

Python 3

1. Python 基础.....	5
2. 函数.....	5
2.1 调用函数.....	5
2.2 定义函数.....	5
2.3 函数的参数.....	6
2.3.1 位置参数.....	6
2.3.2 默认参数.....	6
2.3.3 可变参数.....	7
2.3.4 关键字参数.....	8
2.3.5 命名关键字参数.....	8
2.3.6 组合参数.....	9
2.4 递归函数.....	10
3. 高级特性.....	12
4. 函数式编程.....	12
4.0 概览.....	12
4.1 高阶函数.....	13
4.1.0 概念.....	13
4.1.1 map/reudce.....	13
4.1.2 filter.....	14
4.1.3 sorted.....	15
4.2 返回函数.....	15
4.3 匿名函数.....	19

4.4 装饰器.....	19
4.5 偏函数.....	23
5. 模块.....	24
5.1 使用模块.....	24
5.1.1 作用域.....	24
5.2 安装第三方模块.....	25
6. 面向对象编程.....	26
6.0 前言.....	26
6.1 类和实例.....	26
6.2 访问限制.....	28
6.3 继承和多态.....	30
6.4 获取对象信息.....	34
6.5 实例属性和类属性.....	37
7. 面向对象高级编程.....	39
7.1 使用__slots__.....	39
7.2 使用@property.....	39
7.3 多重继承.....	39
7.4 定制类.....	39
7.5 使用枚举类.....	39
7.6 使用元类.....	39
8. 错误，调试和测试.....	39
9. IO 编程.....	39

10. 进程和线程.....	39
11. 常用内建模块.....	39
12. 常用的第三方模块.....	39
13. 网络编程.....	39
14. 电子邮件.....	39
15. 访问数据库.....	39
16. Web 开发.....	40
17. 异步 IO.....	40
18. 实战.....	40

1. Python 基础

1.1 数据类型与变量

1.2 字符串与编码

1.3 List 与 Tuple

1.4 条件判断

1.5 循环

1.6 Dict 与 Set

2. 函数

2.1 调用函数

函数是一种代码抽象方式；

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”；

小结：

调用 Python 的函数，需要根据函数定义，传入正确的参数。
如果函数调用出错，一定要学会看错误信息，所以英文很重要。

2.2 定义函数

```
def my_abs (x):
    if not isinstance(x,(int,float)):
        raise TypeError('bad operand type') #抛出错误
    if x>0:
        return x
    else:
        return -x

#空函数
def nop(x):
    pass #没想好可以用pass 否则会报错
    if x==1:
        pass #返回None
```

小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用 `return` 随时返回函数结果；

函数执行完毕也没有 `return` 语句时，自动 `return None`。

函数可以同时返回多个值，但其实就是一个 `tuple`。

2.3 函数的参数

2.3.1 位置参数

定义函数的时候，我们把参数的名字和位置确定下来。

```
def power(x): #x的平方
    return x*x

print (power(5))

def power(x,n): #x的n次方
    sum = x
    while n>1:
        sum = sum * x
        n = n - 1
    return sum
```

参数 `x` 即为位置参数。

2.3.2 默认参数

必选参数在前，默认参数在后（必须）防止调用时的歧义。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

```
def power(x,n=2):    #x的平方 默认为2 与n次平方一致
    return x*x

print (power(5))

def add_end(L=[]):
    L.append('end')
    return L

def add_end(L=None): #直接将参数设计为 L = [] 会产生问题
    if L is None:
        L = []
    L.append('end')
    return L
```

默认参数必须指向不变对象# 不变对象 str None。

如果可以设计一个参数对象为不变对象，那就尽量设计成不变对象。

2.3.3 可变参数

在 Python 函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是 1 个、2 个到任意个，还可以是 0 个。

```
def calc(numbers):    #不可变 tuple 或者 list
    sum = 0
    for x in numbers:
        sum = sum + x * x
    return sum

def calc1(*numbers):  #可变参数 可以为空参数
    sum = 0
    for x in numbers:
        sum = sum + x * x
    return sum
```

调用方式可以化简为：

```
T = (1,2,3)
print (calc1(1,2,3))
print (calc(T))
print (calc1(*T))
```

Python 允许你在 list 或 tuple 前面加一个*号，把 list 或 tuple 的元素变成可变参数传进去。

2.3.4 关键字参数

可以扩展函数的功能。

变参数允许你传入 0 个或任意个参数，这些可变参数在函数调用时自动组装为一个 tuple。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict

```
def person1(name, age, **kw):
    if 'city' in kw:
        pass
    if 'job' in kw:
        pass
    print('name:', name, 'age:', age, 'other:', kw)

person1('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)

extra = {'city': 'Xian', 'gender': 'M'}
person('frala', 22, **extra)
```

