

# 数据结构与算法 (Python)

## Python 入门

谢正茂 webg@PKU-Mail

计算机学院数据所

April 22, 2025

# 目录

- 程序设计思维
- Python 的运行和开发环境
- Python 程序
- 数据类型
- 语句和控制流
- 基本输入输出
- 函数和类



# 程序是什么? # 程序是菜谱

- 预热烤箱至 175 度
- 将面粉、苏打、盐、肉桂粉、姜粉、丁香粉混合过筛
- 准备大碗，加入黄油和糖粉，打发
- 打入鸡蛋、水和蜂蜜，搅拌
- 加入过筛混合物
- 取核桃大小面团，卷一层糖，压扁
- 放进烤箱烤 8-10 分钟



```
1 import Oven, Sifter, Bowl
2
3 oven = Oven()
4 bowl = Bowl()
5 sifter = Sifter()
6 oven.preheat(175)
7 ingredients = sifter.sift(
8     [flour, ginger, baking soda,
9      cinnamon, cloves, salt])
10 mixture = bowl.add([margarine, sugar])
11 while not mixture.is_light_fluffy():
12     mixture = bowl.cream()
13 bowl.add([egg, water, molasses])
14 bowl.stir()
15 bowl.add(ingredients)
16 bowl.stir()
17 dough = bowl.get(walnut_size)
18 dough.rollwith([sugar])
19 dough.flatten()
20 oven.add(dough)
21 oven.heat(8)
22 if not dough.welldone():
23     oven.heat(2)
```

# 如何用程序解决问题？求一些数的和

非程序思维是这样

- 有 2 个数

```
print (2+3)
```

- 有 3 个数

```
print (2+3+15)
```

- 有 8 个数

```
print  
(2+3+15+17+1+33+132+76)
```

- 有 1000 个数……?

程序思维是这样：通过不同的参数，  
同一段程序解决一类问题

- 有  $n$  个数

用一个 `sum` 来暂存结果

`sum = 第 1 个数`

对剩下的每一个数  $x$

把  $x$  累加到 `sum`

输出 `sum`

# 目录

- 程序设计思维
- Python 的运行和开发环境
- Python 程序
- 数据类型
- 语句和控制流
- 基本输入输出
- 函数和类



# Python 起名?

- 强大的冷血动物
- python(蟒)/ana'conda(蚺 rán)
- 域-界-门-纲-目-科-属-种
- Python 是标配, Anaconda 是 DS 套餐, 增加了 8000 多与数据科学和机器学习相关的开源包



**Python**  
Temporal range: Miocene–Present

PreC C S D C P T J K PgN

Burmese python (*Python bivittatus*)

**Scientific classification**

Domain:	Eukaryota
Kingdom:	Animalia
Phylum:	Chordata
Class:	Reptilia
Order:	Squamata
Suborder:	Serpentes
Family:	Boidae
Subfamily:	Boinae
Genus:	<i>Eunectes</i>

**Anaconda**  
Temporal range: Miocene-recent<sup>[1]</sup>

Green anaconda, *Eunectes murinus*

**Scientific classification**

Domain:	Eukaryota
Kingdom:	Animalia
Phylum:	Chordata
Class:	Reptilia
Order:	Squamata
Suborder:	Serpentes
Family:	Boidae
Subfamily:	Boinae
Genus:	<i>Eunectes</i>

# 安装 Python



- 除了 Python 的主程序外，还有一些相关工具：pip, ...
- 同一台电脑上有多个 Python 环境，互相“打架”怎么办？
  - Jupiter\_lab 和命令行中，可能就是两个 Python 环境
- Python 中运行 `import sys; print(sys.path)` 查看路径  
命令行运行 `Python -m pip install xpkg`，核对 `xpkg` 是否安在了正确的路径中

# 解决一个 NotFoundError

```
from line_profiler import LineProfiler
def foo(n):
    a = 5
    b = 6
    c = 10
    for i in range(n):
        for j in range(n):
            x = i * i
            y = j * j
            z = i * j
    for k in range(n):
        w = a * k + 45
        v = b * b
    d = 33

if __name__ == "__main__":
    lprofiler = LineProfiler(foo)
    lprofiler.run('foo(10)')
    lprofiler.print_stats()
```

```
ModuleNotFoundError                               Traceback (most recent call last)
Input In [2], in <cell line: 1>()
      1 from line_profiler import LineProfiler
      2 def foo(n):
      3     a = 5
ModuleNotFoundError: No module named 'line_profiler'
```

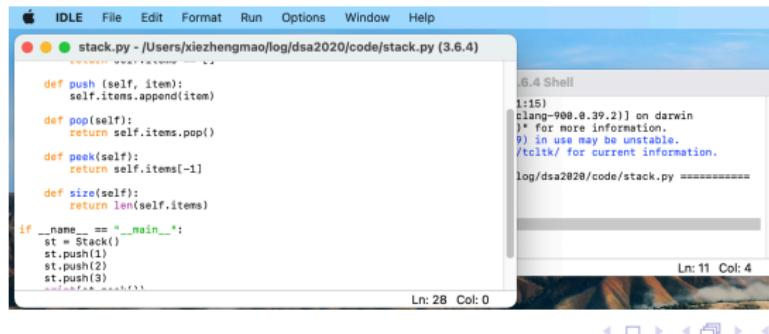
```
import sys
print(sys.path)
```

```
['/Users/zhenmoxie/log/dsa2020', '/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.8/lib/python38.zip', '/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.8/lib/python3.8', '/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.8/lib/python3.8/lib-dynload', '', '/Users/zhenmoxie/Library/Python/3.8/lib/python/site-packages', '/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.8/lib/python3.8/site-packages']
```

```
(python3.12) zhengmoxie@m1book dsa2020 % python -m pip install line_profiler
Requirement already satisfied: line_profiler in /Users/zhenmoxie/.virtualenvs/python3.12/lib/python3.12/site-packages (4.1.2)
```

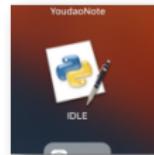
# 各个操作系统里的 Python: Windows

- 各个版本的 Windows 都需要额外安装 Python
  - 安装成功完成后，从程序菜单找 Python
- Command Line 是命令行界面
  - 只能交互式执行单个语句
  - 如何退出：Use quit() or Ctrl-D (i.e. EOF) to exit
- IDLE 是 Python 的“极简”图形界面
  - 拥有两种窗口：
  - 交互式单句执行窗口：Shell
  - 程序代码文件编辑窗口，编辑和保存程序文件，并在 Shell 中执行程序（Run->Run Module）



# 各个操作系统里的 Python: macOS

- macOS 内置有 Python，但可以安装更高版本的 Python
- 命令行界面从“Launchpad-> 其它-> 终端”，输入 python3
- IDLE 从“Launchpad”直接运行
- 命令行界面和 IDLE 跟 Windows 下一样



