# Table of Contents




In [21]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

# 1 面向过程与面向对象

- 程序包括
  - 数据:数据类型,数据结构
  - 处理过程:算法
- 两种程序设计思想
  - 面向过程:以操作为中心

- 面向对象:以数据为中心

## 1.1 面向过程编程

- 把计算机程序视为一系列的命令集合，即一组函数的顺序执行。（C语言）
- 面向过程观点：数据与操作分离， 程序就是对数据进行一系列的操作
  - 先表示数据:常量,变量
  - 再来处理数据
- 特点:数据与操作分离
  - 数据是被动的,操作是主动的

## 1.2 面向对象编程

- 把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息。
- 处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。
- Python中的一切内容皆为对象。
- 数据与操作不可分离。
- 将特定数据值与特定操作捆绑形成一种新型的数据：对象

## 1.3 对象

在Python中，一切都是对象，字符串、列表。函数都是对象。

- 对象拥有特定数据
- 对象能对其数据进行特定操作

In [2]:
```
a=[1,2,3,5]
a.append(1)
a
```

Out[2]:
```
[1, 2, 3, 5, 1]
```

# 2 什么是类

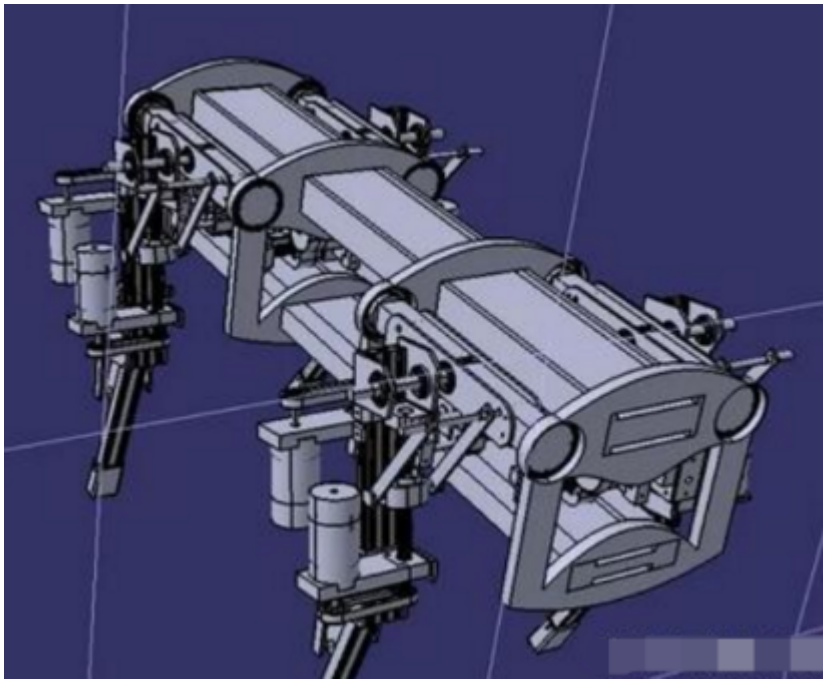
类是封装对象的属性和行为的载体。

- 类是类型概念的发展
  - 对象是广义的"数据值"
  - 对象所属的数据类型就是"类"
  - 用于描述复杂数据的静态和动态行为
- 类(class):描述相似对象的共性。
  - 比如操作方法(method)
  - 数据

## 2.1 定义类

```
class <类名>:
    "类的帮助信息"
    statement            #类体
```

- 类名：一般使用大写字母开头，常使用“驼峰式命名法”。
- 类的帮助信息：用于指定类的文档字符串，定义该字符串后。
- 类体：主要由类变量(或类成员)、方法和属性等定义语句组成。可以用pass语句代替。



In [12]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"                    #类的属性

    def __init__(self,name):         #__init__()方法
        print("%s(机器狗)初始状态正常"%name)

    def run():
        print("奔跑组件激活成功")
```

In [11]:
```
print(Dogs.__doc__)
```

```
这是一只机器狗
```

In [13]:
```
Dogs.colour
```

Out[13]: '黄色'

In [4]:
```
Dogs.run()
```

```
奔跑组件激活成功
```

## 2.2 类的实例

定义完类之后，并不会创建该类的任何实例，如何实例化该类的对象呢?

In [17]:
```
a=list()
a.append(1)
a
```

Out[17]: [1]

In [14]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"        #定义类的属性
```

In [39]:
```
dog_01=Dogs()  #此时创建了类的实例

dog_01.colour
```

Out[39]: '黄色'