Python 允许**extra 表示把 extra 这个 dict 的所有 key-value 用关键字参数传入到函数的**kw 参数。

但是注意：

注意 kw 获得的 dict 是 extra 的一份拷贝，对 kw 的改动不会影响到函数外的 extra。

2.3.5 命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 kw 检查。

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 city 和 job 作为关键字参数。这种方式定义的函数

如下：

```
def person2(name, age, *, city='Xian', job):  
    print(name, age, city, job)  
  
person2('Jack', 24, job='engineer')
```

表示：只接收关键字为 city 以及 job 的关键字参数。

可缺省，默认参数。

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符*了：

```
def person(name, age, *args, city, job):  
    print(name, age, args, city, job)
```

2.3.6 组合参数

5 种参数都可以组合使用，除了可变参数无法和命名关键字参数混合。

参数优先级：参数定义的顺序**必须**是：必选参数、默认参数、可变参数/命名关键字参数和关键字参。

```
def f1(a,b,c=0,*args,**kw):  
    print ('a=',a, 'b=',b, 'c=',c, 'args=',args, 'kw=',kw)  
def f2(a, b, c=0, *, d, **kw):  
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

所以，对于任意函数，都可以通过类似 func(*args, **kw)的形式调用它，无论它的参数是如何定义的。

小结

Python 的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，程序运行时

会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

`*args` 是可变参数，`args` 接收的是一个 tuple；

`**kw` 是关键字参数，`kw` 接收的是一个 dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：`func(1, 2, 3)`，又可以先组装 list 或 tuple，再通过 `*args` 传入：`func(*(1, 2, 3))`；

关键字参数既可以直接传入：`func(a=1, b=2)`，又可以先组装 dict，再通过 `**kw` 传入：`func(**{'a': 1, 'b': 2})`。

使用 `*args` 和 `**kw` 是 Python 的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数不要忘了写分隔符 `*`，否则定义的将是位置参数。

2.4 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 $n! = 1 \times 2 \times 3 \times \dots \times n$ ，用函数 `fact(n)` 表示：

$$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n = (n-1)! \times n = \text{fact}(n-1) \times n$$

```
def fact(n):  
    if n == 1:  
        return 1  
    else:  
        return n*fact(n-1)
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（**stack**）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。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。

通过**尾递归优化**来解决栈溢出的问题。

尾递归：在函数返回的时候，调用自身本身，并且，**return** 语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

```
def fact_iter(number,product):  
    if number == 1:  
        return product  
    else:  
        return fact_iter(number-1,number * product)
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，**Python** 解释器也没有做优化，所以，即使把上面的 **fact(n)** 函数改成尾递归方式，也会导致栈溢出。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导

致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python 标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

3. 高级特性

3.1 切片

3.2 迭代

3.3 列表生成式

3.4 生成器

3.5 迭代器

4. 函数式编程

4.0 概览

面向过程的程序编程，函数就是面向过程的程序设计的基本单元。

语言越低级，越贴近计算机，执行代码越高。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作

用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的特点，允许把函数本身作为函数参数传入另一个函数还允许返回一个函数。

Python 对函数式编程提供部分支持，由于 python 允许使用变量因此 python 不是纯函数编程语言。

4.1 高阶函数

4.1.0 概念

变量可以指向函数，可以通过此变量进行函数调用。

函数名也是变量

变量可以指向函数，那么函数的参数也能接收变量，则一个函数接收另一个函数作为参数，这种函数称为高阶函数。

4.1.1 map/reduce

map 接收两个参数，一个是函数，另一个是 **iterable** 的对象，**map** 将传入的函数依次作用于序列的每个元素，并把结果作为新的 **iterator** 返回。

reduce 把一个函数作用在一个序列上，这个函数也必须接收两个参数，和 **map** 不同的是 **reduce** 把结果继续和下一个元素做累计计算。

reduce 有 三个参数

function	有两个参数的函数， 必需参数
sequence	tuple , list , dictionary, string等可迭代物， 必需参数
initial	初始值， 可选参数

利用 map 和 reduce 定义一个 str2int 函数：

```
def str2int(s):
    def char2int(s):
        digits = {'0':0, '1':1, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8, '9':9}
        return digits[s]
    def f(x,y):
        return x*10+y
    return reduce(f, map(char2int, s))
```

一个练习：str2float

