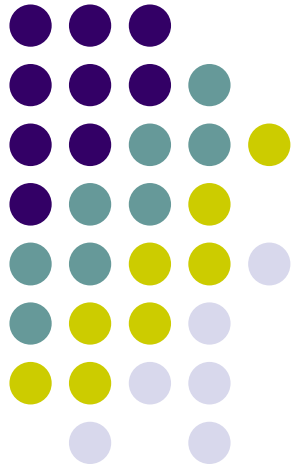
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# 第7章 面向对象思想

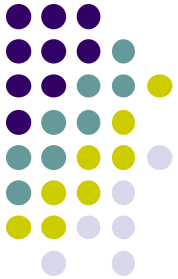
# Object Oriented Programming



申丽萍

lpshen@sjtu.edu.cn

# 第7章 面向对象设计



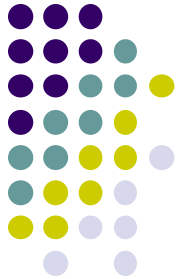
- 面向对象基本概念
- 类的定义
- 面向对象设计
- 编程案例

# 数据与操作:传统观点

- 数据类型
  - 某种值的集合
  - 运算(操作)的集合
- 计算就是对数据进行操作
  - 数据与操作分离
  - 数据是被动的,操作是主动的
- 例如:string类型
  - 值是'abc'等
  - 对串的操作有+,*,len()等

```
>>> list1=[2,5,1,9,6,8]
>>> list2=sorted(list1)
>>> list2
[1, 2, 5, 6, 8, 9]
>>> list1
[2, 5, 1, 9, 6, 8]
```

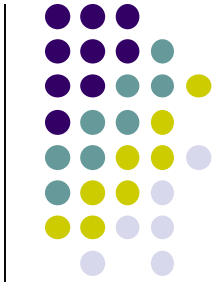
# 数据与操作:面向对象观点



- 对象(*Object*):集数据与操作于一身.
  - 对象知道一些信息
  - 对象能对那些信息进行处理
- 计算:向对象发出请求操作的消息.
  - 消息即请求执行对象的操作
- 面向对象(*Object-Oriented*):软件系统由各种对象组成,对象之间通过消息进行交互.
- 现代软件系统几乎都是OO设计和实现.

```
>>> list1
[2, 5, 1, 9, 6, 8]
>>> list1.sort()
>>> list1
[1, 2, 5, 6, 8, 9]
```

# OO基本概念

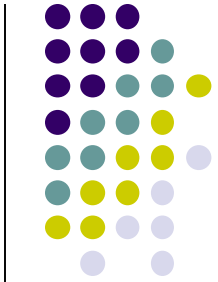


- 类(*class*):描述同类对象的共性
  - 包含的数据：属性（attribute)
    - 任何类型的数据,甚至可以是对其他对象的引用.
  - 能执行的操作:方法(*method*)
- 对象(*object*):类的实例(*instance*)
  - 同类的不同对象可有不同的数据值(实例变量),但能执行的操作是一样的
- 创建对象:使用类的构造器(*constructor*).

  <类名>(<参量1>,<参量2>,…)
- 消息:请求对象执行它的方法.

  <对象>.<方法名>(<参量1>,<参量2>,…)

# 例：函数库与面向对象



```
>>> import string
>>> s="hello world!"
>>> string.upper(s)
'HELLO WORLD!'
>>> s.upper()
'HELLO WORLD!'
```

```
>>> ls = [3,6,2,8]
>>> sorted(ls)
[2, 3, 6, 8]
>>> ls.sort()
>>> ls
[2, 3, 6, 8]
```

# 例：时间的表示

old style:

morning-call time:

hours:minutes:seconds

```
m_hours = 6
m_minutes = 30
m_seconds = 0
```

diner time:

hours:minutes:seconds

```
d_hours = 12
d_minutes = 0
d_seconds = 0
```

exercise time:

hours:minutes:seconds

```
e_hours = 16
e_minutes = 30
e_seconds = 0
```

printTime(hours, minutes, seconds)

# 例：时间的表示 OO方法

myTime:

```
hours
minutes
seconds
----------
printTime()
......
```

```
>>  mtime = myTime()  ------> 构建函数，产生一个 object

    mtime.hours = 6
    mtime.minutes = 30
    mtime.seconds = 0

>>  mtime.printTime()

    dtime = myTime()
    ...

>>  etime = myTime()
    ...
```

比较：

printTime(hours, minutes, seconds)

# 例：学生的信息

**Student_information:**

name,

ID_number,

courses_taken,

campus_address,

home_address,

GPA (grade point average)

---

printSTInfo()

...

