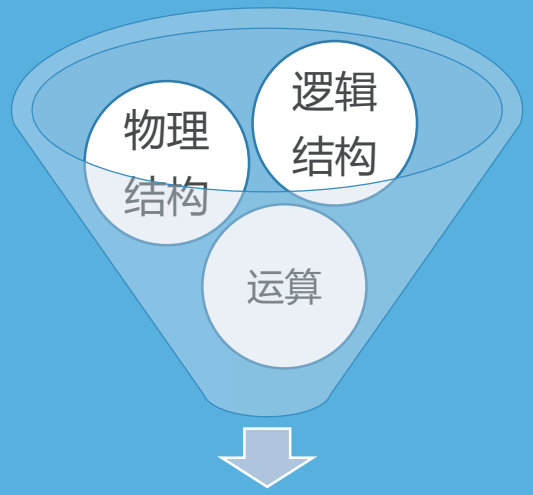


数据结构与算法（Python）-01 Python编程入门

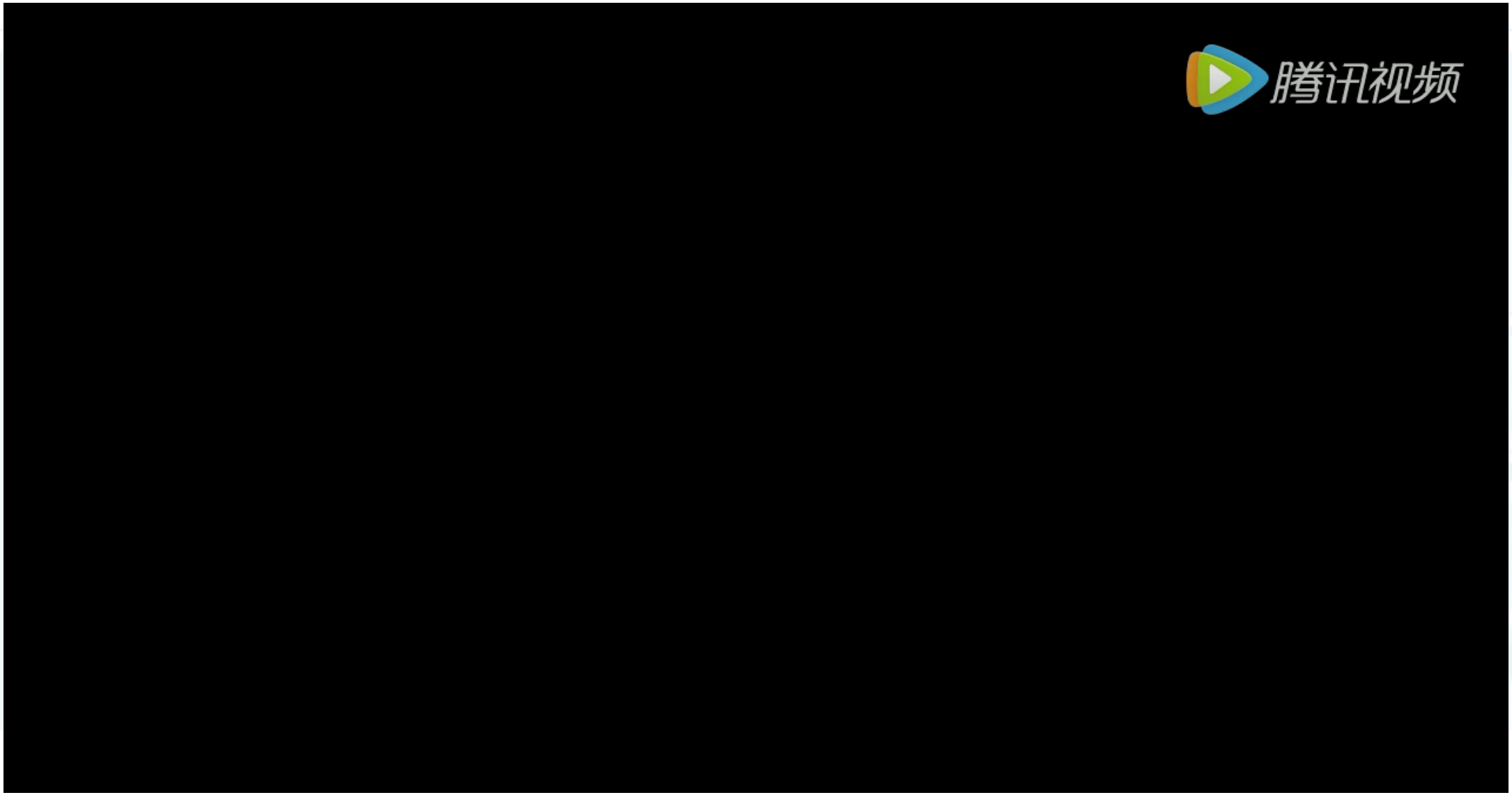
刘云淮 Yunhuai.liu@pku.edu.cn

http://www.yunhuai.net/DSA2023/DSA2023.html

北京大学计算机学院

目录

What Most School Don't Teach



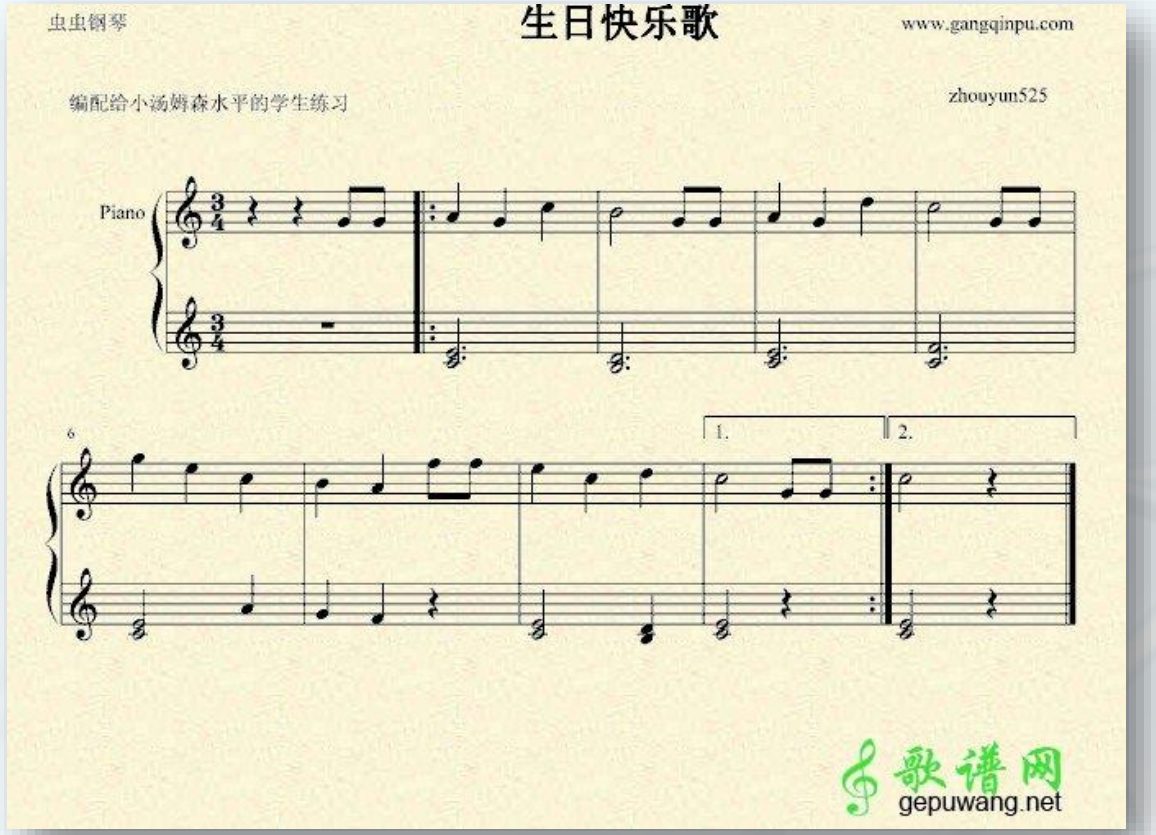
程序是什么？ #程序是菜谱

- 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)
```

程序是什么？ #程序是电影脚本 #程序是乐谱

镜号	景别	拍摄角度	时长	声音	内容	备注
1	全景	平拍、右斜侧	5 秒	小孩子哭啼声	小孩躺在地上哇哇大哭	三四岁的小女孩
2	全景	仰拍、右斜侧	6 秒	孩子依旧哭啼，声音越来越大	夫妻两吵架，丈夫气急败坏摔桌子（茶几）上的东西	以孩子的角度拍摄
3	特写	俯拍、左斜侧	2 秒	孩子依旧哭啼	地上摔掉的东西	以孩子的角度拍摄