A screenshot of the Python IDLE application window. The menu bar shows "IDLE" as the active tab. The "Run" menu is open, with "Python Shell" selected. The main pane displays Python code and its output:

```
Python 3.5.1
[GCC 4.2.1 (GCC 4.2.1 20100531 (release) 2.5666) (dot 3)] on
Type "copyright", "credits" or "license()" for
>>> WARNING: The version of Tcl/Tk (8.5.9) in
Visit http://www.python.org/download/mac/tcltk
.
>>> print(2**100)
1267650600228229401496703205376
>>>
bt.py - /Users/chenbin/pytest/bt.py (3.5.1)

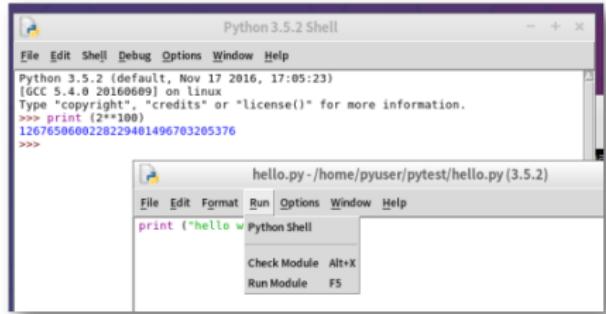
import turtle
turtle.tracer(0)

p= turtle.Pen()
```

# 各个操作系统里的 Python: 各种 Linux

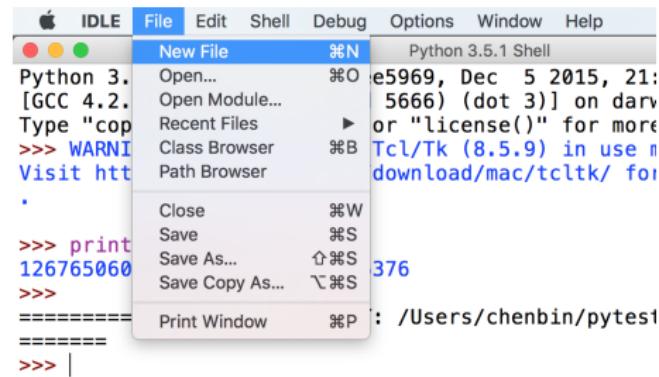
- 各种 Linux 都内置了 Python3
- 命令行界面也是从终端输入 python3 来启动
- 也具备 IDLE 的图形界面
  - 有简单自动安装命令  
sudo apt install idle3
- IDLE 的基本操作也是一样

```
pyuser@pyuser-VirtualBox:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print(2**100)
1267650600228229401496703205376
>>> █
```



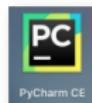
# 在 IDLE 中编辑和运行 Python 程序

- 启动 IDLE
- File->New File
- 在文件编辑窗口中输入代码
- 保存代码文件
  - 作业代码请建子目录保存到 Git 的本地仓库
  - 目录名和文件名不要用中文
- Run->Run Module 运行程序
- 在 Python Shell 中查看结果



# 集成开发环境：PyCharm

- 首先 New Project
- 创建 homework 目录
- 选择好 Python3 的解释器
- 然后 File->New… 来创建 Python File
- 有巨多高级特性帮助快速编写程序
- Run->Run… 来运行程序
- 可以 Tools->Python Console 调出命令行界面来执行单条语句



A screenshot of the PyCharm Community Edition interface. The top menu bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, and Window. The main window shows a project named 'homework' with files like 'bubblesort.py', 'bj2016weather.csv', and 'bjpm25.csv'. The code editor displays a Python script named 'bubblesort.py' with the following code:

```
def bubbleSort(lst):
    for pnum in range(len(lst) - 1,
                       for i in range(pnum):
                           if lst[i] > lst[i + 1]:
                               lst[i], lst[i + 1] =
lst = [23, 45, 12, 100, 10, 0, 9]
bubbleSort(lst)
print(lst)
lst = ['aa', 'bzx', 'kylis', 'google'
bubbleSort(lst)
print(lst)
```

The bottom status bar shows the command line output: 'Process finished with exit code 0'.

# JupyterLab: 一个非常好的 Python 学习工具

- JupyterLab is the latest web-based interactive development environment for notebooks, code, and data.
- <https://jupyter.org/>
- `$ pip install jupyterlab or $ brew install jupyterlab`

The screenshot shows the JupyterLab interface. On the left, there is a file browser window titled 'localhost:8888/lab/tree/python简明教程.ipynb' showing a directory structure with files like ch07.aux, ch07.log, etc. On the right, there is a code editor window titled 'python简明教程.ipynb' containing Python code and its output. The code in cell [2] is:

```
[2]: from platform import python_version
print(python_version())
3.7.4
```

The output is '3.7.4'. Below it, the code in cell [3] is:

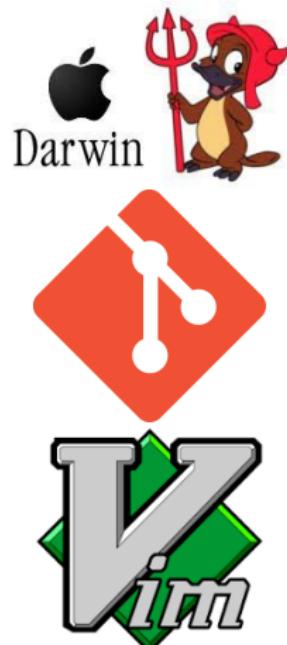
```
[3]: print("Hello World!")
print("Tom")
print("Hello World!",end=' ')
print("Tom", end=' ')
```

The output is:

```
Hello World!
Tom
Hello World! Tom
超级计算机,超极大的数都没问题.
```

# 个人习惯： Unix/Linux/Mac 下的命令行

- Linux 与 Unix 同源，很多程序是通用的。
- Mac 的内核是 darwin, Unix
- 完备的生态，分工合作，大师出品
  - 文本编辑：vim, emacs
  - 文本处理：grep, sed, awk
  - 编程：make, gcc, **python**, ...
  - 论文写作：**LATEX**
  - 文档、代码管理：**Git**
- 小身板，高稳定，力量大；
- 效率高，大量的快捷键，高 APM (Actions Per Minute)
- 要记很多东西，初期学习曲线陡。
- 后面的一些例子采用了这样的开发环境，但不影响代码。



# 第一个 Python 语句；超级计算器

- 打开 IDLE
- 在 Python Shell 中输入语句
  - `print ("Hello World!")`
- 立即看到运行结果！
- 可以计算  $2^{100}$ ！
- 也可以直接输入算式，当计算器用
- 超级大的数都没问题

```
Mini2-1:dsa2020 zhengmaoxie$ python3
Python 3.6.4 (default, Jan  7 2018, 20:51:00)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World!")
Hello World!
>>> print (2**100)
1267650600228229401496703205376
>>> 2**100
1267650600228229401496703205376
>>> █
```

# Python2 or Python3?

- `print'HelloWorld!'` vs. `print("HelloWorld!")`
- 全国计算机等级考试 Python 语言程序设计要求 Python3.5 版本及以上
- Python2.7 在今年初开始已经停止维护
- 有些公司还在坚持用 Python2；一些基础软件依赖于 python2
  - 大量的代码是 Python2 的，升级成本高。
  - 担心兼容性问题。（Mysql5.7 & Mysql8）
- 让自己的代码同时兼容 python2 和 python3？也麻烦！
- 对于我们，选择 Python3，有时电脑上装了几个版本的 Python，需要确认一下。

看看当前的Python版本

```
from platform import python_version  
  
print(python_version())
```