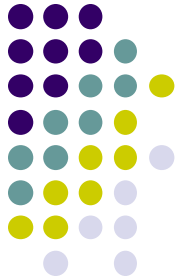
```
s1 = Student_information()

s1.name = "Wang Fang"
s1.ID_number = 123456
...

s1.printSTInfo()
```

```
s2 = Student_information()

s2.name = "Zhang Fang"
s2.ID_number = 123457
...
```

**Course_information:**

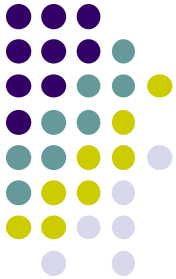
ID_number
instructor_ID
students_list
course_addr_time
...

# 对象



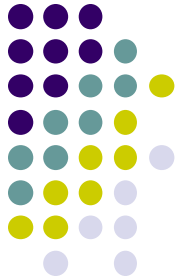
- 对象的构成:
  - 一组相关信息
    - 存储在实例变量中
  - 处理该信息的一组方法
    - 对象内的函数
- 类决定了对象具有哪些信息和方法
  - 对象是类的实例
  - 通过类的构造子创建新对象
- 定义自己的类:即以OO方法来组织自己程序要处理的数据.

# 第7章 面向对象设计



- 面向对象基本概念
- 类的定义
- 面向对象设计
- 编程案例

# 类的定义



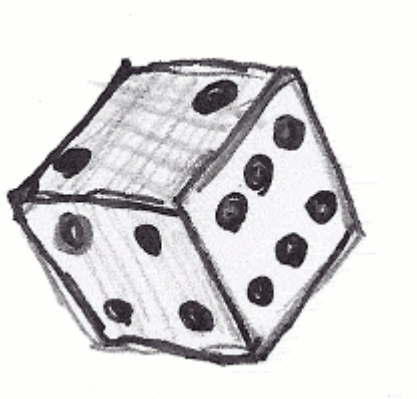
- 语法

  `class` <类名>:

  <方法定义>

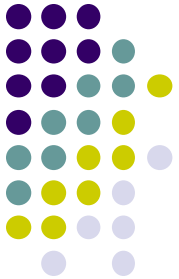
  - 方法定义:同函数定义.
    - 方法是依附于类的函数,普通函数则是独立的.
    - 方法的第一个参量是专用的:代指该方法的作用对象.
      - 此参量习惯用self这个名字.
  - 回忆:对象是数据和操作的结合.
    - 上面的类定义中,方法对应于操作.但数据呢?

# 例:类定义 -多面骰子



```
from random import randrange
class MSDie:
    def __init__(self,s):
            self.sides = s
            self.value = 1
    def roll(self):
            self.value = randrange(1,self.sides+1)
    def getValue(self):
            return self.value
    def setValue(self,v):
            self.value = v
```

# 例:类定义 - person



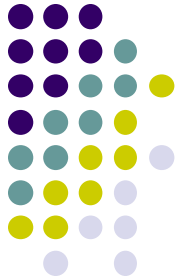
类构造函数

实例变量

特殊必需第一形参代表实例本身

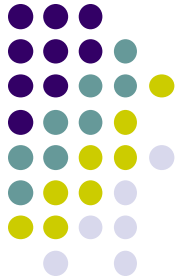
```
class Person:
    def __init__(self,n,y):
        self.name = n
        self.year = y
    def whatName(self):
        print "My name is",self.name
    def howOld(self,y):
        age = y - self.year
        if age > 0:
            print "My age in",y,"is",age
        else:
            print "I was born in",self.year
    def allInfo(self,y):
        self.whatName()
        self.howOld(y)
```

# 实例变量



- 和普通变量一样, Python类并不明显定义实例变量, 而是在方法中直接引入.
  - 主要是在__init__方法中
    - 用self.<实例变量>的方式给出
    - 如MSDie中的sides和value
  - 也可以实时创建
- 每个类的实例(对象)具有自己的实例变量副本,用来存储该对象自己的数据.
- 对实例变量的访问:
  - <对象>.<实例变量>
- 实例变量与函数局部变量不同!

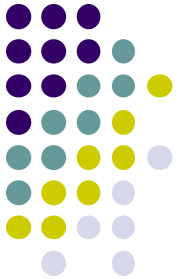
# 方法调用



- 方法调用:类似函数调用,但需指明对象.
  - 不需要为形参self提供实参.

```
class Person:
    def __init__(self,n,y):
        self.name = n
        self.year = y
    def whatName(self):
        print "My name is",self.name
    def howOld(self,y):
        age = y - self.year
        if age > 0:
            print "My age in",y,"is",age
        else:
            print "I was born in",self.year
    def allInfo(self,y):
        self.whatName()
        self.howOld(y)

def main():
    name = raw_input("Enter the name: ")
    birthYear = input("Enter the year when you were born: ")

    print
    p = Person(name, birthYear)
    p.whatName()
    p.howOld(2015)
```