4	特写	平拍、正拍	2 秒	孩子停止哭啼	孩子惊吓的表情，停止哭啼	房屋一片安静
5	全－中－近－全－远	平拍、左斜侧－正侧－右斜侧－背面	6 秒	只有丈夫离开的脚步声	丈夫甩掉东西后生气的离开家，小孩子的视线跟随着丈夫的离开而移动	机位可设在小孩背后
6	近景	平拍，（左或右）斜侧	3 秒		妻子坐在沙发上轻轻的哭泣，默默的流泪	
7	全景	稍俯拍、右斜侧平－仰	10 秒		小孩子从坐的地方慢慢向桌子（茶几）爬去，很不熟练的在一盒清风卫生纸里面抽了一张，站起来，跟跟跄跄的走到妈妈身边，把	爬 5 秒、抽纸 3 秒、递给妈妈 3 秒 俯拍小孩爬、平拍



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

- 有n个数

设置一个sum用来暂存部分和

sum = 第1个数

反复做下列工作，直到所有数完成：

　　取下一个数，累加到sum

输出sum

各个操作系统里的Python：Windows

- **各个版本的Windows都需要额外安装Python（32 / amd64）**

  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

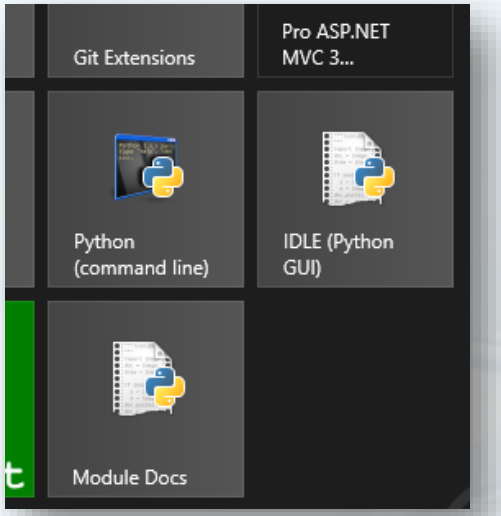
