python2-02-函数进阶

函数基础

1. 创建函数
2. 调用函数
3. 匿名函数

- 创建函数

def 语句创建函数

```
def function_name(arguments):
    function_documentation_string
    function_body_suit
```

前向引用：函数不允许函数未声明之前对其进行引用或者调用
内嵌函数：在函数体内创建另外一个函数

- **调用**函数
  利用一对圆括号来调用函数，如果不加圆括号，只是对函数的引用，不是调用函数。函数所有的参数都必须要放入到圆括号中。

关键字参数：通过调用参数的名字来调用参数，可以允许参数缺失或者不按顺序调用参数

```
def set_age(name, age):
    print('%s is %s years old' % (name, age)

set_age('bob', 23)
set_age()  #参数不够，错误
set_age('bob', 23, 100)  err, 参数太多
set_age(23, 'bob')  语法没有错，语义不正确
set_age(age = 23, name = 'bob')
set_age(age = 23, 'bob') 语法错误，kev=val样式必须在后
set_age('bob', age = 23)  正确
set_age(23 , name = 'bob'  错误，name获得了多个值
```

参数组：允许程序员执行一个没有显式定义的函数，方法是通过元组或字典作为参数传递给函数

```
def myfunc(*args):    *表示args是个元组
    print(args)

def myfunc2(**kwargs):  **表示args是个字典
    print(kwargs)

if __name__ == '__main__':
    myfunc()                        #args=()
    myfunc('hello')                 #参数将会放到元组中args=('hello',)
    myfunc('hello', 123)            #args=('hello', 123)
    myfunc2()
    myfunc2(name='bob', age=23)     #args={'name': 'bob','age': 23}
```

```
def add(x, y)
    print(x + y)

if __name__ == '__main__':
```

```
    add(10, 20)
    nums = [10, 20]
    add(nums[0], nums[1])
    add(*nums)                #*nums 表示把nums拆开
```

- 匿名函数
  python3 使用lambda 关键字在创建匿名函数，意思即不再使用def语句这样标准的形式定义一个函数

语法：

```
lambda 参数： expression
```

普通函数：

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

匿名函数实现以上功能：

```
lambda x, y: x+y
```

filter()函数
filter实现过滤，将nums中的每一项当成func1的参数，func1的结果为真，则把数字留下，否则过滤掉。
案例：

```
import random
def func1(x):
    return x % 2
if __name__ == '__main__':
    nums = [random.randint(1, 100) for i in range(10)]
    print(nums)
    print(list(filter(func1, nums)))
    print(list(filter(lambda x: x % 2, nums)))
```

map()函数
map用来加工数据，将nums中的每一个项目作为func2的参数进行加工，得到结果

```
import random

def func1(x):
    return x % 2

def func2(x)
    return x * 2 + 1

if __name__ == '__main__':
    nums = [random.randint(1, 100) for i in range(10)]
    print(nums)
    print(list(filter(func1, nums)))
    print(list(filter(lambda x: x % 2, nums)))
    print(list(map(func2, nums)))
    print(list(map(lambda x: x * 2 + 1, nums)))
```

函数的高级应用

1. 变量作用域

- 全局变量：函数外面创建的，全局变量在它定义的位置开始，一直到程序结束，任何地方都可见可用。

```
x = 10
def foo():
  print(x)
if __name__ == '__main__':
  foo()
```

- 局部变量：函数内部创建，仅在函数内可见可用。参数也可以看作局部变量

```
def bar():
  hi = hello
  print(hi)
if __name__ == '__main__':
  bar()
  print(hi) 会报错
```

- 如果局部和全局有相同的名字，局部变量名会将全局变量遮盖

```
x = 10
def func1():
  x = 'abc'
  print(x)
if __name__ == '__main__':
  func1()
  print(x)
```

- 如果希望在局部中改变全局变量的值 ，可以在局部中使用关键字global声明

```
x = 10
def func2():
  global x
  x = 100
  print(x)
if __name__ == '__main__':
  func2()
  print(x)
```

标识符的搜索次序：先局部中找，没找到就去全局中找，全局中仍然没有找着就会去内建中找。

1. 函数式编程
   偏函数：改造现有的函数，把其中的一部分参数固定下来，生成一个新的函数

```
from functools import partial
def add(a, b, c, d)
 print(a +b +c +d)
if __name__ == '__main__':
 add(10, 20, 30, 5)
 add(10, 20, 30, 8)
 add(10, 20, 30, 9)

 myadd=partial(add, 10, 20, 30)
 myadd(5)
 myadd(8)
 myadd(9)
```

递归函数：函数内部包含了对自己的调用
案例： 求某一个数的阶乘

```
def func1(x):
    if x == 1
        return 1
```

```
if __name__ == '__main__':
    print(func1(5))
    print(func1(6))
```

案例：快速排序

生成器：本质上就是一个带yield语句的函数，但是函数只能返回一个值，生成器可以返回多个中间值。

```
def mygen()
    yield 10
    a = 100 + 200
    yield a
    yield 'hello'
    a = mygen()
    a.__next__()
    a.__next__()
```

1. 内部函数

   (1). 闭包

   (2). 装饰器

案例4：测试程序运行效率

```
import time
from tqdm import tqdm

def myadd():
    result = 0
    for i in tqdm(range(10000000)):
        result += 1
    print(result)

if __name__ == '__main__':
    start = time.time()
    myadd()
    end = time.time()
    print(end - start)
```

升级版：把myadd()函数当成参数传递到timeit()函数中
方便调用timeit()函数来多次执行myadd()函数。

```
import time
from tqdm import tqdm

def timeit(func):
    start = time.time()
    func()
    end = time.time()
    print(end - start)
def myadd():
    result = 0
    for i in tqdm(range(10000000)):
        result += 1
    print(result)

if __name__ == '__main__':

    timeit(myadd)
```

最终版：闭包版

```
import time
from tqdm import tqdm

def deco(func):
    def timeit():
        start = time.time()
        func()
        end = time.time()
        print(end - start)
    return timeit

@deco

def myadd():
    result = 0
    for i in tqdm(range(10000000)):
        result += 1
    print(result)

if __name__ == '__main__':
    #myadd = deco(myadd)  相当于上面deco函数的@deco
    myadd()
```