高级程序设计
---Python与深度学习
6. 函数、模块
6. Function & module

李冰
副研究员
交叉科学研究院

课程内容

- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

函数（Function）

- 通用程序部件，一处编写，多处使用。
- 函数的好处
  - 减少重复代码
  - 方便代码阅读
  - 方便调试

```
def name(arg1, arg2,... argN):
    ...
    return value
```

- def 首行定义了函数名，赋值给了函数对象，并在括号中包含了0个或多个参数。
- 代码块紧随其后，并会缩进。
- 函数主体往往都是包含了一条 **return** 语句，它表示函数调用的结束，并将结果返回至函数调用处。
- **return** 语句可选，如果没有 **return** 语句，函数运行结束时返回 None 对象。

函数例子

- 定义

```
def times(x, y):                    # Create and assign function
    return x * y                    # Body executed when called
```

```
multip = times
type(multip)
```

```
function
```

函数例子

- 定义

```
def times(x, y):                    # Create and assign function
    return x * y                    # Body executed when called
```

- 调用

```
times(2, 4)                         # Arguments in parentheses
```

```
8
```

```
multip(2, 5)
```

```
10
```

```
times('Ni', 4)                      # Functions are "typeless"
```

```
'NiNiNiNi'
```

可以将 times() 用作数字的乘法或是序列的重复。

课程内容

- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

函数多态例子：寻找序列的交集

```
def intersect(seq1, seq2):
    res = []                          # Start empty
    for x in seq1:                    # Scan seq1
        if x in seq2:                 # Common item?
            res.append(x)             # Add to end
    return res
```

用来搜索两个序列的公共元素。

```
s1 = "SPAM"
s2 = "SCAM"
intersect(s1, s2)                     # Strings
```

字符串

```
['S', 'A', 'M']
```

函数多态例子：寻找序列的交集

```
def intersect(seq1, seq2):
    res = []                          # Start empty
    for x in seq1:                    # Scan seq1
        if x in seq2:                 # Common item?
            res.append(x)             # Add to end
    return res
```

用来搜索两个序列的公共元素。

```
s1 = "SPAM"
s2 = "SCAM"
intersect(s1, s2)                     # Strings
```

```
['S', 'A', 'M']
```

字符串

```
x = intersect([1, 2, 3], (1, 4))      # Mixed types
x                                     # Saved result object
```

```
[1]
```

列表

函数多态例子：寻找序列的交集

```
def intersect(seq1, seq2):
    res = []                          # Start empty
    for x in seq1:                    # Scan seq1
        if x in seq2:                 # Common item?
            res.append(x)             # Add to end
    return res
```

用来搜索两个序列的公共元素。

```
s1 = "SPAM"
s2 = "SCAM"
intersect(s1, s2)                     # Strings
```

```
['S', 'A', 'M']
```

```
x = intersect([1, 2, 3], (1, 4))      # Mixed types
x                                     # Saved result object
```

```
[1]
```

列表

多态(polymorphism)

- Python 将对某一对象在某种语法的合理性由对象自身来判断。
- 自动地适用于所有类别的对象类型。只要对象支持所预期的接口就能处理。
- 运算符多态
  - +， *， 运算类型可以是数字，字符串。
- 函数多态
  - 函数自动地适用于所有类别的对象类型。只要对象支持所预期的接口，那么函数就能处理。
    - times() 函数中表达式 x * y 的意义完全取决于 x 和 y 的对象类型。
    - 可以实现两种功能：数字的相乘和字符重复

多态(polymorphism)

- 函数多态
  - 函数自动地适用于所有类别的对象类型。只要对象支持所预期的接口，那么函数就能处理。

```
def name(arg1, arg2,... argN):
    ...
    return value
```

  - 在 Python 中，代码不关心特定的数据类型。
  - 在 Python 中为对象编写接口，而不是数据类型。
  - 如果不支持这种预期的接口，Python 将会在运行时检测到错误，并抛出一个异常。

课程内容

- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

函数：作用域

- “在函数里有个变量，那函数外这个变量是否起作用？”
- 变量作用域
  - 变量作用域的含义就是这个变量名能被访问到的范围。
  - 当在程序中使用变量名时，Python 创建、改变或查找变量名都是在所谓的命名空间（一个保存变量名的地方）中进行的。

