- eval(string)，eval可将string转换为express（表达式）

- lambda表达式的特殊用法

```
def test(a,b,func)
    return func(a,b)

func_new = input("请输入一个表达式") #resume func_new: lambda x,y:x+y
func_new = eval(func_new) #python 3

result = test(3,4,func_new)
```

- num += num 和 num = num +num 不同

```
#不的数据类型结果不同：
#如果是可变类型：字典、列表传引用；基本数据类型传值
a = [100]

def test(num):
    num += num
    num = num + num
    print(num)

test(a)
print(a)
```

- 返回多个值，本质是返回元组

```
def div(a,b)
    shang = a//b
    yushu = a%b
    return shang, yushu

sh ,yu = div(5,2)
asdf
```

- 测量引用计数的方式

```
import sys
a = "adsf"
b = a
sys.getrefcount(a)
```

- 子类调用父类的重写方法

```
class Animal:
    def eat(self):
        print("-----吃----")
    def drink(self):
        print("-----喝----")
    def sleep(self):
        print("-----睡觉----")
    def run(self):
        print("-----跑----")

class Dog(Animal):
    def bark(self):
        print("----汪汪叫---")


class Xiaotq(Dog):
    def fly(self):
        print("----飞----")

    def bark(self):
        print("----狂叫-----")

        #第1种调用被重写的父类的方法
        #Dog.bark(self)

        #第2种
        super().bark()


xiaotq = Xiaotq()
xiaotq.fly()
xiaotq.bark()
xiaotq.eat()
```

- python实现类似C++伪多态

```
class Dog(object):
```

```
    def print_self(self):
        print("Dog...")

class Xiaotq(Dog):
    def print_self(self):
        print("Xiaotq...")


def introduce(temp):
    temp.print_self()


dog1 = Dog()
dog2 = Xiaotq()

introduce(dog1)
introduce(dog2)
```

- 类属性、实例属性

```
class Tool(object):

    #类属性
    num = 0

    #方法
    def __init__(self, new_name):
        #实例属性
        self.name = new_name
        #对类属性+=1
        Tool.num += 1


tool1 = Tool("铁锹")
tool2 = Tool("工兵铲")
tool3 = Tool("水桶")

print(Tool.num)
```

- 类方法、实例方法、静态方法

```
class Game(object):

    #类属性
    num = 0
```

```
    #实例方法
    def __init__(self):
        #实例属性
        self.name = "laowang"

    #类方法
    @classmethod
    def add_num(cls):
        cls.num = 100

    #静态方法
    @staticmethod
    def print_menu():
        print("----------------------")
        print(" 穿越火线V11.1 ")
        print(" 1. 开始游戏")
        print(" 2. 结束游戏")
        print("----------------------")

game = Game()
# 调用类方法的方式一：可以通过类的名字调用类方法
Game.add_num()#
# 调用类方法的方式二：还可以通过这个类创建出来的对象 去调用这个类方法
game.add_num()
print(Game.num)

#Game.print_menu()#通过类 去调用静态方法
game.print_menu()#通过实例对象 去调用静态方法
```

- **init**()方法和__new__()方法
  - **new**()只负责创建对象，然后return对象
  - **init**()利用__new__()创建后的对象，对其初始化

- 简单工厂模式：通过剥离CarStore类，解耦特定的方法

```
class CarStore(object):
    def __init__(self):
        self.factory = Factory()

    def order(self, car_type):
        return self.factory.select_car_by_type(car_type)

class Factory(object):
    def select_car_by_type(self, car_type):
        if car_type=="索纳塔":
            return Suonata()
```

```
        elif car_type=="名图":
            return Mingtu()
        elif car_type=="ix35":
            return Ix35()

class Car(object):
    def move(self):
        print("车在移动....")
    def music(self):
        print("正在播放音乐....")
    def stop(self):
        print("车在停止....")

class Suonata(Car):
    pass

class Mingtu(Car):
    pass

class Ix35(Car):
    pass

car_store = CarStore()
car = car_store.order("索纳塔")
car.move()
car.music()
car.stop()
```

- 工厂方法模式:父类只设计接口，子类通过继承负责实现

```
from abc import ABCMeta, abstractmethod
class Store(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def select_car(self):
        pass

    def order(self, car_type):
        return self.select_car(car_type)

class BMWCarStore(Store):
    def select_car(self, car_type):
        return BMWFactory().select_car_by_type(car_type)

class BMWFactory(object):
    def select_car_by_type(self, car_type):
        pass
```

```
bmw_store = BMWCarStore()
bmw = bmw_store.order("720li")
```

- 单例模式

```
class Dog(object):
    # 类的私有属性，用来保存初次创建的对象实例
    __instance = None
    __init_flag = False
    def __new__(cls, *more):
        if cls.__instance is None:
            cls.__instance = object.__new__(cls)
            return cls.__instance
        else:
            # return 上一次初次创建对象的引用
            return cls.__instance

    def __init__(self, name):
        if Dog.__init_flag is False:
            self.name = name
            Dog.__init_flag = True


if "__main__" == __name__:
    a = Dog("旺财")
    b = Dog("哮天犬")
    print(id(a))
    print(a.name)
    print(id(b), b.name)
```

- Python中的异常

```
#coding=utf-8

try:
    num = input("xxx:")
    int(num)
    #11/0
    #open("xxx.txt")
    #print(num)
    print("-----1----")
# python2这样写：except NameError,FileNotFoundError：
except (NameError,FileNotFoundError):
    print("如果捕获到异常后做的 处理....")
# python2这样写：except
except Exception as ret:
```

```
    print("如果用了Exception,那么意味着只要上面的except没有捕获到异常,这个except一定会捕获到")
    print(ret)
else:
    print("没有异常才会执行的功能")
finally:
    print("------finally-----")

print("-----2----")
```

- 抛出自定义异常类

```
class ShortInputException(Exception):
    # 用户自定义异常
    def __init__(self, length, atleast):
        self.length = length
        self.atleast = atleast

def main():
    try:
        s = input("请输入")
        if len(s) < 3:
            # 引发自己的异常
            raise ShortInputException(len(s), 3)
    # result指向了创建的ShortInputExcception对象
    except ShortInputExcception as result:
        print("ShortInputException:输入的长度是 %d,长度至少应该是 %d",%(result.length,result.atleast))
    else:
        print("没有发生异常")
```

- __all__的使用

```
# test2不可被其他模块使用
__all__ = ['test1','num','Test']

def test1():
    print('-------test1---------')

def test2():
    print('-------test2---------')

num = 100
```

```
class Test(object)
    pass
```

- __init__.py在包中的使用

```
__all__ = ['sendmsg']

from . import sendmsg
```

- python模块的发布
  - 在要发布包（一个目录）的同级目录建立setup.py文件
  - 编辑setup.py文件,py_modules指明包中的模块

```
from distutils.core import setup

setup(name="willxin",
      version="1.0",
      description="willxin's module",
      author="willxin",
      py_modules=['TestMsg.sendmsg', 'TestMsg.recvmsg'])
```

  - 构建bulid目录：$ python setup.py build
  - 创建tar包：$ python setup.py sdist
  - 他人安装包：python setup.py install