  输入quit()来退出Python命令行

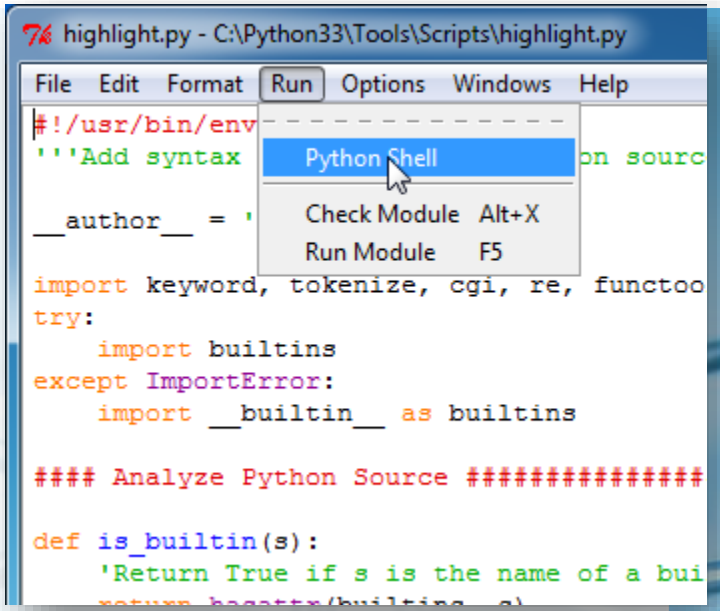
- **IDLE是Python的图形界面**

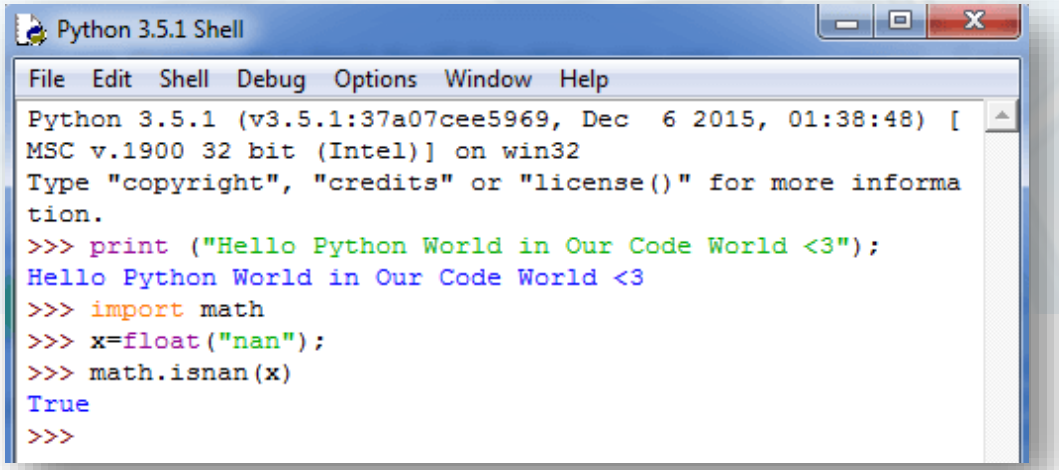
  拥有两种窗口：

  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）

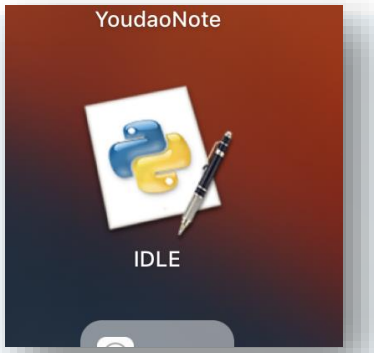


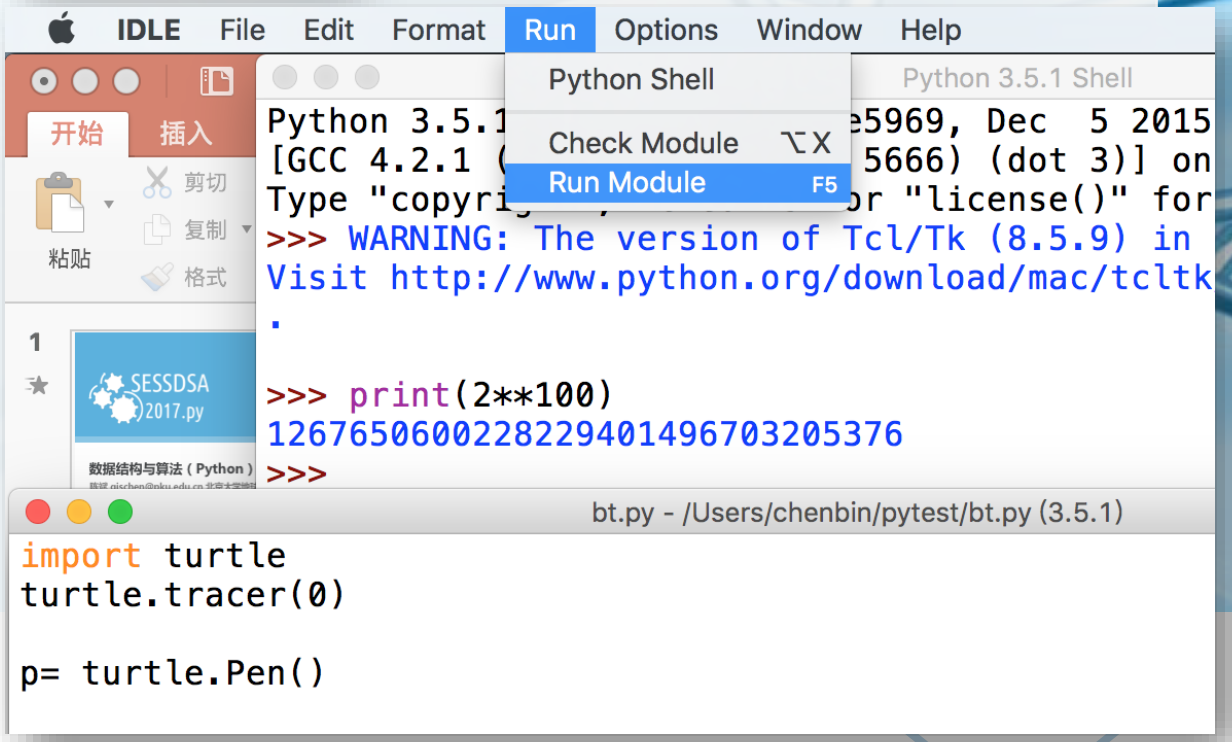




各个操作系统里的Python：macOS

- 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





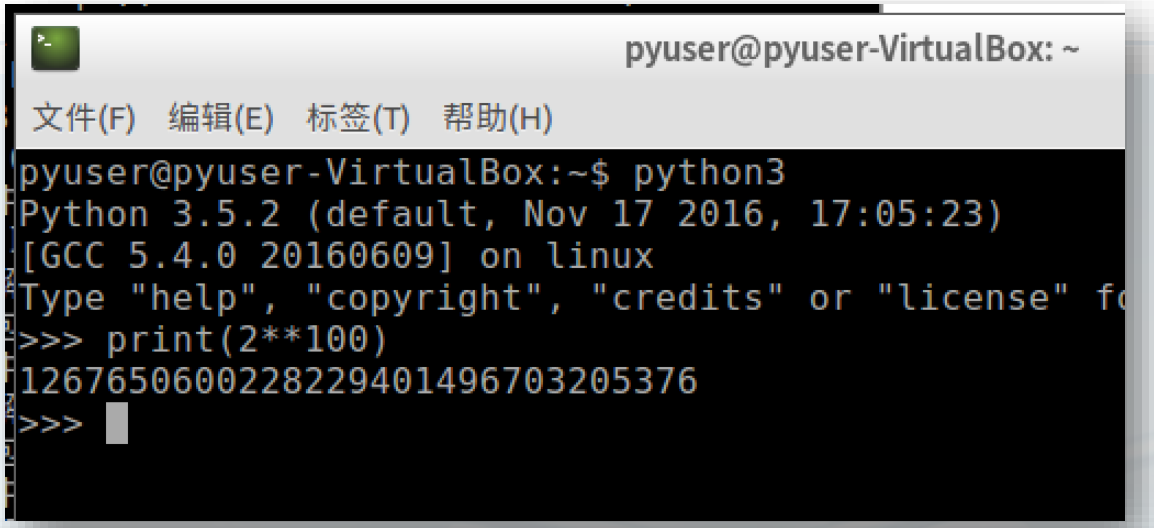
各个操作系统里的Python：各种Linux

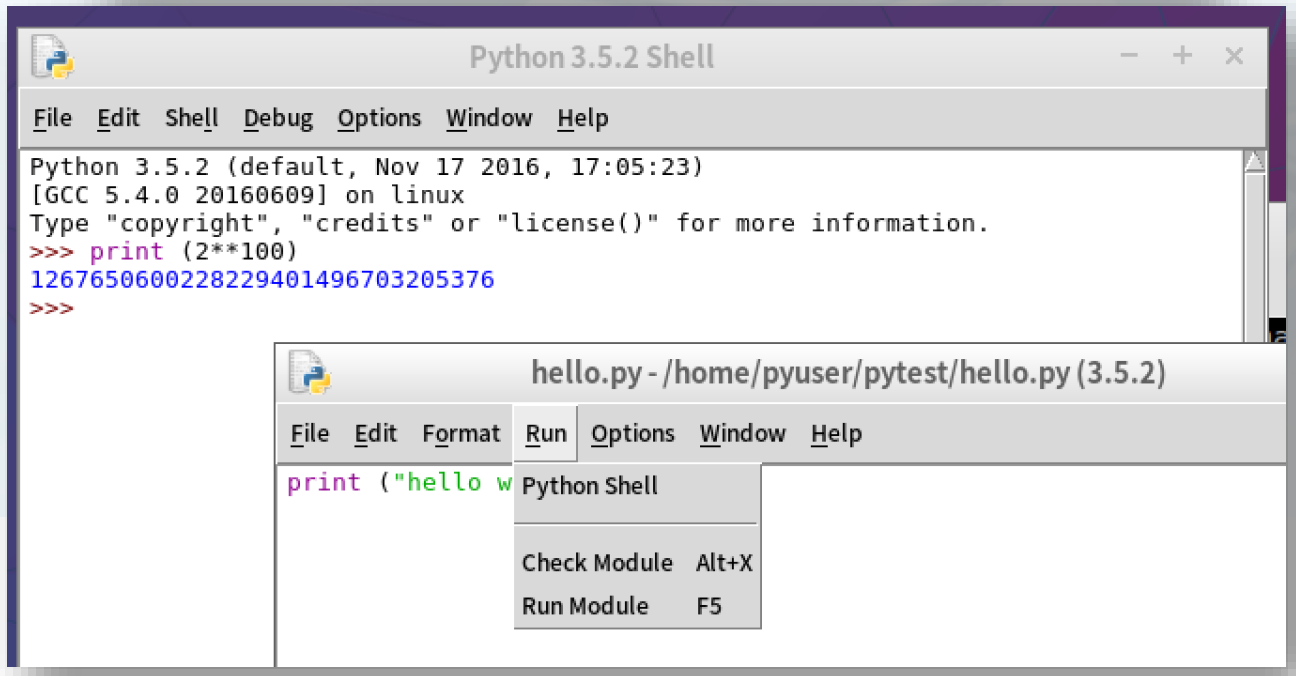
- 各种Linux都内置了Python3
- 命令行界面也是从终端输入python3来启动
- 也具备IDLE的图形界面

  （需要一个简单命令自动安装idle3）

