10. 前方高能-函数的进阶

本节主要内容:

1. 函数参数--动态传参
2. 名称空间, 局部名称空间, 全局名称空间, 作用域, 加载顺序.
3. 函数的嵌套
4. gloabal, nonlocal关键字

一. 函数参数--动态传参

之前我们说过了传参, 如果我们需要给一个函数传参, 而参数又是不确定的. 或者我给一个函数传很多参数, 我的形参就要写很多, 很麻烦, 怎么办呢. 我们可以考虑使用动态参数.

形参的第三种: 动态参数

动态参数分成两种:

1. 动态接收位置参数

首先我们先回顾一下位置参数, 位置参数, 按照位置进行传参

```
def chi(quality_food, junk_food):
    print("我要吃", quality_food, junk_food)

chi("大米饭", "小米饭")    # "大米饭"传递给quality_food    "小米饭"传递给junk_food
按照位置传
```

现在问题来了. 我想吃任意的食物. 数量是任意的, 食物也是任意的. 这时我们就要用到动态参数了.

在参数位置编写*表示接收任意内容

```
def chi(*food):
    print("我要吃", food)

chi("大米饭", "小米饭")

结果:
我要吃 ('大米饭', '小米饭')    # 多个参数传递进去. 收到的内容是元组tuple
```

动态接收参数的时候要注意: 动态参数必须在位置参数后面

```
def chi(*food, a, b):
    print("我要吃", food, a, b)

chi("大米饭", "小米饭", "黄瓜", "茄子")
```

这时程序运行会报错. 因为前面传递进去的所有位置参数都被*food接收了. a和b永远接收不到参数

```
Traceback (most recent call last):
  File "/Users/sylar/PycharmProjects/oldboy/fun.py", line 95, in <module>
    chi("大米饭", "小米饭", "黄瓜", "茄子")
TypeError: chi() missing 2 required keyword-only arguments: 'a' and 'b'
```

所以必须改写成以下代码:

```
def chi(*food, a, b):
    print("我要吃", food, a, b)

chi("大米饭", "小米饭", a="黄瓜", b="茄子")    # 必须用关键字参数来指定
```

这个时候a和b就有值了, 但是这样写呢位置参数就不能用了. 所以. 我们要先写位置参数, 然后再用动态参数

```
def chi(a, b, *food):
    print("我要吃", a, b, food)

chi("大米饭", "小米饭", "馒头", "面条")    # 前两个参数用位置参数来接收, 后面的参数用动态参数接收
```

那默认值参数呢?

```
def chi(a, b, c='馒头', *food):
    print(a, b, c, food)

chi("香蕉", "菠萝")  # 香蕉 菠萝 馒头 (). 默认值生效
chi("香蕉", "菠萝", "葫芦娃")  # 香蕉 菠萝 葫芦娃 ()     默认值不生效
chi("香蕉", "菠萝", "葫芦娃", "口罩")     # 香蕉 菠萝 葫芦娃 ('口罩',) 默认值不生效
```

我们发现默认值参数写在动态参数前面. 默认值只有一种情况可能会生效.

```
def chi(a, b, *food, c="娃哈哈"):
    print(a, b, food, c)

chi("香蕉", "菠萝")  # 香蕉 菠萝 () 娃哈哈    默认值生效
chi("香蕉", "菠萝", "葫芦娃")  # 香蕉 菠萝 ('葫芦娃',) 娃哈哈     默认值生效
chi("香蕉", "菠萝", "葫芦娃", "口罩")     # 香蕉 菠萝 ('葫芦娃', '口罩') 娃哈哈   默认值生效
```

这个时候我们发现所有的默认值都生效了. 这个时候如果不给出关键字传参. 那么你的默认值是永远都生效的.

顺序: 位置参数, 动态参数*, 默认值参数

2. 动态接收关键字参数

在python中可以动态的位置参数, 但是*这种情况只能接收位置参数无法接收关键字参数.

在python中使用**来接收动态关键字参数

```
def func(**kwargs):
    print(kwargs)

func(a=1, b=2, c=3)
func(a=1, b=2)

结果:
{'a': 1, 'b': 2, 'c': 3}
{'a': 1, 'b': 2}
```

这个时候接收的是一个dict

顺序的问题, 在函数调用的时候, 如果先给出关键字参数, 则整个参数列表会报错.

```
def func(a, b, c, d):
    print(a, b, c, d)


# 关键字参数必须在位置参数后面, 否则参数会混乱
func(1, 2, c=3, 4)
```

所以关键字参数必须在位置参数后面. 由于实参是这个顺序. 所以形参接收的时候也是这个顺序. 也就是说位置参数必须在关键字参数前面. 动态接收关键字参数也要在后面

最终顺序(*):

位置参数 > *args > 默认值参数 > **kwargs

这四种参数可以任意的进行使用.

