PYTHON程序设计

计算机学院 王纯

五 函数

五 函数

- 函数的定义和调用
- 函数的参数与返回值
- 函数的测试
- 变量的作用域
- 递归函数
- 函数式编程
- 综合示例
- 函数库

函数的定义和调用

Python的标准内置函数，我们已经接触了一些。下面我们主要讨论如何编写新的函数。

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

函数的定义和调用

函数是一种功能抽象

- ✓ 函数是一段具有特定功能的、可重用的语句组，用函数名来表示并通过函数名来完成功能调用。
- ✓ 函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要在每个执行地方重复编写这些语句。
- ✓ 函数的作用，是可以减少重复的代码，可以让程序***更短、更易读、更容易修改bug***。

```
def hello():
        print('嗨！‘ )
        print('你好！‘ )
        print('很高兴认识！‘ )
hello()
hello()
#重复调用时候，每一行就相当于函数中的三行。修改时也只需要修改函数中的内容，每次调用的逻辑就会跟着变化。
```

函数的定义和调用

定义函数:

def <函数名> (<参数列表>):

<函数体>

调用函数:

<函数名> (<参数列表>)

```
def hello(name):
    ''' say hello to name '''
    print("Hello ", name)

hello("World")
```

文档字符串

函数命名的注意事项

- ✓ 函数名称应该小写，以下划线分隔。提倡描述性的名称。
- ✓ 函数名称通常反映需要对参数应用的操作（例如print,add,square），或者结果（例如max,abs,sum）。
- ✓ 参数名称应小写，以下划线分隔。提倡单个词的名称。
- ✓ 参数名称应该反映参数在函数中的作用，并不仅仅是满足的值的类型。
- ✓ 当作用非常明确时，单个字母的参数名称可以接受，但是永远不要使用***l***（小写的***L***）和***O***（大写的***o***），或者***I***（大写的***i***）来避免和数字混淆。

函数的定义和调用

前向引用

Python中不允许在函数未声明之前对其进行引用或调用，但在函数定义中引用后面定义的函数是允许的，称为*前向引用*。

```
def func1():
    print("func1")
    func2()

def func2():
    print("func2")

func1()
```

编写良好的函数

- ✓ 每个函数都应该只做一个任务。这个任务可以使用短小的名称来定义，使用一行文本来标识。顺序执行多个任务的函数应该拆分在多个函数中。
- ✓ 不要重复劳动（***DRY-Don’t Repeat Yourself***）是软件工程的中心法则。原则上，逻辑应该只实现一次，指定一个名称，并且多次使用。如果你发现自己在复制粘贴一段代码，你可能发现了一个使用函数抽象的机会。
- ✓ 函数应该定义得通用一些，比如平方并未在 Python 库中，因为它是power函数的一个特例。

*将复杂的任务拆分为简洁的函数是一种技巧，需要**实践经验**的积累。*

函数的参数与返回值：形参、实参与参数传递

被操作的数据对象通过参数的形式传递给函数体。

- ✓ Python函数调用时采用赋值传递(***pass by assignment***) 的策略，与其它高级语言的*pass by value*或*pass by reference*不同，***形参变量=实参变量***， 即形参变量与实参变量指向同一个对象。
- ✓ 特殊的情况下实参的值有可能在函数内部被修改，如参数为可变类型，并且在函数内部使用下标或其他方式为可变序列增加、删除元素或修改元素值时， 同时也修改了实参。*这种情况可通过切片调用避免*。

```
def func1(n):       #n为形参 (parameter)
    n = 20
    print(n)
a = 10
func1(a)       #a为实参 (argument)
def func2(n):
    n[1] = 10      #修改了参数中的元素!
    print(n)
a = [1,2,3]
func2(a)   #实参为列表对象，存在被修改的风险
func2(a[:])   #切片后调用，避免被篡改
```

函数的参数与返回值：参数传递方式

Python允许定义多个不同类型的参数，参数传递的规则多样：

- ✓ 根据调用方式区分：
  - ✓ 按位置传递(***positional argument***)：当一个函数有多个参数时，实参默认按位置顺序传递给形参。
  - ✓ 按名称传递(***keyword argument***)：Python允许指定参数的名称，也称为关键字实参。**实参与形参顺序可以不一致。**
- ✓ 根据定义方式区分：
  - ✓ 允许给函数的参数指定默认值 (***default argument***)，否则就是非默认参数。
  - ✓ 可变参数：函数还可以接受非固定数目的参数。一种是在参数名前加**1个***；一种是在参数名前加**2个***。

```
#函数定义
def func1(a,b,c=1):
    print(a,b,c)
#按位置传递调用
func1(1,2,3)
#按名称传递调用
func1(b=2,a=1,c=3)
#使用默认值
func1(b=2,a=1)
```

函数的参数与返回值：参数的顺序问题

✓函数调用时实参顺序：

位置参数->关键字参数

✓函数定义时形参顺序：

非默认参数->默认值参数

为什么这样设计？ *函数调用时默认值参数可能并没有包含在实参中！另外，关键字参数可以区分多个默认值参数。*

```
def func1(a,b,c):
    print(a,b,c)
func(b=1,2,c=3)   #错误的实参顺序
#SyntaxError: positional argument follows keyword argument
func(1,c=3,b=2)  #正确的实参顺序
def func1(a=3,b,c=2):    #错误的形参顺序
    print(a,b,c)
#SyntaxError: non-default argument follows default argument
```

函数的参数与返回值：参数默认值的隐患

- ✓ 如果默认值参数的默认值为**可变对象**，且函数体中原地修改了该对象，默认值也会被修改
- ✓ 这种特性可能导致很难发现的逻辑错误。如Digg著名的V4升级失败案例
- ✓ *也可能是故意这样来设计在函数的多次调用间保存状态信息*

注意： 多次调用函数并且不为默认值参数传递值时，默认值参数值在调用时被更改， 可以使用函数名。__defaults__查看默认参数的当前值如何修改?

```
>>> def demo(newitem, old_list=[]) :
        old_list. append(newitem)
        return old_list
>>> print (demo(' 5' , [1, 2, 3, 4]) ) #right
[1, 2, 3, 4, ' 5' ]
>>> print (demo(' aaa' , [' a' , ' b' ]) ) #right
[' a' , ' b' , ' aaa' ]
>>> print (demo(' a' ) ) #right
[' a' ]
>>> print (demo(' b' ) )
[' a' , ' b' ]
>>> print (demo. __defaults__)
([' a' , ' b' ], )
```

def get_user_by_ids(ids=[]) #Digg的问题API

函数的参数与返回值：参数默认值的隐患

修改前面的函数，参照以下原则：

- ✓ 默认值参数的默认值避免采用可变对象
- ✓ 采用可变对象但是函数体不要进行原地操作
- ✓ 除非特意设计以利用可变对象的这一特性，但需要充分考虑程序的副作用和由此带来的可维护性问题

```
def demo(newitem, old_list=None) :
    if old_list is None:
        old_list=[]
    old_list. append(newitem)
    return old_list
print (demo(' 5' , [1, 2, 3, 4]) ) #right
print (demo(' aaa' , [' a' , ' b' ]) ) #right
print (demo(' a' ) ) #right
print (demo. __defaults__)
print (demo(' b' ) )
```

函数的参数与返回值：可变参数

Python中函数可以接受非固定数目的参数，有两种形式：

- ✓ 参数名前加***1个****，将尚没有匹配形参的位置实参组合成元组赋值给该参数
- ✓ 参数名前加***2个****，将尚没有匹配形参的关键字实参组合成字典赋值给该参数

- 如果存在， ***args**形式的参数应该出现在默认值参数之后
- 如果存在， ****kwargs**形式的参数应该出现在最后
- 每种类型的可变参数只能出现一次

```
>>> def func1(*a):
        print(a)
>>> func1(1,2,3)
(1, 2, 3)
>>> func1(1,2)
(1, 2)
>>> def func2(**p):
        print(p)
>>> func2(a=1,b=2,c=3)
{('a', 1), ('b', 2), ('c', 3)}
```