  ```
  sudo apt install idle3
  ```

- 操作也是一样





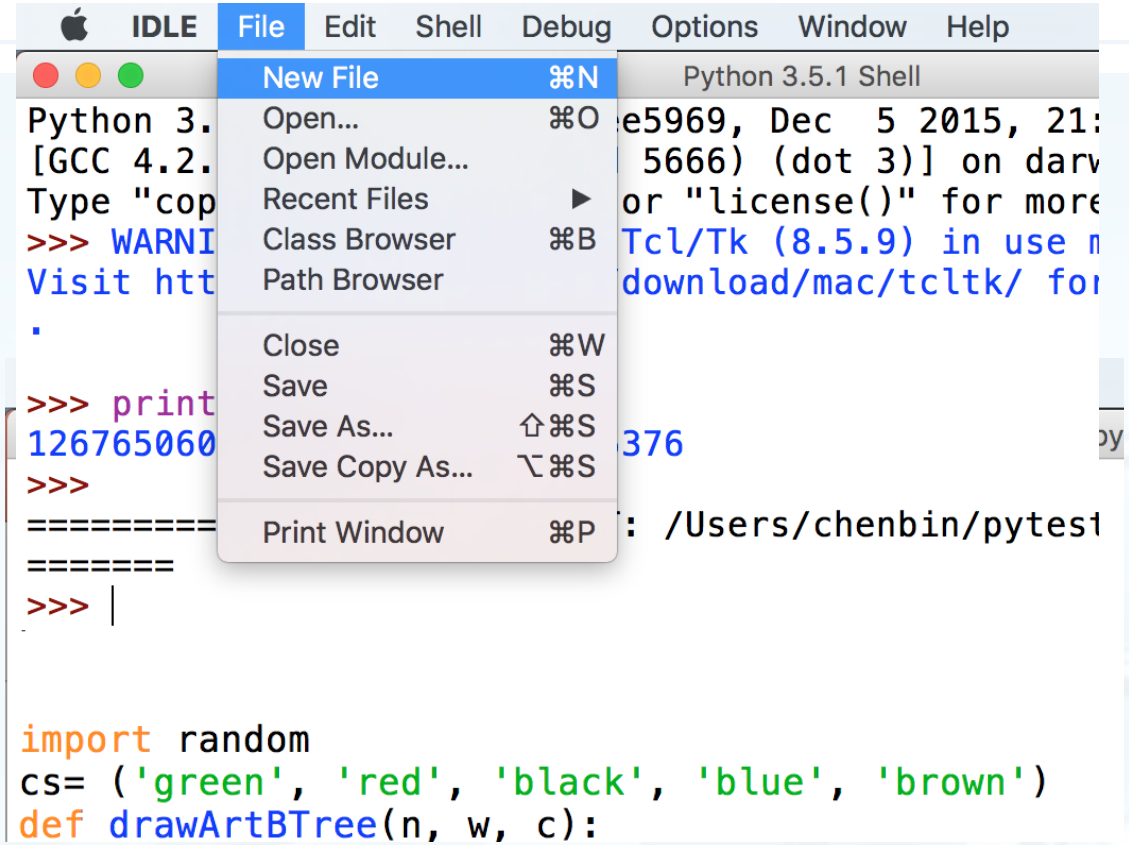
在IDLE中编辑和运行Python程序

- 启动IDLE
- File->New File
- 在文件编辑窗口中输入代码
- 保存代码文件

  注意保存在“D:\homework”或者“文稿/homework”这样的目录下

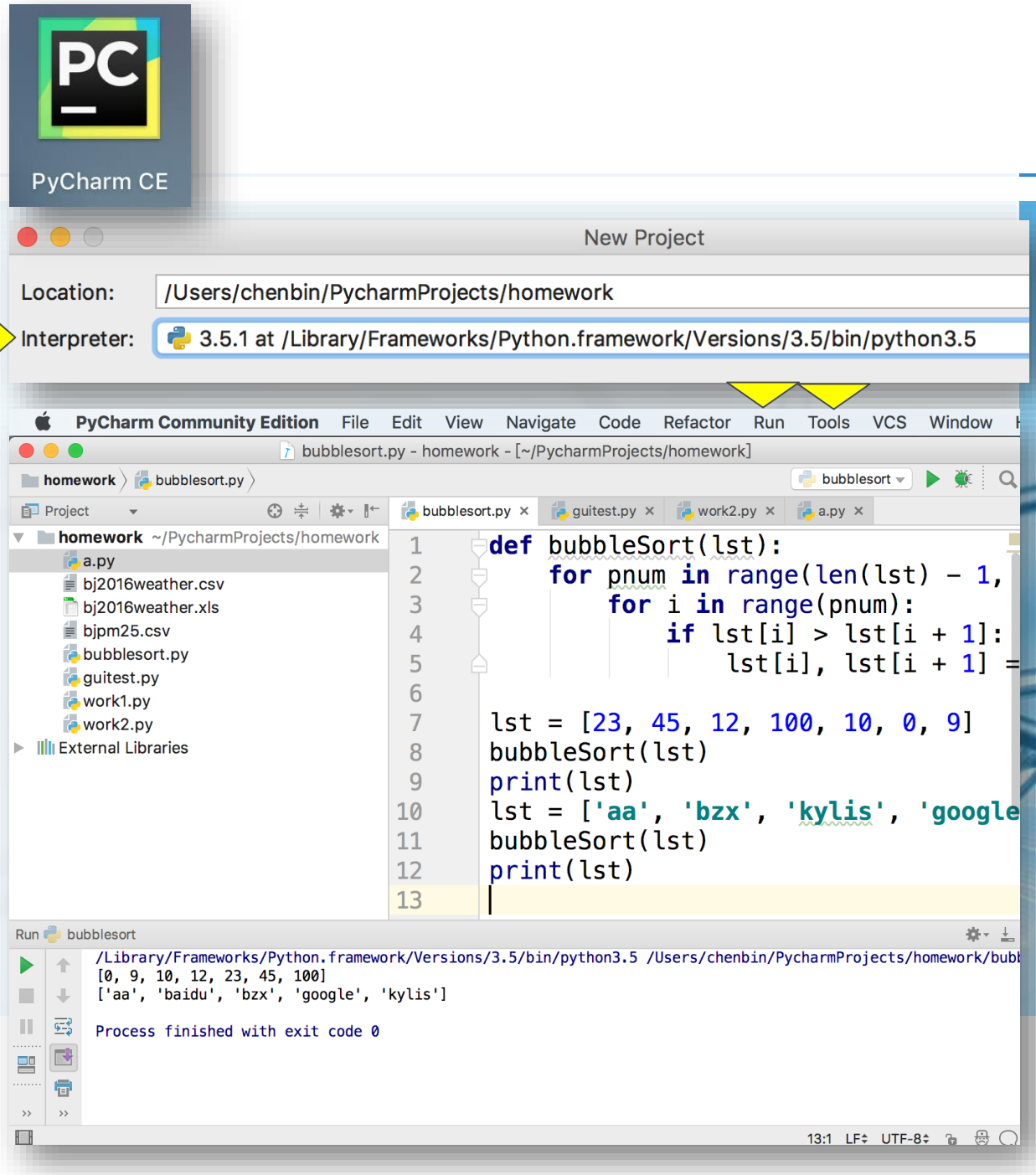
  目录名和文件名不要用中文

- Run->Run Module运行程序
- 在Python Shell中查看运行结果



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

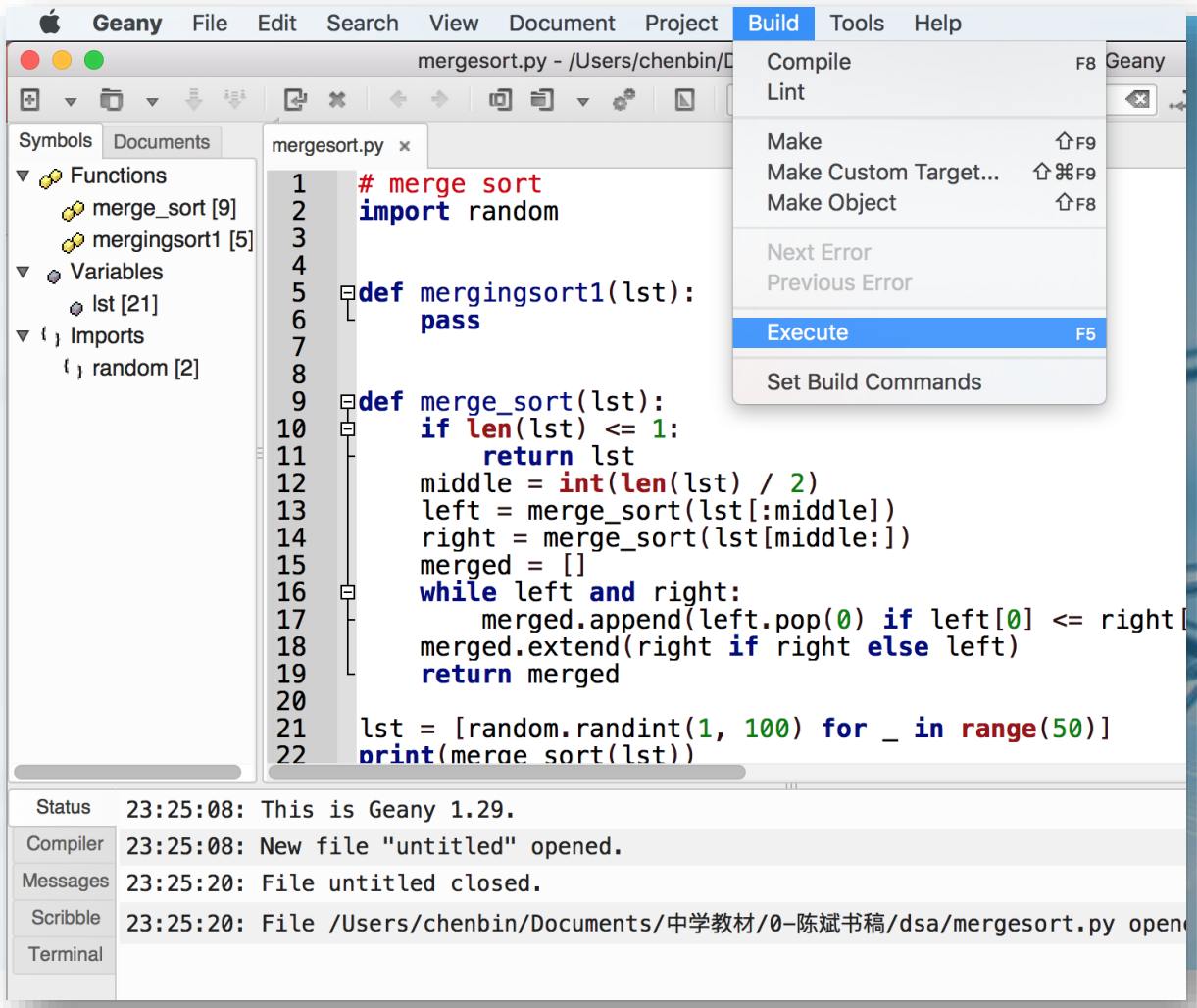


集成开发环境：Geany

- 本质上是一个带执行程序功能的代码编辑器
- 可以编辑执行各种语言的程序

  Python / C / C++ / Java / HTML等等
- 保存程序后，Build->Execute

