人工智能实践教程
从Python入门到机器学习

第一部分Python核心编程技术
第 2 章 Python 面向对象

- **所有代码及ppt均可以由以下链接下载**
- **https://github.com/shao1chuan/pythonbook**
- **https://gitee.com/shao1chuan/pythonbook**

面向对象基本特性

目录

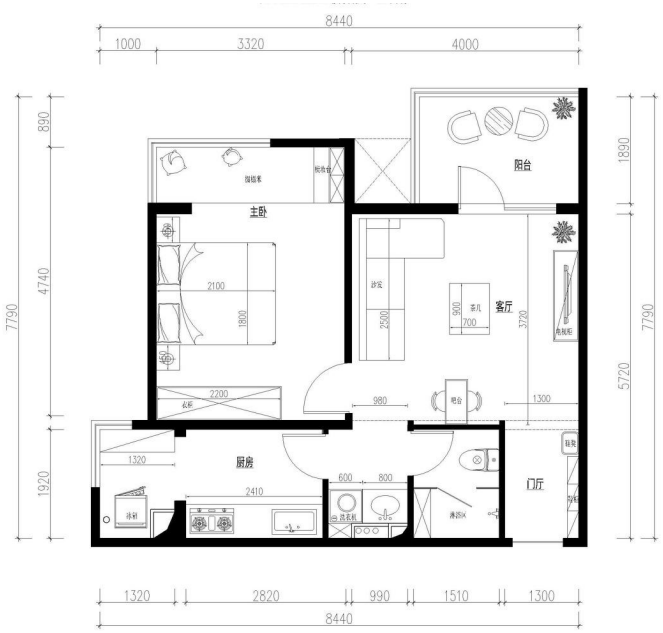
Contents

01 对象和类

标题描述

对象和类

图纸与真实的房子哪个是类，哪个是对象？



对象和类

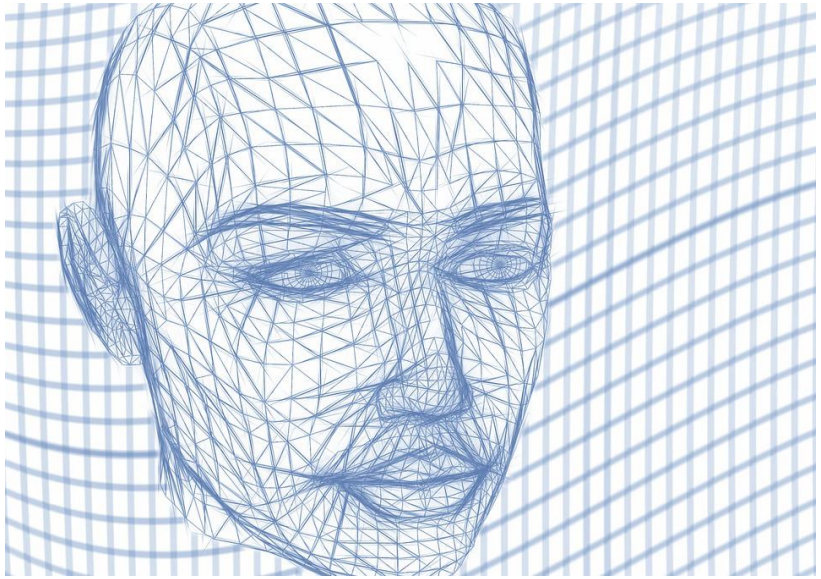
类（Class）是现实或思维世界中的实体在计算机中的反映，它将数据以及这些数据上的操作封装在一起。