# 构造器



- 对象构造器(*constructor*)

  __init__ （注意前后分别两个'_')

- 用法:

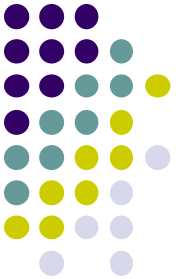
  (1)在类外部用类名生成新实例:

  ```
  die1 = MSDie(6)
  ```

  (2)Python创建一个MSDie新实例,并对该实例调用__init__(),从而初始化其实例变量:

  ```
  die1.sides = 6
  die1.value = 1
  ```

# 类模块文件



- 类定义可以单独构成模块,以提供给其他所有程序使用.
  - 就如同函数库一样.
  - 很多OO语言都提供类库.
- 良好程序设计风格:使用文档注释(*Documentation*)来说明类功能和用法.
  - Python提供专用注释: 文档注释串(*三个引号*)
    - 模块/类/函数下面的第一行可以是一个注释字符串. 形如

```
#projectile.py
""" This module provides ..."""
class Projectile:
""" This class ..."""
    def getX(self)
    " This function ..."
```

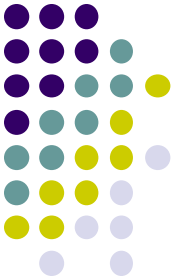
    - *docstring*被系统保存在模块/类/函数的属性__doc__中,可访问.

# 堆栈

- 堆栈是一种数据集合体，具有“后进先出”LIFO的特点。
- 主要操作：
  - push(x)：在堆栈顶部推入一个新数据x，x即成为新的栈顶元素；
  - pop()：从堆栈中取出栈顶元素，显然被取出的元素只能是最后加入堆栈的元素。
  - isFull()：检查堆栈是否已满。如果堆栈具有固定大小，那么满了之后是无法执行push()的；
  - isEmpty()：检查堆栈是否为空。如果堆栈是空的，那么pop()操作将出错。



# 堆栈



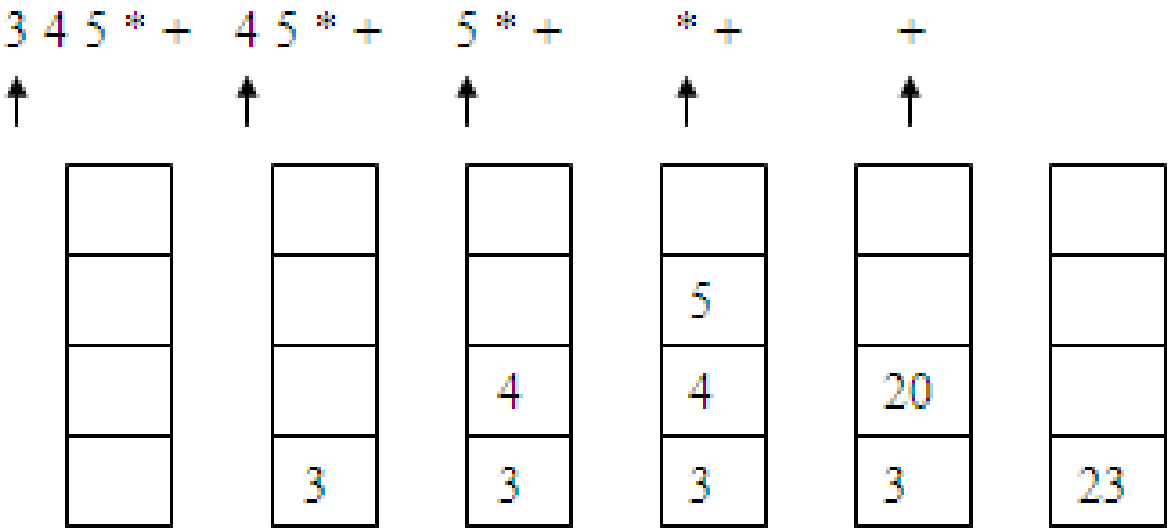
- 可以采用多种不同的方式来实现堆栈这个抽象数据类型。
- 用列表来实现:令列表stack是存放数据的堆栈，以列表尾为栈顶，那么向堆栈中放入元素就只能在尾部添加，可以用列表append方法来实现堆栈push()，列表pop方法来实现堆栈pop()

```
>>> from demo_complexDatatype_listStack import Stack
>>> s=Stack()
>>> s.items
[]
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> s.items
[1, 2, 3]
>>> s.pop()
3
>>> s.pop()
2
>>> s.items
[1]
>>> s.pop()
1
>>> s.isEmpty()
True
```