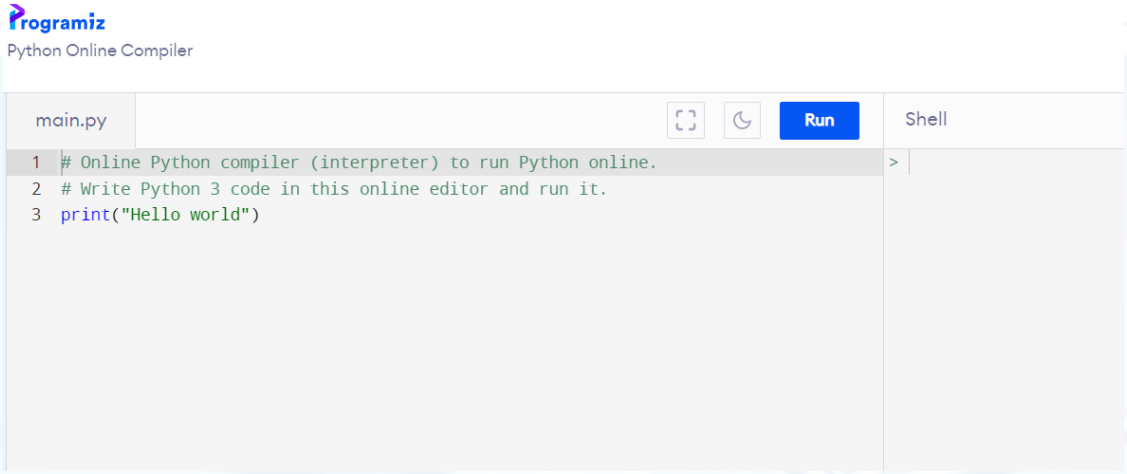
  （可以在Set Build Commands中设置Python解释器的版本）



一些在线的Python解释器

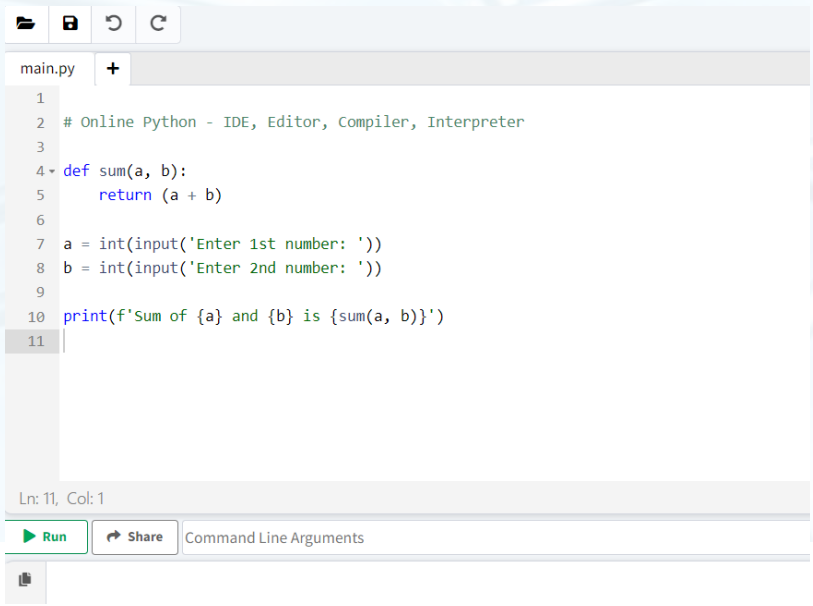
- **Python Online Complier**

  https://www.programiz.com/python-programming/online-compiler/



- **Online Python**

  https://www.online-python.com/

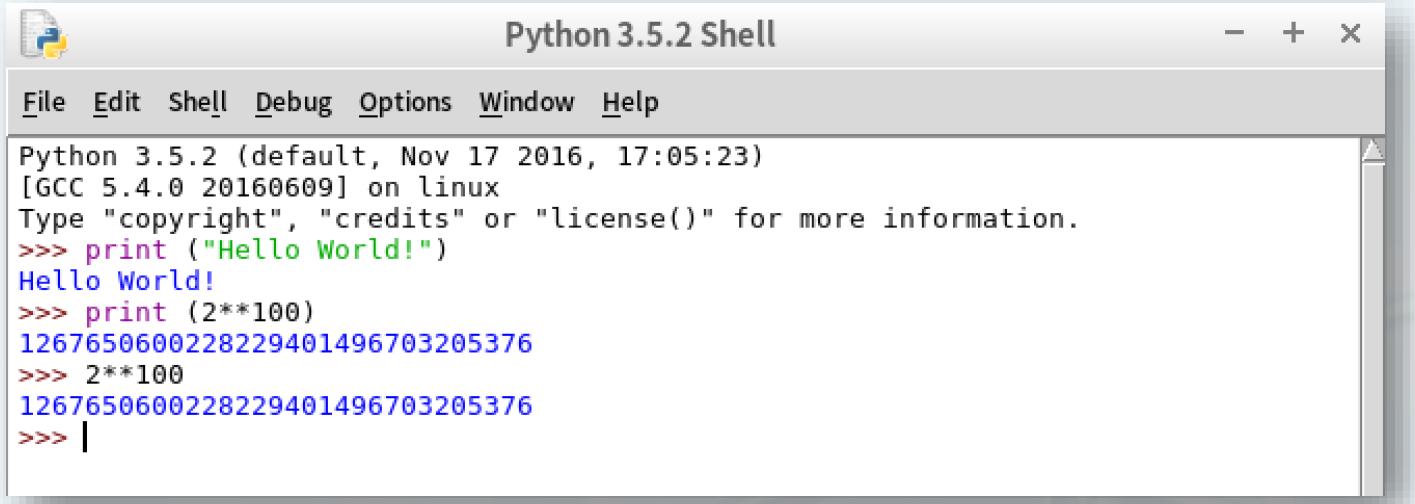


第一个Python语句：超级计算器

- 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

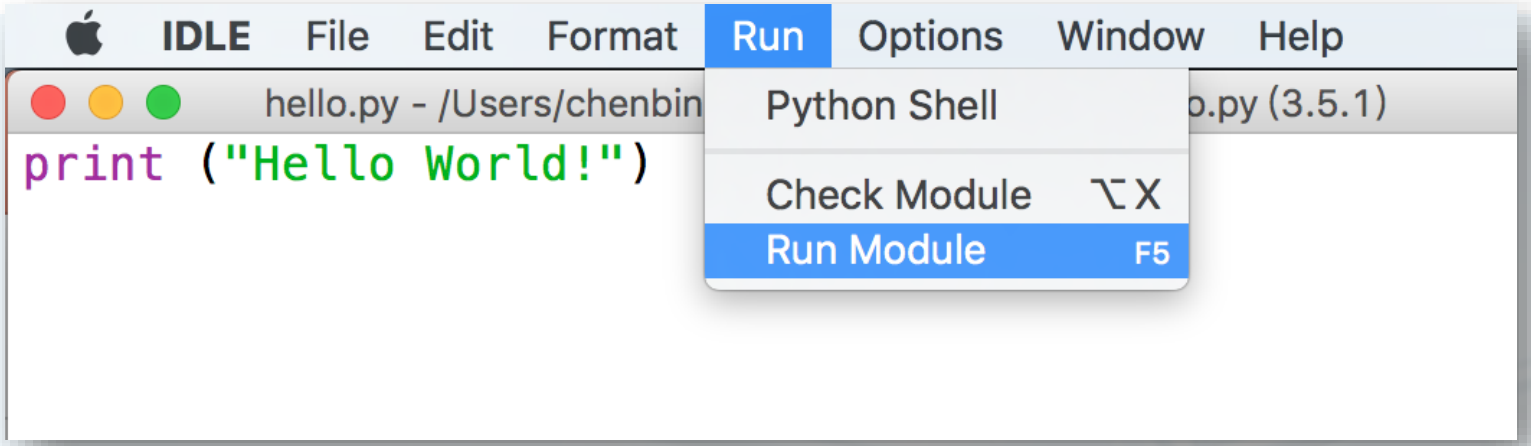


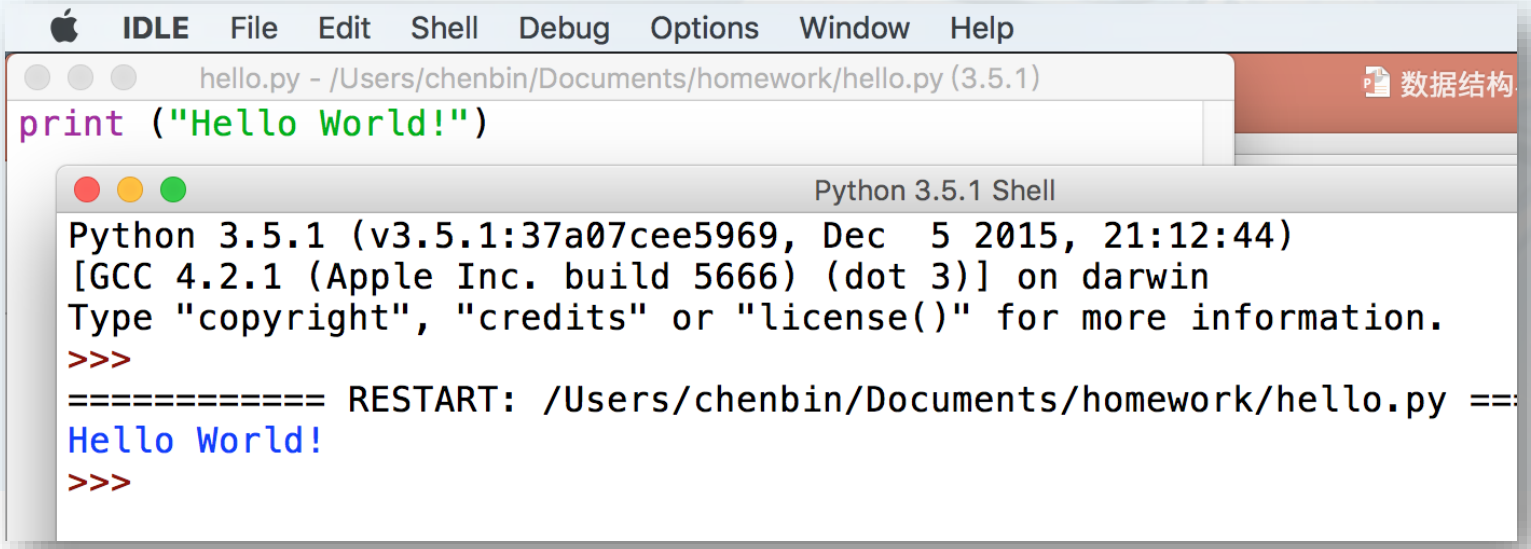
第一个Python程序：Hello World!

- 打开IDLE
- File->New File
- 输入代码