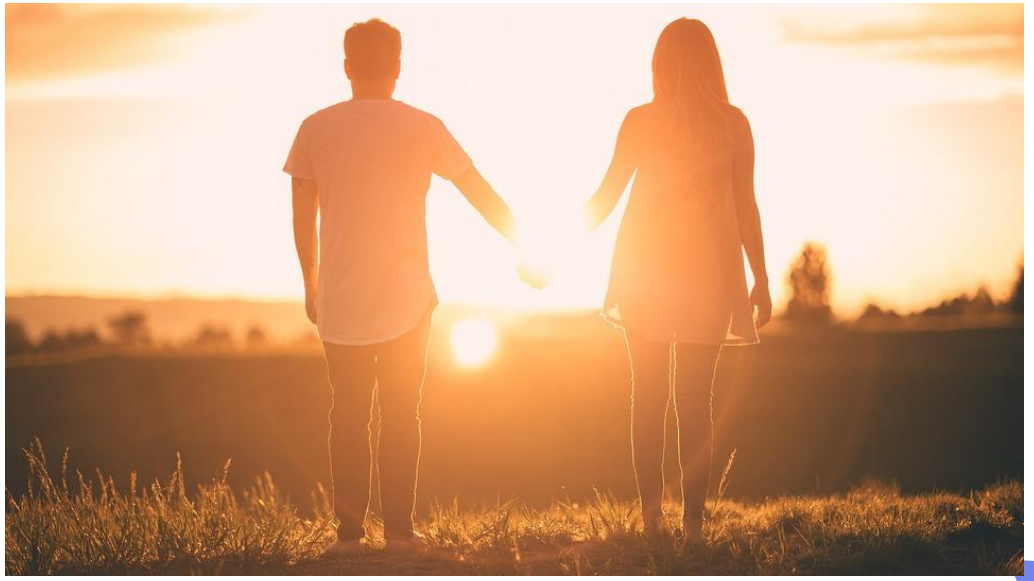
对象(Object)是具有类类型的变量。类和对象是面向对象编程技术中的最基本的概念。

类和对象 的区别就是 鱼和三文鱼 的区别; 就是 猫和蓝猫 的区别。

对象和类

哪个是类，哪个是对象?



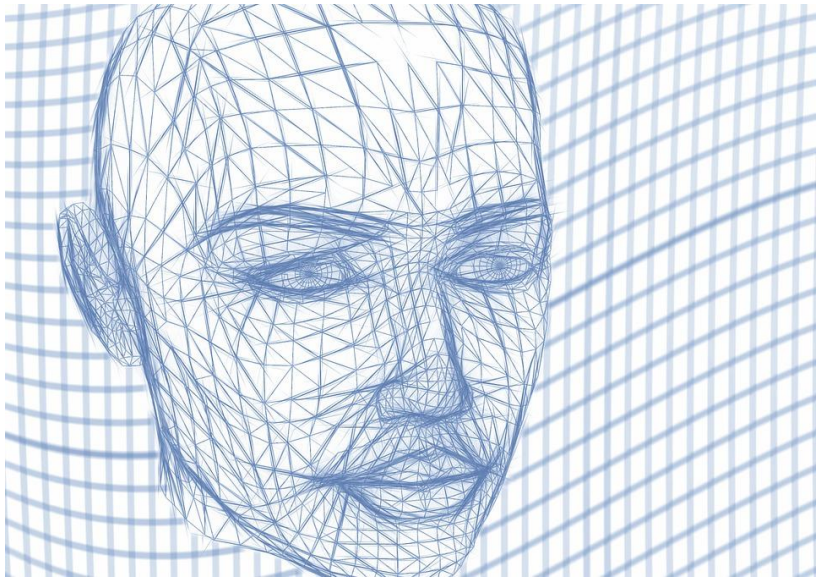


对象和类

哪个是类，哪个是对象?