3.7.4

# 目录

- 程序设计思维
- Python 的运行和开发环境
- Python 程序
- 数据类型
- 语句和控制流
- 基本输入输出
- 函数和类



# 第一个 Python 程序：Hello World！

- 编辑文件 `hw.py`
- 在文件中写入一行代码，保存退出。  
`print ("Hello World!")`
- 查看一下 py 文件  
`vimcat hw.py`
- 执行程序  
`python hw.py`  
`python3 hw.py`
- 对比 C 程序，省事很多。
  - 脏活、累活别人帮你做了。
  - 语法的确简洁，属于“人狠话不多”。

```
Mini2-1:code zhengmaoxie$ vimcat hw.py
print ("Hello World!")
Mini2-1:code zhengmaoxie$ python hw.py
Hello World!
Mini2-1:code zhengmaoxie$ python3 hw.py
Hello World!
Mini2-1:code zhengmaoxie$ █

Mini2-1:code zhengmaoxie$ vimcat hw.c
#include <stdio.h>

int
main()
{
    printf("Hello World!\n");
    return 0;
}
Mini2-1:code zhengmaoxie$ gcc -g -o hw hw.c
Mini2-1:code zhengmaoxie$ ./hw
Hello World!
Mini2-1:code zhengmaoxie$ █
```

# 代码缩进：视觉效果和功能的统一

- 程序块 block 是代码中经常出现的组织方式
- 用'{}'容易出现的问题：C/java
  - 第 12 行的 **go fail;** 被误以为属于 if\_block；实际上被无条件执行。
  - 整个函数始终以 *fail* 返回；
- 让眼睛更快、更轻松：
  - 缩进 vs. '{}'
  - 颜色
- 缩进产生的 block 更加直观

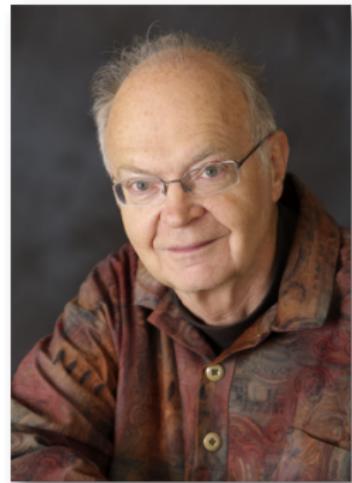
```
1 static OSStatus
2 SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedPa
3                                     uint8_t *signature, UInt16 signatureLen)
4 {
5     ...
6     ...
7     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
8         goto fail;
9     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
10        goto fail;
11    if ((err = SSLHashSHA1.final(&hashCtx, &hashout)) != 0)
12        goto fail;
13    err = sslRawVerify(ctx,
14                      ...
15                      ...
16                      ...
17                      ...
18                      ...
19                      ...
20                      ...
21                      ...
22                      ...
23                      ...
24                      ...
25                      ...
26                      ...
27 fail:
28     SSLFreeBuffer(&sig);
29     SSLFreeBuffer(&hash);
30     return err;
31 }
```

```
7     if ((err = SSLHashSHA1
8         goto fail;
9     if ((err = SSLHashSHA1
10        goto fail;
11        goto fail;
12        goto fail;
13        goto fail;
14        goto fail;
15        goto fail;
16        ...
17        ...
18        ...
19        ...
20        ...
21        ...
22        ...
23        ...
24        ...
25        ...
26        ...
27 fail:
28     SSLFreeBuffer(&sig);
29     SSLFreeBuffer(&hash);
30     return err;
31 }
```

- Python 的强制缩进规范完成了关键部分
- 我们还需要良好的编程规范
  - 变量、函数、类命名
  - 注释和文档
  - 一些编程设计上的良好风格
- Programs are meant to be read by humans and only incidentally for computers to execute.

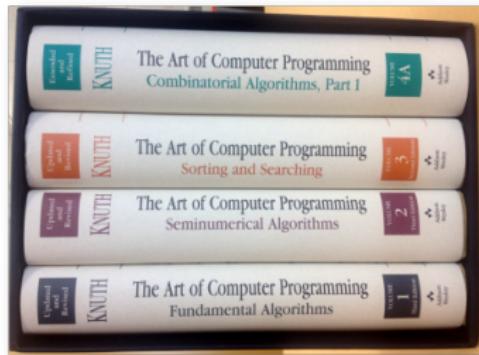
—Donald Ervin Knuth

- 大师万万没有想到，“软件开发”作为工程，
  - 不是所有人都有良好的素质和足够的训练。
  - 如何减少局部缺陷和防止风险外溢
  - ddl counts



# 程序员的精彩人生

- 鸿篇巨著《计算机程序设计艺术》
- 为了巨著印刷漂亮，开发了伟大的排版软件  $\text{\TeX}$ 
  - 有趣的版本号 3.14159265，最后是  $\pi$
  - 奖励 bug 提交者，从 128 美分开始翻倍，但得到奖金的人往往不愿拿支票去兑现
- 字符串快速匹配 KMP 算法
- 1974 年获得图灵奖



# Python 语言的几个要件

## 数据对象和组织

- 对现实世界实体和概念的抽象
- 分为简单类型和容器类型
- 简单类型用来表示值
  - 整数 int、浮点数 float、复数 complex、逻辑值 bool、字符串 str
- 容器类型用来组织这些值
  - 列表 list、元组 tuple、集合 set、字典 dict
- 数据类型之间几乎都可以转换

## 赋值和控制流

- 对现实世界处理和过程的抽象
- 分为运算语句和控制流语句
- 运算语句用来实现处理与暂存
  - 表达式计算、函数调用、赋值
- 控制流语句用来组织语句描述过程
  - 顺序、条件分支、循环
- 定义语句也用来组织语句，描述一个包含一系列处理过程的计算单元
  - 函数定义、类定义

# 目录

- 程序设计思维
- Python 的运行和开发环境
- Python 程序
- 数据类型
- 语句和控制流
- 基本输入输出
- 函数和类



# Python 可变类型和不可变类型

- 不可变类型

- 数字
- 字符串
- 元组

- 变量只是对象的标签
- 每次运算都生成一个新对象
- 元组不能够元素赋值
- 为什么元组？元组的出现让 Python 程序更优雅

```
def 碾米(谷子) :  
    return 米, 糜
```

米, 糜 = 碾米(谷子)

```
>>> a=20  
>>> b=a  
>>> id(a)  
4498775232  
>>> id(b)  
4498775232  
>>> b=b+1  
>>> id(b)  
4498775264  
>>> id(a)  
4498775232  
>>> |
```

- 可变类型

- 列表
- 集合
- 字典
- ...