```
print ("Hello World!")
```

- File->Save
- Run->Run Module





Python程序：简洁、优雅

Python版归并排序

```python
def merge_sort(lst):
    if len(lst) <= 1:
        return lst
    middle = int(len(lst) / 2)
    left = merge_sort(lst[:middle])
    right = merge_sort(lst[middle:])
    merged = []
    while left and right:
        merged.append(left.pop(0) if left[0]
    merged.extend(right if right else left)
    return merged
```

C语言版归并排序

```c
void merge_sort_recursive(int arr[], int reg[], int start, int end) {
    if (start >= end)
        return;
    int len = end - start, mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    merge_sort_recursive(arr, reg, start1, end1);
    merge_sort_recursive(arr, reg, start2, end2);
    int k = start;
    while (start1 <= end1 && start2 <= end2)
        reg[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[sta
    while (start1 <= end1)
        reg[k++] = arr[start1++];
    while (start2 <= end2)
        reg[k++] = arr[start2++];
    for (k = start; k <= end; k++)
        arr[k] = reg[k];
}
void merge_sort(int arr[], const int len) {
    int reg[len];
    merge_sort_recursive(arr, reg, 0, len - 1);
}
```

代码缩进：视觉效果和功能的统一


```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedPa
3                                   uint8_t *signature, UInt16 signatureLen)
4  {
5      OSStatus        err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14         goto fail;
15     err = sslRawVerify(ctx,
16                        ctx->peerPubKey,
17                        dataToSign,              /* plaintext */
18                        dataToSignLen,           /* plaintext length */
19                        signature,
20                        signatureLen);
21     if(err) {
22         sslErrorLog("SSLDecodeSignedServerK
23                     "returned %d\n", (int)e
24         goto fail;
25     }
26
27 fail:
28     SSLFreeBuffer(&signedHashes);
29     SSLFreeBuffer(&hashCtx);
30     return err;
31 }
```

```
7
8        if ((err = SSLHashSHA1
9            goto fail;
10       if ((err = SSLHashSHA1
11           goto fail;
12           goto fail;
13       if ((err = SSLHashSHA1
14           goto fail;
15       err = sslRawVerify(ctx
```

程序是写给人读的，只是偶尔让计算机执行一下

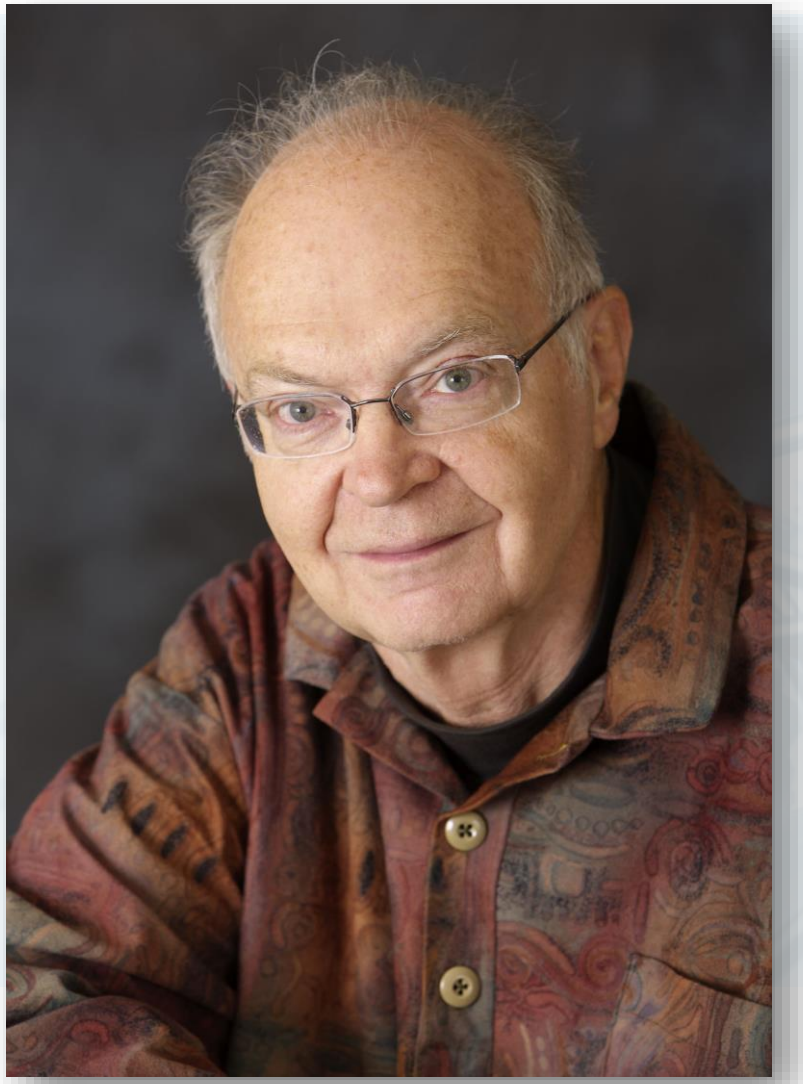
- Python的强制缩进规范完成了关键部分
- 我们还需要良好的编程规范

  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



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T_EX

  有趣的版本号3.14159265，最后是π

  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

Python 可变类型和不可变类型

- **不可变类型**
  - 数字
  - 字符串
  - 元组
- **每次赋值操作都是生成一个新对象**
- **元组不支持赋值操作**

- **可变类型**
  - 列表
  - 集和
  - 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

- 最大的特点是不限制大小
- 浮点数受到17位有效数字的限制
- 常见的运算包括加、减、乘、除、整除、求余、幂指数等
- 浮点数的操作也差不多
- 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)
>>>
```