函数的参数与返回值：仅允许名称传递的形参

keyword-only参数，即这些参数只能通过关键字参数的形式来传递

- ✓ keyword-only参数出现在可变长度的位置参数之后
- ✓ 如果没有可变长度位置参数， 则添加一个*
- ✓ keyword-only参数也可以指定默认值
- ✓ 调用时必须采取关键字参数传递， 或者不传递表示采用默认值

```
>>>def kwonly(a, *b, c, d=5) :
        return (a, b, c, d)
>>>def kwonly2(a, b, *, c, d=5) :
        return (a, b, c, d)

>>>print(kwonly(1, 2, c=3) ) #使用默认值

>>>print(kwonly2(1, 2, c=3,d=4) )
```

函数的参数与返回值：形参与实参的匹配

- ✓ 函数定义中的形参顺序：**位置参数→默认值参数→可变长度位置参数→仅允许关键字传递参数→可变长度关键字参数**
- ✓ 函数调用中的实参顺序: **位置参数→关键字参数**
- ✓ 函数调用时， 根据传递的实参按照以下规则匹配形参：
  - 首先按照位置匹配形参中的位置参数和默认值参数
  - 接下来根据关键字参数的名字匹配形参中的各个参数
  - *剩下的位置参数组成**tuple**赋值给可变长度位置参数*
  - *剩下的关键字参数组成**dict**赋值给可变长度字典参数*
  - *函数定义中**尚未匹配**的参数设置为默认值*
  - 如果仍然有尚未匹配的形参和实参则报错
  - 每个参数只能匹配一次

```
>>>def params(a, *b, c, d=5, **e) :
    return (print(a,b,c,d,e))
>>>params(1,2,3,4,c=5,f=6,k=8,j=9)
1 (2, 3, 4) 5 5 {'f': 6, 'k': 8, 'j': 9}
```

函数的返回值

- ✓ 定义函数时不需要声明函数的返回值类型；
- ✓ 函数返回值类型与**return**语句返回的表达式类型一致；
- ✓ 没有**return**语句时函数的返回值都为**None**，即返回空值；
- ✓ 可以返回元组类型，类似返回多个值。

```
def func1():
    print("Hello World")
def func2():
    return 1,2,3
```

关于函数设计的补充

一种可参考的函数参数设计的准则：

一：参数必须是某个基础类型

二：对于需要比较复杂的数据的情况，可以使用嵌套的元组、字典等进行传递

三：不要修改或保存传入的数据

四：重要的参数在前，次要的参数在后且有默认值

参数设计带来的解耦

- ✓ 并非不要用面向对象，不要设计类，而是相反
- ✓ 尽量不要把一个类的实例当作另一个类的参数来使用，尤其当它们属于不同的模块的时候。这就带来了模块之间的解耦，从：**类 <=> 类** 变成了 **类 <=> 最小数据集合 <=> 类** 的关系
- ✓ **最小数据集合**也可以理解为一种**协议**，它清晰地描述了我们当前问题中，需要传递哪些信息。这个集合很容易扩展，只要在参数列表末位添加新的带有缺省值的参数就可以了
- ✓ 便于扩展到其他应用场景

函数的测试

函数的测试是验证函数的行为是否符合预期的操作。当我们的函数足够复杂的时候，我们需要测试我们的实现。

- ✓ 测试是一种系统化执行验证的机制。 通常编写另一个函数，这个函数包含一个或多个被测函数的样例调用，调用后的返回值会和预期结果进行比对。
- ✓ 与大多数通用的函数不同，测试涉及到挑选特殊的参数值，并使用它来验证调用。
- ✓ 测试也可作为文档：描述如何调用函数，以及什么参数值是合理的。

常用的函数测试方法：

- *文档字符串(docstring)*
- *断言(assert)*

高效测试的关键是在实现新的函数之后（甚至是之前）立即编写（以及执行）测试。只调用一个函数的测试叫做**单元测试**。详尽的单元测试是良好程序设计的标志。

函数的测试：文档字符串

编写 Python 程序时，除了最简单的函数之外，都要包含**文档字符串**。请注意，***代码只编写一次，但是会阅读多次***。Python文档包含了*文档字符串准则（PEP257）*，是一种事实的标准。

Doctest。Python提供了一个便利的方法，将简单的测试直接写到函数的文档字符串内。文档字符串的第一行应该包含单行的函数描述，参数和行为的详细描述一般跟随在后面，包括测试用例的描述。

这样就可以使用 *doctest* 模块来测试函数。*globals*函数返回全局变量的表示，解释器需要它来求解表达式。

```
def sum_naturals(n):
    """Return the sum of the first n natural numbers

    >>> sum_naturals(10)
    55
    >>> sum_naturals(100)
    5050
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + k, k + 1
    return total

>>> from doctest import run_docstring_examples
>>> run_docstring_examples(sum_naturals,
globals())
```

函数的测试：断言

断言assert

语法：***assert <布尔表达式> [, '表达异常信息的文本']***

通常使用assert语句来验证预期，测试函数的输出是其中的一种应用。assert语句在只有一个布尔表达式，后面是带引号的一行文本。当被断言的表达式求值为真时，断言语句的执行没有任何“效果”；当它为假时，assert会造成执行中断，并显示语句中的文本。

```
assert fib(8) == 13, 'The 8th Fibonacci number should be 13’

AssertionError: The 8th Fibonacci number should be 13

def fib_test():
    assert fib(2) == 1, 'The 2nd Fibonacci number should be 1'
    assert fib(3) == 1, 'The 3nd Fibonacci number should be 1'
    assert fib(50) == 7778742049, 'Error at the 50th Fibonacci number'
```

变量的作用域：全局变量和局部变量

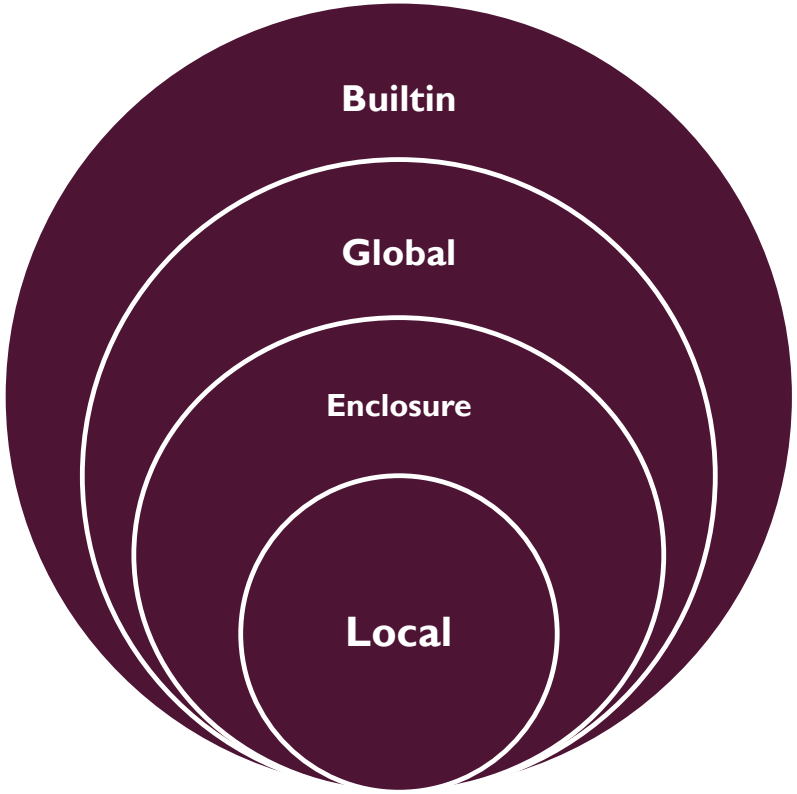
变量在程序执行过程中起作用的范围称为**变量作用域**。

- ✓ 只有*定义（赋值）*、*global*和*nonlocal*才会建立或者改变变量的作用域
- ✓ **局部变量**：在函数内部定义的普通变量，有效范围为函数体
- ✓ **全局变量**：一般在函数外定义，通常没有缩进，在程序的执行全程有效
- ✓ 在函数内部定义或修改全局变量必须使用关键字*global*
  - 如果一个变量已在函数外定义，在函数内需要为这个变量赋值，可以在函数内使用global关键字将其声明为全局变量；
  - 如果一个变量在函数外没有定义，在函数内部同样也可以使用global关键字直接将一个变量定义为全局变量，该函数执行后，将*增加一个新的全局变量*；

```
a = 1 #全局变量
def func1():
    b = 10 #局部变量
    global c #在函数内部声明的全局变量
    c = 20
    print("Local:",b)
```