函数：作用域

- 一个在 def 内定义的变量名能够被 def 内的代码使用，不能在函数的外部引用这个变量名。
- def 内的变量名与 def 外的变量名并不冲突，即使是使用在别处的相同的变量名。
- 一个在 def 外被赋值的变量 X 与 在这个 def 内赋值的变量 X 是完全不同的变量。

```
X = 99
def func():
    X = 88
#func()
print(X)
```

```
99
```

函数：作用域

```
# Global scope
X = 99                          # X and func assigned in module: global

def func(Y):                    # Y and Z assigned in function: locals
    # Local scope
    Z = X + Y                   # X is a global
    return Z

func(1)
```

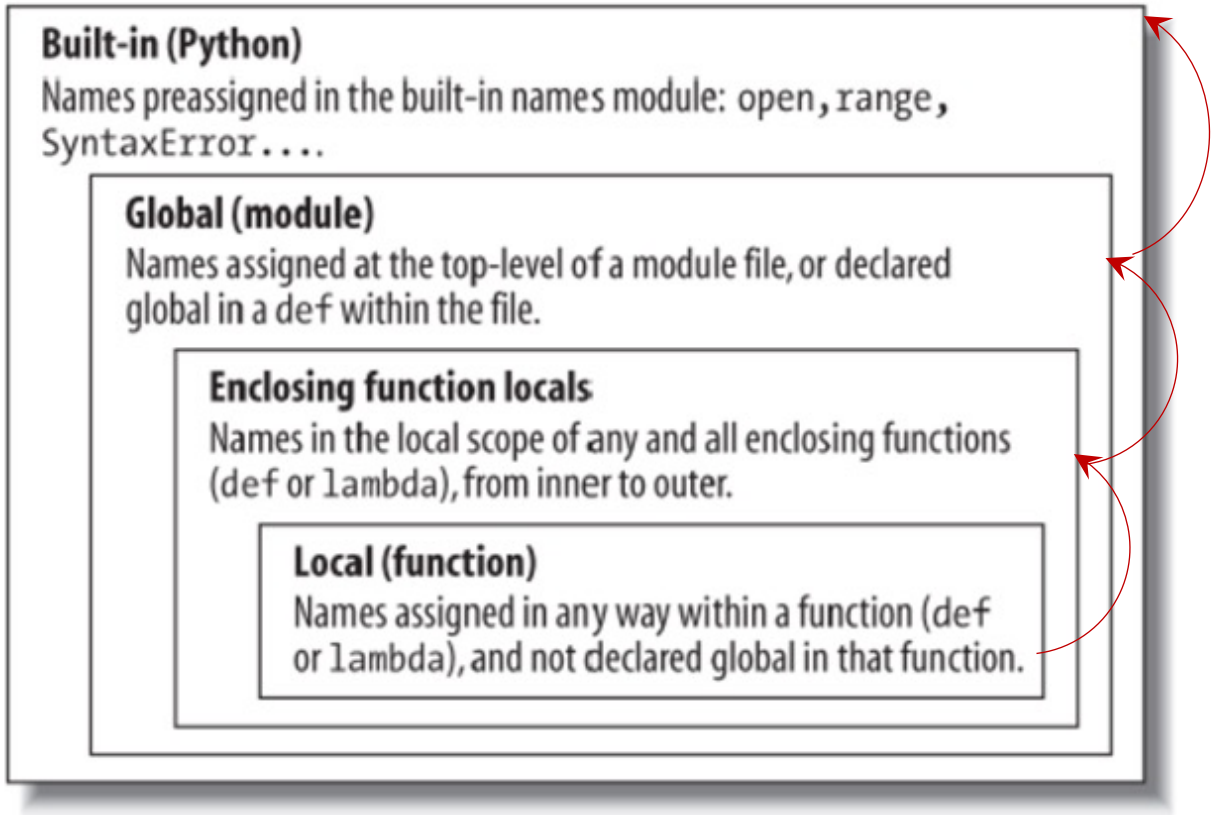
- 全局变量名：X, func
  - X 和func是在模块文件顶层创建的，是全局变量
  - X能够在函数内部进行引用，不需要特意声明
  - 在整个文件中都有效
- 局部变量名：Y, Z
  - Y 和 Z 都是在**函数定义内部进行赋值**，是局部变量，只在函数内有效

作用域法则

- 单个文件对应全局作用域（global）
  - 一个文件的顶层的变量名仅对于这个文件内部的代码而言是全局的。
- 每次对函数的调用都创建了一个新的局部作用域(local)
  - 也就是说，将会存在由那个函数创建的命名空间。
- 变量作用域的关键字
  - `global, nonlocal,` 改变变量的作用域
  - 比如，在局部作用域(一个函数里)声明一个全局变量，那这个变量在整个文件都有作用。