- 可以改变，可以赋值
- 不同的变量可能指向同一个对象

# Python 数据类型：整数 int、浮点数 float

- Python 语言本身没有限制整数的大小

下面两行程序的运行结果，Out Of:  
Memory/Paper/Patience?

```
7***(7***7)  
700***(700***700)
```

- 不得已的情况下，用 `sys.maxsize` 表示  $\infty$
- 运算：加/减/乘/除、整除、求余、求幂...
  - 除法“/”在 python2 和 3 中定义不同
- 浮点数的操作也差不多，但受到 17 位有效数字的限制，存在精度丢失问题。
- 一些常用的数学函数如 `sqrt/sin/cos` 等都在 `math` 模块中

```
import math  
math.sqrt(2)
```

```
>>> 5  
5  
>>> -100  
-100  
>>> 5 + 8  
13  
>>> 90 - 10  
80  
>>> 4 * 7  
28  
>>> 7 / 2  
3.5  
>>> 7 // 2  
3  
>>> 7 % 3  
1  
>>> 3 ** 4  
81  
>>> 2 ** 100  
1267650600228229401496703205376  
>>> divmod(9, 5)  
(1, 4)  
>>> |
```

```
|>>> 0.1+0.2  
0.30000000000000004
```

# Python 数据类型：复数

- Python 内置对复数的计算
- 支持所有常见的复数计算
- 对复数处理的数学函数在模块 cmath 中

```
import cmath  
cmath.sqrt(1+2j)
```

- 复数的表示:  $m + nj$  ( $n = 1$  时不能省略)

```
>>> 1+3j  
(1+3j)  
>>> (1+2j)*(2+3j)  
(-4+7j)  
>>> (1+2j)/(2+3j)  
(0.6153846153846154+0.07692307692307691j)  
>>> (1+2j)**2  
(-3+4j)  
>>> (1+2j).imag  
2.0  
>>> (1+2j).real  
1.0  
>>>
```

```
>>> 1+1j  
(1+1j)  
>>> 1+j  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'j' is not defined  
>>> |
```

# Python 数据类型：逻辑值

- 逻辑值仅包括 *True*/*False* 两个
- 用来配合 *if*/*while* 等语句做条件判断
- 其它数据类型可以转换为逻辑值：
  - 例如数值： $0 \Rightarrow False$
  - 非  $0 \Rightarrow True$

```
>>> True
True
>>> False
False
>>> 1>2
False
>>> 23<=34
True
>>> bool(0)
False
>>> bool(999)
True
>>> if (2>1):
    print ("OK")
```

OK

>>> |

# 字符串：Python 最精彩的地方之一

- 最大的特点是 Python 字符串不可修改，只能生成新的字符串
- 用双引号或者单引号都可以表示字符串
- 多行字符串用三个连续单引号表示
- 特殊字符用转义符号“\”表示
  - 制表符"\t"，换行符号\n
- 字符串操作：
  - + 连接、\* 复制、len 长度
  - [start : end : step] 用来提取一部分：同 list，最“好用”的操作。

```
>>> 'abc'
'abc'
>>> "abc"
'abc'
>>> '''abc def
ghi jk'''
'abc def\nghi jk'
>>> "Hello\nWorld!"
'Hello\nWorld!'
>>> print ("Hello\nWorld!")
Hello
World!
>>> 'abc' + 'def'
'abcdef'
>>> 'abc' * 4
'abcabcabcabc'
>>> len('abc')
3
>>> 'abcd'[0:2]
'ab'
>>> 'abcd'[0::2]
'ac'
```

# Python 数据类型：字符串

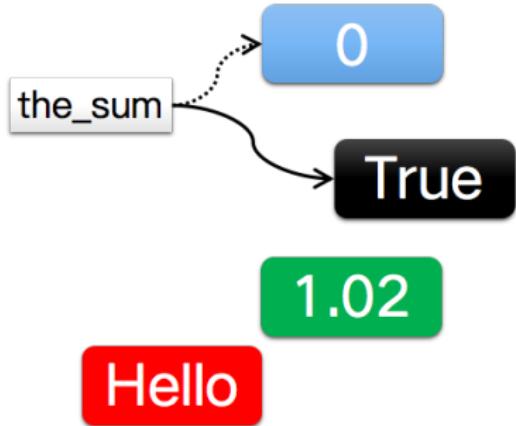
- 一些高级操作：

- `split`: 分割; `join`: 合并
- `upper/lower/swapcase`: 大小写相关
- `ljust/center/rjust`: 排版左中右对齐
- `replace`: 替换子串

```
>>> 'You are my sunshine.'.split(' ')
['You', 'are', 'my', 'sunshine.']
>>> '-'.join(["One", "for", "Two"])
'One-for-Two'
>>> 'abc'.upper()
'ABC'
>>> 'aBC'.lower()
'abc'
>>> 'Abc'.swapcase()
'aBC'
>>> 'Hello World!'.center(20)
'    Hello World!    '
>>> 'Tom smiled, Tom cried, Tom shouted'.replace('Tom', 'Jane')
'Jane smiled, Jane cried, Jane shouted'
```

# Python 变量机制：数据对象的引用

- `the_sum=0` 实际上把数据对象“0”的引用/指针赋值给 `the_sum`
- 变量可以指向任何一个数据对象，比如 `True`, `1.02`, 或者“`Hello`”
- 变量的类型随着指向的数据对象类型改变而改变！
- Python 的类型是“动态”的
- 强类型 vs. 弱类型，  
静态类型 vs. 动态类型
- 对于两个变量的判断：`==`, `is`
  - `a is b` means `id(a)==id(b)`



# Python 容器类型：列表和元组

- Python 中有几种类型是一系列元素组成的序列，采用了顺序表的存储结构，以整数作为下标访问
- 字符串 str 就是一种同类元素的序列
- 列表 list 和元组 tuple 则可以容纳不同类型的元素，构成序列
- 元组是不能再更新（不可变）序列
  - 字符串也是不能再更新的序列
- 列表则可以删除、添加、替换、重排列中的元素
  - 可变类型
  - 字符串也借用它来做变形
- 列表或元祖之间的比较操作“<”，同字符串一样采用字典排序的规则

字符串str									
0	1	2	3	4	5	6	7	8	
H	e	l	l	o		T	o	m	
列表list									
0	1	2	3	4	5	6	7	8	
123	2.4	'ab'	True	None	[1, 2]	(2, 3)	556	0	
元组tuple									
0	1	2	3	4	5	6	7	8	
123	2.4	'ab'	True	None	[1, 2]	(2, 3)	556	0	

# 比较：列表和元组，很多共同的操作

- 创建列表：[] 或者 list()
- 创建元组：() 或者 tuple()
- 用索引 [n] 获取元素（列表可变）
- +：连接两个列表/元组
- \*：复制 n 次生成新列表/元组
- len()：列表/元组中元素的个数
- in：某个元素是否存在
- [start : end : step]：切片
- 元素赋值：列表可以，元组不行

```
>>> alist[0] = False
>>> alist
[False, True, 0.234]
```

```
>>> []
[]
>>> list()
[]
>>> alist = [1, True, 0.234]
>>> alist[0]
1
>>> alist + ["Hello"]
[1, True, 0.234, 'Hello']
>>> alist * 2
[1, True, 0.234, 1, True, 0.234]
>>> len(alist)
3
>>> 1 in alist
True
>>> alist
[1, True, 0.234]
>>> alist[1:3]
[True, 0.234]
>>> alist[0:3:2]
[1, 0.234]
>>> alist[::-1]
[0.234, True, 1]

>>> ()()
>>> tuple()
()
>>> atuple = (1, True, 0.234)
>>> atuple[0]
1
>>> atuple + ("Hello",)
(1, True, 0.234, 'Hello')
>>> atuple * 2
(1, True, 0.234, 1, True, 0.234)
>>> len(atuple)
3
>>> 1 in atuple
True
>>> atuple
(1, True, 0.234)
>>> atuple[1:3]
(True, 0.234)
>>> atuple[0:3:2]
(1, 0.234)
>>> atuple[::-1]
(0.234, True, 1)

>>> atuple[0] = False
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in <module>
    atuple[0] = False
TypeError: 'tuple' object does not support item assignment
```

