形参定义方式 parameter

```
    函数: [一个] 功能
    定义:
       def 函数名称(形式参数):
           ...
           return 数据
    调用:
       函数名称(实际参数)
    参数:调用者给定义者传递的消息
    返回值:定义者给调用者传递的结果
        return 作用:返回数据,退出函数
    可变与不可变类型对象传参:
       不可变,函数内部修改栈帧的变量,不影响函数执行以后。
       可变,函数内部修改可变对象,影响函数执行以后。
    函数参数
       实参
          位置实参:根据位置进行对应(实-->形).
             序列实参:用星号拆分序列进行对应(元素-->形)
          关键字实参:根据名字进行对应(实-->形).
             字典实参:用双星号拆分字典进行对应(键-->形,传递值)
"""
def fun01(a,b):
   pass
fun01(1,2)
tuple01 = (3,5)
fun01(*tuple01)
fun01(b = 22,a= 11)
dict01 = {"b":222,"a":111}
fun01(**dict01)
```

缺省参数

1. 语法:
   def 函数名(形参名 1=默认实参 1, 形参名 2=默认实参 2, ...):

函数体

2. 说明：

缺省参数必须自右至左依次存在，如果一个参数有缺省参数，则其右侧的所有参数都必须有缺省参数。

缺省参数可以有 0 个或多个，甚至全部都有缺省参数。

位置形参

语法：

```
def 函数名(形参名 1, 形参名 2, ...):
    函数体
```

星号元组形参

1. 语法：

```
def 函数名(*元组形参名):
    函数体
```

2. 作用：

收集多余的位置传参。

3. 说明：

一般命名为'args'

形参列表中最多只能有一个

命名关键字形参

1. 语法：

```
def 函数名(*, 命名关键字形参 1, 命名关键字形参 2, ...):
    函数体
def 函数名(*args, 命名关键字形参 1, 命名关键字形参 2, ...):
    函数体
```

2. 作用：

强制实参使用关键字传参

双星号字典形参

1. 语法：

```
def 函数名(**字典形参名):
    函数体
```

2. 作用：

收集多余的关键字传参

3. 说明:

一般命名为'kwargs'

形参列表中最多只能有一个

参数自左至右的顺序

位置形参 --> 星号元组形参 --> 命名关键字形参 --> 双星号字典形参

```
"""
    函数参数
        形参
            默认形参：实参可以不传递数据。
            位置形参：实参根据位置进行对应
                星号元组形参：实参数量无限（将实参合并为元组）
            关键字形参：实参根据名称进行对应
                双星号字典形参:实参数量无限（将实参合并为字典）
                命名关键字形参：实参必须是关键字实参
"""
# 1. 默认形参：实参可以不传递数据（从右向左依次存在）
def fun01(a=0, b="bb", c=1.5):
    print(a)
    print(b)
    print(c)
#
fun01()
fun01(1, "b")
# 关键实参 + 默认形参：调用者可以随意指定参数进行传递
fun01(b="bbbbb")
# 2. 星号元组形参：让位置实参的数量无限
def fun02(p1, p2, *args):
    print(args)
fun02(1, 2)
fun02(1, 2, 3)
fun02(1, 2, 3, 4, 5)
# 3. 命名关键字形参：传递的实参必须是关键字实参。
# 写法1：星号元组形参以后的参数是命名关键字形参
#    p1  p2
def fun03(*args, p1="", p2):
    print(args)
    print(p1)
    print(p2)
fun03(2, 2, p1=111, p2=222)
fun03(p1=111, p2=222)
fun03(p2=222)
# 案例：
# def print(*args, sep=' ', end='\n', file=None):
```

```python
# 1---fff---3.5---4---55---6---67 ok
print(1, "fff", 3.5, 4, 55, 6, 67, sep="---", end="
")
print("ok")
# 写法2:星号以后的位置形参是命名关键字形参
def fun04(*, p1=0, p2):
    print(p1, p2)
fun04(p1=1, p2=2)
fun04(p2=2)
# 4. 双星号字典形参:让关键字实参的数量无限
def fun05(**kwargs):
    print(kwargs)
fun05(a=1)  # {'a': 1}
fun05(a=1, b=2)
fun05(a=1, b=2, qtx=3)  # {'a': 1, 'b': 2, 'qtx': 3}
```

作用域 LEGB

1. 作用域：变量起作用的范围。
2. Local 局部作用域：函数内部。
3. Enclosing 外部嵌套作用域：函数嵌套。
4. Global 全局作用域：模块(.py 文件)内部。
5. Builtin 内置模块作用域：builtins.py 文件。