作用域法则

- 变量名引用分为四个作用域依次查找：
  - 局部(local)， 嵌套函数(enclosing functions)，全局(global)，内置(built-in)



全局变量

- 全局变量是位于模块文件内部的顶层的变量名
- 全局变量在函数内部不经过声明也可以被引用
- 在函数内创建/修改一个全局变量的话，必须通过关键
  - global允许我们创建/修改一个全局变量。

```
X = 88                          # Global X

def func():
    global X
    X = 99                      # Global X: outside def

func()
print(X)                        # Prints 99
```

```
99
```

全局变量

- 全局变量是位于模块文件内部的顶层的变量名
- 全局变量在函数内部不经过声明也可以被引用
- 在函数内创建/修改一个全局变量的话，必须通过关键
  - global允许我们创建/修改一个全局变量。

```
# Global variables in module
y, z = 1, 2

def all_global():
    global x                          # Declare globals assigned
    x = y + z                         # No need to declare y, z: LEGB rule

all_global()
print(x)
```

```
3
```

全局变量

- 过多使用全局变量可能会引发一些软件工程问题
  - 由于变量的值取决于函数调用的顺序，而函数自身是任意顺序进行排序的，导致了程序调试起来非常困难。

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

```
func2()
func1()
print(X)

func2()
print(X)
```

这里 X 的值将会是多少？

全局变量

- 过多使用全局变量可能会引发一些软件工程问题
  - 由于变量的值取决于函数调用的顺序，而函数自身是任意顺序进行排序的，导致了程序调试起来非常困难。

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

```
func2()
func1()
print(X)

func2()
print(X)
```

这里 X 的值将会是多少？

这里 X 的值将会是多少？

在不熟悉编程的情况下，最好尽可能避免使用全局变量。

嵌套函数及嵌套作用域

- 对于一个函数：

```
X = 99                                  # Global scope name: not used

def f1():
    X = 88                              # Enclosing def local
    def f2():
        print(X)                        # Reference made in nested def
    f2()

f1()                                    # Prints 88: enclosing def local
```

```
88
```

嵌套函数及嵌套作用域

- 对于一个函数：
  - 一个引用（X）首先在局部（函数内）作用域查找变量名 X；
  - 之后会在代码的语法上嵌套了的函数中的局部作用域，从内到外查找；
  - 之后查找当前模块文件的全局作用域；最后搜索内置作用域（**builtin**）
  - 默认情况下，一个赋值（X = value）创建或改变了变量名 X 的作用域。如果 X 在函数内部声明为全局变量，它将创建或改变变量名 X 为整个模块的作用域。
  - 另一方面，如果 X 在函数内声明为 nonlocal，赋值会修改最近的嵌套函数的局部作用域中的名称 X。

嵌套函数及嵌套作用域

```
def tester(begin):
    state = begin                    # Referencing nonlocals works normally
    def nested(label):
        print(label, state)          # Remembers state in enclosing scope
    return nested

F = tester(0)
F('spam')
```

```
spam 0
```

嵌套函数及嵌套作用域

```
def tester(begin):
    state = begin
    def nested(label):
        print(label, state)
        state += 1                          # Cannot change by default
    return nested

F = tester(0)
# F('spam')
```

嵌套函数及嵌套作用域

- nonlocal 语句

```
def tester(begin):
    state = begin                          # Each call gets its own state
    def nested(label):
        nonlocal state                     # Remembers state in enclosing scope
        print(label, state)
        state += 1                         # Allowed to change it if nonlocal
    return nested

F = tester(0)
F('spam')                                  # Increments state on each call
```

```
spam 0
```

nonlocal 名称必须已经在一个嵌套的 def 作用域中赋值过，否则将会得到一个错误提示。
nested 中将 state 声明为一个 nonlocal，就可以在 nested 函数中修改它了。

嵌套函数及嵌套作用域

- nonlocal 语句

```
def tester(begin):
    state = begin                          # Each call gets its own state
    def nested(label):
        nonlocal state                     # Remembers state in enclosing scope
        print(label, state)
        state += 1                         # Allowed to change it if nonlocal
    return nested

F = tester(0)
F('spam')                                  # Increments state on each call
```

```
spam 0
```

tester已经返回退出了，这依然是有效的。

```
F('ham')
```

```
ham 1
```

```
F('eggs')
```

```
eggs 2
```