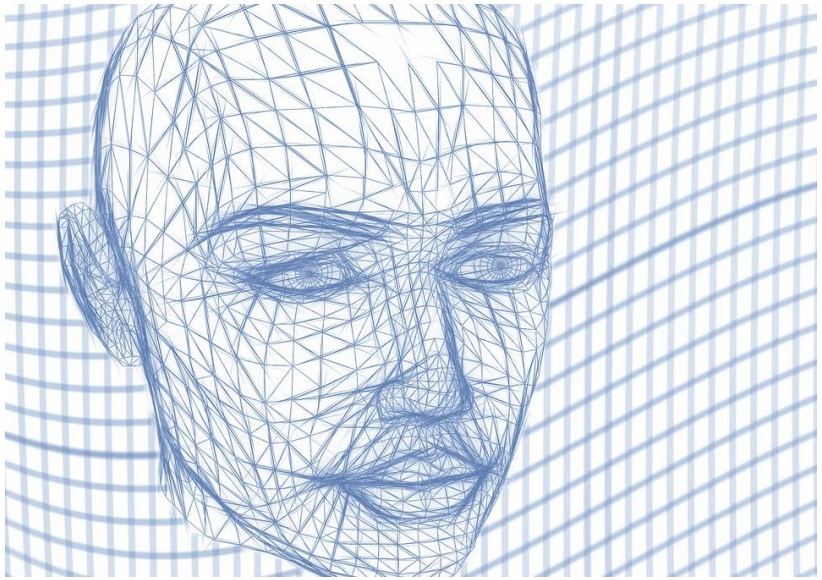
对象和类

人类的属性: 你能说出几个?



对象和类

人类的行为/**方法**: 你能说出几个?



对象和类

1. 如何定义类？　　class 类():　pass
2. 如何将类转换成对象？

实例化是指在面向对象的编程中，把用类创建对象的过程称为实例化。是将一个抽象的概念类，具体到该类实物的过程。实例化过程中一般由类名 对象名 = 类名（参数1，参数2...参数n）构成。

对象和类

类（Class）是是创建实例的模板

对象(Object)是一个一个具体的实例

类和对象 的区别就是 鱼和三文鱼 的区别; 就是 猫和蓝猫 的区别。

02 封装特性

面向对象的三大特性是指：封装、继承和多态

封装特性

封装，顾名思义就是将内容封装到某个地方，以后再去调用被封装在某处的内容。

所以，在使用面向对象的封装特性时，需要：

1). 将内容封装到某处

2). 从某处调用被封装的内容

1). 通过对象直接调用被封装的内容： 对象.属性名

2). 通过self间接调用被封装的内容： self.属性名

3). 通过self间接调用被封装的内容: self.方法名()

对象和类

构造方法__init__与其他普通方法不同的地方在于，当一个对象被创建后，会立即调用构造方法。自动执行构造方法里面的内容。

封装特性

对于面向对象的封装来说，其实就是使用构造方法将内容封装到 对象 中，然后通过对象直接或者self间接获取被封装的内容。

对象和类

创建一个类People,拥有的**属性**为姓名， 性别和年龄， 拥有的**方法**为购物,玩游戏,学习;实例化对象,执行相应的方法。 显示如下:

小明,18岁,男,去西安赛格购物广场购物

小王,22岁,男,去西安赛格购物广场购物

小红,10岁,女,在西部开源学习

提示:

属性:name,age,gender

方法:shopping(), playGame(), learning()

03 继承特性

面向对象的三大特性是指：封装、继承和多态

1. 继承

2. 多继承

3. 私有属性与私有方法

继承特性

在现实生活中,继承一般指的是子女继承父辈的财产,如下图:

继承特性

在现实生活中,继承一般指的是子女继承父辈的财产,如下图:

继承特性

搞不好,结果如下..

继承特性

继承描述的是事物之间的所属关系,当我们定义一个class的时候,可以从某个现有的class继承,新的class称为**子类、扩展类**(Subclass),而被继承的class称为**基类、父类或超类**(Baseclass、Superclass)。

继承特性

问题一: 如何让实现继承?

子类在继承的时候,在定义类时,小括号()中为父类的名字

问题二: 继承的工作机制是什么?