```
def str2float(s):
    def char2floatnumber(s):
        digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5,
                  '6': 6, '7': 7, '8': 8, '9': 9, '.': -1}
        return digits[s]
    point = 0
    def to_float(f, n):
        nonlocal point
        if n == -1:
            point = 1
            return f
        if point == 0:
            return f * 10 + n
        else:
            point = point * 10
            return f + n / point
    return reduce(to_float, map(char2floatnumber, s))
```

4.1.2 filter

Python 内置的 filter () 用于过滤序列。

Filter 也接收一个函数和一个序列和 map 不同的是 filter 把传入函数作用于每一个元素之后，然后根据返回值 True 还是 false 决定保留还是丢弃。

实质是实现筛选函数。

```

#用filter求素数

#先构造以3为始的奇数无限序列 生成器
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n

#define filter_function
def _not_divisible(n):
    return lambda x: x % n > 0

#define prime
def primes():
    yield 2
    it = _odd_iter() #初始序列
    while True:
        n = next(it)
        yield n
        it = filter(_not_divisible(n),it)

```

iterator 是一个惰性序列，所以我们可以用 python 表示全体自然数，全体素数等无限序列，且非常简洁。

小结：

Filter 的作用是从一个序列中筛选出符合条件的元素，由于使用了惰性计算，所以只有在取 filter 结果时，才会真正筛选并每次返回下一个筛选出来的元素。

4.1.3 sorted

排序算法，比较过程通过函数抽象出来。

Python 内置的 sorted 函数可以对 list 排序。

此外 sorted 也是一个高阶函数，可以接收一个 key 参数，实现自定义排序。

Sorted 函数传入三个参数，待排序序列，key 映射函数参数，reverse 参数升降序参数。

4.2 返回函数

高阶函数除了可以接受函数作为参数外，还可以把函数作为返回值。

```
In [12]: #实现可变参数求和
...: def calc_sum(*args):
...:     ax = 0
...:     for x in args:
...:         ax = ax + x
...:     return ax
...:
...: #如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？
...: #可以不返回求和的结果，而是返回求和的函数
...: def lazy_sum(*args):
...:     def sum():
...:         ax = 0
...:         for x in args:
...:             ax = ax + x
...:         return ax
...:     return sum
...:
...: #当调用 lazy_sum() 时返回求和函数而非和值
...: f = lazy_sum(0,1,2,3,4,5,6,7,8,9)
...: print (f)
...:
...: sum = f()
...: print (sum)
...:
<function lazy_sum.<locals>.sum at 0x0000011217746D90>
45
```

在此例中，在函数 `lazy_sum` 函数中定义了函数 `sum` 函数，并且 `sum` 函数可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（closure）”的程序结构拥有极大的威力。

注：当两次调用 `lazy_sum` 时互不影响。

```
In [23]: f1 = lazy_sum(1,3,5,7,9)
...: f2 = lazy_sum(1,3,5,7,9)
...: f1 == f2
...:
Out[23]: False
```

闭包：

注意点 1，返回的函数之中引用了 `args` 局部变量。也就是当一个函数返回另一个函数后，其内部的局部变量还是被新函数

使用。

注意点 2，返回的函数没有立即执行，当它在被调用时采取执行。

例：

```
In [85]: def count():
...:     L = []
...:     for i in range(1,4):
...:         def f():
...:             return i*i
...:         L.append(f)
...:     return L
...:
...:
...:
...: [f1,f2,f3] = count()

In [86]: f1(),f2(),f3()
Out[86]: (9, 9, 9)
```

count()返回函数列表与 f1,f2,f3.

返回全部是 9，原因是返回的函数全都引用了变量 i，但它并非立即执行。等到三个函数都返回时它们所引用的变量 i 已经变成了 3，因此最终的结果为 9.

注：返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

若定要引用循环变量，则：

方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变。例：

```

In [83]: def count():
...:     def f(j):#绑定函数
...:         def g():
...:             return j*j
...:         return g
...:     L = []
...:     for i in range(1,4):
...:         L.append(f(i)) #f(i)被立即执行
...:     return L
...:
...:
...: f1, f2, f3 = count()
...:

In [84]: f1(),f2(),f3()
Out[84]: (1, 4, 9)

```

有明显缺点：代码太长，可利用 `lambda` 缩短代码。

练习：

利用闭包返回一个计数器函数，每次调用它返回递增整数