课程内容

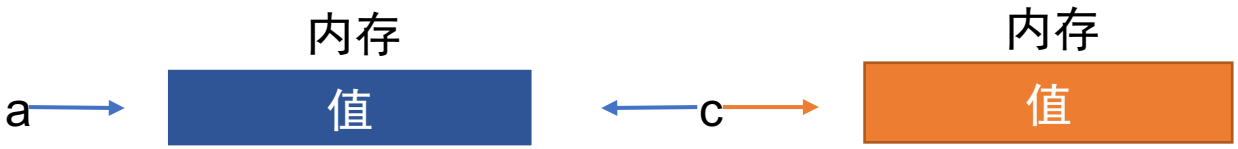
- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

函数：参数

- 参数传递
  - 函数的传参方式是共享传参，即函数的形参是实参中各个引用的副本。
  - 不可变对象，变量的改变相当于指向的改变，直接换一块内存
  - 可变对象，变量的改变相当于指向的那块内存值的改变。

```
def f(c)
    ……
a
f(a)
```

如果a是不可变对象：int，string，float，tuple， f(c)中修改了c， 那么：



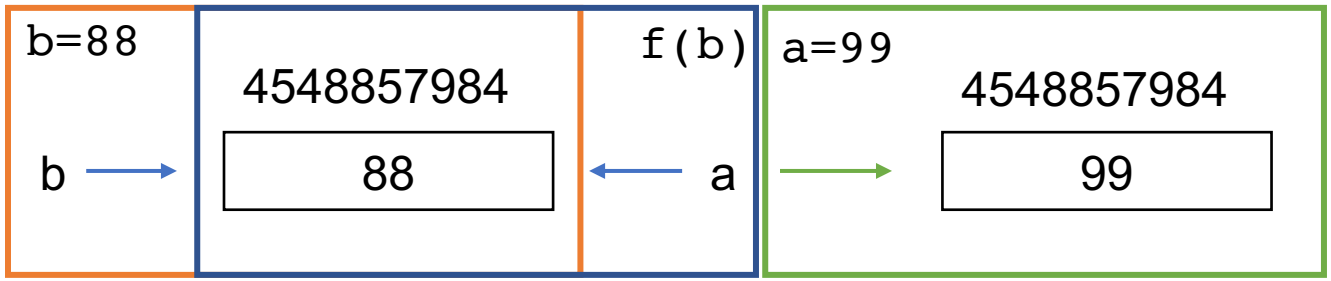
如果a是可变对象：list, set, dict， f(c)中修改了c， 那么：

函数：参数

```
def f(a):                                   # a is assigned to (references) the passed object
    print('f before change, ', id(a))
    a = 99                                  # Changes local variable a only
    print('f after change, ', id(a))

b = 88
print('main before invoke f , ',id(b))
f(b)                                        # a and b both reference same 88 initially
print('main after invoke f , ',id(b))
print(b)                                    # b is not changed
```

```
main before invoke f ,  4548857984
f before change,  4548857984
f after change,  4548858336
main after invoke f ,  4548857984
88
```



函数：参数

```
def changer(a, b):                      # Arguments assigned references to objects
    a = 2                               # Changes local name's value only
    b[0] = 'spam'                       # Changes shared object in place

X = 1
L = [1, 2]                              # Caller:
changer(X, L)                           # Pass immutable and mutable objects
X, L                                    # X is unchanged, L is different!
```

```
(1, ['spam', 2])
```

