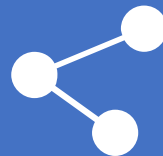


# Python

周嵩林

函数、类、模块





1

# 函数



## 函数 (function)

`def <函数名> (参数):`

`...`

```
def f():  
    pass
```

# 函数中至少要有有一个语句，可用pass占位

```
def f(a,b):  
    if a > b:  
        return a,b  
    else:  
        return b,a
```

# 函数可以返回多个参数，实际上是返回一个元组并把其中元素依次赋值给待返回变量

```
x,y = f(1,2)
```

# x = 2   y = 1

```
def pow(x,n=2):
```

# 默认参数

```
    return x**n
```

```
pow(3)
```

# 9

```
def f(*nums):
```

# 可变参数——元组参数，可以理解为将所有参数打包为tuple传入

```
    print(nums)
```

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

# (1,2)

```
f(*range(100))
```

# \*range(100)拆为100个数作为参数 (0,1,...99)

```
def f(a,**b):
```

# 关键字参数——字典参数，调用时必须给出键与值生成字典传入

```
    print('a=',a)
```

```
    print('b=',b)
```

```
f(0)
```

# a=0 b={}

```
f(a=0,b=1,c=2)
```

# a=0 b={ 'b' :1, 'c' :2 }



```
def f(a,b,*,c,d):
```

```
    pass
```

```
f(0,1)
```

```
f(0,1,c=2,e=3)
```

```
f(0,1,c=2,d=3)
```

# 命名关键字参数——\*后的参数必须有值（默认或传入）

# Error, 缺少关键字参数

# Error, 键错误

# 通过

```
def f(a,*b,c):
```

```
    print('a=',a,'b=',b,'c=',c)
```

```
f(0,2,4)
```

```
f(0,2,c=4)
```

```
f(0,2,4,c=8)
```

# 可变参数后的一般参数默认为命名关键字参数

# Error 未传入c

# a=0 b=(2,) c=4

# a=0 b=(2,4) c=8



匿名函数：使用lambda运算符快速定义匿名函数

```
f = lambda x,y:x*y
```

# 定义了函数f，传入参数x和y，返回x\*y

map函数：对整个列表所有元素执行同一函数

```
def f(x,y):  
    return x*y  
a = list(range(100))  
b = list(range(0,200,2))  
list(map(f,a,b))
```

# [ 0, 2, 8, 18, ...]

结合以上两个函数，以下式子

```
list(map(lambda x,y:x*y, list(range(100)), list(range(0,200,2))))
```

# [ 0, 2, 8, 18, ...]

也可用列表生成器，一般而言更简单明了

```
[x*y for x,y in list(zip(range(100),range(0,200,2)))]
```

还有很多很好用的特殊函数（filter, reduce等）可以自己探索学习



2

# 类与对象



**class <类名>:**

**def \_\_init\_\_(self, <参数>):**

...

**def <成员函数名>(self, <参数>):**

...

例 class Node:

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_\_y = y

def print(self):

print(self.\_\_x, self.\_\_y,

end = ',')

成员函数与变量命名规则:

字母开头 —— public

双下划线开头 —— private

单下划线开头 —— public

双下划线开头结尾 —— public

常用特殊成员函数有: \_\_init\_\_ \_\_del\_\_ \_\_str\_\_

\_\_eq\_\_ \_\_add\_\_ \_\_len\_\_

# 不能以双下划线结尾

# 避免使用这种命名

# 特殊成员函数

# private成员并不绝对, 使用\_<类名>\_\_<变量名>也可以访问, 但极不推荐





```
class Particle(Node):  
    def __init__(self, x, y, h):  
        Node.__init__(self, x, y)    # 也可以写 super().__init__(x, y)  
        self.__heavy = h  
    def print(self):  
        Node.print(self)             # 也可以写 super().print()  
        print(self.__heavy, end='')  
  
p1 = Particle(1, 2, 3)  
p1.print()                          # 1 2,3
```