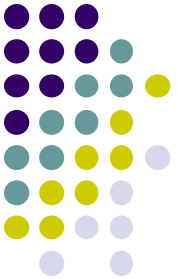
```
# implement stack using list.

class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def getLen(self):
        return (len(self.items))
    def isEmpty(self):
        return (self.items == [])
```

# 堆栈应用

- 算术表达式的中缀形式和后缀形式（无需括号）
- 例如“1＋2”可写成“1 2 +”、“3＋4＊5”可写成“3 4 5 * +”。后缀形式的表达式可以利用堆栈来非常方便地求值。

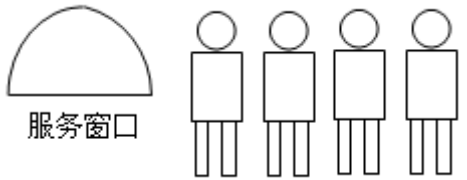


# 队列

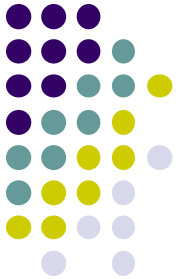


- 队列是一种数据集合体，具有“先进先出”FIFO的特点。
- 主要操作是：
  - enqueue：入队，即在队列尾部添加数据；
  - dequeue：出队，即将队列头部的数据移出队列作为返回值。
- 队列的具体实现有多种方式，例如可以用顺序列表、链表来实现队列。

```
>>> q=Queue()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> q.enqueue(3)
>>> q.items
[1, 2, 3]
>>> q.dequeue()
1
>>> q.items
[2, 3]
```



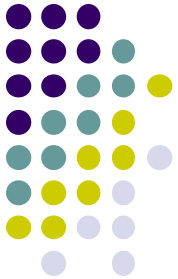
# 例：TOP GPA

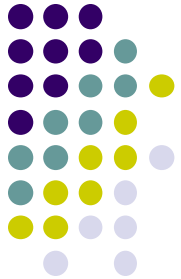


- Suppose we have a data file that contains student grade information.
- Each line of the file consists of a student's name, credit-hours, and grade points.
- Our job is to write a program that reads this file to find the student with the best GPA and print out their name, credit-hours, and GPA.

```
Adams, Henry          127     228
Comptewell, Susan     100     400
DibbleBit, Denny      18      41.5
Jones, Jim            48.5    155
Smith, Frank          37      125.33
```

# 例：TOP GPA



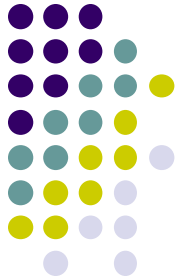
- ```
  class Student:
      def __init__(self, name, hours, qpoints):
          self.name = name
          self.hours = float(hours)
          self.qpoints = float(qpoints)
  ```
- The values for `hours` are converted to `float` to handle parameters that may be floats, ints, or strings.
- To create a student record:
  ```
  aStudent = Student("Adams, Henry", 127, 228)
  ```
- The coolest thing is that we can store all the information about a student in a single variable!
  ```

# 例：TOP GPA



- We need to be able to access this information, so we need to define a set of accessor methods.
- ```
  def getName(self):
      return self.name

  def getHours(self):
      return self.hours

  def getGPoints(self):
      return self.qpoints

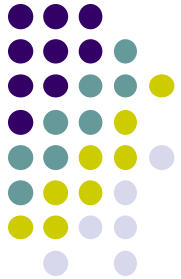
  def gpa(self):
      return self.qpoints/self.hours
  ```
- For example, to print a student's name you could write: `print aStudent.getName()`
  ```

# 例：TOP GPA



- How can we use these tools to find the student with the best GPA?
- We can use an algorithm similar to finding the max of *n* numbers! We could look through the list one by one, keeping track of the best student seen so far!

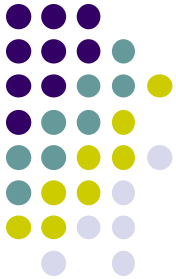
# 例：TOP GPA



```
# gpa.py
#    Program to find student with highest GPA

class Student:

    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)

    def getName(self):
        return self.name

    def getHours(self):
        return self.hours

    def getQPoints(self):
        return self.qpoints

    def gpa(self):
        return self.qpoints/self.hours

def makeStudent(infoStr):
    name, hours, qpoints = infoStr.split("\t")
    return Student(name, hours, qpoints)
```

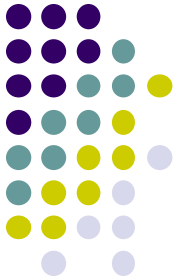
```
def main():
    filename = input("Enter name the grade file: ")
    infile = open(filename, 'r')
    best = makeStudent(infile.readline())
    for line in infile.readline():
        s = makeStudent(line)
        if s.gpa() > best.gpa():
            best = s
    infile.close()
    print("The best student is:", best.getName())
    print ("hours:", best.getHours())
    print("GPA:", best.gpa())

if __name__ == '__main__':
    main()
```