父类的属性、方法,会被继承给子类。 举例如下: 如果**子类没有**定义__init__方法,**父类有**,那么在子类继承父类的时候这个方法就被继承了,所以只要创建对象,就默认执行了那个继承过来的__init__方法。

继承特性

重写父类方法: 就是子类中,有一个和父类相同名字的方法,在子类中的方法会覆盖掉父类中同名的方法。

继承特性

调用父类的方法:

1. 父类名.父类的方法名()
2. super(): py2.2+的功能

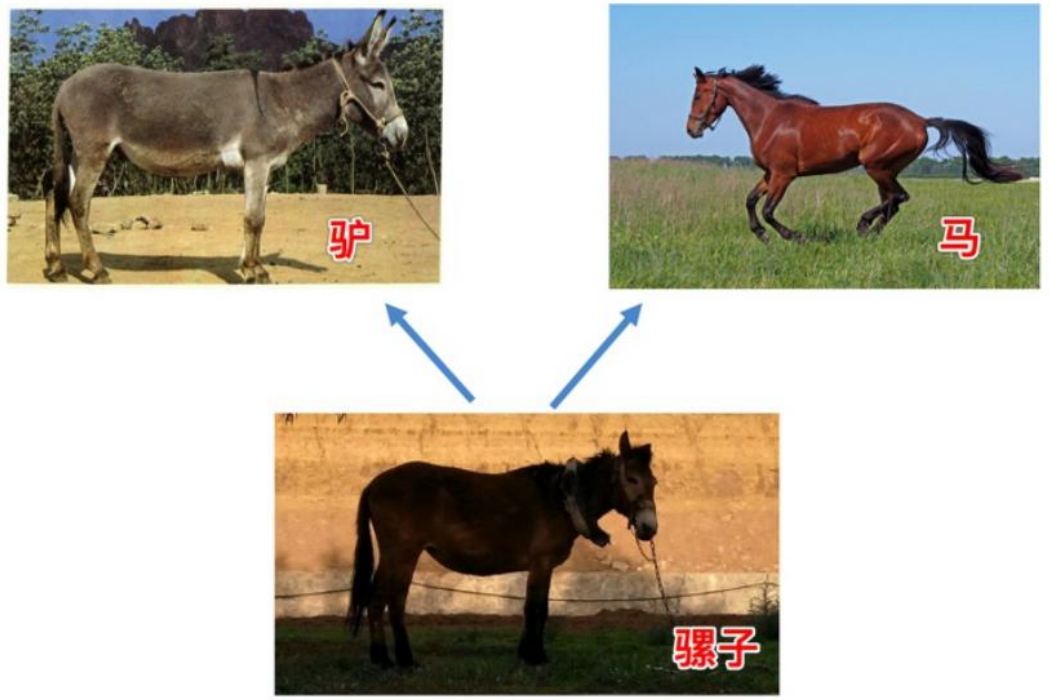
继承特性

调用父类的方法:

1. 父类名.父类的方法名()
2. super(): py2.2+的功能

多继承

多继承,即子类有多个父类,并且具有它们的特征



多继承

新式类与经典类

在Python 2及以前的版本中，由任意内置类型派生出的类，都属于“新式类”，都会获得所有“新式类”的特性；反之，即不由任意内置类型派生出的类，则称之为“经典类”。

```
class 类名(object):
    pass
```

新式类

```
class 类名:
    pass
```

经典类

多继承

“新式类”和“经典类”的区分在Python 3之后就已经不存在，在**Python 3.x之后**的版本，因为所有的类都派生自内置类型object(即使没有显示的继承object类型)，即**所有的类都是“新式类”**。