### @staticmethod

# 静态成员函数，无法获得对象任何信息

```
class Particle(Node):
```

```
...
```

```
    @staticmethod
```

```
    def type():
```

```
        print('Particle')
```

# 不必加上self参数

```
p1 = Particle(1,2,3)
```

```
p1.type()
```

# Particle

### @classmethod

# 类成员函数，只能获得对象的类信息

```
class Particle(Node):
```

```
...
```

```
    @classmethod
```

```
    def type(cls):
```

```
        print(cls)
```

# 第一个参数必须为cls, 为Particle类

```
p1 = Particle(1,2,3)
```

```
p1.type()
```

# < class '\_\_main\_\_.Particle' >



### @property

# 将函数变为变量，用于直接调用

```
class Particle(Node):
```

```
...
```

```
@property
```

# 将函数变为可读变量

```
def heavy(self):
```

```
    return self.__heavy
```

```
@heavy.setter
```

# 为该变量增加可写属性

```
def heavy(self,h):
```

```
    self.__heavy = h
```

```
p1 = Particle(1,2,3)
```

```
print(p1.heavy)
```

# 3

```
p1.heavy = 5
```

```
print(p1.heavy)
```

# 5

通过使用@property，不必再为每个private变量都设计两个函数，每次调用函数读写同时，又可以在setter中判断输入数据合法性，相比直接使用public变量更安全还可以不声明 setter 使某个变量为只读属性



```
class Node:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def print(self):
        print(self.x, self.y)
```

```
n1 = Node (1,2)
```

```
n2 = n1
```

```
n2.x = 3
```

```
n1.print()
```

# 3 2 为什么出现这种结果?

# python 中的赋值, 实际都只是增加了一个**引用**, 不会复制数据

# 实际运用中, 此类问题大量存在于列表, 字典等容器中

```
a = [1,2,3]
```

```
b = a
```

```
b[0] = 4
```

```
print(a)    # [4,2,3]
```

解决: import copy

# import 会在后面讲到

```
a = [1,2,3]
```

```
b = copy.copy(a)
```

```
b[0] = 4
```

```
print(a)
```

# [1,2,3]



# copy也并非完全复制

```
a = [1,2,[3,4,5]]
```

```
b = copy.copy(a)
```

```
b[2][0] = 6
```

```
print(a)          # [1,2,[6,4,5]] 为什么?
```

# copy只复制最浅层, b从a中复制了整数 1,2 和列表 [3,4,5] 的引用, 解决方法为deepcopy

# 对象中有其他类的对象作为成员时, 与上述情况等同

```
class A:
```

```
    def __init__(self,x):
```

```
        self.x = x
```

```
class B:
```

```
    def __init__(self,a,y):
```

```
        self.a = a
```

```
        self.y = y
```

```
a1 = A(3)
```

```
b1 = B(a1,4)
```

```
b2 = copy.copy(b1)
```

```
b2.a.x = 8
```

```
print(b1.a.x)      # 8
```



3

## 文件操作



```
f = open('文件路径与名称', '标示符')
```

```
f = open('123.txt', 'r')
```

```
f.readline()
```

```
f.read()
```

```
f.close()
```

# 同一路径下可直接写文件名, r为只读, w为写, wb为写二进制

# 读取一行

# 读取所有内容

# 关闭

读文件时可能出错, 为了严谨, 应使用try获取异常, 但太过繁琐, 可使用以下方法:

```
with open('123.txt', 'r') as f:
```

```
    f.read()
```