# OO回顾



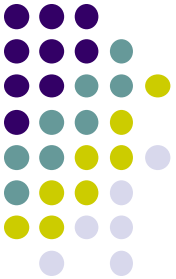
- 类是数据和操作的抽象，对象是类的实例
- 任何一个对象都有：
  - 唯一不变的id:　id(object) 函数查看
  - 类型：　用 type(object)　或者 object.__class__查看
  - 实例变量值
    - 可变的: lists,  dictionaries, instances
    - 不可变的：　numbers, strings, tuples,
    - 实例变量可以动态创建
- 实用的工具
  - dir(object) , help(object) 查看对象属性和方法
  - sys.getrefcount（object) 查看对象引用次数

# 第7章 面向对象设计



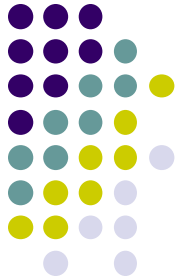
- 面向对象基本概念
- 类的定义
- 面向对象设计
- 编程案例

# 设计中的SoC



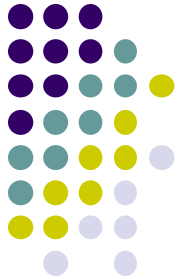
- 设计的本质是用黑箱及其接口描述系统.
  - 每个部件通过其接口提供一些服务,其他部件是这些服务的用户(客户).
  - 客户只需了解服务的接口,而实现细节对客户无关紧要.
  - 服务组件只管提供服务的实现,不管客户如何应用.

# 模块化设计与面向对象设计



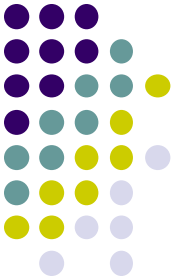
- 自顶向下设计:
  - 函数是黑箱.
  - 客户只要知道函数接口即能使用之.
  - 函数实现细节被封装在函数定义中.
- OOD:
  - 黑箱是对象.
  - 对象的能力由类定义.
  - 类对外提供的接口即方法.
  - 方法的实现对外部客户是不重要的.

# OOD设计



- OOD:对给定问题找出并定义一组有用的类的过程.
  - 确定有用的对象
    - 考虑问题描述中的名词(事物)
    - 这些事物有什么行为
  - 确定实例变量（数据）
  - 确定接口
    - 考虑问题描述中的动词(对象方法)
  - 复杂方法的自顶向下逐步求精
  - 反复设计
  - 尝试其他途径
  - 力求简单

# OO概念:封装



- 将数据以及相关操作打包在一起的过程. 封装的结果就是对象概念.
  - 数据的安全性：防止使用者直接操作对象数据而造成错误。
  - 使用的透明性：调用方法操作数据无需了解实现细节，维护也非常点单容易。
  - 代码的重用性：封装使得代码重用成为可能.
  - 界面的标准化：同类对象具有标准的操作界面。
- 对Python来说,封装只是一种程序设计思想和方法,语言本身并不提供强制性的要求.
  - 在程序的任何地方可以随意访问实例变量.
  - 严格意义的OO:内部细节在类外部是不可见的,只能通过方法(接口)访问.

# OO概念:继承

- 可以从现有的类出发,定义新类，以实现代码重用.
  - 超类与子类
  - 子类继承超类的变量和方法,并且可有自己的变量和方法.
  - 子类可以覆写（override）超类的方法。

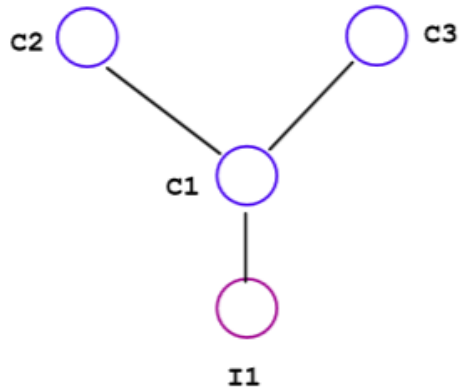
```
class Person:  #超类,2个属性，2个方法
    def __init__(self,name,birthYear):
        self.name = name
        self.year = birthYear
    def whatName(self):
        print "My name is",self.name
    def howOld(self,thisYear):
        age = thisYear-self.year
        if age > 0:
            print "My age in",y,"is",age
        else:
            print "I was born in",self.year
```

```
class Student(Person):  #子类，4个属性，4个方法
    def __init__(self,n,y,u,id):
        Person.__init__(self,n,y)  #调用超类的构造函数
        self.univ = u
        self.snum = id
    def getUniv(self):          #定义新的方法
        return self.univ
    def getNum(self):           #定义新的方法
        return self.snum
```

```
class Star(Person):   #子类
    def howOld(self,y): #覆写howOld方法
        print "You guess?"
```

```
Person.howOld(jack,2015) #强制调用超类的方法
```