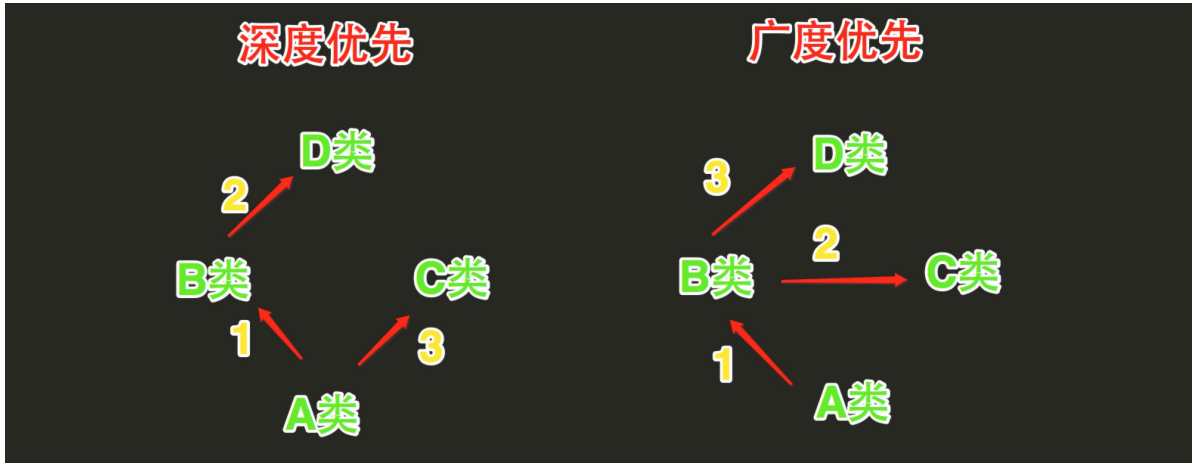
多继承

新式类与经典类

最明显的区别在于继承搜索的顺序不同，即：

经典类多继承搜索顺序(深度优先算法):先深入继承树左侧查找，然后再返回，开始查找右侧。

新式类多继承搜索顺序(广度优先算法):先在水平方向查找，然后再向上查找，



多继承

新式类与经典类

私有属性与私有方法

默认情况下,属性在 Python 中都是“public”， 大多数 OO 语言提供“访问控制符”来限定成员函数的访问。

在 Python 中,实例的变量名如果以 __ 开头,就变成了一个私有变量/属性(private),实例的函数名如果以 __ 开头,就变成了一个私有函数/方法(private)只有内部可以访问,外部不能访问。

私有属性与私有方法

问题: 私有属性一定不能从外部访问吗?

python2版本不能直接访问 __属性名 是因为 Python 解释器对外把 __属性名 改成了 _类名__属性名 ,所以,仍然可以通过 _类名__属性名 来访问 __属性名 。

因为不同版本的 Python 解释器可能会把 __属性名 改成不同的变量名。

私有属性与私有方法

优势

1. 确保了外部代码不能随意修改对象内部的状态,这样通过访问限制的保护,代码更加健壮。

2. 如果又要允许外部代码修改属性怎么办?可以给类增加专门设置属性方法。 为什么大费周折?因为在方法中,可以对参数做检查,避免传入无效的参数。

04 多态特性

标题描述

多态特性

多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。通俗来说： 同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

多态特性

多态的好处就是，当我们需要传入更多的子类，只需要继承父类就可以了，而方法既可以直接不重写（即使用父类的），也可以重写一个特有的。这就是多态的意思。调用方只管调用，不管细节，而当我们新增一种的子类时，只要确保新方法编写正确，而不用管原来的代码。这就是著名的“开闭”原则：