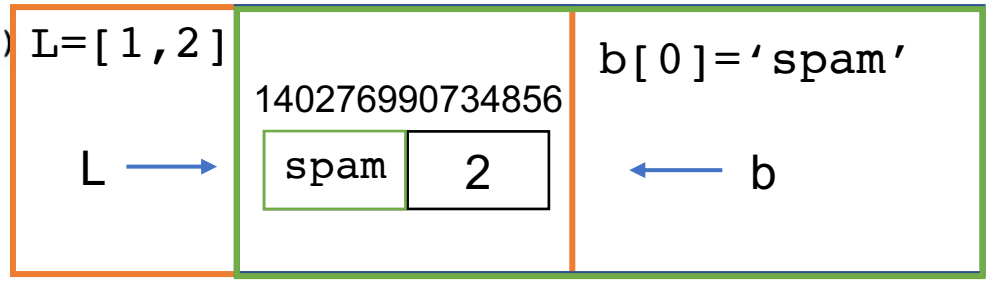
第二个变量变化了，为什么呢？

函数：参数

```
def changer(a, b):                          # Arguments assigned references to objects
    a = 2                                   # Changes local name's value only
    print('changer before change, ',id(b))
    b[0] = 'spam'                           # Changes shared object in place
    print('changer after change, ',id(b))

X = 1
L = [1, 2]                                  # Caller:
print('main before invoke changer , ',id(L))
changer(X, L)                               # Pass immutable and mutable objects
print('main before invoke changer , ',id(L))
X, L                                        # X is unchanged, L is different!
```

```
main before invoke changer ,  140276991366920
changer before change,  140276991366920
changer after change,  140276991366920
main before invoke changer ,  140276991366920
(1, ['spam', 2])
```



b是局部变量名

函数：参数

- 如果不想改变传入的可变对象参数，我们需要进行拷贝。
- 两种方式：传入拷贝，在函数内拷贝。

```
def changer(a, b):                          # Arguments assigned references to objects
    a = 2                                   # Changes local name's value only
    b[0] = 'spam'                           # Changes shared object in place

L = [1, 2]
changer(X, L[:])                            # Pass a copy, so our 'L' does not change
X, L
```

```
(1, [1, 2])
```

```
def changer(a, b):
    b = b[:]                                # Copy input list so we don't impact caller
    a = 2
    b[0] = 'spam'                           # Changes our list copy only

L = [1,2]
changer(X,L)
X,L
```

函数：参数匹配

- 在默认情况下，参数是通过其位置进行匹配的，从左至右，而且必须精确地传递和函数头部参数名一样多的参数。

函数：参数匹配

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

func(1, 2, 3)
```

```
1 2 3
```

```
func(c = 3, b = 2, a = 1)                    # 不推荐随意交换参数顺序
```

```
1 2 3
```

```
def func(a, b=2, c=3):                       # a required, b and c optional
    print(a, b, c)
```

```
func(1, 4)                                   # Override defaults
```

```
1 4 3
```

函数：不定长参数/可变参数

- 如果有不同数量的参数，如何定义函数？
  - Python 模块库代码，会发现很多函数的参数定义，都会跟上 *args 和 **kwargs（不定参数的另一种形式，后面会讲到）。
- **Python提供方法，用来支持任意参数**

函数：不定长参数/可变参数

- **Python提供方法，用来支持任意参数**
- 第一种用法：在函数定义中，在**元组**中收集不匹配的位置参数。
- 下面的例子中，Python 将所有位置相关的参数收集到一个新的**元组**中，并将这个元组赋值给变量args （args的名称可以自行定义）：

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

f()
```

```
()
```

```
f(1)
```

```
(1,)
```

```
f(1, 2, 3, 4)
```

```
(1, 2, 3, 4)
```

函数：不定长参数/可变参数

- 第二种用法：形参名前加两个*表示，参数在函数内部将被存放在以形式名为标识符的 **字典**中，调用函数的方法则需要采用 arg1=value1,arg2=value2 这样的形式。
- 如下面程序所示：

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

f()
```

```
{}
```

```
f(a=1, b=2)
```

```
{'a': 1, 'b': 2}
```

注意，采用 **kwargs 传递参数的时候，不能传递数组参数

```
f(1,2)
```

函数：不定长参数/可变参数

- 函数参数列表能够混合一般参数、* 参数以及 ** 来实现更加灵活的调用方式。

```
def f(a, *pargs, **kargs):
    print(a, pargs, kargs)

f(1, 2, 3, x=4, y=5)
```

```
1 (2, 3) {'x': 4, 'y': 5}
```

函数：参数

- 解包参数：
  - 在调用函数时使用 * 语法
  - 它会解包参数的集合，而不是创建参数的集合。
- 例如，我们可以给函数传递一个包含四个元素的元组，让 Python 将它们解包成四个独立的参数。

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

args = (1, 2)
args += (3, 4)
func(*args)                          # Same as func(1, 2, 3, 4)
```

```
1 2 3 4
```

如果args是个列表或者数组时，如何解包？

函数：参数匹配

- 解包参数：
  - 相似的，在函数调用时，** 会以键值对的形式解包一个字典，使其成为独立的关键字参数。

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

args = {'a': 1, 'b': 2, 'c': 3}
args['d'] = 4
func(**args)                                    # Same as func(a=1, b=2, c=3, d=4)
```