```

In [108]: def create_counter():
...:     k = [0]
...:     def counter():
...:         k[0]=k[0]+1
...:         return k[0]
...:     return counter
...:
...:
...: counter = create_counter()
...: counter(),counter(),counter()
...:

Out[108]: (1, 2, 3)

```

因为 `counter= create_counter()` 这一句, `counter` 指向了 `counter()` 函数，因此调用 `counter` 时，即是执行 `counter()`，而不会对 `s` 进行再次赋值。

注：

- 1.内部函数一般无法修改外部函数的参数
- 2.想要修改需要声明 `nonlocal`
- 3.内部函数可以修改外部 `list` 中的元素

小结：

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

4.3 匿名函数

在传入函数时，不必使用显式函数，使用匿名函数更为方便。

```
In [3]: list(map(lambda x:x*x,range(1,10)))  
Out[3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`lambda` 关键字表示匿名函数，匿名函数限制就是不能使用 `return` 返回，返回值就是该表达式的值。

匿名函数有个好处，因为函数没有名字不必担心函数名冲突。

此外匿名函数也是个函数对象，可以把函数赋给一个变量，再利用变量调用该函数。同样的，也可以把匿名函数当做返回值返回。

Python 对匿名函数的支持有限，只有在一些简单的情况下才可以使用匿名函数。

注：`lambda` 中不能有赋值操作。

4.4 装饰器

由于函数也是一个对象，可以被变量赋值。通过变量可以调用该函数。

```
In [34]: def now():  
...:     print ('2018/4/13')  
...:  
...: n = now  
...: n()  
...:  
2018/4/13
```

函数的对象有一个 `name` 属性，可以拿到函数名。

```
In [43]: now.__name__  
Out[43]: 'now'
```

```
In [44]: n.__name__  
Out[44]: 'now'
```

现在，要增强 `now` 函数的功能，例如，在函数调用前后打印自动日志但又不希望修改 `now` 函数的定义，这种在代码运行期间动态增加功能的方式叫做**装饰器（decorator）**。

本质上 `decorator` 是一个返回函数的高阶函数。

```
In [56]: def log(func):  
...:     def wrapper(*args,**kw):  
...:         print ('call %s():'%func.__name__)  
...:         return func(*args,**kw)  
...:     return wrapper  
...:  
...:  
...: @log  
...: def now1():  
...:     print ('2018/4/13')  
...: now1()  
...:  
call now1():  
2018/4/13
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个 `decorator`，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)`，因此，`wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

```

In [66]: def log(text):
...:     def decorator(func):
...:         def wrapper(*args,**kw):
...:             print ('%s %s():'%(text,func.__name__))
...:             return func(*args,**kw)
...:         return wrapper
...:     return decorator
...:
...: @log('excute')
...: def now2():
...:     print ('2018/4/13')

In [67]: now2()
excute now2():
2018/4/13

```

和两层嵌套的 decorator 相比，3 层嵌套的效果是这样的：

```
now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')`，返回的是 decorator 函数，再调用返回的函数，参数是 now 函数，返回值最终是 wrapper 函数。

以上两种 decorator 的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你去看经过 decorator 装饰之后的函数，它们的 `__name__` 已经从原来的 'now' 变成了 'wrapper'。

```

In [76]: now1.__name__
Out[76]: 'wrapper'

In [77]: now2.__name__
Out[77]: 'wrapper'

```

因为返回的那个 `wrapper()` 函数名字就是 'wrapper'，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python 内置的 `functools.wraps` 就是干这个事的，所以，一个完整

的 decorator 的写法如下：

```
In [93]: import functools
...: def log(text):
...:     def decorator(func):
...:         @functools.wraps(func)
...:         def wrapper(*args,**kw):
...:             print ('%s %s():'%(text,func.__name__))
...:             return func(*args,**kw)
...:         return wrapper
...:     return decorator
...:
...: @log('excute')
...: def now():
...:     print ('2018/4/13')
...:
```

```
In [94]: now()
excute now():
2018/4/13
```

```
In [95]: now.__name__
Out[95]: 'now'
```

小结

在面向对象（OOP）的设计模式中，decorator 被称为装饰模式。OOP 的装饰模式需要通过继承和组合来实现，而 Python 除了能支持 OOP 的 decorator 外，直接从语法层次支持 decorator。Python 的 decorator 可以用函数实现，也可以用类实现。

decorator 可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

思考题：



交作业

蓝墨灰 created at 1分钟前, Last updated at 1分钟前

有参无参decorator合并

```
def log(text=None):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(args,**kw):
            if callable(text):
                print ('call %s():'%func.name)
            else:
                print ('%s %s():'%(text,func.name))
            return func(args,**kw)
        return wrapper
    return decorator(text) if callable(text) else decorator

@log('excute')

def test4(i,j):
    return i*j

@log

def test5(m,n):
    return m+n
```