变量的作用域：LEGB法则

- ✓ Python引用变量的顺序依次为***局部作用域、嵌套作用域、全局作用域和内置作用域***。这就是所谓的LEGB法则；
- ✓ 函数内部声明的标识符具有***局部作用域***，其优先级最高；
- ✓ ***嵌套作用域***也是在函数内部，它和局部作用域是相对的，嵌套作用域相对于更上层的函数而言是局部作用域，相对于它内部的函数则是嵌套作用域；
- ✓ ***全局变量***具有全局作用域；
- ✓ 系统内固定模块里定义的标识符具有***内置作用域***。



变量的作用域：LEGB法则

在函数内部，如果不修改全局变量（该变量也没有被局部变量覆盖），只是读取全局变量的值，则可以正常使用全局变量。此规则对于嵌套作用域也是同样适用：

```
>>> a = 1 #全局作用域
>>> def func():
        b = 10 #嵌套作用域
        def func1():
            print(a) #局部作用域
            print(b) #局部作用域
```

变量的作用域：LEGB法则

在Python中当在某个作用域内任意位置只要有为变量赋值的操作，该变量在这个作用域内就是局部变量（***赋值改变作用域***）。如果要在局部作用域内修改全局作用域或嵌套作用域内变量，就需要事先声明。修改全局作用域内的变量要用关键字***global***声明，修改嵌套作用域内的变量要用关键字***nonlocal***声明。

```
def func():
    def func1():
        nonlocal a #这里要求a必须是已存在的变量
        a = 2 #嵌套作用域变量赋值
    def func2():
        global a #如果全局作用域内没有a，就自动新建一个
        a = 3 #全局变量赋值
    a = 1 #嵌套作用域变量初始值
```

变量的作用域：闭包

允许在函数内部创建另一个函数

```
def linear(a, b) :
    def result(x) :
        return a * x + b
    return result
func_1_1 = linear(1, 1)
func_2_1 = linear(2, 1)
print(' f(x) = x+1, f(10) = ', func_1_1(10) )
print(' f(x) = 2x+1, f(10) = ' , func_2_1(10))
```

闭包(closure)

- ✓ 闭包函数必须返回一个函数对象
- ✓ 闭包函数返回的那个函数必须引用外部变量（一般不能是全局变量），而返回的那个函数内部不一定要return

- 示例中外部的linear函数对象中的***__closure__***属性，包含了闭包引用的外部变量。
- 若主函数内的闭包不引用外部变量，则不存在闭包，主函数***__closure__***属性永远为***None***
- 若主函数没有***return***子函数，则不存在闭包，主函数不存***__closure__***属性
- 闭包在被返回时，它的所有变量就已经固定，形成一个**封闭的对象**，这个对象包含了其引用的所有外部、内部变量和表达式

变量的作用域：闭包

循环体内定义的函数无法保存循环执行过程中不停变化的外部变量

```
funcs = []
for i in range(3):
    j = [i]
    print(hex(id(j)), '  ', j)
    def func(n):
        print(n * j[0])
        print(hex(id(j)), ' in func  ', j)
    funcs.append(func)
print (funcs)
[f(10) for f in funcs]
```

```
0x252bb373d80    [0]
0x252bb373e40    [1]
0x252bb373cc0    [2]
[<function func at 0x00000252BB37F430>, <function func at 0x00000252BB37F550>,
 <function func at 0x00000252BB37F4C0>]
20
0x252bb373cc0  in func   [2]
20
0x252bb373cc0  in func   [2]
20
0x252bb373cc0  in func   [2]
```

```
funcs = []
for i in range(3):
    def func(i):
        j = [i]
        print(hex(id(j)), '  ', j)
        def f_clo(n):
            print(n * j[0])
            print(hex(id(j)), ' in func  ', j)
        return f_clo
    funcs.append(func(i))
print (funcs)
[f(10) for f in funcs]
```

```
0x252a737fec0    [0]
0x252bb373d80    [1]
0x252bb373300    [2]
[<function func.<locals>.f_clo at 0x00000252BB37F4C0>, <function func.<locals>
 .f_clo at 0x00000252BB37F430>, <function func.<locals>.f_clo at
 0x00000252BB37F550>]
0
0x252a737fec0  in func   [0]
10
0x252bb373d80  in func   [1]
20
0x252bb373300  in func   [2]
```

变量的作用域：闭包

闭包的作用

- 保存运行状态
- 惰性求值
- 修饰器的基础

可应用于数值计算、处理。。。

```
55  def create(pos=origin):
56
57      def go(direction,step):
58          new_x = pos[0] + direction[0]*step
59          new_y = pos[1] + direction[1]*step
60          pos[0] = new_x
61          pos[1] = new_y
62          return pos
63
64      return go
65
66  player = create()
67  print player([1,0],10)
68  print player([0,1],20)
69  print player([-1,0],10)
```

变量的作用域：修饰器

修饰器（Decorator）的一般实现形式

```
def wrapper ( func ) :
    def helper ( *args, **kwargs ) :
        # 修饰器本身的功能实现
        。。。
        return func ( *args, **kwargs )
    return helper
```

此处的函数嵌套可以在完成自身逻辑后完整的返回作为外层函数参数的函数对象及其调用的全部参数（可变参数可覆盖所有可能），并且能够获得func(*,**)调用的全部外部信息，因此修饰器的作用就是基本无损的与其他函数叠加。

完成修饰器函数与被修饰函数的捆绑可以这样做

```
func = wrapper ( func )
```

Python中还可以更方便的这样写（使用修饰器语法符号@）

```
@wrapper
def func ( ) :
 # 被修饰函数的函数体
```

递归函数：概念

从递推到递归：

- ✓ 最初级的代码复用就是基于***for/while***这样的递推(迭代)，其策略在减少代码行的重复
- ✓ 难以避免变量的副作用。尽可能减少这样的副作用的一个方法就是让函数自我引用，即**递归**
- ✓ 递归并非没有副作用，也不与函数式编程等同，但也是**函数式编程**中实现循环的一种基本操作
- ✓ 递归的典型典型场景：数据定义是递归的（fibonacci）；问题求解过程是递归的（Hanoi）；数据结构是递归的（二叉树遍历）

```
def power1(x: float, n: int) -> float:
    res = 1
    for i in range(n):
        res *= x
    return res
```

```
def power2(x: float, n: int) -> float:
    if n == 0:
        return 1
    else:
        return x * power2(x, n - 1)
```

递归函数：定义和特性

如果一个函数中包含自身的引用，那么我们就说这个函数是递归的。

- ✓ 基线条件（针对最小的问题）： 可以存在一个或多个基例，满足条件时直接返回确定值；
- ✓ 递归条件： 包含一个或多个调用， 这些调用旨在解决问题的**一部分**。