# 列表 list 的其它方法，只有它是可变的

方法名称	使用例子	说明
append	alist.append(item)	列表末尾添加元素
insert	alist.insert(i,item)	列表中i位置插入元素
pop	alist.pop()	删除最后一个元素，并返回其值
pop	alist.pop(i)	删除第i个元素，并返回其值
sort	alist.sort()	将表中元素排序
reverse	alist.reverse()	将表中元素反向排列
del	del alist[i]	删除第i个元素
index	alist.index(item)	找到item的首次出现位置
count	alist.count(item)	返回item在列表中出现的次数
remove	alist.remove(item)	将item的首次出现删除

# 可变类型的变量引用情况

- 由于变量的引用特性可变类型的变量操作需要注意
- 变量赋值不会复制对象，只会引用同一个对象
- 通过其中一个变量修改了可变类型对象，其它变量也看得到

- alist = [1,2,3,4]
- blist = alist
- blist[0] = 'abc'

- 在下面的 A=[myList]\*3 中，是变量被复制了三份，而不是对象。
- 如何获得一个二维数组？

Python 3.6

```
1 myList = [1,2,3,4]
2 A = [myList]*3
3 print(A)
4 myList[2]=45
5 print(A)
```

→ line that has just executed  
→ next line to execute

< Back Program terminated Forward >

Visualized using [Python Tutor](#) by Philip Guo

Print output (drag lower right corner to resize)

```
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
[[1, 2, 45, 4], [1, 2, 45, 4], [1, 2, 45, 4]]
```

Frames Objects

Global frame

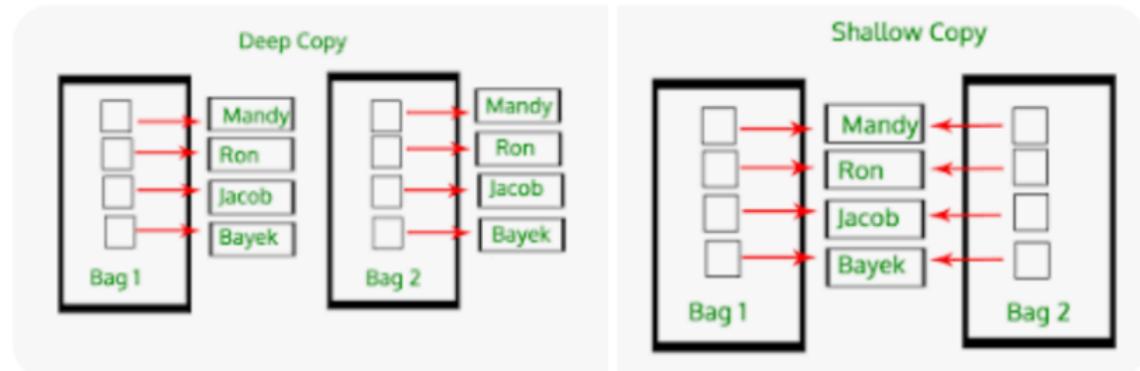
list

0	1	2	3
1	2	45	4

list

0	1	2
---	---	---

# 如何进行对象的拷贝



- `import copy`
- 浅拷贝: `Bag2 = copy.copy(Bag1)`
- 深拷贝: `Bag2 = copy.deepcopy(Bag1)`

# 常用的连续序列生成器：range 函数

- range(n)
  - 从 0 到 n-1 的序列
- range(start, end)
  - 从 start 到 end-1 的序列
- range(start, end, step)
  - 从 start 到 end-1,
  - 步长间隔 step, 可以是负数
- range 函数返回 range 类型的对象，可以直接当做序列用，也可以转换为 list 或者 tuple 等容器类型
- 用在 for\_loop 里面：  
`for i in range(10)`

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
>>> list(range(10, 1, -2))
[10, 8, 6, 4, 2]
>>> range(10)
range(0, 10)
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

# Python 容器类型：集合 set

- 集合是不重复元素的无序组合
- 用 `set()` 创建空集
- 可用 `set()` 从其它序列转换生成集合
- 集合的常见操作
  - `in`: 判断元素是否属于集合
  - `|, union()`: 并集
  - `&, intersection()`: 交集
  - `-, difference()`: 差集
  - `^, symmetric_difference()`: 异或
  - `<=,<,>=,>`: 子集/真子集/超集/真超集

```
>>> set()
set()
>>> asset = set('abc')
>>> asset
{'c', 'a', 'b'}
>>> 'a' in asset
True
>>> asset | set('bcd')
{'c', 'd', 'a', 'b'}
>>> asset & set(['b', 'c', 'd'])
{'c', 'b'}
>>> asset - set(['b', 'c', 'd'])
{'a'}
>>> asset ^ set('bcd')
{'a', 'd'}
>>> asset <= set('abcd')
True
>>> asset > set('abcd')
False
```

# Python 容器类型：集合 set

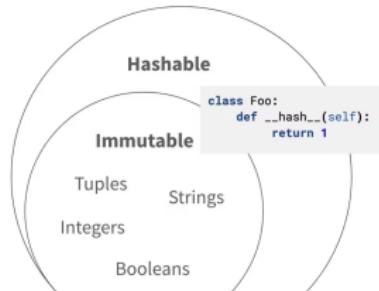
- `add(x)`: 集合中添加元素
- `remove(x)`: 集合中删除指定元素
- `pop()`: 删除集合中任意元素并返回其值
- `clear()`: 清空集合成为空集
- `in` 操作列表也有，但是集合的更高效

```
>>> aset
{'c', 'a', 'b'}
>>> aset.add(1.23)
>>> aset
{'c', 1.23, 'a', 'b'}
>>> aset.remove('b')
>>> aset
{'c', 1.23, 'a'}
>>> aset.pop()
'c'
>>> aset
{1.23, 'a'}
>>> aset.clear()
>>> aset
set()
```

# Python 容器类型：字典 dict

- 字典是通过键值 **key** 来索引元素 **value**, 而不是象列表是通过连续的整数来索引
- 字典是可变类型，可以添加、删除、替换元素
- 字典中的元素 **value** 没有顺序，可以是任意类型
- 字典中的键值 **key** 和集合的元素都必须是 **hashable**
- 数值、字符串、元组、布尔值是不可变类型 **immutable**
- **immutable** ⊂ **hashable**
- **in** 操作判断的是 **key**

```
>>> student = {'name':'Tom', 'age':20, 'gender':'Male', 'course':['math', 'computer']}
>>> student
{'name': 'Tom', 'age': 20, 'course': ['math', 'computer'], 'gender': 'Male'}
>>> student['name']
'Tom'
>>> student['age']
20
>>> student['age'] = 19
>>> student
{'name': 'Tom', 'age': 19, 'course': ['math', 'computer'], 'gender': 'Male'}
>>> student['course'].append('chemistry')
>>> student
{'name': 'Tom', 'age': 19, 'course': ['math', 'computer', 'chemistry'], 'gender': 'Male'}
>>> 'gender' in student
True
>>> student.keys()
dict_keys(['name', 'age', 'course', 'gender'])
>>> student.values()
dict_values(['Tom', 19, ['math', 'computer', 'chemistry'], 'Male'])
>>> student.items()
dict_items([('name', 'Tom'), ('age', 19), ('course', ['math', 'computer', 'chemistry']), ('gender', 'Male')])
```