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)
```

```
>>> 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
>>>
```

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与非0等

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


OK
>>>
```

Python数据类型：字符串

- 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

```
>>> '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

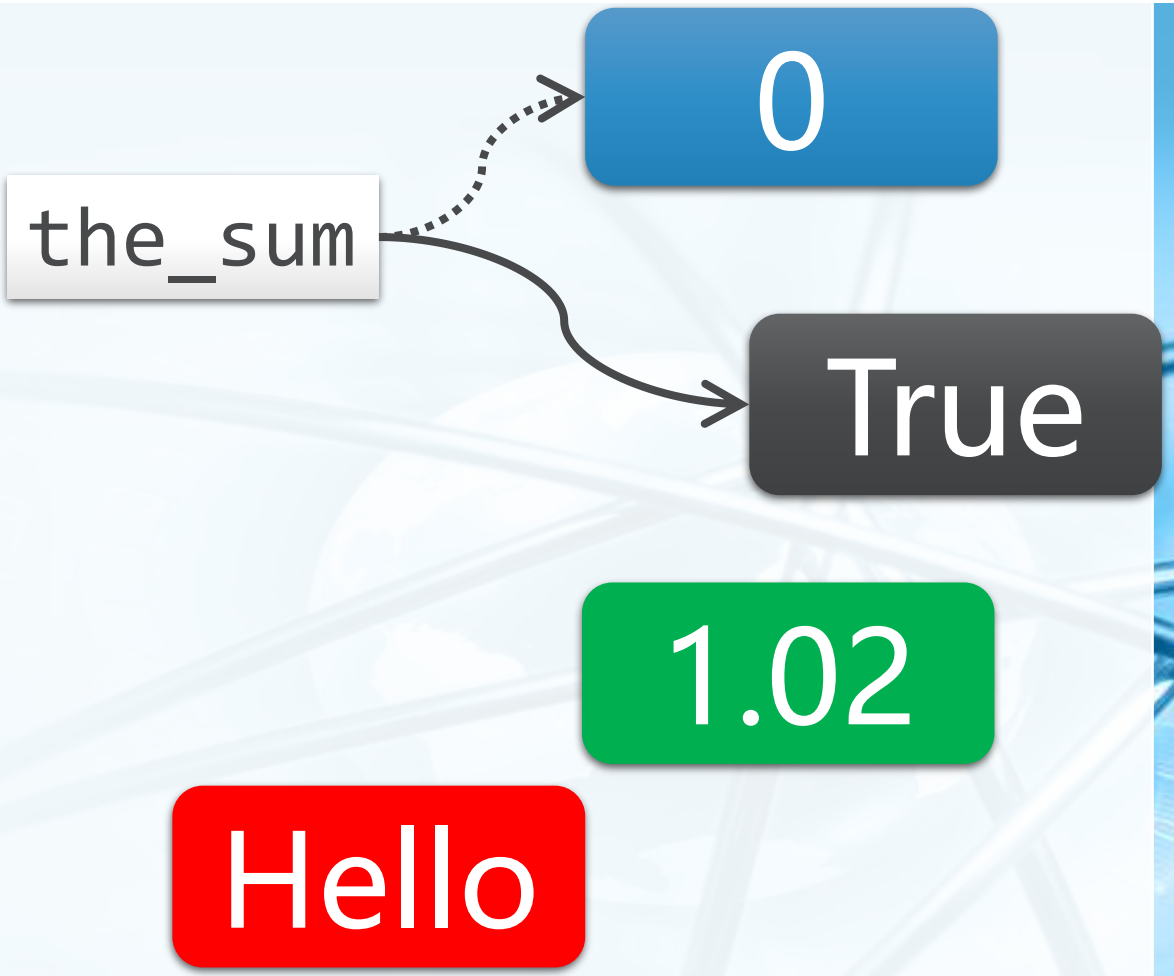
  ljust/center/rjust：排版左中右对齐

  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：引用数据对象

- 赋值语句`the_sum = 0`，实际上是创建了名为the_sum的变量，然后指向数据对象“0”
- 所以变量可以随时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，以整数作为索引
- 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**序列中的元素
  可变类型

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

容器类型：列表和元组

- 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

```
>>> []
[]
>>> list()
[]
>>> alist = [1, True, 
>>> alist[0]
1
>>> alist + ["Hello"]
[1, True, 0.234, 'Hell
>>> alist * 2
[1, True, 0.234, 1, Tr
>>> 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)
```

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

```
>>> 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
```

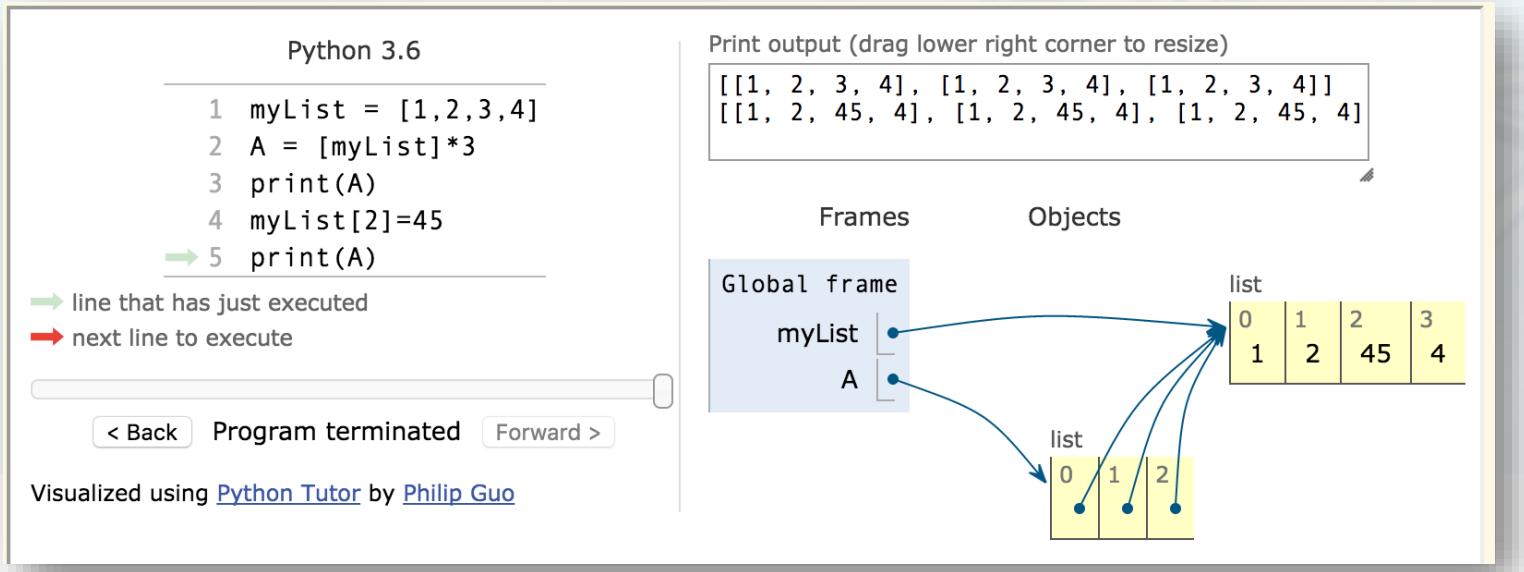
列表list的其它方法

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
- 多个变量通过赋值引用同一个可变类型对象时
- 通过其中任何一个变量改变了可变类型对象
- 其它变量也看到了改变

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



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
  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**