```
1 2 3 4
```

函数：参数匹配

- Keyword-Only Arguments表示给函数传参的时候必须指定参数名，也就是关键字， 这是Python3 新加特性
- “*”星号后面的参数必须以指定参数名的方式传参！

```
def kwonly(a, *b, c):
    print(a, b, c)

kwonly(1, 2, c=3)
```

```
1 (2,) 3
```

```
kwonly(a=1, c=3)
```

```
1 () 3
```

```
# kwonly(1, 2, 3)                    # kwonly() missing 1 required keyword-only argument: 'c'
```

代码中，a 可以按照位置或者关键字传递，b 收集任何额外的位置参数，c 必须只按照关键字传递。

函数：参数匹配

- Keyword-Only Arguments表示给函数传参的时候必须指定参数名，也就是关键字， 这是Python3 新加特性
- “*” 星号后面的参数必须以指定参数名的方式传参！
- 可以对 keyword-only 参数使用默认值。

```
def kwonly(a, *, b, c='spam'):
    print(a, b, c)

kwonly(1, b='eggs')
```

```
1 eggs spam
```

```
# kwonly(1, c='eggs')                    # kwonly() missing 1 required keyword-only argument: 'b'
```

参数匹配小结

- 位置：从左至右进行匹配
  - 最常使用的方法
- 关键字参数：通过参数名进行匹配
  - 调用者可以定义函数的哪一个参数接受这个值，通过在调用时使用参数的变量名，使用name=value这种语法
- 默认参数：为没有传入值的参数定义参数值
  - 函数能为自己设定参数的默认值，同样使用语法name=value
- 不定长参数：收集任意多基于位置或关键字的参数
  - 函数能够使用特定的参数，它们是以字符 * 开头，收集任意多的额外参
- 不定长参数解包：传递任意多的基于位置或关键字的参数
  - 在函数头部 * 意味着收集任意多的参数，而在调用者中意味着传递任意多的参数
- Keyword-only 参数：参数必须按照关键字传递
  - Keyword-only 参数通常用来定义实际参数以外的配置选项

实例：求最小值的函数

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res
```

```
def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first
```

```
def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]
```

```
print(min1(3, 4, 1, 2))
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

```
1
aa
[1, 1]
```

课程内容

- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

匿名函数 lambda

- 除了 def 语句之外，Python 还提供了一种生成函数对象的表达式形式，它返回了一个函数而不是将这个函数赋值给一个变量名。这也是 lambda 也叫作匿名函数的原因。
  - lambda 是一个表达式，而不是一个语句
    - 正因为这点，lambda 能够出现在 Python 语法不允许 def 出现的地方，例如在一个列表常量或者函数调用的参数中。
  - lambda 的主体是一个单个的表达式，而不是一个代码块
    - 主体简单的就好像放在 def 主体的 return 语句中的代码一样。因为它仅限于表达式，lambda 通常要比 def 功能要小，连 if 这样的语句都不能使用。

```
lambda argument1, argument2,... argumentN : expression using arguments
```

关键字 lambda 后面跟一个或多个参数，紧跟的是一个冒号以及表达式

lambda表达式

```
def func(x, y, z):
    return x + y + z

func(2, 3, 4)
```

```
9
```

lambda 表达式所返回的函数对象与由 def 创建并赋值的函数对象工作起来是完全一样的。

```
f = lambda x, y, z: x + y + z
f(2, 3, 4)
```

```
9
```

```
x = (lambda a="fee", b="fie", c="foe": a + b + c)
x("wee")
```

```
'weefiefoe'
```

参数的用法

lambda表达式

- 通常来说，lambda 起到了一种函数速写的作用，允许在使用的代码内嵌入一个函数的定义。
- 和def等价，但更简洁
  - 作用域，参数传递等准则对lambda适用

```
L = [lambda x: x ** 2,                  # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]                  # A list of three callable functions

for f in L:
    print(f(2))
```

```
4
8
16
```

```
print(L[0](3))
```

```
9
```

lambda表达式

- 通常来说，lambda 起到了一种函数速写的作用，允许在使用的代码内嵌入一个函数的定义。
- 和def等价，但更简洁
  - 作用域，参数传递等准则对lambda适用

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]

for f in L:
    print(f(2))
```

```
4
8
16
```

```
print(L[0](3))
```

```
9
```

```
def f1(x): return x ** 2
def f2(x): return x ** 3          # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]                  # Reference by name

for f in L:
    print(f(2))                   # Prints 4, 8, 16
print(L[0](3))                    # Prints 9
```

```
4
8
16
9
```

lambda表达式

- Python 中的字典或者其它的数据结构来构建更多种类的行为表，从而做同样的事情

```
key = 'got'
{'already': (lambda: 2 + 2),
'got': (lambda: 2 * 4),
'one': (lambda: 2 ** 6)}[key]()
```

```
8
```

```
def f1(): return 2 + 2
def f2(): return 2 * 4
def f3(): return 2 ** 6

key = 'one'
{'already': f1, 'got': f2, 'one': f3}[key]()
```

```
64
```