# 建立大型数据结构（嵌套/复合）

- 嵌套列表

- 列表的元素是一些列表:  $alist[i][j]$

- 字典的元素可以是任意类型，甚至也可以是字典

- $bands = \{'Marxes': ['Moe', 'Curly']\}$

- 字典的键值可以是任意不可变类型，例如用元组来作为坐标，索引元素

- $poi = \{(100, 100): 'busstop'\}$

```
>>> alist=[[23, 34, 45], [True, 'ab']]  
>>> alist[0][2]  
45  
>>> bands={'Marxes': ['Moe', 'Curly'], 'KK': [True, 'moon']}  
>>> bands['KK'][0]  
True  
>>> poi={(100,100):'Zhongguancun', (123,23):'Pizza'}  
>>> poi[(100,100)]  
'Zhongguancun'
```

```
>>> grades = {'Tom': {'English': 95}}  
>>> grades  
{'Tom': {'English': 95}}  
>>> grades['Tom']['math']=100  
>>> grades  
{'Tom': {'English': 95, 'math': 100}}  
>>> 
```

# 输入和输出：input/print 函数

- **input(prompt)**

- 显示提示信息 **prompt**, 用户输入的内容以字符串形式返回

- **print(v1, v2, v3, ...)**

- 打印各变量的值输出
- 可选参数 **end**, 表示打印后以这个字符串结尾, 缺省为换行符'\n', 打印不想换行的话, 则 **end=""** 空串
- 可选参数 **sep**, 表示变量之间用什么字符串隔开, 缺省为空格

- 格式化字符串

- **format** 方法 (参考 ipynb)

- 文件的读写: **open()**

```
>>> yname = input ("Please input your name")
Please input your nameTom Hanks
>>> yname
'Tom Hanks'
>>> print (1, 23, 'Hello')
1 23 Hello
>>> print (1, 23, 'Hello', end='')
1 23 Hello
>>> print (1, 23, 'Hello', sep=',')
1,23>Hello
>>> '%d %s' % (23, 'Hello')
'23 Hello'
>>> '%d' % (23,)
'23'
>>> '(%4d):%s' % (12, 'Hello')
'( 12):K>Hello'
>>> '(%04d):%s' % (12, 'Hello')
'(0012):K:      Hello'
```

# 输入和输出：OJ-新人易犯错误

样例输入

10

样例输出

3 7

```
1 def isprime(n):
2     for i in range(2,n):
3         if n % i == 0:
4             return False
5     return True
6
7 def Goldbach_conjecture(m):
8     if not 0 < m <= 10000 :
9         print("the number must be not more than 10000")
10    else :
11        for j in range(2,int(m/2)+1):
12            if isprime(j):
13                k = m - j
14                if isprime(k):
15                    return (j,k)
16                continue
17        return('not matched')
18
19
20 m = int(input('请输入一个10000以内的正整数'))
21 print(Goldbach_conjecture(m))
```

关于 `input()`, 你回去看一下。



请问是犯了什么低级错误吗?



好的好的，谢谢老师，我检查一下！



看一下录像，说了每年都有同学犯这个错误。



好的，我坐后面有时候会有点听不清，下次挪到前面来

3/7/24 4:08 PM



谢谢老师，我解决了！输出部分的格式要求也特别严格，3 7就不能说3, 7，也不能用函数输出none值或其他值



再次感谢老师

# 运算语句：表达式、函数调用和赋值

- 各种类型的数据对象，可以通过各种运算组织成复杂的表达式
- 调用函数或者对象，也可以返回数据，所有可调用的事物称为 **callable**
  - 调用函数或者对象，需要在其名称后加圆括号，如果有参数，写在圆括号里
  - 不加圆括号的函数或者对象名称仅是表示自己，不是调用
- 将表达式或者调用返回值传递给变量进行引用，称为赋值
- 函数也是对象，可以把他的指针传递给变量

```
>>> 12 * 34.5 + 23.4
437.4
>>> ('abc' + '123') * 3
'abc123abc123abc123'
>>>
>>> import math
>>> math.sqrt(12)
3.4641016151377544
>>> math.sqrt
<built-in function sqrt>
>>>
>>> n = 12 * 34
>>> n
408
>>> p2 = math.sqrt(2)
>>> p2
1.4142135623730951
>>> pfg = math.sqrt
>>> pfg
<built-in function sqrt>
>>> pfg(2)
1.4142135623730951
```

# 赋值语句的小技巧

- 级联赋值语句

- $x = y = z = 1$

- 多个变量分解赋值

- $a, b = ['hello', 'world']$
  - $a, b = 'hello', 'world'$

- 变量交换

- $a, b = b, a$

- 增量赋值运算

- $i += 1$
  - $n *= 45$

```
>>> x = y = z = 1
>>> x,y,z
(1, 1, 1)
>>> a, b= ['hello', 'world']
>>> a
'hello'
>>> b
'world'
>>>
>>> a, b = b, a
>>> a
'world'
>>> b
'hello'
>>>
>>> a += 'cup'
>>> a
'worldcup'
```

# 控制流语句：条件 if

- 条件语句

- if <逻辑条件>:  
    <语句块>  
elif <逻辑条件>: # 可多个 elif  
    <语句块>  
else: # 仅 1 个  
    <语句块>

- 逻辑条件，各种类型中某些值等效为 False，其它值则是 True:

- None, 0, 0.0, 空字符串, [], (), {}, set()
- x 等效为 False 或者 True，并不意味着它等于 ==
  - None == True, None == False, '' == True, '' == False
  - 1 == True, 2 == True
- 所以 if result: 和 if result == True: 的效果是不同的

```
>>> a = 12
>>> if a > 10:
...     print ("Great!")
... elif a > 6:
...     print ("Middle!")
... else:
...     print ("Low!")
```

Great!

# 控制流语句：while 循环

- 条件循环 while
  - while <逻辑条件>:  
    <语句块>  
    break # 跳出整个 while 结构（包括 else 部分）  
    continue # 略过余下循环语句，继续 while  
    <语句块>  
    else: #while 的条件不满足退出循环，执行一次：  
        <语句块>
- C 语言中的 while 没有 else 部分，后面的代码常常需要想其他办法去判断循环是如何退出的

# 控制流语句：for 循环

- 迭代循环 for:

- for < 变量 > in < 可迭代对象 >:  
    < 语句块 >  
    break # 跳出循环  
    continue # 略过余下  
    else: # 迭代完毕执行  
        < 语句块 >

- 可迭代对象有很多类型

- 象字符串、列表、元组、字典、集合等
- 也可以有后面提到的生成器、迭代器等

```
>>> for n in range(5):
    print (n)
0
1
2
3
4
>>> alist = ['a', 123, True]
>>> for v in alist:
    print (v)
a
123
True

>>> adic = {'name':'Tom', 'age':18, 'gender':'Male'}
>>> for k in adic:
    print (k, adic[k])
name Tom
age 18
gender Male

>>> for k, v in adic.items():
    print (k, v)
name Tom
age 18
gender Male
```