# OO概念:多重继承




```
class C2: ...

class C3: ...

class C1(C2, C3): ...

I1 = C1()

I2 = C1()
```

multiple inheritance

```
class C2:
    def meth1(self): self.x=88
    def meth2(self): print(self.x)

class C3:
    def metha(self): self.x=99
    def methb(self): print(self.x)

class C1(C2,C3):
    pass
```

```
>>> i=C1()
>>> dir(i)
['__doc__', '__module__', 'meth1', 'meth2', 'metha', 'methb']
>>> i.meth1()
>>> dir(i)
['__doc__', '__module__', 'meth1', 'meth2', 'metha', 'methb', 'x']
>>> i.x
88
>>> i.metha()
>>> dir(i)
['__doc__', '__module__', 'meth1', 'meth2', 'metha', 'methb', 'x']
>>> i.x
99
```

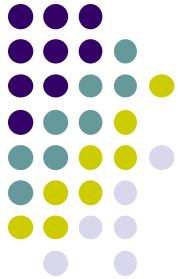
# only one x in i !

```
class C22:
    # __x is a private virable
    def meth1(self): self.__x=88
    def meth2(self): print(self.__x)

class C33:
    def metha(self): self.__x=99
    def methb(self): print(self.__x)

class C11(C22,C33):
    pass
```

```
>>> ii=C11()
>>> dir(ii)
['__doc__', '__module__', 'meth1', 'meth2', 'metha'
methb']
>>> ii.meth1()
>>> dir(ii)
['_C22__x', '__doc__', '__module__', 'meth1', 'meth
'metha', 'methb']
>>> ii._C22__x
88
>>> ii.metha()
>>> dir(ii)
['_C22__x', '_C33__x', '__doc__', '__module__', 'me
', 'meth2', 'metha', 'methb']
>>> ii._C33__x
99
```

# two __x in ii !

# OO概念:多态性



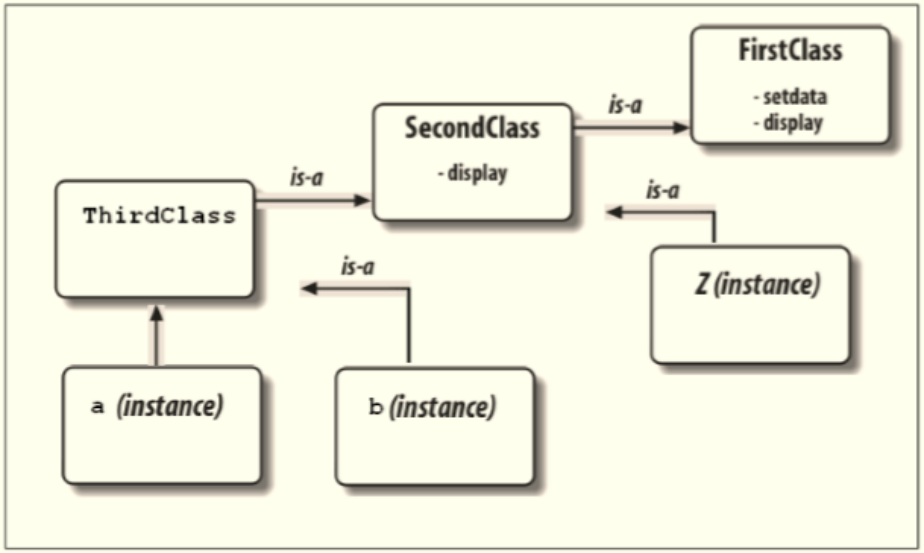
- 不同的对象支持相同的消息，但响应消息的行为不同。给对象发消息,具体做什么取决于该对象的类型.
  - 例如:obj.draw(win)对不同图形对象obj将画出不同的图形.
  - 下例中student和teacher都有getNum方法，但返回值却不同

```
class Rectangle(_BBox):

    def __init__(self, p1, p2):
        _BBox.__init__(self, p1, p2)

    def _draw(self, canvas, options):
        p1 = self.p1
        p2 = self.p2
        x1,y1 = canvas.toScreen(p1.x,p1.y)
        x2,y2 = canvas.toScreen(p2.x,p2.y)
        return canvas.create_rectangle(x1,y1,x2,y2,options)

class Line(_BBox):

    def __init__(self, p1, p2):
        _BBox.__init__(self, p1, p2, ["arrow","fill","width"])
        self.setFill(DEFAULT_CONFIG['outline'])
        self.setOutline = self.setFill

    def _draw(self, canvas, options):
        p1 = self.p1
        p2 = self.p2
        x1,y1 = canvas.toScreen(p1.x,p1.y)
        x2,y2 = canvas.toScreen(p2.x,p2.y)
        return canvas.create_line(x1,y1,x2,y2,options)
```

```
class Student(Person):  #子类，4个属性，4个方法
    def __init__(self,n,y,u,id):
        Person.__init__(self,n,y)  #调用超类的构造函数
        self.univ = u
        self.snum = id
    def getUniv(self):          #定义新的方法
        return self.univ
    def getNum(self):           #返回学生学号
        return self.snum

class Teacher(Person):  #子类，2个属性，4个方法
    def setNum(self,n):
        self.snum = n
    def getNum(self):           #返回班级学生数
        return self.snum
```