# 不用再手动关闭

遇到非utf-8编码的文件:

```
with open('123.txt', 'r', encoding='ansi') as f:
```

写文件时, 只在close后才能完全写入, 否则会遗失在缓存区, 推荐都使用 **with** 语句



```
import os
```

# 首先import os包

```
os.path.abspath('.')
```

# 获取绝对路径

```
os.mkdir('dir1')
```

# 创建文件夹

```
os.rmdir('dir1')
```

# 删除文件夹

```
if not os.path.exists('t1.txt'):  
    with open('t1.txt', 'w'):  
        print('create t1.txt')
```

# 判断是否存在文件

# 创建文件，必须为可写属性

```
os.system('mkdir dir1 && cd dir1')
```

# 创建目录dir1并进入

# 使用 os.system 传入字符串等价于执行cmd命令

其他操作参考`help(os)`，以及搜索引擎





# 4

## 异常处理



```
try:                                # 运行直到检测出异常，跳到except
```

```
...
```

```
except <异常类型> [as x]:          # 捕获异常并处理
```

```
...
```

```
finally:                            # 不管有无异常，最后一定执行
```

```
...
```

```
try:
```

```
    a = 3/0
```

```
except ZeroDivisionError as z:
```

```
    print(z)
```

```
finally:
```

```
    print( "final" )
```

# 捕获异常时可以捕获该异常子类，若捕获Exception，则可捕获所有异常  
# division by zero

# final

```
try:
```

```
    a = 3/0
```

```
except ZeroDivisionError as z:
```

```
    raise
```

# 重新抛出异常给上层



```
class NodeException(Exception):  
    def __init__(self, err):  
        super().__init__(self)  
        self.err = err  
    def __str__(self):  
        return self.err
```

# 继承自Exception类

# \_\_str\_\_方法返回用于print的参数

```
class Node:  
    def __init__(self, x, y):  
        try:  
            if (type(x) != int or type(y) != int):  
                raise NodeException('type is not int')  
            self.x = x  
            self.y = y  
        except NodeException as e:  
            print(e)  
  
    def print(self):  
        print(self.x, self.y)
```

# 抛出异常

# 异常处理

```
n1 = Node('a', 2)
```

# type is not int



5

# 模块与包



## 模块 (Module)

一个.py文件就是一个模块

二者的调用方法相同

```
import math
```

```
math.sin(math.pi/2) # 1.0
```

```
from math import sin
```

```
sin(1) # 0.84147...
```

```
from math import *
```

```
sin(pi/2) # 1.0
```

```
import math as m
```

```
m.sin(m.pi/2) # 1.0
```

## 包 (Package)

包含多个模块与一个 `__init__.py` 文件的文件夹

常用模块:

os, sys

time

math

threading

json

re

socket

ctypes

pillow

numpy

scipy

系统相关函数

时间相关函数

数学函数

多线程

数据解析交换

正则表达式

socket通信

c/c++混合编程

图像处理

数值计算

科学计算

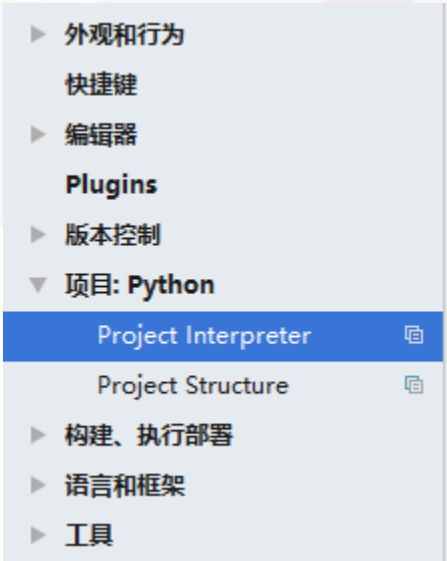
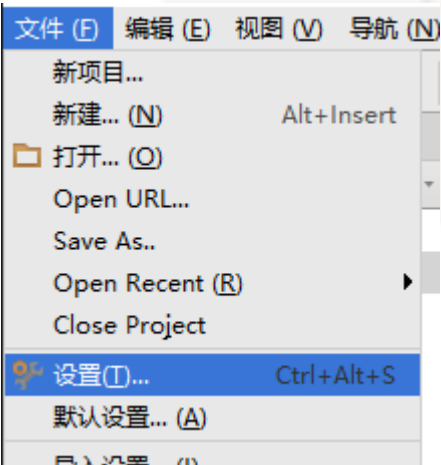