- ✓ 递归可实现的程序均可采用循环实现，绝大多数情况下循环效率更高。但递归可读性好，实现更简洁容易，当然还有栈溢出可能。
- ✓ 函数调用是通过栈实现，每进入一次函数调用，就会加一层栈帧，每当函数返回，就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

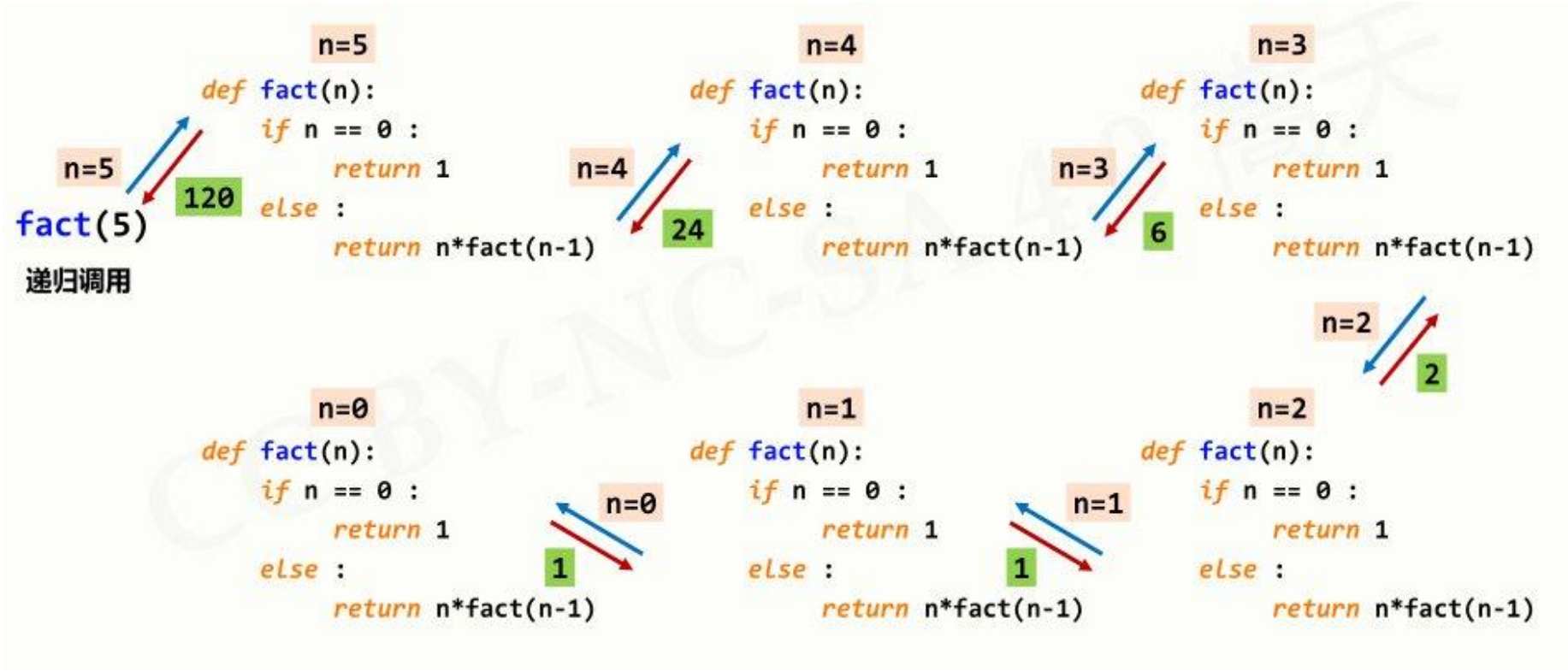
#字符串反转示例

```
def rvs(s):
    if s == "" :
        return s
    else :
        return rvs(s[1:])+s[0]
```

递归函数：执行过程

示例：阶乘　$n! = n\,(n-1)(n-2).....(1)$

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & otherwise \end{cases}$$



递归函数：示例

二分查找的例子
Python提供了***bisect***模块提供序列的二分查找功能，性能好于***index***，实际上是二分查找与顺序查找的区别。

```
def search(sequence, number, lower, upper):
    if lower == upper:
        assert number == sequence[upper]
        return upper
    else:
        middle = (lower + upper) // 2
        if number > sequence[middle]:
            return search(sequence, number, middle + 1, upper)
        else:
            return search(sequence, number, lower, middle)
```

递归函数：效率问题

- ✓ 递归在执行效率上不能让人满意的是延迟计算和由此导致的栈空间的占用和堆栈操作的时间消耗。
- ✓ 如右图，以前面的幂运算为例，***power2***函数被调用后需要到最后算到***power2(x, 1)***的时候才能真的发生运算，之前都是在做表达式展开（或者说堆栈）的工作。

```
power2(2.0, 5)
2.0 * power2(2.0, 4)
2.0 * (2.0 * power2(2.0, 3))
2.0 * (2.0 * (2.0 * power2(2.0, 2)))
2.0 * (2.0 * (2.0 * (2.0 * power2(2.0, 1))))
2.0 * (2.0 * (2.0 * (2.0 * (2.0 * power2(2.0, 0)))))
2.0 * (2.0 * (2.0 * (2.0 * (2.0 * 1.0))))
2.0 * (2.0 * (2.0 * (2.0 * 2.0)))
2.0 * (2.0 * (2.0 * 4.0))
2.0 * (2.0 * 8.0)
2.0 * 16.0
32.0
```

递归函数：优化

一个解决方案就是让它**立即求值**，对于求值优先而不是展开优先的过程来说（减少了堆栈的部分时间），就能提升效率。此时我们需要一个保存临时值的量，来保存即时计算的结果。如下

```
def power3(x: float, n: int, acc: float = 1.0) -> float:
    if n == 0:
        return acc
    else:
        return power3(x, n - 1, acc * x)
```

*程序的执行效果如右图所示，称为**尾递归**。尾递归成立的必要条件是*

递归调用在函数结尾

```
power3(2.0, 5, 1)
power3(2.0, 4, 1 * 2.0)
power3(2.0, 4, 2.0)
power3(2.0, 3, 2.0 * 2.0)
power3(2.0, 3, 4.0)
power3(2.0, 2, 4.0 * 2.0)
power3(2.0, 2, 8.0)
power3(2.0, 1, 8.0 * 2.0)
power3(2.0, 1, 16.0)
power3(2.0, 0, 16.0 * 2.0)
power3(2.0, 0, 32.0)
32.0
```

递归函数：优化

在**尾递归**的基础上优化递归调用的**栈帧操作**

- ✓ 优化原理：当递归函数被该修饰器修饰后，递归调用在修饰器while循环内部进行，每当产生新的递归调用栈帧时***f.f_back.f_back.f_code == f.f_code***就捕获当前尾调用函数的参数，并抛出异常，从而销毁递归栈并使用捕获的参数手动调用递归函数，所以递归的过程中始终只存在一个栈帧对象，达到优化的目的。
- ✓ 使用该优化函数去修饰尾递归函数就可以实现尾递归优先求值的效果。