4.5 偏函数

Python `functools` 中有很多有用的功能，例如偏函数，（`partial function`）。

在介绍函数参数时，通过设定参数默认值可以降低调用的难度。而偏函数便能做到这一点。

创建偏函数时实际上可以接收函数对象，`*args`，`**kw` 这 3 个参数。

```
In [20]: import functools
...: int2 = functools.partial(int ,base=2)
...: print (int2('10101010001010111001010'))
...:
...: max10 = functools.partial(max,10)#传入*args = 10    #max(*args)
...: print (max10(5,7,2))
5576138
10
```

小结：

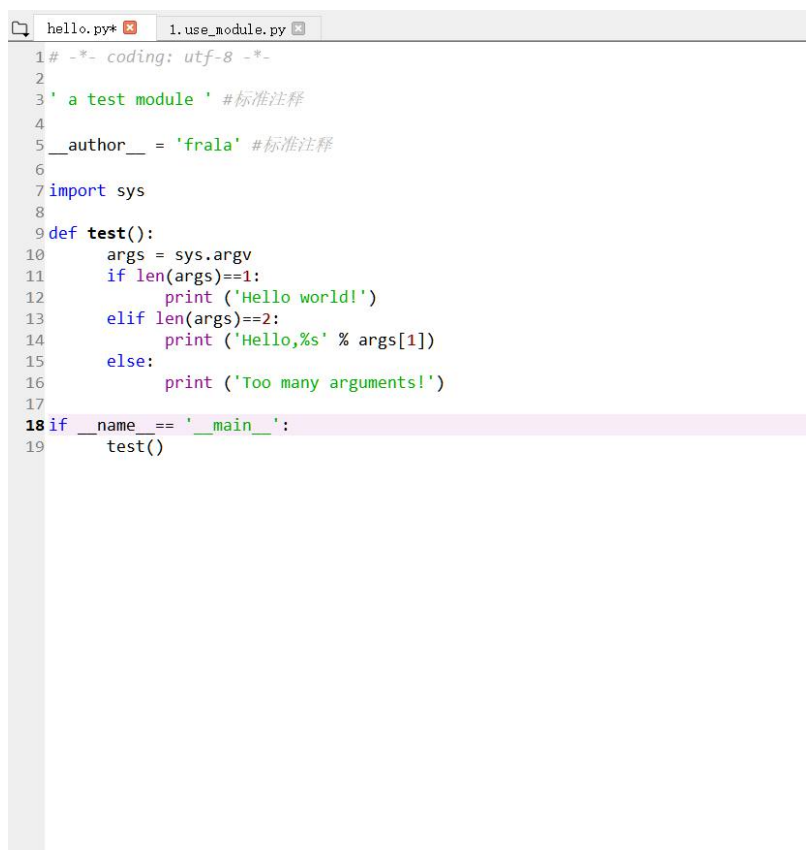
当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

5. 模块

5.1 使用模块

Python 中有很多有用的模块，只要安装完毕，这些模块就可以立即使用。

以内建的 `sys` 模块为例，编写一个 `hello` 模块。



```
1 # -*- coding: utf-8 -*-
2
3 ' a test module ' #标准注释
4
5 __author__ = 'frala' #标准注释
6
7 import sys
8
9 def test():
10     args = sys.argv
11     if len(args)==1:
12         print ('Hello world!')
13     elif len(args)==2:
14         print ('Hello,%s' % args[1])
15     else:
16         print ('Too many arguments!')
17
18 if __name__ == '__main__':
19     test()
```

```
In [24]: import hello
...: hello.test()
...:
Hello world!
```

5.1.1 作用域

一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在 Python 中，是通过 `_` 前缀来实现的。

正常的函数和变量名是公开的（**public**），可以被直接引用，比如：`abc`，`x123`，`PI` 等；

类似 `__xxx__` 这样的变量是**特殊变量**，可以被直接引用，但是有特殊用途，比如上面的 `__author__`，`__name__` 就是特殊变量，`hello` 模块定义的文档注释也可以用特殊变量 `__doc__` 访问，我们自己的变量一般不要用这种变量名；