# OO概念:继承与多态性

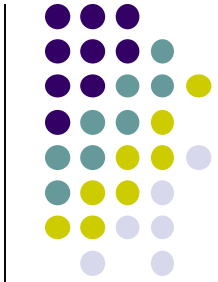
Upward search:
Instances->subclasses->superclasses, stopping at the first appearance




```
class FirstClass:
    def setdata(self,value):
        self.data=value
    def display(self):
        print(self.data)

class SecondClass(FirstClass):
    def display(self):
        print 'Current Value= %s' % (self.data)

class ThirdClass(SecondClass):
    def __init__(self,value):
        self.data=value

    def __add__(self,other):  #  override __add__ or '+'
        return ThirdClass(self.data+other)

    def __str__(self):    # override 'print self' or 'str()'
        return '[ThirdClass: %s]' % self.data

    def mul(self,other):
        self.data *=other
```

```
>>> a=ThirdClass('abc')
>>> a.display()
Current Value= abc
>>> b=a+'xyz'
>>> b.display()
Current Value= abcxyz
>>> print a
[ThirdClass: abc]
>>> print b
[ThirdClass: abcxyz]
>>> a.mul(3)
>>> print a
[ThirdClass: abcabcabc]
```

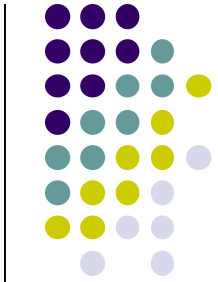
# 类与对象属性/变量



- 类属性
- 实例/对象属性
- 实例/对象的私有属性
- 方法的局部变量

```
class Person:
    '''This is a Person class, object factory'''
    intelligent = True
    def __init__(self,name,birthyear):
        self.name = name
        self.ID = 0
        self.__year = birthyear
    def whatName(self):
        return self.name
    def howOld(self,currentyear):
        age = currentyear - self.__year
        if age > 0:
            return age
        else:
            return -1
    def setID(self, id):
         self.ID = id
    def getID():
         return self.ID
```

# 类的special hooks



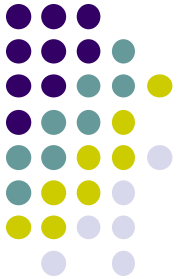
- 以双写下划线开头和结尾的类的方法(__xxx__)
- 这些方法当执行内嵌操作时自动执行：
  - __init__() : 创建实例时
  - __str__() ： print
  - __add__ ： +
  - __sub__ ： -
  - __mul__: *
  - __floordiv__: /
  - __lt__ ： <
  - __eq__ ： ==
  - ……
  - dir([])可以查看

# 类的special hooks

- 以双写下划线开头和结尾的类的方法(__xxx__)
- 这些方法当执行内嵌操作时自动执行：
  - __init__() : 创建实例时
  - __str__() ： print
  - __add__ ： +
  - __sub__ ： -
  - __mul__: *
  - __lt__ ： <
  - __eq__ ： ==
  - ……
  - dir([])可以查看

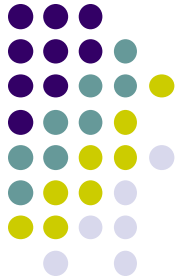
```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)

object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
```