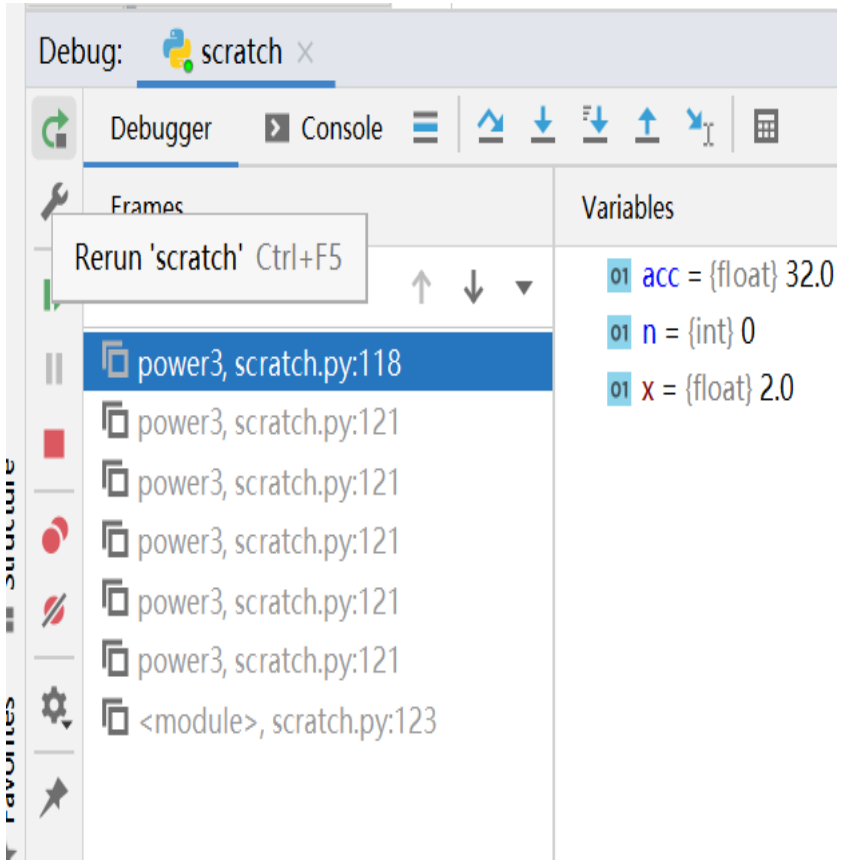
@tail_call_optimized

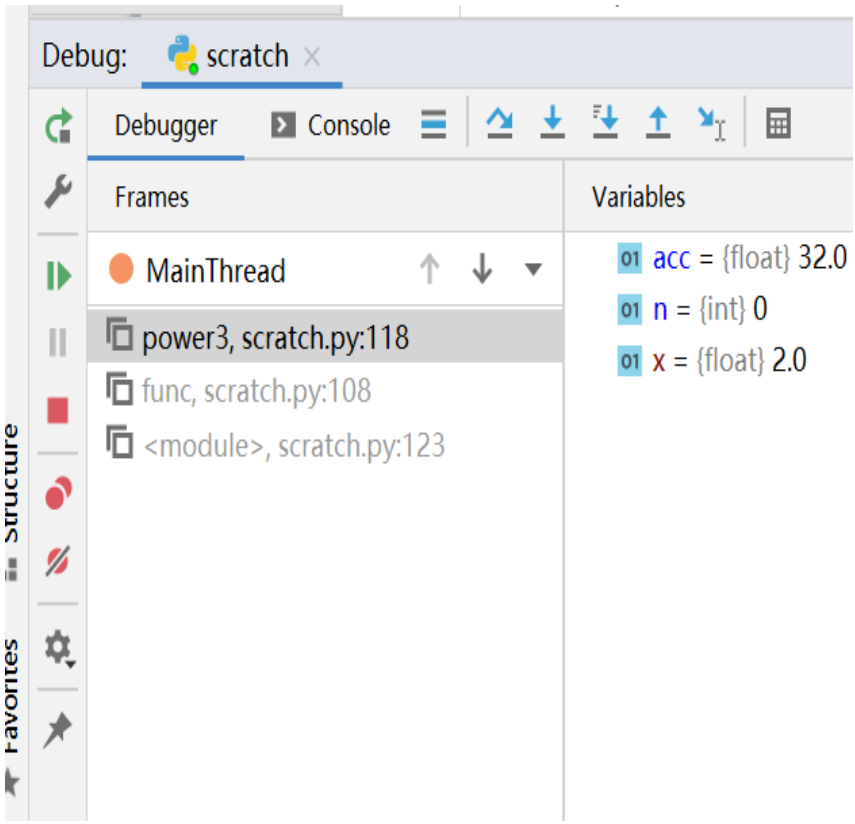
def power3():

```
import sys

class TailRecurseException(BaseException):
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code
== f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func
```

递归函数：优化

优化前后的运行栈对比（右图为优化后）





函数式编程

内置函数sorted可以提供排序功能，其语法格式为：

sorted(iterable, key=None, reverse=False)

其中iterable为可迭代对象，也是被排序的对象；

key可以接受一个函数，用来指定排序规则，缺省时直接比较可迭代对象的元素；

reverse为True时表示按降序排列，为False时表示按升序排列。

```
>>> countries = ['Canada', 'China', 'Cuba','Germany','United Kingdom','United States']
>>> sorted(countries,key=len)
['Cuba', 'China', 'Canada', 'Germany', 'United States' ,
'United Kingdom']
```

函数式编程：LAMBDA表达式

lambda表达式用来创建一个匿名函数，因此匿名函数又叫lambda函数。其语法格式如下：

<函数名> = lambda <参数列表>:<表达式>

lambda函数与正常函数是一样的，其等价于下面的形式：

```
def <函数名> (<参数列表>):
    return <表达式>
```

```
>>>f = lambda x,y,z : x + y + z
>>>f(1,2,3)
6
>>>f(4,5,6)
15

>>> students = [ ('20181213001', 'Li', 'M', 21,
'CS'),('20181213002', 'Liu', 'F', 20, 'CS'),
('20181213003', 'Wang', 'F', 19, 'IS'),('20181213004',
'Zhang', 'M', 20, 'IS'),
('20181213005', 'Chen', 'M', 19, 'IS') ]
>>> sorted(students, key=lambda student: student[1])
[('20181213005', 'Chen', 'M', 19, 'IS'), ('20181213001', 'Li',
'M', 21, 'CS'), ('20181213002', 'Liu', 'F', 20, 'CS'),
('20181213003', 'Wang', 'F', 19, 'IS'), ('20181213004',
'Zhang', 'M', 20, 'IS')]
```

函数式编程：LAMBDA表达式

世界杯小组赛的32支参赛队分为八个小组，每组四队进行比赛。每支球队都必须和其他三支球队进行且只进行一场比赛，胜者得三分，负者不得分，打平双方各得一分。小组排名规则的前3条如下：

a、积分高者排名靠前；

b、小组中总净胜球（总进球数减去总失球数）高者排名靠前；

c、小组中总进球数高者排名靠前。

```
>>> Teams = [ ('Iran', 2, 2, 4),('Morocco', 2, 4, 1),('Portugal', 5, 4, 5),('Spain', 6, 5, 5)]
>>> sorted(Teams,key = lambda x :(x[3],x[1]-x[2],x[1]),reverse = True)
[('Spain', 6, 5, 5), ('Portugal', 5, 4, 5), ('Iran', 2, 2, 4), ('Morocco', 2, 4, 1)]
```

函数式编程：LAMBDA表达式

lambda表达式嵌套

<函数名> = lambda <参数列表>：lambda <参数列表>：<表达式>

显然等同于下面的函数定义，其实相当于一个闭包

```
def f1(*x):
    def f2(*y):
        return expression with any in x,y param list
    return f2
```

```
#示例
square = lambda x: x**2
product = lambda f, n: lambda x: f(x)*n
ans = product(square, 2)(10)
```

函数式编程：函数的副作用

显式(***Explicit***)与隐式(***Implicit***)*数据流交换*：

- ✓ 函数与外部交换数据只有一个渠道——参数和返回值，即**显式**数据流。
- ✓ 函数通过参数和返回值以外的渠道，和外部进行数据交换。比如读取/修改全局变量，都叫作以**隐式**的方式和外部进行数据交换。

具有隐式交换的函数可能有**副作用**，即影响函数行为的确定性。

```
def f(x):                    #无副作用
    return x + 1

def f(ls, a):                #有副作用（列表的原地操作）
    ls.append(a)
    return ls
```