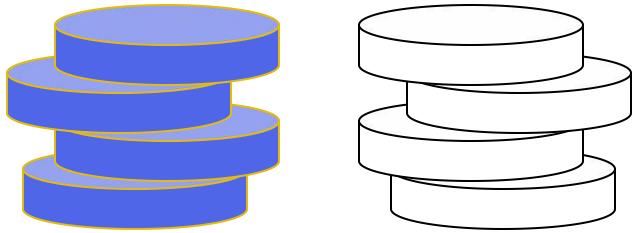
对扩展开放（Open for extension）：允许子类重写方法函数

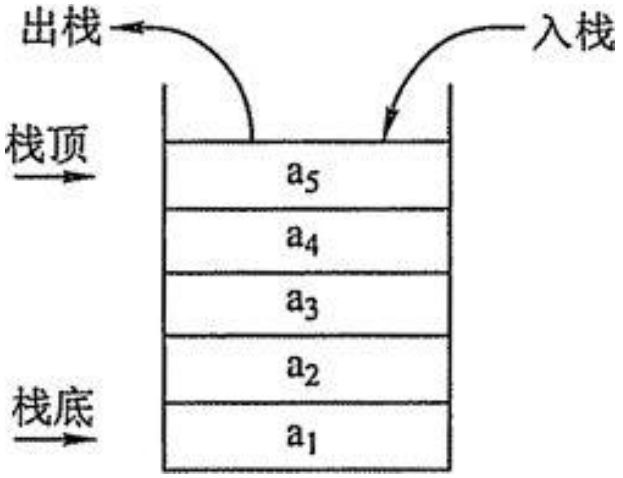
对修改封闭（Closed for modification）：不重写，直接继承父类方法函数

05 项目案例: 栈与队列的封装

标题描述

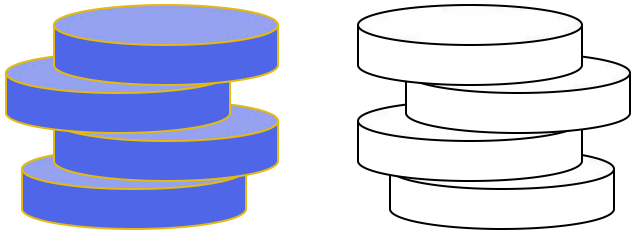
项目案例: 栈与队列的封装





项目案例: 栈与队列的封装

栈是限制在一端进行插入操作和删除操作的线性表（俗称堆栈），允许进行操作的一端称为“栈顶”，另一固定端称为“栈底”，当栈中没有元素时称为“空栈”。向一个栈内插入元素称为是进栈，push；从一个栈删除元素称为是出栈，pop。特点：后进先出（LIFO）。



项目案例: 栈与队列的封装

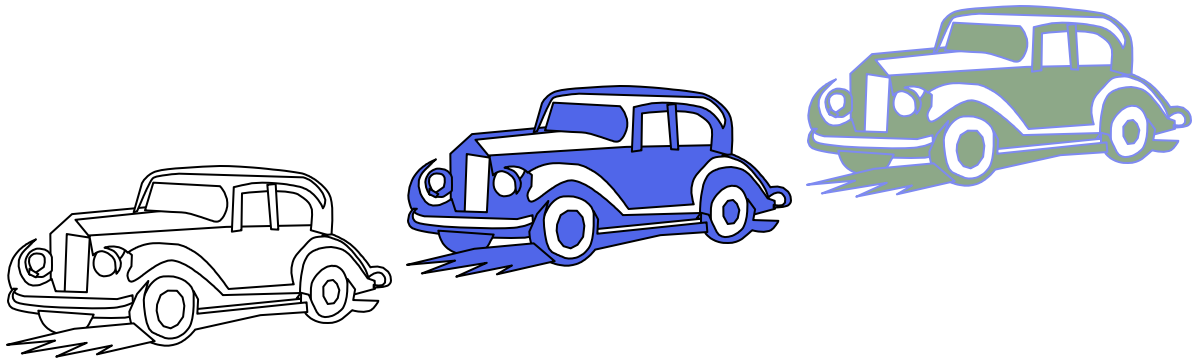
Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	“error”	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

项目案例: 栈与队列的封装

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
```

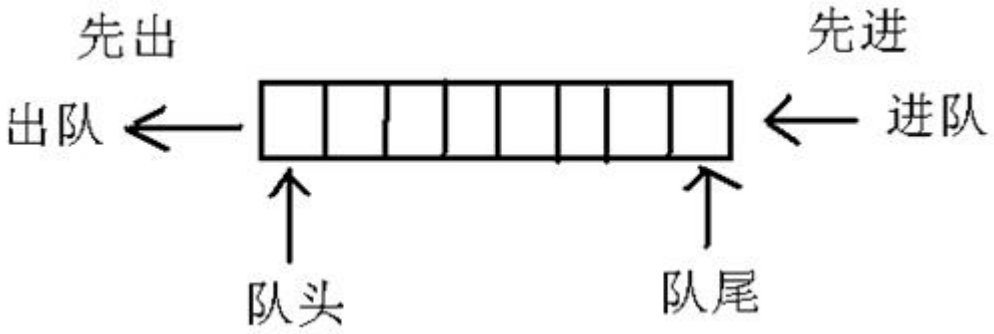
```
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