# 推导式 comprehensions (又称解析式)

- 可以用来生成列表、字典和集合的语句
  - [`<表达式>for <变量>in <可迭代对象>if <逻辑条件>`]
  - {`<键值表达式>:<元素表达式>for <变量>in <可迭代对象>if <逻辑条件>`}
  - {`<元素表达式>for <变量>in <可迭代对象>if <逻辑条件>`}
- 代码简洁易懂

```
>>> [x**x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> {'K%d'%(x,):x**3 for x in range(10)}
{'K2': 8, 'K8': 512, 'K5': 125, 'K6': 216, 'K3': 27, 'K9': 729, 'K0': 0,
'K7': 343, 'K1': 1, 'K4': 64}
>>>
>>> {x**x for x in range(10)}
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
>>>
>>> {x+y for x in range(10) for y in range(x)}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

# 推导式 comprehensions (又称解析式)

- 可以使用多个 for 结构
- 通过 if 结构进行过滤
- 万人坑：如何获得一个初始值为 0， 大小为  $10 \times 10$  的二维数组？
  - `x=[ [0] * 10] * 10`
  - `y=[ [0] * 10 for _ in range(10)]`

```
>>> [x+y for x in range(10) for y in range(x)]
[1, 2, 3, 3, 4, 5, 4, 5, 6, 7, 5, 6, 7, 8, 9, 6, 7, 8, 9, 10, 11, 7, 8, 9,
 , 10, 11, 12, 13, 8, 9, 10, 11, 12, 13, 14, 15, 9, 10, 11, 12, 13, 14, 15
 , 16, 17]
>>>
>>> [x*x for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>>
>>> [x.upper() for x in [1, 'abc', 'xyz', True] if isinstance(x, str)]
['ABC', 'XYZ']
```

# 生成器推导式

- 与推导式一样语法：
  - (<元素表达式> for <变量> in <可迭代对象> if <逻辑条件>)
- 返回一个生成器对象，也是可迭代对象，但并不立即产生全部元素，仅在要用到元素的时候才生成，可以极大节省内存

```
>>> agen = (x*x for x in range(10))
>>> agen
<generator object <genexpr> at 0x10c5bbf10>
>>> for n in agen:
...     print(n, end=', ')
...
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, >>>
>>>
```

# 括号的用法：灵活多用也有坑

- 方括号 []：列表 list
  - l = [1, 2, 'sina']
- 大括号 {}：字典 dict、集合 set
  - d = {'Tom':25, 'John':26}
  - party = {'Tom', 'Bob', 'Alice'}
- 圆括号 ()：可以表示元组 tuple，生成器 generator，也可以只是分隔符

```
>>> g = (i for i in range(0))
>>> g
<generator object <genexpr> at 0x10c791f68>
>>> g1 = (1, 2, 4)
>>> g1
(1, 2, 4)
>>> help(g1)

>>> type(g1)
<class 'tuple'>
```

```
>>> type((3))
<class 'int'>
>>> type((3,))
<class 'tuple'>
>>> (3)**4
12
>>> (3,)*4
(3, 3, 3, 3)
>>> 
```

# 例外处理 Exception Handling

- 代码运行可能会意外各种错误：
  - 语法错误： Syntax Error
  - 除以 0 错误： ZeroDivisionError
  - 列表下标越界： IndexError
  - 类型错误： TypeError...
- 错误会引起程序中止退出，如果希望掌控意外，就需要在可能出错误的地方设置陷阱捕捉错误

```
try:  
    <可能出错的代码>  
except:  
    <处理错误的代码>  
else:  
    <没有出错执行的代码>  
finally:  
    <无论出错否，都执行的代码>
```

- try, except (可多个) 必须， else, finally 可选

# 函数 function

- 函数用来对具有明确功能的代码段命名，以便复用 (reuse)
- 定义函数：def 语句；

```
def <函数名> (<参数表>):  
    <缩进的代码段>  
    return <函数返回值>
```

- 无 return = “return”= “return None”
- 调用函数：<函数名> (<参数>)
  - 注意括号！
  - 丢弃返回值：<函数名>(<参数表>)
  - 返回值赋值：v=<函数名>(<参数表>)

```
1 def sum_list(alist): # 定义一个带参数的函数  
2     sum_temp = 0  
3     for i in alist:  
4         sum_temp += i  
5     return sum_temp # 函数返回值  
6  
7  
8 print(sum_list) # 查看函数对象sum_list  
9  
10 my_list = [23, 45, 67, 89, 100]  
11 # 调用函数，将返回值赋值给my_sum  
12 my_sum = sum_list(my_list)  
13 print("sum of my list:%d" % (my_sum))  
<function sum_list at 0x10067a620>  
sum of my list:324  
  
sum = int(input())  
if sum % 2 == 1:  
    print("even number required.")  
    exit # 并不会按预想退出  
#do sth interesting with sum  
half = sum // 2
```

# 定义函数的参数：固定参数／可变参数

定义函数时，参数数量可以有两种情况：

- 一种是在参数表中写明参数名 `key` 的参数，固定了顺序和数量  
调用函数时可以根据位置或 `key`，来提供参数。
  - `def func(key1, key2, key3…):`
  - `def func(key1, key2=value2…):`
- 另一种在参数表中不明确参数名，不确定参数数量
  - 数组参数，不带 `key` 的多个参数，

```
def func(*args):
```

- 字典参数，`key=val` 形式的多个参数，

```
def func(**kwargs):
```

```
|>>> print(1, 2, 'xzm', [1,2,3])
1 2 xzm [1, 2, 3]
```

# 定义函数的参数：固定参数

- 参数的传入顺序和缺省值的使用

```
16 def func_test(key1, key2, key3=23):
17     print("k1=%s,k2=%s,k3=%s" % (key1, key2, key3))
18
19
20     print("====func_test")
21 # 没有传入key3, 用了缺省值
22 func_test('v1', 'v2')
23 # 传入了key3
24 func_test('ab', 'cd', 768)
25 # 使用参数名称就可以不管顺序
26 func_test(key2='KK', key1='K')
```

```
====func_test
k1=v1,k2=v2,k3=23
k1=ab,k2=cd,k3=768
k1=K,k2=KK,k3=23
```

## 定义函数的参数：可变无名参数

- args 在函数里面是一个元组 tuple

```
29     # 可以随意传入0个或多个无名参数
30     def func_test2(*args):
31         for arg, i in zip(args, range(len(args))):
32             print("arg%d=%s" % (i, arg))
33
34
35     print("====func_test2")
36     func_test2(12, 34, 'abcd', True)
```

```
====func_test2
arg0=12
arg1=34
arg2=abcd
arg3=True
```

## 定义函数的参数：可变带名参数

- kwargs 在函数里面是一个字典 dict

```
39     # 可以随意传入0个或多个带名参数
40     def func_test3(**kwargs):
41         for key, val in kwargs.items():
42             print("%s=%s" % (key, val))
43
44
45     print("====func_test3")
46     func_test3(myname="Tom", sep="comma", age=23)
```