- 以下情况优先考虑lambda：
  - 三个函数**不会在其它的地方使用到**
  - 这三个函数创建变量名也许会与文件中的其它变量名发生冲突

嵌套 lambda 和作用域

- lambda 是嵌套函数作用域查找的最大受益者。
- 很典型的情况是：lambda 出现在 def 中，嵌套的 lambda 能够获取到上层函数作用域中的变量名 x 的值。

```
def action(x):
    return (lambda y: x + y) # Make and return function, remember x

act = action(99)
act
# act()  #TypeError: <lambda>() missing 1 required positional argument: 'y'
```

```
<function __main__.action.<locals>.<lambda>(y)>
```

```
act(2)
```

```
101
```

嵌套lambda和作用域

- 那我们省去def 会怎么样？

```
action = (lambda x: (lambda y: x + y))
act = action(99)
act(3)
```

```
102
```

出于对代码可读性的要求，通常最好避免使用嵌套的 lambda。

课程内容

- 函数
  - 定义def
  - 关键字
  - 多态
  - 作用域
  - 参数
  - lambda
- 模块
  - 用法
  - 加载与重载
  - 注意事项
  - 调试

模块module

- 什么是模块
  - Python 文件，以.py结尾，包含定义的函数、变量和类。

- 模块的好处
  - 方便维护程序
    - 程序很长的时候，把脚本拆成多个文件
  - 代码复用
    - 不同程序调用同一个函数时，不用每次把函数复制到各个程序。

模块module

- 用法 demo_code › pizza.py

```
# pizza.py
def make_pizza(size, *toppings):
    print('Making a ' + str(size) + '-inch pizza with the following toppings:')
    for topping in toppings:
        print("- " + topping)
```

使用 import 语句导入刚创建的 pizza.py 模块，再调用模块内的 make_pizza() 函数两次：

```
from demo_code import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

import 用法

- 导入特定的函数

```
from module_name import function_name
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

```
from demo_code import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
from demo_code.pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

- as 函数别名

```
from module_name import function_name as fn
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

- as 函数别名

```
from module_name import function_name as fn
```

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from demo_code.pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

- as 函数别名

```
from module_name import function_name as fn
```

- 从模块中导入所有函数

```
from module_name import *
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

- as 函数别名

```
from module_name import function_name as fn
```

- 从模块中导入所有函数

```
from module_name import *
```

```
from demo_code.pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

import 用法

- 导入特定的函数

```
from module_name import function_name

from module_name import function_0, function_1, function_2
```

- as 函数别名

```
from module_name import function_name as fn
```

- 从模块中导入所有函数

```
from module_name import *
```

最好**不要**采用这种导入方法

要么只导入你需要使用的函数，
要么导入整个模块并使用句点表示法
让代码更清晰，更容易阅读和理解

import如何工作

- 程序第一次导入指定文件时，会执行三个步骤：
  - 找到模块文件
    - 首先，Python 必须搜索到 import 语句所引用的模块文件。
    - Python 解析器对模块文件的搜索顺序是：
    - 1、当前目录
    - 2、如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。
    - 3、如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为 /usr/local/lib/python/， Windows 下默认路径一般为：c:\python27\lib；
  - 编译成字节码（可选）
    - Python 会检查文件的时间戳，如果发现字节码文件比源代码文件要旧，就会在程序运行时自动重新生成字节码；反之，则跳过从源代码到字节码的编译步骤。
  - 运行模块代码
    - import 操作的最后步骤是执行模块的字节码。
    - 文件中所有语句都会从头至尾顺序执行。如果模块文件中顶层代码缺失做了什么实际的工作，你就会在导入时看见其结果。
    - 例如，模块内顶层的 print 语句会显示其输出；而函数的 def 语句只是简单地定义了稍后使用的对象。