类似 `_xxx` 和 `__xxx` 这样的函数或变量就是非公开的（**private**），不应该被直接引用，比如 `_abc`，`__abc` 等；

之所以我们说，**private** 函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为 Python 并没有一种方法可以完全限制访问 **private** 函数或变量，但是，从编程习惯上不应该引用 **private** 函数或变量。

外部不需要引用的函数全部定义成 **private**，只有外部需要引用的函数才定义为 **public**，即这也是一种**非常有用的代码封装和抽象的方法**。

5.2 安装第三方模块

6. 面向对象编程

6.0 前言

面向对象编程——Object Oriented Programming，简称 OOP，是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程：大块函数切分成小块降低复杂度。

面向对象：把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

面向对象三大特点：

数据封装

继承

多态

6.1 类和实例

面向对象编程最重要的是类（Class）和实例（Instance）的概念：

类是抽象的概念，像是模板。

实例是根据类创建出来的一个个具体的对象。每个对象都具有相同的方法。但各自的数据可能不同。

定义：

```

class Student(object): #自定义类
    pass

frala = Student() #创建实例

#可以自由的给实例绑定一些属性
frala.name = 'FZ'

#创建类时可以绑定属性
#定义方法 与普通函数没有什么区别
#仍然可用可变参数, 默认参数, 关键字参数, 命名关键字参数
class Student(object):
    def __init__(self, name, score): #与普通函数相比
        self.name = name #第一个参数永远是实例变量 且调用时不用传入
        self.score = score #解释器自动传入

frala = Student('FZ', 97)

```

数据封装:

面向对象编程的一个重要特点就是数据封装。在上面的 Student 类中, 每个实例就拥有各自的 name 和 score 这些数据。我们可以通过函数来访问这些数据。

```

class Student(object):
    def __init__(self, name, score): #与普通函数相比
        self.name = name #第一个参数永远是实例变量 且调用时不用传入
        self.score = score #解释器自动传入

    def print_score(self):
        print ('%s : %d' % (self.name, self.score))

frala = Student('FZ', 97)

frala.print_score()

```

这样一来, 我们从外部看 Student 类, 就只需要知道, 创建实例需要给出 name 和 score, 而如何打印, 都是在 Student 类的内部定义的, 这些数据和逻辑被“封装”起来了, 调用很容易, 但却不用知道内部实现的细节。

封装的另一个好处是可以给 Student 类增加新的方法, 比如 get_grade:

```

class Student(object):
    def __init__(self, name, score): #与普通函数相比
        self.name = name           #第一个参数永远是实例变量 且调用时不用传入
        self.score = score         #解释器自动传入

    def print_score(self):
        print ('%s : %d' % (self.name, self.score))

    def get_grade(self):
        if self.score > 90:
            return 'A'
        elif self.score > 80:
            return 'B'
        elif self.score > 60:
            return 'C'
        elif self.score < 60:
            return 'D'

frala = Student('FZ', 97)

frala.print_score()
frala.get_grade() #返回值 为 'A'

```

小结：

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python 允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同。

6.2 访问限制

在 Class 内部，可以有属性和方法，而外部代码可以通过直接

调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

如果要想内部属性不被外部访问，可以把属性的名称前加上两个下划线__，在 Python 中，实例的变量名如果以__开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问。

私有变量设置便于参数检查。

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name    #私有变量
        self.__score = score
    def print_score(self):
        print('%s : %d' % (self.__name, self.__score))

frala = Student('FZ', 99)    #解释器si私有变量改名
frala.__name = 'Zl'         #frala.__name => frala._Student__name

In [30]: frala.__name
Out[30]: 'Zl'

In [31]: frala._Student__name
Out[31]: 'FZ'
```

需要注意的是，在 Python 中，变量名类似__xxx__的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是 private 变量，所以，不能用__name__、__score__这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如__name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其

实也不是。不能直接访问__name 是因为 Python 解释器对外把__name 变量改成了 _Student__name，所以，仍然可以通过 _Student__name 来访问__name 变量。但是不能这么做。

6.3 继承和多态

在 OOP 程序设计中，当我们定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）。

继承最大的好处是子类获得了父类的全部功能。

继承的第二个好处需要我们对代码做一点改进。重写父类方法使得它更具体。

```
class Animal(object):
    def run(self):
        print ('Animal is running.')

class Dog(Animal):
    def run(self):
        print ('dog is running.')