如果想接收所有的参数:

```
def func(*args, **kwargs):
    print(args, kwargs)

func("麻花藤","马晕",wtf="胡辣汤")
```

动态参数的另一种传参方式:

```
def fun(*args):
    print(args)


lst = [1, 4, 7]
fun(lst[0], lst[1], lst[2])

fun(*lst)   # 可以使用*把一个列表按顺序打散
s = "臣妾做不到"
fun(*s)     # 字符串也可以打散, (可迭代对象)
```

在实参位置上给一个序列,列表,可迭代对象前面加个*表示把这个序列按顺序打散.

在形参的位置上的* 表示把接收到的参数组合成一个元组
如果是一个字典, 那么也可以打散. 不过需要用两个*

```
def fun(**kwargs):
    print(kwargs)

dic = {'a':1, 'b':2}
fun(**dic)
```

函数的注释:

```
def chi(food, drink):
    """
    这里是函数的注释, 先写一下当前这个函数是干什么的, 比如我这个函数就是一个吃
    :param food: 参数food是什么意思
    :param drink: 参数drink是什么意思
    :return: 返回的是什么东东
    """
    print(food, drink)
    return "very good"
```

二. 命名空间

在python解释器开始执行之后, 就会在内存中开辟一个空间, 每当遇到一个变量的时候, 就把变量名和值之间的关系记录下来, 但是当遇到函数定义的时候, 解释器只是把函数名读入内存, 表示这个函数存在了, 至于函数内部的变量和逻辑, 解释器是不关心的. 也就是说一开始的时候函数只是加载进来, 仅此而已, 只有当函数被调用和访问的时候, 解释器才会根据函数内部声明的变量来进行开辟变量的内部空间. 随着函数执行完毕, 这些函数内部变量占用的空间也会随着函数执行完毕而被清空.

```
def fun():
    a = 10
    print(a)

fun()
print(a)    # a不存在了已经..
```

我们给存放名字和值的关系的空间起一个名字叫: 命名空间. 我们的变量在存储的时候就是存储在这片空间中的.

命名空间分类:

1. 全局命名空间--> 我们直接在py文件中, 函数外声明的变量都属于全局命名空间
2. 局部命名空间--> 在函数中声明的变量会放在局部命名空间
3. 内置命名空间--> 存放python解释器为我们提供的名字, list, tuple, str, int这些都是内置命名空间

加载顺序:

1. 内置命名空间

2. 全局命名空间
3. 局部命名空间(函数被执行的时候)

取值顺序:

1. 局部命名空间
2. 全局命名空间
3. 内置命名空间

```
a = 10
def func():
    a = 20
    print(a)


func()  # 20
```

作用域: 作用域就是作用范围, 按照生效范围来看分为 全局作用域和局部作用域

全局作用域: 包含内置命名空间和全局命名空间. 在整个文件的任何位置都可以使用(遵循从上到下逐行执行). 局部作用域: 在函数内部可以使用.

作用域命名空间:

1. 全局作用域: 全局命名空间 + 内置命名空间
2. 局部作用域: 局部命名空间

我们可以通过globals()函数来查看全局作用域中的内容, 也可以通过locals()来查看局部作用域中的变量和函数信息

```
a = 10
def func():
    a = 40
    b = 20
    def abc():
        print("哈哈")
    print(a, b)      # 这里使用的是局部作用域
    print(globals())    # 打印全局作用域中的内容
    print(locals())     # 打印局部作用域中的内容

func()
```

三. 函数的嵌套

1. 只要遇见了()就是函数的调用. 如果没有()就不是函数的调用
2. 函数的执行顺序

```
def fun1():
    print(111)
```

```
def fun2():
    print(222)
    fun1()

fun2()
print(111)

# 函数的嵌套
def fun2():
    print(222)
    def fun3():
        print(666)
    print(444)
    fun3()
    print(888)
print(33)
fun2()
print(555)
```

四. 关键字global和nonlocal

首先我们写这样一个代码, 首先在全局声明一个变量, 然后再局部调用这个变量, 并改变这个变量的值

```
a = 100
def func():
    global a    # 加了个global表示不再局部创建这个变量了. 而是直接使用全局的a
    a = 28
    print(a)

func()
print(a)
```

global表示. 不再使用局部作用域中的内容了. 而改用全局作用域中的变量

```
lst = ["麻花藤", "刘嘉玲", "詹姆斯"]
def func():
    lst.append("马云云")    # 对于可变数据类型可以直接进行访问. 但是不能改地址. 说白了. 不能赋值
    print(lst)

func()
print(lst)
```

nonlocal 表示在局部作用域中, 调用父级命名空间中的变量.

```
a = 10
def func1():
    a = 20
    def func2():
```

```
        nonlocal a
        a = 30
        print(a)
    func2()
    print(a)


func1()
```

```
结果:
加了nonlocal
30
30
不加nonlocal
30
20
```

再看, 如果嵌套了很多层, 会是一种什么效果:

```
a = 1
def fun_1():
    a = 2
    def fun_2():
        nonlocal a
        a = 3
        def fun_3():
            a = 4
            print(a)
        print(a)
        fun_3()
        print(a)
    print(a)
    fun_2()
    print(a)


print(a)
fun_1()
print(a)
```

这样的程序如果能分析明白. 那么作用域, global, nonlocal就没问题了