变量名的查找规则

1. 由内到外：L -> E -> G -> B
2. 在访问变量时，先查找本地变量，然后是包裹此函数外部的函数内部的变量，之后是全局变量，最后是内置变量。

局部变量

1. 定义在函数内部的变量(形参也是局部变量)
2. 只能在函数内部使用
3. 调用函数时才被创建，函数结束后自动销毁

全局变量

1. 定义在函数外部,模块内部的变量。
2. 在整个模块(py 文件)范围内访问（但函数内不能将其直接赋值）。

global 语句

1. 作用:
   在函数内部修改全局变量。
   在函数内部定义全局变量(全局声明)。
2. 语法:
   global 变量 1, 变量 2, …
3. 说明
   在函数内直接为全局变量赋值，视为创建新的局部变量。
   不能先声明局部的变量，再用 global 声明为全局变量。

nonlocal 语句

1. 作用:
   在内层函数修改外层嵌套函数内的变量
2. 语法
   nonlocal 变量名 1,变量名 2, ...
3. 说明
   在被嵌套的内函数中进行使用

```
price:
            print("购买成功，找回：%d 元。" % (money -
total_price))
            list_order.clear()
            break
        else:
            print("金额不足.")
def get_total_price():
    """
        获取总价格
    """
    total_price = 0
    for item in list_order:
        commodity = dict_commodity_info[item["cid"]]
        total_price += commodity["price"] *
item["count"]
```

```
    return total_price
def print_orders_info():
    """
        打印所有订单信息
    """
    for item in list_order:
        commodity = dict_commodity_info[item["cid"]]
        print("商品：%s，单价：%d,数量:%d." %
(commodity["name"], commodity["price"],
item["count"]))
shopping()
```

面向对象 Object Oriented

概述

面向过程

1. 分析出解决问题的步骤，然后逐步实现。
   例如：婚礼筹办
   -- 发请柬（选照片、措词、制作）
   -- 宴席（场地、找厨师、准备桌椅餐具、计划菜品、购买食材）
   -- 婚礼仪式（定婚礼仪式流程、请主持人）
2. 公式：程序 = 算法 + 数据结构
3. 优点：所有环节、细节自己掌控。
4. 缺点：考虑所有细节，工作量大。

面向对象

1. 找出解决问题的人，然后分配职责。
   例如：婚礼筹办
   -- 发请柬：找摄影公司（拍照片、制作请柬）
   -- 宴席：找酒店（告诉对方标准、数量、挑选菜品）
   -- 婚礼仪式：找婚庆公司（对方提供司仪、制定流程、提供设备、帮助执行）
2. 公式：程序 = 对象 + 交互
3. 优点
   (1) 思想层面：
      -- 可模拟现实情景，更接近于人类思维。
      -- 有利于梳理归纳、分析解决问题。
   (2) 技术层面：
      -- 高复用：对重复的代码进行封装，提高开发效率。
      -- 高扩展：增加新的功能，不修改以前的代码。
      -- 高维护：代码可读性好，逻辑清晰，结构规整。
4. 缺点：学习曲线陡峭。

类和对象

1. 类：一个抽象的概念，即生活中的”类别”。
2. 对象：类的具体实例，即归属于某个类别的”个体”。
3. 类是创建对象的”模板”。
   -- 数据成员：名词类型的状态。
   -- 方法成员：动词类型的行为。
4. 类与类行为不同，对象与对象数据不同。

语法

定义类

1. 代码
   class 类名:
   “””文档说明”””
   def _init_(self,参数列表):
   self.实例变量 = 参数
   方法成员
2. 说明
   -- 类名所有单词首字母大写.
   -- _init_ 也叫构造函数，创建对象时被调用，也可以省略。
   -- self 变量绑定的是被创建的对象，名称可以随意。

创建对象(实例化)

变量 = 构造函数 (参数列表)

实例成员

实例变量

1. 语法
   (1) 定义：对象.变量名
   (2) 调用：对象.变量名

2. 说明
   (1) 首次通过对象赋值为创建，再次赋值为修改.

w01 = Wife()
w01.name = “丽丽”
w01.name = “莉莉”

(2) 通常在构造函数(_init_)中创建。

w01 = Wife(“丽丽”,24)
print(w01.name)

(3) 每个对象存储一份，通过对象地址访问。

3. 作用：描述所有对象的共有数据。
4. __dict__：对象的属性，用于存储自身实例变量的字典。

实例方法

1. 语法
   (1) 定义： def 方法名称(self, 参数列表):
   方法体
   (2) 调用： 对象地址.实例方法名(参数列表)
   不建议通过类名访问实例方法
2. 说明
   (1) 至少有一个形参，第一个参数绑定调用这个方法的对象,一般命名为"self"。
   (2) 无论创建多少对象，方法只有一份，并且被所有对象共享。
3. 作用：表示对象行为。

```
"""
    类:抽象的概念       类别
        水果类成员:
            数据(名词)成员:重量/水分/味道/价格...
            行为(动词)成员:生长/腐烂...
        狗类成员:
            数据(名词)成员:重量/体味/高度...
            行为(动词)成员:尿尿/吃/叫...
    对象:具体的实例    个体
        水果类对象:香蕉/苹果/哈密瓜
        狗类对象:拉布拉多/金毛
    类与类行为不同,对象与对象数据不同.
"""
class Wife:
    """
        老婆类   ---- 抽象
    """
    # 数据
    def __init__(self,name,height,weight):
        self.height = height
        self.weight = weight
```

```
        self.name = name
    # 行为
    def play(self):
        # 方法可以访问数据
        print(self.name + "在玩耍")
# 创建对象
# 类名(参数...)  调用__init__方法
w01 = Wife("翠花",1.8,180)
w01.play()# 通过对象地址调用方法,会自动传递对象地址.
w02 = Wife("如花",2.1,20)
w02.play()
# 同一个方法,可以访问不同的对象数据(方法中self指向了不同对象).
```

4.