项目案例: 栈与队列的封装



项目案例: 栈与队列的封装

队列是限制在一端进行插入操作和另一端删除操作的线性表，允许进行插入操作的一端称为“队尾”，允许进行删除操作的一端称为“队头”，，当队列中没有元素时称为“空队”。特点：先进先出（FIFO）。



项目案例: 栈与队列的封装

Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

项目案例: 栈与队列的封装

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
```

```
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
27
28   def dequeue(self):
29     """Remove and return the first element of the queue (i.e., FIFO).
30
31     Raise Empty exception if the queue is empty.
32     """
33     if self.is_empty():
34       raise Empty('Queue is empty')
35     answer = self._data[self._front]
36     self._data[self._front] = None          # help garbage collection
37     self._front = (self._front + 1) % len(self._data)
38     self._size -= 1
39     return answer
```

项目案例: 栈与队列的封装

```
40    def enqueue(self, e):
41      """Add an element to the back of queue."""
42      if self._size == len(self._data):
43        self._resize(2 * len(self.data))          # double the array size
44      avail = (self._front + self._size) % len(self._data)
45      self._data[avail] = e
46      self._size += 1
47
48    def _resize(self, cap):                       # we assume cap >= len(self)
49      """Resize to a new list of capacity >= len(self)."""
50      old = self._data                            # keep track of existing list
51      self._data = [None] * cap                   # allocate list with new capacity
52      walk = self._front
53      for k in range(self._size):                 # only consider existing elements
54        self._data[k] = old[walk]                 # intentionally shift indices
55        walk = (1 + walk) % len(old)              # use old size as modulus
56      self._front = 0                             # front has been realigned
```

06 项目案例: 乌龟吃鱼游戏

标题描述

乌龟吃鱼游戏

乌龟吃鱼游戏

西部开源

游戏规则：

1). 假设游戏场景为范围（x,y）为0<=x<=10,0<=y<=10

2). 游戏生成1只乌龟和10条鱼, 它们的移动方向均随机

3). 乌龟的最大移动能力为2（它可以随机选择1还是2移动），

鱼儿的最大移动能力是1当移动到场景边缘，自动向反方向移动

4). 乌龟初始化体力为100（上限）, 乌龟每移动一次，体力消耗1

当乌龟和鱼坐标重叠，乌龟吃掉鱼，乌龟体力增加20, 鱼暂不计算体力

5). 当乌龟体力值为0（挂掉）或者鱼儿的数量为0游戏结束

乌龟吃鱼游戏

乌龟吃鱼游戏

Pygame是跨平台Python模块，专为电子游戏设计，包含图像、声音。允许实时电子游戏研发而无需被低级语言（如机器语言和汇编语言）束缚。

乌龟吃鱼游戏

一个游戏循环（也可以称为主循环）就做下面这三件事：

处理事件

更新游戏状态

绘制游戏状态到屏幕上

乌龟吃鱼游戏

Pygame常用模块

```
import pygame
import sys

pygame.init()  # 初始化pygame
size = width, height = 320, 240  # 设置窗口大小
screen = pygame.display.set_mode(size)  # 显示窗口

while True:  # 死循环确保窗口一直显示
    for event in pygame.event.get():  # 遍历所有事件
        if event.type == pygame.QUIT:  # 如果单击关闭窗口，则退出
            sys.exit()

pygame.quit()  # 退出pygame
```

乌龟吃鱼游戏

乌龟吃鱼游戏

图片素材处理

```
import os
import os.path
from PIL import Image
img = Image.open('turtle.png')
# 图片缩放尺度大了就会失真，PIL带ANTIALIAS滤镜缩放结果，
out = img.resize((80, 80),Image.ANTIALIAS)
#resize image with high-quality
out.save('turtle1.png', 'png')
```

乌龟吃鱼游戏

乌龟类