class Cat(Animal):
    pass

hua = Dog() #重写父类方法
zl = Cat()  #继承父类方法

hua.run()
zl.run()

dog is running.
Animal is running.
```

当子类 and 父类都存在相同的 run() 方法时，我们说，子类的 run() 覆盖了父类的 run()，在代码运行的时候，总是会调用子类的 run()。这样，我们就获得了继承的另一个好处：多态。

```
a = list()
b = Animal()
c = Dog()

isinstance(a,list)    #true
isinstance(b,Animal)  #true
isinstance(c,Dog)     #true
isinstance(c, Animal) #true
```

在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行。

理解多态的好处。

```
def run_twice(Animal):
    Animal.run()
    Animal.run()

run_twice(b)    #Animal is running.
run_twice(c)    #dog is running.

class Tortoise(Animal):
    def run(self):
        print ('tortoise is running slowly.')

run_twice(Tortoise())    #tortoise is running slowly.
```

新增一个 `Animal` 的子类，不必对 `run_twice()` 做任何修改，实际上，任何依赖 `Animal` 作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

多态的好处就是，当我们需要传入 `Dog`、`Cat`、`Tortoise`……时，我们只需要接收 `Animal` 类型就可以了，因为 `Dog`、`Cat`、`Tortoise`……都是 `Animal` 类型，然后，按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()` 方法，因此，传入的任意类型，只要是 `Animal` 类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

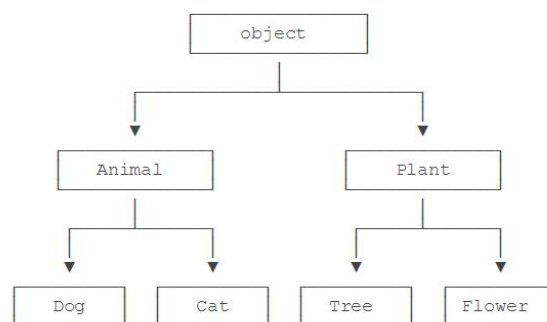
对于一个变量，我们只需要知道它是 `Animal` 类型，无需确切地知道它的子类型，就可以放心地调用 `run()` 方法，而具体调用

的 `run()` 方法是作用在 `Animal`、`Dog`、`Cat` 还是 `Tortoise` 对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种 `Animal` 的子类时，只要确保 `run()` 方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增 `Animal` 子类；

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

继承还可以一级一级地继承下来。



静态语言 vs 动态语言：

对于静态语言（例如 `Java`）来说，如果需要传入 `Animal` 类型，则传入的对象必须是 `Animal` 类型或者它的子类，否则，将无法调用 `run()` 方法。

对于 `Python` 这样的动态语言来说，则不一定需要传入 `Animal` 类型。我们只需要保证传入的对象有一个 `run()` 方法就可以了。动态语言的‘鸭子类型’。它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。


```
class Timer(object):
    def run(self):
        print ('start...')

run_twice(Timer()) #starting...
```

Python 的 “file-like object “就是一种鸭子类型。对真正的文件对象，它有一个 read()方法，返回其内容。但是，许多对象，只要有 read()方法，都被视为 “file-like object “。许多函数接收的参数就是 “file-like object “，你不一定要传入真正的文件对象，完全可以传入任何实现了 read()方法的对象。

小结

1. 继承：

（1）子类将继承父类的全部功能

（2）此外，在子类继承父类功能（方法）的时候，子类具有很强的灵活性，既可以对继承自父类的方法进行修改，也可以在父类的基础上增加新的功能（方法）#不知道能不能有选择性的继承。

2.多态 对于一个方法，在使用（或者调用更专业一点？）的时候，其结果取决于实例。我的理解是，对于继承自一个父类的各个子类，均有 A 方法（A 方法是 public 的），A 方法在基于这一个体系上的不同的类，其内容可以是不同的，这个不同就体现出了多态。

3.开闭原则：

（1）对扩展开放：允许新增父类的子类

（2）对修改封闭：不需要依赖 Animal 类型的 run_time()等函

数 //私以为中 `run_time(Animal)`函数，函数的参数上体现了继承的特性，而在函数体中，体现了多态的特性

4.动态语言特有的“鸭子类型”

(1) 静态语言：对于静态语言来讲，定义一个函数 `func(Animal)`，则传入的类型必须是 `Animal` 类型或者他的子类

(2) 动态语言：然而对于 `python` 为代表的动态语言来讲，只要传入的类型能够让我的函数运行正常，此类型中有该函数所需要的方法，那 OK，可以传进来。`python` 中"file-like object"就属于鸭子类型。

6.4 获取对象信息

判断对象类型用 `type` 函数。