```
====func_test3
sep=comma
age=23
myname=Tom
```

# 调用函数的参数：位置参数／关键字参数

- 调用函数的时候，可以传进两种参数；
- 一种是没有名字的位置参数
  - `func(arg1, arg2, arg3…)`
  - 会按照前后顺序对应到函数参数传入
- 一种是带 `key` 的关键字参数
  - `func(key1[arg1], key2[arg2]…)`
  - 由于指定了 `key`，可以不按照顺序对应
- 如果混用，所有位置参数必须在前，关键字参数必须在后

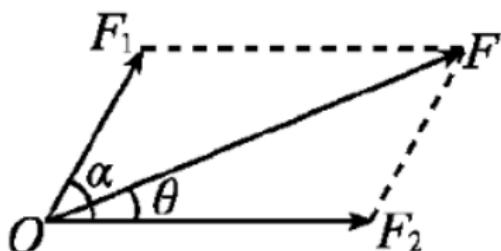
# 面向对象：类的定义与调用

- 类用来实现抽象数据类型 ADT，封装实体的属性和行为
- 定义类：class 语句；
  - class <类名>:  
    def \_\_init\_\_(self, <参数表>): # 前后连着两个'.'  
    def <方法名>(self, <参数表>):
- 调用类：<类名>(<参数>)
  - 必须有括号，即使里面为空！
  - obj = <类名>(<参数表>)
  - 返回一个对象实例，
  - 类方法中的 self 指这个对象实例！

# class Force

```
1 class Force: # 力
2     def __init__(self, x, y): # x,y方向分量
3         self.fx, self.fy = x, y
4
5     def show(self): # 打印出力的值
6         print("Force<%s,%s>" % (self.fx, self.fy))
7
8     def add(self, force2): # 与另一个力合成
9         x = self.fx + force2.fx
10        y = self.fy + force2.fy
11        return Force(x, y)
12
13
14     # 生成一个力对象
15     f1 = Force(0, 1)
16     f1.show()
17
18     # 生成另一个力对象
19     f2 = Force(3, 4)
20     # 合成为新的力
21     f3 = f1.add(f2)
22     f3.show()
```

Force<0,1>  
Force<3,5>



# 类定义中的特殊方法

- 在类定义中实现一些特殊方法，可以方便地使用 python 一些内置操作

- 所有特殊方法以两个下划线开始结束
- `__str__(self)`: 自动转换为字符串
- `__add__(self, other)`: 支持 `+` 操作
- `__mul__(self, other)`: 支持 `*` 操作
- `__eq__(self, other)`: 支持 `==` 操作
- `__hash__(self)`: 支持 `hash(obj)` 操作

- 更多特殊方法：

[rszalski.github.io/magicmethods/](https://rszalski.github.io/magicmethods/)

- `__eq__` 和 `__hash__` 要一起出现，只有前者会变成 unhashable
- 它们都没有定义的话，缺省的相等和哈希针对 `id(obj)` 进行
- 自定义了 `__eq__` 和 `__hash__` 的类，把对象放入字典再修改它的值，可能造成“元素不是元素”的迷局
  - [http://github.com/dsaClass/dsa2020.git, course/eq\\_hash.py](http://github.com/dsaClass/dsa2020.git, course/eq_hash.py)

```
13     __add__ = add
14
15     def __str__(self):
16         return "F<%s,%s>" % (self.fx, self.fy)
17
18     def __mul__(self, n):
19         x, y = self.fx * n, self.fy * n
20         return Force(x, y)
21
22     def __eq__(self, force2):
23         return (self.fx == force2.fx) and \
24                (self.fy == force2.fy)
25
26
27 # 操作符使用
28 f3 = f1 + f2
29 print("Fadd=%s" % (f3,))
30 f3 = f1 * 4.5
31 print("Fmul=%s" % (f3,))
32 print("%s==%s? -> %s" % (f1, f2, f1 == f2))
```

Fadd=F<3,5>

Fmul=F<0.0,4.5>

F<0,1>==F<3,4>? -> False

# 代码复用：类的继承

- 如果两个类具有“一般-特殊”的逻辑关系，那么特殊类就可以作为一般类的“子类”来定义，从“父类”继承属性和方法
  - “子类” is a kind of “父类”； NOT “a part of” nor “an attribute of”
  - class < 子类名 >(< 父类名 >):  
    def < 新定义方法 >(self, …):  
    def < 重定义方法 >(self, …): # 可以覆盖父类中的同名方法
- 子类对象可以调用父类方法，除非这个方法在子类中重新定义了
- Python 与 C++ 类似，可以多继承

# GasCar and ElecCar are all Cars

```
45 class Car:
46     def __init__(self, name):
47         self.name = name
48         self.remain_mile = 0
49
50     def fill_fuel(self, miles): # 加燃料里程
51         self.remain_mile = miles
52
53     def run(self, miles): # 跑miles英里
54         print(self.name, end=':')
55         if self.remain_mile >= miles:
56             self.remain_mile -= miles
57             print("run %d miles!" % (miles,))
58         else:
59             print("fuel out!")
60
61
62     class GasCar(Car):
63         def fill_fuel(self, gas): # 加汽油gas升
64             self.remain_mile = gas * 6.0 # 每升跑6英里
65
66
67     class ElecCar(Car):
68         def fill_fuel(self, power): # 充电power度
69             self.remain_mile = power * 3.0 # 每度电3英里
```

```
71 gcar=GasCar("BMW")
72 gcar.fill_fuel(50.0)
73 gcar.run(200.0)
```

```
75 ecar=ElecCar("Tesla")
76 ecar.fill_fuel(60.0)
77 ecar.run(200.0)
```

BMW: run 200 miles!  
Tesla: fuel out!

# 自己写模块 module

- 我们不能把所有的代码放在一起（一个文件里）
- 把一组相关功能集中在一个'.py'文件里，它就叫一个模块
- 需要相关功能，`import` 对应的模块。
- 一个模块是这样组织的，`__name__` 是模块的属性：
- 使用判断 `if __name__ == "__main__":` 是一种常见的实践
  - 在下面放一些本模块的测试代码。模块被 `python` 直接运行时，测试代码会运行；被 `import` 时，测试代码不运行

```
2. Default (vim)
Default (vim) 301 Default (bash) 302
1 #foo.py
2
3 print(""" 我总是会被执行。模块里一些功能在使用前有一些初始化的工作,
4 可以把他们放到这里，会自动完成。""")
5
6 def hello():
7     print("我实现了一个say hello的功能")
8     print("我所在的模块名字叫做: {}".format(__name__))
9
10 if __name__ == '__main__':
11     print(""" 只有模块被Python直接调用（非import）时，我才会被执行）。
12 所以一些本模块的测试代码，常常放到这里。""")
13     hello()
foo.py
1 #bar.py
2 import foo
3
4 if __name__ == '__main__':
5     foo.hello()
bar.py
```

```
Mini2-1:tmp zhengmaoxie$ python3 foo.py
我总是会被执行。模块里一些功能在使用前有一些初始化的工作,
可以把他放到这里，会自动完成。
只有模块被Python直接调用（非import）时，我才会被执行）。
所以一些本模块的测试代码，常常放到这里。
我实现了一个say hello的功能
我所在的模块名字叫做: __main__
Mini2-1:tmp zhengmaoxie$ python3 bar.py
我总是会被执行。模块里一些功能在使用前有一些初始化的工作,
可以把他放到这里，会自动完成。
我实现了一个say hello的功能
我所在的模块名字叫做: foo
Mini2-1:tmp zhengmaoxie$
```