# 第7章 面向对象设计



- 面向对象基本概念
- 类的定义
- 面向对象设计
- 编程案例

# 编程实例:炮弹模拟

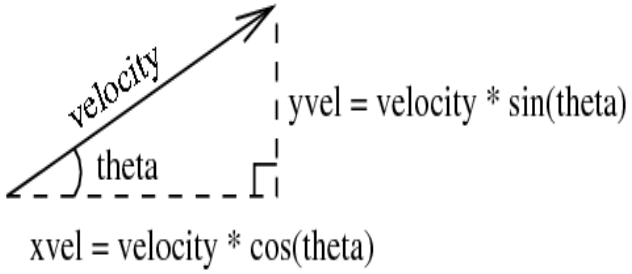


- 程序规格
  - 输入:发射角,初速度,初始高度
  - 输出:射程

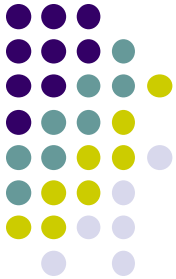
  注:不用微积分,只用一些基本知识来算法化解决.
- 算法
  1. 输入模拟参数:角度,速度,高度,计算位置变化的时间间隔
  2. 计算炮弹初始位置xpos,ypos
  3. 计算炮弹初始水平和垂直速度xvel, yvel
  4. 当炮弹还在飞行,循环:
     1. 更新一个时间间隔之后的xpos, ypos, yvel
  5. 输出xpos

# 编程实例:抛物体模拟(续)

```
def main():
    angle = input("...(in degrees)")
    vel = input("...(in meters/sec")
    h0 = input("...(in meters)")
    time = input("...(in seconds)")
    xpos, ypos = 0, h0
    theta = angel * math.pi / 180.0
    xvel = vel * math.cos(theta)
    yvel = vel * math.sin(theta)
    while ypos >= 0.0:
        更新
    print "Distance: %0.1f meters." % (xpos)
```



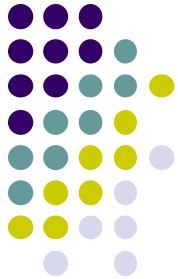
# 编程实例:抛物体模拟(续)



- 算法核心部分:不断更新各变量的值

```
xpos = xpos + xvel * time
yvel_new = yvel – 9.8 * time
ypos = ypos + time * (yvel + yvel_new) / 2
yvel = yvel_new
```

# 编程实例:抛物体模拟(续)

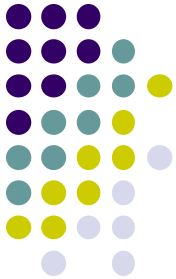


- 模块化设计:

```
def main():
    angle, vel, h0, time = getInput()
    xpos, ypos = 0, h0
    xvel, yvel = getXYComponents(vel, angle)
    while ypos >= 0.0:
        xpos,ypos,yvel = updatePos(time,xpos,ypos,xvel,yvel)
    print "Distance: %0.1f meters." % (xpos)
```

  - 问: 变量theta和vyel_new呢?
    - 这正是自顶向下逐步求精的SoC带来的好处
  - 但updateData()似乎不太好?

# 编程实例:抛物体模拟(续)



- 函数updateData()的弊端
  - 参数过多: 5个参数, 3个返回值.
  - 函数参量过多通常意味着有更好的组织方式
- OO设计:设计一个抛物体类Projectile.从而:

```
def main():
    angle, vel, h0, time = getInput()
    cball = Porjectile(angle,vel,h0)
    while cball.getY() >= 0.0:
        cball.update(time)
    print "Distance: %0.1f meters." % (cball.getX())
```

  - 隐藏了对炮弹的描述信息:**xpos, ypos, xvel, yvel**

# 编程实例:Porjectile类

- 构造器
  - 实例变量:xpos,ypos,xvel,yvel
  - 构造器需要三个初值来为实例变量初始化:

    ```
    cball = Projectile(angle,vel,h0)
    ```
  - 因此得到:

    ```
    def __init__(self,a,v,h):
        self.xpos = 0
        self.ypos = h
        theta = math.pi * a / 180
        self.xvel = v * math.cos(theta)
        self.yvel = v * math.sin(theta)
    ```

# 编程实例:Porjectile类(续)

- 读取实例变量的方法

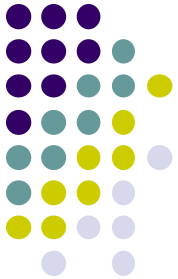
```
def getX(self):
    return self.xpos
def getY(self):
    return self.ypos
```

- 更新实例变量的方法:

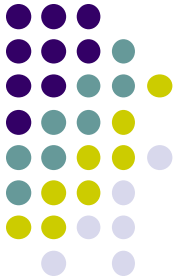
```
def update(self,time):
    self.xpos = self.xpos + self.xvel * time
    yvelnew = self.yvel - 9.8 * time
    yvelavg = (self.yvel + yvelnew) / 2
    self.ypos = self.ypos + yvelavg * time
    self.yvel = yvelnew
```

- **OO版抛物体模拟程序:cball3.py**

# assign6



- 注意从ftp://public.sjtu.edu.cn/ct/assignments 下载详细作业6文档
- 上机时间：12月3日 8：00~9：40
- 上机地点：电院4号楼313机房
- 截止日期：12月3日 24：00



# End