# 3 类的属性

## 3.1 类属性的创建

类的属性就是指定义在类中，并且在函数体外的属性，这些属性在该类的所有实例都可以共享。

In [19]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"      #类的属性
    size="中等体型"    #类的属性
    def __init__(self,name):
        print("%s(机器狗)初始状态正常"%name)
```

In [96]:
```
dog_01=Dogs("阿花")

dog_01.colour

dog_01.size
```

阿花(机器狗)初始状态正常

Out[96]: '黄色'

Out[96]: '中等体型'

这些属性在该类的所有实例都可以共享。

In [97]:
```
dog_02=Dogs("阿旺")

dog_02.colour

dog_02.size
```

阿旺(机器狗)初始状态正常

Out[97]: '黄色'

Out[97]: '中等体型'

In [22]:
```
dog_03=Dogs("二哈")

dog_03.colour

dog_03.size
```

二哈(机器狗)初始状态正常

Out[22]: '黄色'

Out[22]: '中等体型'

## 3.2 类属性的修改

In [29]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"      #类的属性
    size="中等体型"    #类的属性
    def __init__(self,name):
        print("%s(机器狗)初始状态正常"%name)

Dogs.colour="白色"
```

In [30]:
```
Dogs.colour
```

Out[30]: '白色'

## 3.3 类属性的添加

**和属性修改的方法一样，修改一个类中不存在的属性的话，就会变成添加操作：**

In [31]:
```
Dogs.eyes=2
```

In [33]:
```
Dogs.eyes
```

Out[33]: 2

## 3.4 类属性的删除

删除类属性的操作也是用del方法就可以：

In [34]:
```
del Dogs.eyes
```

In [35]:
```
Dogs.eyes
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-35-cdbc50a41042> in <module>
----> 1 Dogs.eyes

AttributeError: type object 'Dogs' has no attribute 'eyes'
```

# 4 创建__init__()方法

__init__() 方法是一种特殊的方法，每当创建一个类的新实例的时候，Python都会自动执行它。

- __init__() 方法必须包含一个self参数,并且必须是第一个参数。
- self参数是一个指向实例本身的引用，用于访问类中的属性和方法。
- 在方法调用时，会自动通过self传递实际参数。
- __init__() 方法前后两个下划线，是一种约定习惯写法，用来区分Python默认方法和普通方法。

In [46]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"
    def __init__(self):      #定义类的__init__()方法
        print("初始状态正常")
    def run(self):
        print("机器狗在跑")
```

In [48]:
```
dogs_88=Dogs()
dogs_88.run()
```

```
初始状态正常
机器狗在跑
```

In [38]:
```
Dogs().colour
```

```
初始状态正常
```

Out[38]: '黄色'

**定义类Dogs实例的时候，即使没有为 __init__() 方法指定参数，但是 __init__() 方法依然会被自动执行：**

In [39]:
```
dogs_01=Dogs()    #你会发现，当你创建Dog的实例的时候，会自动运行__init__()方法缩进下的代码
```

```
初始状态正常
```

**实例是否属于某个类的判断：可以使用内置方法isinstance()来测试一个对象是否为某个类的实例。**

In [51]:
```
isinstance(Dogs_01,Dogs)
```

Out[51]:
```
True
```

# 5 动态添加类的方法

In [49]:
```
class Dogs:
    """这是一只机器狗"""
    colour="黄色"
    def __init__(self):       #定义类的__init__()方法
        print("初始状态正常")
```

定义完类Dogs之后，如果想要将一个自定义函数添加到Dogs中，作为类的方法，怎么办？

In [51]:
```
def fly(self):      #必须要有self
    print("展开翅膀，起飞")
```

用types库的方法“MethodType”将自定义函数fly作为实例方法添加到Dogs中，重新以实例方法名字fly_add命名。

In [52]:
```
from types import MethodType

Dogs.fly_add=MethodType(fly,Dogs)
```

In [53]:
```
Dogs.fly_add()
```

```
展开翅膀，起飞
```

当然，如果你创建Dogs的实例，实例也是可以访问到fly_add方法的：

In [54]:
```
dog_05=Dogs()
dog_05.fly_add()
```

```
初始状态正常
展开翅膀，起飞
```

# 6 实例方法和属性

类的成员主要由实例方法和数据成员组成。

- 实例方法，在类的实例上操作的函数。
- 实例方法的第一个参数必须是self

# 6.1 实例方法

## 6.1.1 创建实例方法

和 __init__() 方法一样，实例方法的第一个参数必须是self,语法:

```
def 实例方法名(self,其他参数1,其他参数2...):
    方法体
```

In [55]:
```
class Dogs:
    def __init__(self,name):       #__init__()方法,注意__init__()方法不单只可以定义self参数
        print("%s(机器狗)初始状态正常"%name)
    def run(self,speed):           #这里定义了实例方法run()
        print("机器狗正在%s地跑"%(speed))

dog_01=Dogs("阿花")
```

```
阿花(机器狗)初始状态正常
```

实例方法是类的一部分，实例方法创建完成后，可以通过该类的实例进行访问，语法:

```
类的实例名.实例方法名(相应的参数)
```

In [56]:
```
dog_01.run("快速")
```

```
机器狗正在快速地跑
```

为什么叫实例的方法呢？因为我们创建完这些方法之后，是需要通过实例来访问的，而不能通过类来访问:

In [69]:
```
Dogs.run("快速")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-69-4f075b4e896f> in <module>
----> 1 Dogs.run("快速")

TypeError: run() missing 1 required positional argument: 'speed'
```

**创建实例方法的时候，也可以和创建函数的时候一样，为参数设定默认值：**

In [57]:
```
class Dogs:
    def __init__(self,name):       #__init__()方法
        print("%s(机器狗)初始状态正常"%name)
    def run(self,speed="优哉游哉"):          #这里定义了实例方法run()
        print("机器狗正在%s地跑"%(speed))
```

In [71]:
```
dog_06=Dogs("阿花")
dog_06.run()
```

```
阿花(机器狗)初始状态正常
机器狗正在优哉游哉地跑
```

### 6.1.2 动态添加实例方法

如果创建完一个类以后，又开发了新的自定义函数，想以该自定义函数作为类的实例方法，该怎么办?

比如一开始定义了类：

In [58]:
```
class Dogs:
    def __init__(self,name):
        print("%s(机器狗)初始状态正常"%name)
    def run(self,speed="优哉游哉"):
        print("机器狗正在%s地跑"%(speed))

dogs_06=Dogs("金毛")
```

```
金毛(机器狗)初始状态正常
```

定义完毕之后，又开发出了自定义函数escape：

In [59]:
```
def escape(self,name):
    print(name,"激活逃跑状态")
```

可以用types库的方法“MethodType”将自定义函数escape作为实例方法添加到Dogs_06中,新的实例方法名定义为escape_add：

In [60]:
```
from  types  import MethodType

dogs_06.escape_add=MethodType(escape,dogs_06)
```

In [61]:
```
dogs_06.escape_add("金毛")
```

金毛 激活逃跑状态

In [62]:
```
Dogs.escape_add()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-62-e53af503e242> in <module>
----> 1 Dogs.escape_add()

AttributeError: type object 'Dogs' has no attribute 'escape_add'
```

## 6.2 实例属性

实例属性，顾名思义，就是只能通过实例才能够访问的属性，如果通过类名来访问，就会报错：

In [63]:
```
class Dogs:
    colour="黄色"                          #类的属性
    def __init__(self,name):
        self.init_power="初始电量100%"      #实例属性
        print("%s(机器狗)初始状态正常"%name)

dog_06=Dogs("金毛")
```

金毛(机器狗)初始状态正常

In [68]:
```
dog_06.nose=2
```

In [69]:
```
dog_06.nose
```

Out[69]: 2

In [66]:
```
dog_06.colour
```

Out[66]: '黄色'

In [67]:
```
Dogs.colour
```

Out[67]: '黄色'

# 7 类的继承

在面向对象编程中，被继承的类成为父类或基类，新的类称为子类或派生类。

In [70]:
```
class father:
    eyes_num=2
    eyes_colour="blue"
    def __init__(self):
        self.birth="哭着出生"
```

In [71]:
```
class son(father):
    eyes_colour="brown"
```

In [72]:
```
son.eyes_colour
```

Out[72]: 'brown'

In [73]:
```
son.eyes_num
```

Out[73]: 2

In [74]:
```
John=son()

John.birth
```

Out[74]: '哭着出生'

# 8 练习

尝试自定义一种类，完成以下操作：

- 创建该类的实例
- 动态添加一个实例方法
- 添加一种类的属性
- 创建该类的派生类

In [ ]:
```
from types import MethodType
```

## 8.1 创建类

In [80]:
```

```

## 8.2 创建派生类

In [81]:
```

```

## 8.3 动态添加类的方法

In [87]:
```

```

## 8.4 添加类的属性

In [91]:
```

```

## 8.5 动态添加实例的方法

In [93]:
```

```