函数式编程：函数的副作用

副作用的***利弊***：

1. 回溯问题

无副作用函数更容易找到问题、定位问题和复现问题，也意味着非常强的**可测性**。甚至对于一个静态的函数式语言（可惜Python不是），编译阶段就能暴露和解决绝大部分的问题。

2. 无法和环境交互的程序大概率没有应用价值

纯函数式的概念，构造的是一个逻辑符号运算系统。如果没有和外部环境交互，则欠缺实用性，甚至使用*print*都是在产生一个函数外的屏幕的副作用。因此，无副作用也意味着它的应用很难。当然把副作用限缩在一个非常小的范围里的方法可以提高对程序的把握并兼顾交互，是更实用的编程思路。

函数式编程：函数的副作用

3. 效率

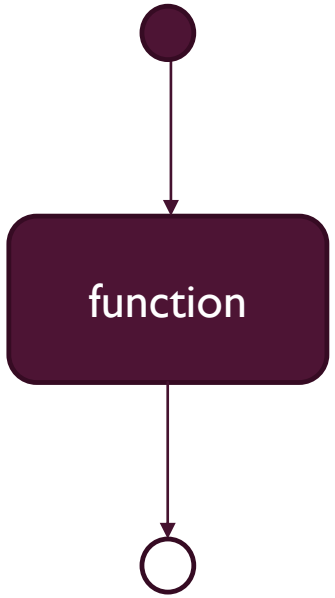
计算机本身的概念是基于**环境**或者说基于**副作用**的。而函数式编程在一个副作用机器上实现，本身就会效率下降。更何况，如果我们不使用类似*append*之类的原地操作，这就意味着更多空间，更多的值的复制，这些都让程序的效率大打折扣。此外，Python对诸如递归等函数式特点的程序优化效果并不好，这也使得副作用可能更容易让人青睐。

4. 表达能力

使用**环境**的概念衍生出很多概念，譬如**传值**、**传址**、**可变变量**、**全局变量**，并且这些概念是必须内生在语言内的。某种意义上，这是表达能力弱的体现。事实上，函数式编程仅靠值和函数两个概念，以及基本的类型、运算就能实现几乎所有的事（或者说图灵完全的）。

函数式编程：编程范式

那些接受函数作为参数的函数或者把函数作为返回结果的函数叫**高阶函数**（***Higher-order function***）。函数式编程就是指这种高度抽象的编程范式。



函数式编程思维-水管模型

编造一系列的水管，水管就是函数式中的函数。我们的目标就是事先将各种函数水管架设成管道系统，然后将水（**数据/不可变参数**）倒入，等待水管的另一头流出结果。

函数式编程：编程范式

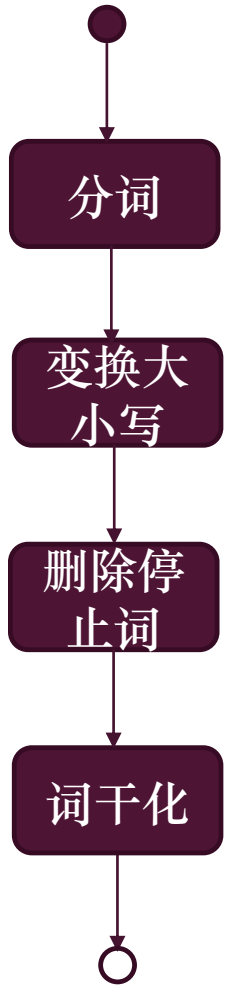
函数的组合（*Compose*）

对于复杂一点的情况需要将多个不同的水管相连。以一个文本处理的任务为例，如右图需要分词、变换大小写、删除定冠词之类的停止词、提取词干等函数串接执行。

这个操作也被称为***Compose***（符号∘）数学表述如下：

$$(f \circ g)x = f(g(x))$$

下面的函数起到了组合的效果 *args*是要参与组合的函数




```
def and_then(*args):
    return reduce(lambda f, g: lambda x: g(f(x)), args)
```

函数式编程：一等对象

在 Python 中，函数是**一等对象**。编程语言学者把“一等对象”定义为满足下述条件的程序实体：

- ✓ 在运行时创建
- ✓ 能赋值给变量或数据结构中的元素
- ✓ 能作为参数传给函数
- ✓ 能作为函数的返回结果

> 不管别人怎么说或怎么想，我从未觉得*Python*受到来自函数式语言 的太多影响。我非常熟悉命令式语言，如 *C* 和 *Algol 68*，虽然我把函数定为一等对象，但是我并不把*Python*当作函数式编程语言。
>
> ——*Guido van Rossum 仁慈的独裁者*

函数式编程：一等对象

函数作为参数和返回值的好处是可以对函数本身进行抽象处理。

比如计算两点间距离

```
def distance_between_two_ls(ls1, ls2, distance_func):
    return [distance_func(i, j) for i, j in zip(ls1, ls2)]
```

两个实用的特性：提前计算和延迟计算：

- ✓ 我们需要靠one()来得到实际的数值1，这个就是最小的**延迟计算**的例子。
- ✓ **科里化**（***Currying***）-这种函数嵌套的写法是闭包的一种形式，其另外一个好处，就是**提前计算**。

```
#延迟计算
one = lambda : 1

one()

#提前计算
def f(x):
    def f_(y):
        z = x ** 2 + x + 1
        return y * z
    return f_

f1 = f(1)
# 调用
f1(2)
f1(3)
```

函数式编程：一等对象

高阶函数的优势：

- ✓ 仅需要函数和值的概念，就能保存状态，而不需要更多诸如实例化、对象、可变变量、属性等概念
- ✓ 可以延迟计算，体现最基本的惰性计算的概念
- ✓ 可以提前计算，更高效率地复用代码

Callable：Callable的对象是指可以被调用执行的对象，并且可以传入参数，包括：

- ✓ 函数
- ✓ 类
- ✓ 类里的函数
- ✓ 实现了__call__方法的实例对象

函数式编程：示例

定义自然数

要求：仅用字符串“e”，函数，if-else分支，==“e”运算，这四个概念来实现一个自然数的概念（实际中还用到了bool值，不过bool值本身也可以用"e"和f("e")表示）。

思路：引入皮亚诺公理构建自然数生成规则，应用函数式编程范式进行实现。皮亚诺公理定义的是无限可数集的概念。

皮亚诺公理定义

1 $e \in S$ 表示我们需要一个初始值，在这个符号集里叫e。对应于自然数的**1**的概念

2 $(\forall a \in S)(f(a) \in S)$ 表示往下数一个数的操作，这个符号集里用ff表述，也把这个操作叫**后继**。对应自然数中**加一/往下数一**的概念

3 $(\forall b \in S)(\forall c \in S)(f(b) = f(c) \rightarrow b = c)$ 确定恒等关系

4 $(\forall \in S)(f(a) \neq e)$ 确定e不是任何数的后继，保证它是第一个被数的数

5 $(\forall A \subseteq S)(((e \in A) \wedge (\forall a \in A)(f(a) \in A)) \rightarrow (A = S))$ **归纳公理**--如果A是包含自然数的集合，且A内所有整数的后继数也在A内，则A包含了所有自然数

函数式编程：示例

实现

仅两行代码实现自然数的全部定义:我们使用递归表示向下数数，用 **“e”** 表达起始值1 。我们可以快速地给数字命名如下，不难看出每个后继数字以函数形式存在，其中保留了前序数字的信息。

比如 **two()** 就是 **one** 也就是**"e “**

这样，我们就可以实现**判断相等**、**加法**、**乘法**这三个最简单的操作了

```
#示例

one = "e" # 1
f = lambda x: lambda : x # 后继


one = "e"
two = f(one)
three = f(two)
four = f(three)
。 。 。
seventeen = f(sixteen)
eighteen = f(seventeen)
nineteen = f(eighteen)
```