```
print (type(123)==int) #ture
print (type('123')==type('abc')) #true
print (type('123')) #<class 'str'>
```

函数返回所属类名。

判断对象是否是函数，利用 `types` 模块中定义的常量。

```
print (type(fn)==types.FunctionType) #true
print (type(abs)==types.BuiltinFunctionType) #true
print (type(lambda x:x)==types.LambdaType) #true
print (type(x for x in range(10))==types.GeneratorType) #ture
```

判断是实例或者继承关系，用 `isinstance` 函数。

```

class Animal:
    pass

class Dog(Animal):
    pass

class Cat(Animal):
    pass

a = Animal()
b = Dog()
c = Cat()

print (isinstance(a,Animal))    #ture
print (isinstance(b,Dog))      #ture
print (isinstance(b,Animal))    #true
print (isinstance(c,Dog))      #false

```

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断。

并且还可以判断一个变量是否是某些类型中的一种。

```

print (isinstance([1,2,3],(list,tuple)))    #true
print (isinstance((1,2,3),(list,tuple)))    #ture

```

注：总是优先使用 `isinstance()` 判断类型，可以将指定类型及其子类“一网打尽”。

获取对象所有属性和方法使用 `dir` 函数

```

print (dir('123'))    #[ '_add_', '_class_', '_contains_', ...]

```

类似于 `__xxx__` 的方法或者属性，在 `python` 中都是有特殊用途的，例如 `__len__` 方法返回长度。在 `python` 中如果你调用 `len` 函数试图获取此对象的长度，实际上在 `len` 函数的内部它自动的调用了该对象的 `__len__` 方法。

所以下面代码是等价的。

```

'zhe'.__len__()
3
len('zhe')
3

```

我们自己写的类，如果也想用 `len(myObj)` 的话，就自己写一个 `__len__()` 方法。

剩下的都是普通的方法和属性。

仅仅把属性和方法列出来是不够的，配合 `getattr()`、`setattr()` 以及 `hasattr()`，我们可以直接操作一个对象的状态：

```
class MyObject():
    def __init__(self):
        self.x = 16
    def power(self):
        return self.x*self.x

obj = MyObject()

print (hasattr(obj, 'x')) #true    实例有属性x
print (obj.x)            #16

print (hasattr(obj, 'y')) #false
setattr(obj, 'y', 32)
print (hasattr(obj, 'y')) #true
print (getattr(obj, 'y')) #32
print (obj.y)

#print (getattr(obj, 'z')) #抛出error

print (hasattr(obj, 'power')) #true
print (getattr(obj, 'power')) #bound method MyObject.power

fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
print (fn) #fn = obj.power

print (fn()) #16*16
```

小结

通过内置的一系列函数，我们可以对任意一个 Python 对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

```
sum = obj.x + obj.y
```

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):  
    if hasattr(fp, 'read'):  
        return readData(fp)  
    return None
```

假设我们希望从文件流 `fp` 中读取图像，我们首先要判断该 `fp` 对象是否存在 `read` 方法，如果存在，则该对象是一个流，如果不存在，则无法读取。`hasattr()`就派上了用场。

请注意，在 `Python` 这类动态语言中，根据鸭子类型，有 `read()` 方法，不代表该 `fp` 对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要 `read()`方法返回的是有效的图像数据，就不影响读取图像的功能。

6.5 实例属性和类属性

由于 `Python` 是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过 `self` 变量。

```
class Student():
    name = 'student' #belongs to class
    def __init__(self,name):
        self.name = name

s = Student('fz') #via self
s.score = 100 #via instance variable

print (Student.name) #student
```

如果 Student 类本身需要绑定一个属性呢？可以直接在 class 中定义属性，这种属性是类属性，归 Student 类所有。

```
class Student(object):
    name = 'Student'

s = Student()
print (s.name) #Student
s.name = 'fz' #绑定变量，优先级高于类变量
print (s.name) #fz
del s.name #删除后 重现类变量
print (s.name) #Student
```

注：在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

小结

实例属性属于各个实例所有，互不干扰；

类属性属于类所有，所有实例共享一个属性；

不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。

7. 面向对象高级编程

7.1 使用__slots__

7.2 使用@property

7.3 多重继承

7.4 定制类

7.5 使用枚举类

7.6 使用元类

8. 错误，调试和测试

9. IO 编程

10. 进程和线程

11. 常用内建模块

12. 常用的第三方模块

13. 网络编程

14. 电子邮件

15. 访问数据库

16. Web 开发

17. 异步 IO

18. 实战