```
# 乌龟类
class Turtle:
    def __init__(self):
        self.power = 100  # 体力
        # 乌龟坐标
        self.x = random.randint(0, width - 10)
        self.y = random.randint(0, height - 10)

    # 乌龟移动的方法：移动方向均随机 第四条
    def move(self, new_x, new_y):
        # 判断移动后是否超出边界
        if new_x < 0:
            self.x = 0 - new_x
        elif new_x > width:
            # self.x=width-(new_x-width)
            self.x = 0
        else:
            # 不越界则移动乌龟的位置
            self.x = new_x
        if new_y < 0:
            self.y = 0 - new_y
        elif new_y > height:
            # self.y=height-(new_y-height)
            self.y = 0
        else:
            # 不越界则移动乌龟的位置
            self.y = new_y
        self.power -= 1  # 乌龟每移动一次，体力消耗1

    def eat(self):
        self.power += 20  # 乌龟吃掉鱼，乌龟体力增加20
        if self.power > 100:
            self.power = 100  # 乌龟体力100（上限）
```

乌龟吃鱼游戏

鱼类

```
# 鱼类
class Fish:
    def __init__(self):
        # 鱼坐标
        self.x = random.randint(0, width - 20)
        self.y = random.randint(0, height - 20)

    def move(self):
        new_x = self.x + random.choice([-10])
        if new_x < 0:
            self.x = width
        else:
            self.x = new_x
```

乌龟吃鱼游戏

实例化游戏对象

```
tur = Turtle()  # 生成1只乌龟
fish = []  # 生成10条鱼
for item in range(10):
    newfish = Fish()
    fish.append(newfish)  # 把生成的鱼放到鱼缸里
```

乌龟吃鱼游戏

键盘控制

```
# pygame有一个事件循环，不断检查用户在做什么。事件循环中，如何让循环中断下来
#（pygame形成的窗口中右边的插号在未定义前是不起作用的）
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        if event.type == KEYDOWN:
            # 通过上下左右方向键控制乌龟的动向
            if event.key == pygame.K_LEFT:
                tur.move(tur.x - 10, tur.y)
            if event.key == pygame.K_RIGHT:
                tur.move(tur.x + 10, tur.y)
            if event.key == pygame.K_UP:
                tur.move(tur.x, tur.y - 10)
            if event.key == pygame.K_DOWN:
                tur.move(tur.x, tur.y + 10)
```

乌龟吃鱼游戏

游戏逻辑处理

```
screen.blit(background, (0, 0))  # 绘制背景图片
screen.blit(score, (500, 20))  # 绘制分数
# 绘制鱼
for item in fish:
    screen.blit(fishImg, (item.x, item.y))
    # pygame.time.delay(100)
    item.move()  # 鱼移动
screen.blit(wuguiImg, (tur.x, tur.y))  # 绘制乌龟
# 判断游戏是否结束：当乌龟体力值为0（挂掉）或者鱼儿的数量为0游戏结束
if tur.power < 0 or len(fish) == 0:
    print("Game Over ~")
    sys.exit()
for item in fish:
    print("鱼",item.x,item.y,y_width,y_height)
    print("乌龟",tur.x,tur.y,w_width,w_height)
    if ((tur.x < item.x + y_width) and (tur.x + w_width > item.x) and
        (tur.y < item.y + y_height) and (w_height + tur.y > item.y)) :
        tur.eat()  # 乌龟吃鱼的方法
        fish.remove(item)  # 鱼死掉
        # 吃鱼音乐
        # eatsound.play()
        count = count + 1  # 累加
        score = font.render("score %d" % count, True, (255, 255, 255))
        print("死了一只鱼")
        print("乌龟最新体力值为 %d" % tur.power)

pygame.display.update()  # 更新到游戏窗口
fpsClock.tick(10)  # 通过每帧调用一次fpsClock.tick(10)，这个程序就永远不会以超过每秒10帧的速度运行
```