模块的加载

- 为了保证运行效率，每次解释器会话只导入一次模块。
- 如果更改了模块内容，必须重启解释器；

```
#simple.py file
print('hello')
spam = 1
```

```
from demo_code import simple     # First import
simple.spam          # Assignment makes an attribute
```

```
1
```

```
simple.spam = 2          # Change attribute in module
from demo_code import simple      # Just feteches already loaded module
simple.spam             # Code wasn't rerun
```

```
2
```

模块

- 如何重载模块

```
#simple.py file
print('hello')
spam = 1
```

```
simple.spam = 2          # Change attribute in module
from demo_code import simple      # Just feteches already loaded module
# # simple.spam             # Code wasn't rerun
from imp import reload
reload(simple)
simple.spam
```

```
hello

1
```

模块需要注意

- 如果出现同样的变量名，怎么办？

```
# nested.py
X = 99
def printer():
    print(X)
```

```
# nested2.py
from demo_code.nested import X, printer # Copy names out
X = 88 # Changes my "X" only!
printer() # nested1's X is still 99
```

```
99
```

模块需要注意

- 如果出现同样的变量名，怎么办？

```
# nested.py
X = 99
def printer():
    print(X)
```

```
import demo_code.nested as nested # Get module as a whole
nested.X = 88 # OK: change nested's X
nested.printer()
```

```
88
```

模块需要注意

- **from module import ***
- **不建议这么做。**

```
from module1 import *    # Bad: may overwrite my names silently
from module2 import *    # Worse: no way to tell what we get!
from module3 import *

func()                   # Huh???
```

试着在 from 语句中明确列出想要的属性，而且限制在每个文件中最多只有一个被导入的模块使用 from * 这种形式。

模块 – 调试

- 如何对模块单独调试？

```
# runme.py
def tester():
    print("It's Christmas in Heaven...")
```

```
$ python runme.py
```

模块 – 调试

- 如何对模块单独调试？

```
# runme.py
def tester():
    print("It's Christmas in Heaven...")
```

```
$ python runme.py
```

- 以__name__进行单元测试

```
# runme.py
def tester():
    print("It's Christmas in Heaven...")

if __name__ == '__main__': # Only when run
    tester() # Not when imported
```

```
It's Christmas in Heaven...
```

在文件末尾加个__name__测试，把测试模块导出的程序代码放在模块中。

- 如果文件是以顶层程序文件执行，在启动时，__name__ 就会设置为字符串"__main__"
- 如果文件被他人当做模块导入，__name__就会改设成调用者所了解的模块名

模块 – 调试

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

```
1
6
```

这种写法的问题在于，每次这个文件被其他文件导入时，
都会出现调用自我测试所得的输出。

模块 – 调试

改进之后，我们在__name__检查区块内封装了自我测试的调用，使其在文件作为顶层脚本执行时才会启动，而在被导入时不会启动。

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Self-test code
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

模块 – 小节

- 当编写包含多个文件的较大 Python 系统时，设计理念就会变得重要起来：
  - 总是在 Python 的模块内编写代码
    - 封装
  - 模块间耦合性要降到最低
    - 除了与从其它模块导入的函数和类
    - 一个模块应该尽可能与其它模块的全局变量无关
  - 最大化模块的内聚性
    - 通过最大化模块内的内聚性来最小化模块间的耦合性。如果一个模块的所有元素都享有共同的目的，就不太可能依赖外部的变量名
  - 模块应该少去修改其它模块的变量
    - 应该试着通过函数参数返回值去传递结果，而不是跨模块的修改。
    - 否则全局变量的值会变成依赖于其它文件内的赋值语句的顺序，从而模块会变的难以理解和再利用

练习

1.写一个函数，判断其输入是奇数还是偶数

2.给定一个整数数组 nums 和一个目标值 target，请在该数组中找出和为目标值的那两个整数，并返回它们的数组下标。（假设每种输入有且只有一个答案。不能重复利用数组中同一个元素）

- 示例 1：给定 nums = [2, 7, 11, 15, 6], target = 9
  - 因为 nums[0] + nums[1] = 2 + 7 = 9，所以返回 [0, 1]
- 示例 2：nums = [3, 3], target = 6
  - 返回 [0, 1]

```
def twoSum(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: List[int]
    """
```

练习

3.实现 mySqrt(x) 函数，计算并返回 x 的平方根，其中 x 是非负整数。函数返回类型是浮点数，结果保留小数点后一位即可。

- 示例 1：输入: 4 输出: 2.0
- 示例 2：输入: 8 输出: 2.8
- 注：不允许使用内置 sqrt 函数、pow 函数和 x**0.5 方法