[The Python Standard Library](#) 列出了所有的内建包

安装第三方包

可使用 `pip3 install <包名称>` 安装第三方包，需要提前安装pip

pycharm中进入 文件->设置->项目->Project Interpreter也可以进行外部包管理



Package	Version	Latest
PyInstaller	3.3.1	3.3.1
Theano	1.0.1	➡ 1.0.2
altgraph	0.15	0.15
cycler	0.10.0	
future	0.16.0	0.16.0
kiwisolver	1.0.1	1.0.1
macholib	1.9	1.9
matplotlib	2.2.2	2.2.2
numpy	1.14.2	➡ 1.15.0rc1
opencv-python	3.4.1.15	3.4.1.15
pefile	2017.11.5	2017.11.5
pip	10.0.1	10.0.1
pyparsing	2.2.0	2.2.0
pywin32	223	223
python-dateutil	2.7.3	2.7.3
pytz	2018.4	➡ 2018.5
pywin32	223	223
scipy	1.0.1	➡ 1.1.0rc1
setuptools	18.1	➡ 39.2.0
six	1.11.0	1.11.0

万能方法：搜索引擎



6

# python脚本



python脚本除了使用一般.py文件的写法外，还有以下内容

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def main():
    print('Hello,world')

if __name__ == '__main__':
    main()
```





```
#!/usr/bin/env python3
```

#!/ 开头为选择执行这个脚本的解释器  
通过env打开python，原因为python位置不确定  
在win系统下，此行被忽略

```
# -*- coding: utf-8 -*-
```

指定此的脚本编码  
一般情况下均为 utf-8  
脚本出现乱码时请想起这一行



```
def main():
```

```
    print('Hello,world')
```

定义了main函数，与其他函数没有本质区别

并不会在执行时优先调用只是习惯上将其作为这一程序的主函数

```
if __name__ == '__main__':
```

```
    main()
```

这句话保证了这个脚本在被import的时候不会执行main

而在直接被执行的时候执行main函数

```
import a
```

```
a.__name__          # a
```

运行a, \_\_name\_\_ 为 '\_\_main\_\_'



一个脚本最前方的多行注释为它的\_\_doc\_\_

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''
abcd...
'''
```

通过a.\_\_doc\_\_访问

help()函数打印的就是\_\_doc\_\_



7

## 作业2



1. 实现一个类，拥有以下功能：

- (1) 开始计时
- (2) 暂停计时
- (3) 返回当前计时数
- (4) 停止计时并清零

然后分别对同样规模的循环和递归计时（尾递归会被优化为循环，不使用尾递归）

思考出现该计时结果的原因

2. 选做：有效利用搜索引擎

实现一个脚本，依次调用不同的py文件，

- (1) 将他们的**打印**输出重定向到一个文件中（例如txt）
- (2) 为所有不同时刻的输入添加输入时间（例如：2018.7.2 18:52）格式任意，并按时间排序
- (3) 在该文件中注明所有数据的输入来源，并按来源分类到文件中集中的某几行或不同文件中
- (4) 尝试在调用的文件中为打印输出添加各种噪声（无关打印内容），然后提取需要的信息到输出文件中
- (5) 考虑将py文件换为c/c++编译后的exe文件（或者shell脚本等）

必做小学期结束前，选做暑假结束前

提交 姓名-x-y.py 文件（或打包所有作业一起提交）到 [zhousl16@mails.tsinghua.edu.cn](mailto:zhousl16@mails.tsinghua.edu.cn)



8

# 深入学习



两天时间，并不足以讲python讲透，需要接下来更深入地学习

以下是一些途径：

1. [廖雪峰python教程](#)

2. [python官方文档](#)

3. [Stack Overflow](#)

4. Baidu, Google (\*)