input>", line 3, in foo
ZeroDivisionError: division by zero
```