函数式编程：示例

判断全等：我们将两个函数无限地求值（前序）下去，看到最后是不是同时得到**"e"**值，这对应了性质**3**

注意在上面的实现中使用了**x ="e"** 这种前面带空格而后面不带空格的写法，其实是为了强调**="e"**是一个一元运算，我们仅使用它而不需要其他概念。仔细探究这个算式，我们发现其实它隐式地用到性质**4**，只要一个不为 **“e”** 我们就可以确定它们是不相等的。

然后是**加法、乘法和幂运算**

```
def equal(x, y) -> bool:
    if x =="e" and y =="e":
        return True
    elif x =="e":
        return False
    elif y =="e":
        return False
    else:
        return equal(x(), y())

not_equal = lambda x, y: not(equal(x, y))
```

```
def power(x, y):  #幂运算
    if y =="e":
        return x
    else:
        return multiply(power(x, y()), x)
```

```
def add(x, y):   #加法
    if y =="e":
        return f(x)
    else:
        return add(f(x), y())
def multiply(x, y):  #乘法
    if y =="e":
        return x
    else:
        return add(multiply(x, y()), x)
```

函数式编程：示例

最后，验证一下

```
>>> assert equal(add(two, one), three)
>>> assert not_equal(power(two, three), seven)
>>> assert equal(power(two, three), eight)
>>> assert equal(multiply(three, five), fifteen)
```

函数式编程的魅力在于：

- ✓ 我们可以用非常少的概念（在这个例子中是4个）自举实现足够多的事情。这个也是早期**LISP**语言会那么在AI领域或者一些对语言内核大小非常敏感的领域受青睐的原因。
- ✓ 因为函数式编程中的函数和数学上的函数非常接近，这使得在数学上使用的代数运算可以非常方便的实现

函数式编程：LAMBDA演算和Y算子

```
>>> power_of_2 = lambda x: 1 if (x == 0) else 2 * power_of_2(x - 1)
>>> power_of_2(5)
32
```

如何用lambda演算实现上述的递归调用？Curry提出**组合子**（***Y Combinator***）

f(g)=g(f(g))=g(g(f(g)))=...

(λx.N)g=g((λx.N)g)

Y=λf.(λx.f(xx))(λx.f(xx))

从下面幂运算函数的写法可以看到，通过Y组合子使用lambda演算即可实现**递归**。因此，可以论证**递归**在函数式编程语言中可以是**派生**的，而不需要**原生**支持。这个例子让我们进一步看到函数式编程的表达能力以及和数学特别是抽象代数、符号逻辑的联系。

```
Y = lambda f: (lambda g: f(g(g)))(lambda g: f(lambda y: g(g)(y)))
power_of_2 = Y(lambda f: lambda x: 1 if (x == 0) else 2 * f(x - 1))
```

综合示例：基于元组实现一个列表数据类型

问题

实现一个列表数据类型及其关键操作，要求不用类来实现，而在基本的数据类型二元元组*(a, b)*和空元组*()*的基础上实现

分析

- ✓ 自举产生一个列表类型，关键是创建一个模式，可以无穷展开自己，保存一个值和下一个数据。例如*[1, 2, 3, 4]*我们可以用*(1, (2, (3, (4, ()))))*
- ✓ 必须指定一个结尾，这个就是*()*在其中的作用，*()*同时代表空列表和列表结尾的含义。

综合示例：基于元组实现一个列表数据类型

实现

- ✓ 首先将列表**定义**如下（这里闭包的形式是为将数据隔离，防止我们使用内置的比较来实现一些功能）；然后我们定义两个函数，来**获取数据**
- ✓ 定义一个函数表示**空变量***empty_list_base*。为了方便计算，我们可以写一个生成***cons***的的方便的方法，这样我们就可以很方便地完成新建列表了。

```
def cons(head, tail):
    def helper():
        return (head, tail)
    return helper

head = lambda cons_list: cons_list()[0]
tail = lambda cons_list: cons_list()[1]
```

```
def cons_apply(*args):
    if len(args) == 0:
        return empty_list_base
    else:
        return cons(args[0], cons_apply(*args[1:]))
```

综合示例：基于元组实现一个列表数据类型

实现（续）

- ✓定义一个判断列表是否相等的函数并验证
- ✓分别实现*map*、*fold_left*和*filter*
  - **map**：对列表的每一项应用带入的函数参数***f***
  - **filter**：应用***f***作为列表筛选函数，返回过滤后的列表
  - **fold_left**：相当于***reduce***

```
# 判断列表是否相等
def equal_cons(this: ListBase[S], that: ) -> bool:
    if this == empty_list_base and that != empty_list_base:
        return False
    elif this != empty_list_base and that == empty_list_base:
        return False
    elif this == empty_list_base and that == empty_list_base:
        return True
    else:
        return head(this) == head(that) and equal_cons(tail(this), tail(that))

>>> assert equal_cons(cons_apply(1, 2, 3), cons(1, cons(2, cons(3, ()))))
```

```
#map操作
def map_cons(f, cons_list):
    if cons_list == ():
        return empty_list_base
    else:
        return cons(f(head(cons_list)), map_cons(f, tail(cons_list)))
```

综合示例：基于元组实现一个列表数据类型

验证

可以测试一些基本功能了，比如将*[1, 2, 3, 4, 5]*每个元素加一，筛选偶数并求和。

#测试

```
>>> res = fold_left_cons(lambda x, y: x + y, 0,
...         filter_cons(lambda x: x % 2 == 0,
...         map_cons(lambda x: x + 1,
...         cons_apply(1, 2, 3, 4, 5)
...         )))
>>> res == 12
```

```
def filter_cons(f, cons_list):     # filter操作
    if cons_list == ():
        return empty_list_base
    else:
        hd, tl = head(cons_list), tail(cons_list)
        if f(hd):
            return cons(hd, filter_cons(f, tl))
        else:
            return tl

def fold_left_cons(f, init, cons_list): #fold_left操作
    if cons_list == ():
        return init
    else:
        return fold_left_cons(f, f(init, head(cons_list)),
tail(cons_list))
```

函数库：常用库

1）**标准库**。Python官方发布的库***Python 标准库 — Python 3.10.7 文档***。标准库非常庞大，所提供的组件涉及范围十分广泛。其中包含了多个内置模块 (以 C 编写)，Python 程序员依靠它们来实现系统级功能， 例如文件 I/O；此外还有大量以 Python 编写的模块，提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计，通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。 Windows 版本的 Python 安装程序通常包含整个标准库，往往还包含许多额外组件。

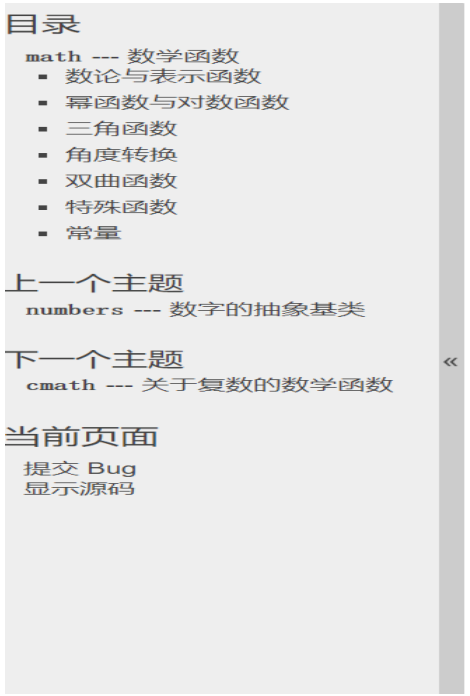
2）选择版本，查看库的列表，查看每个库里面的 函数列表，和每个函数的使用方法。

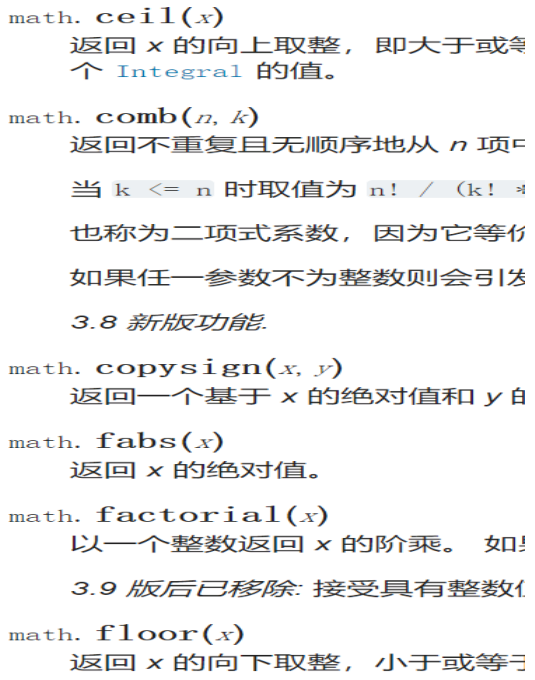
函数库：常用库

以***math***库为例

- ✓ 查看*math*库提供的列表，找到需要使用的函数。
- ✓ 搞清楚函数作用，搞明白参数和返回值定义。
- ✓ 在代码顶部，用*import*语句导入*math*库。
- ✓ 在代码中，就可以正常使用相关函数了。

如右侧，有常数π，有各种函数等。甚至还有比较两个数“是否比较接近”。 也可以仅导入自己需要的函数或常量，如右侧。此时， 代码中可以直接写函数名字和常量名字，而不需要带上模块的名字，如pi。如果调用没有导入的函数或常量， 则出错。——推荐使用前者，导入整个模块。（避免搞不清楚函数来自于哪里，变量名看不懂）






```
In[42]: import math
In[43]: a=math.factorial(5)
In[44]: a*=math.pi
In[45]: print(a)
376.99111843077515
```

导入整个math模块

```
In[36]: from math import factorial,pi
In[37]: print(factorial(5)*pi)
376.99111843077515
```

仅导入math中的阶乘函数和pi常量

函数库：第三方库

1）***PyPI (Python PackageIndex)*** 包索引是Python官方的第三方库的 仓库，区别与本地的标准库，任何 人都可以上传和下载。 ***https://pypi.org/***

2）如搜索“废话”，***PyPI*** 列出各 种与之有关的第三方库。选择 “*write-composition1.7.4*” （废话生成器），点开可查看简介、版本、安装方法。

安装

- ✓ 确认安装*pip*(Python的包管理工具)。在命令行界面(不是Python脚本 界面)，输入“pip -V”，看是否输出pip版本。Python3.5后默认安装pip。
- ✓ 在线安装。在命令行界面直接运行：pip install 包名。直接从PyPI 在线安装，*pip install write-composition*。也可以更换国内镜像(阿里云:https://mirrors.aliyun.com/pypi/simple； 如果包比较大，建议选择国内镜像，如阿里云、清华等)。 *pip install write-composition -i https://mirrors.aliyun.com/pypi/simple*
- ✓ 离线安装。pip install 路径和名字。注：很多包默认下载下来是***whl*** 格式，为了能安装whl格式压缩包，需要提前安装*wheel*：*pip install wheel*。

函数库：第三方库

1）以“*write-composition1.7.4*” （废话生 成器）为例。

2）安装后，正常导入，就可以按照该库的说明，进行调用：

```
from write_composition import generator
content = generator("刷题无用")
print(content)
```

可是，即使是这样，刷题无用的出现仍然代表了一定的意义．既然如何，刷题无用的发生，到底需要如何做到，不刷题无用的发生，又会如何产生．放弃自己，相信别人，这就是失败的原因。叔本华在不经意间这样说过，意志是一个强壮的盲人，倚靠在明眼的跛子肩上。这句话语虽然很短，但令我浮想联翩．人生至善，就是对生活乐观，对工作愉快，对事业兴奋。成功不是将来才有的，而是从决定去做的那一刻起，持续累积而成
刷题无用，发生了会如何，不发生又会如何．对我个人而言，刷题无用不仅仅是一个重大的事件，还可能会改变我的人生．一个人没有钱并不必须就穷，但没有梦想那就穷定了。那么，洛克曾经说过，学到很多东西的诀窍，就是一下子不要学很多。这句话看似简单，但其中的阴郁不禁让人深思．成大事不在于力量多少，而在能坚持多久。我命由我不由天，天欲灭我我灭天。在这种困难的抉择下，本人思来想去，寝食难安.在这种不可避免的冲突下，我们必须解决这个问题．塞内加曾经说过，勇气通往天堂，怯懦通往地狱。这不禁令我深思．自己打败自己是最可悲的失败，自己战胜自己是最可贵的胜利人格的完善是本，财富的确立是末。前有阻碍，奋力把它冲开，运用炙热的活力，转动心中的期待，血在澎湃，吃苦流汗算什么。这种事实对本人来说意义重大，相信对这个世界也是有一定意义的.刷题无用似乎是一种巧合，但如果我们从一个更大的角度看待问题，这似乎是一种不可避免的事实．在这种困难的抉择下，本人思来想去，寝食难安.那么，博在不经意间这样说过，一次失败，只是证明我们成功的决心还够坚强。 维这句话把我们带到了一个新的维度去思考这个问题：行动是治愈恐惧的良药，而犹豫、拖延将不断滋养恐惧。行动是治愈恐惧的良药，而犹豫、拖延将不断滋养恐惧。刷题无用，到底应该如何实现．山顶对我们半山腰的人来说并不遥远。塞涅卡曾经提到过，真正的人生，只有在经过艰难卓绝的斗争之后才能实现。这启发了我．不能去骗别人，因为你能骗到的人，都是信任你的。莫找借口失败，只找理由成功。

洋洋洒洒一堆废话 中间替换很多个给定的关键词，读起来似是而非。 对什么有用？ 对于搜索引擎按关键词随便产生一个页面有用。

函数库：库的使用原则

1）基本上可以假定，你想写的逻辑（尤其是需要封装为函数，便于多次使用的逻辑）已经有人写过了。

2）不要重复造轮子。

3）无需记忆，用到再查。

4）先搜标准库，再搜第三方库。

5）如果搜到的和自己的需求接近，但是不完全相同怎么办？简单的情况下，可以对函数 进行“装修”改造。 ——函数的“修饰器”。

简单叠加两个函数的功能。*注意，绑定后在当前程序中不能解绑。*

```
@func1
def func2():
        ……
```

函数库：自己的库

1）一般来说，用的时候查阅标准库和第三方库就可以。

2）如果将来从事某一个业务领域的开发，可能会遇到一些从业务视角上、特殊数据模型视角上常用的需要封装的逻辑，类似情况下可以建立一个自己的库（zhangsan.py模块）， 将这些函数封装到里面。

3）使用的时候，和其他库一样，在代码顶部导入你的模块名字即可:import zhangsan。然后就可以正常使用库里面的函数了。 （模块文件，放在代码同路径，或者Python\Lib\site-packages下面都可以）

日积月累，可以让自己在所从事领域的代码写得更简单、更快

多选题 1分

设置

下面哪些说法错误?

A 函数参数可以靠位置进行识别

B 函数参与可以靠关键字名字识别

C 返回值类型与return语句返回的表达式类型一致

D 没有return语句时函数的返回值为False

提交

多选题 1分

设置

下面哪些说法正确?

A *p方式，函数把传入参数，存入一个元组的类型

B **p方式，传入的是键值对，函数把所有实参存入一个字典的类型

C 函数内可以直接（未作声明）给全局变量赋值

D 函数内可以直接访问全局变量

提交

五 函数

- 函数的定义和调用
- 函数的参数与返回值
- 函数的测试
- 变量的作用域
- 递归函数
- 函数式编程
- 综合示例
- 函数库

谢谢