```
>>> 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()
>>> aset = set('abc')
>>> aset
{'c', 'a', 'b'}
>>> 'a' in aset
True
>>> aset | set('bcd')
{'c', 'd', 'a', 'b'}
>>> aset & set(['b', 'c', 'd'])
{'c', 'b'}
>>> aset - set(('b', 'c', 'd'))
{'a'}
>>> aset ^ set('bcd')
{'a', 'd'}
>>> aset <= set('abcd')
True
>>> aset > 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
- 如果经常需要判断元素是否在一组数据中，这些数据的次序不重要的话，推荐使用集合，可以获得比列表更好的性能

```
>>> 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可以是任何不可变类型（数值 / 字符串 / 元组）

```
>>> 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', 'c
hemistry'], '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):'bus stop'}
  ```

```
>>> 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'
```

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

  可以带参数end='\n'，缺省为换行，表示打印后以这个字符串结尾

  带参数sep=' '，缺省是空格，表示变量之间用什么字符串隔开

- **格式化字符串**

  '%d %s' % (v1, v2)

```
>>> 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):K:%s' % (12, 'Hello')
'(  12):K:Hello'
>>> '(%04d):K:%10s' % (12, 'Hello')
'(0012):K:     Hello'
```

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

```
>>> 12 * 34.5 + 23.4
437.4
>>> ('abc' + '123') * 3
'abc123abc123abc123'
>>>
>>> import math
>>> math.sqrt(12)
3.464101615137754 4
>>> 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']
```

- **变量交换**

```
a, b = b, a
```

- **自操作**

```
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>: #可以多个elif
    <语句块>
else: #仅1个
    <语句块>
```

- **各种类型中某些值会自动被转换为False，其它值则是True：**

```
None, 0, 0.0, '',
[], (), {}, set()
```

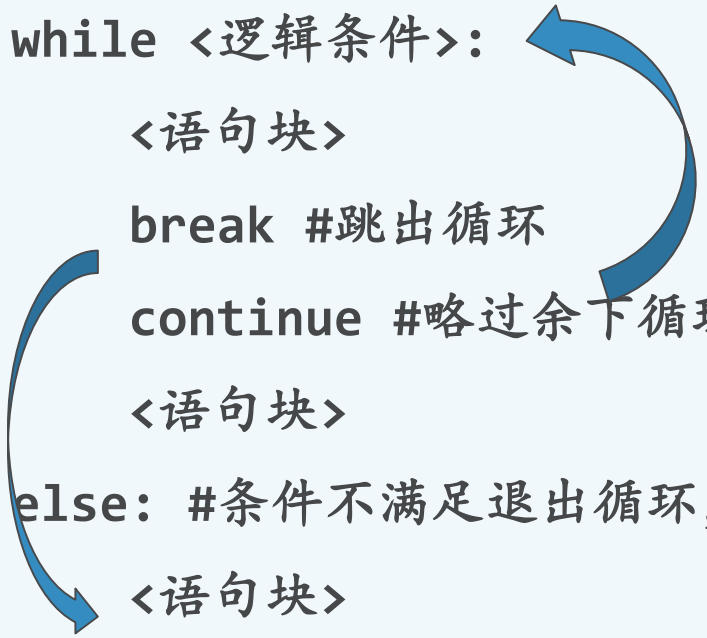
```
>>> a = 12
>>> if a > 10:
        print ("Great!")
elif a > 6:
        print ("Middle!")
else:
        print ("Low!")

Great!
```

控制流语句：while循环

- 条件循环while

```
while <逻辑条件>:
    <语句块>
    break #跳出循环
    continue #略过余下循环语句
    <语句块>
else: #条件不满足退出循环，则执行
    <语句块>
```



- else中可以判断循环是否遭遇了break

```
>>> n = 5
>>> while n > 0:
        n = n - 1
        if n < 2:
            break
        print (n)

4
3
2
```

```
>>> n = 5
>>> while n > 0:
        n = n - 1
        if n < 2:
            continue
        print (n)
else:
        print ('END!')

4
3
2
END!
```

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 #略过余下循环语句
else: #迭代完毕，则执行
    <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

- 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>}

```
>>> [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


```
>>> [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 0x1078f5620>
>>> for n in agen:
        print (n)


0
1
4
9
16
25
36
49
64
81
```

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: # 为缩进的代码设置陷阱

  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

```
1   try:
2       print('try...')
3       r = 10 / 'xyz'
4       print('result:', r)
5   except TypeError as e:
6       print('TypeError:', e)
7   except ZeroDivisionError as e:
8       print('ZeroDivisionError:', e)
9   else:
10      print('no error!')
11  finally:
12      print('finally...')
13  print('END')
```

```
try...
TypeError: unsupported operand type(s) for /: 'int' and 'str'
finally...
END
```

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>
```

- 调用函数：<函数名>（<参数>）

注意括号！

无返回值：<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
```

定义函数的参数：固定参数 / 可变参数

- 定义函数时，参数可以有两种；
- 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

```
def func(key1, key2, key3…):
def func(key1, key2=value2…):
```

- 一种是定义时还不知道会有多少参数传入的可变参数

```
def func(*args): #不带key的多个参数
def func(**kwargs): #key=val形式的多个参数
```

```
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：固定参数 / 可变参数


```
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
```

```
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

```
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
```

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>):
    def <方法名>(self, <参数表>):
```

- 调用类：<类名>（<参数>）

注意括号！

```
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！

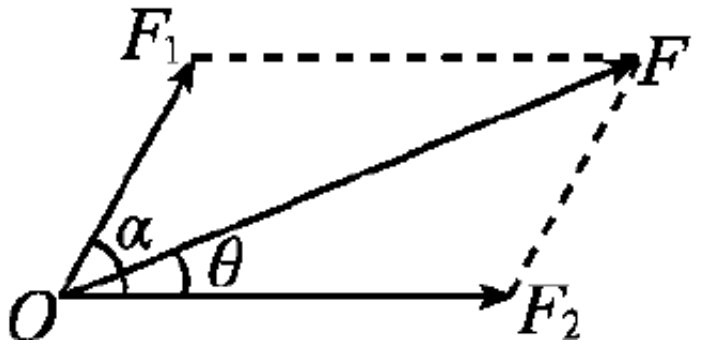
```python
class Force:  # 力
    def __init__(self, x, y):  # x,y方向分量
        self.fx, self.fy = x, y

    def show(self):  # 打印出力的值
        print("Force<%s,%s>" % (self.fx, self.fy))

    def add(self, force2):  # 与另一个力合成
        x = self.fx + force2.fx
        y = self.fy + force2.fy
        return Force(x, y)


# 生成一个力对象
f1 = Force(0, 1)
f1.show()

# 生成另一个力对象
f2 = Force(3, 4)
# 合成为新的力
f3 = f1.add(f2)
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)：使用+操作符

  __mul__(self, other)：使用*操作符

  __eq__(self, other)：使用==操作符

- **其它特殊方法参见Python网站**

  https://rszalski.github.io/magicmethods/

```
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)
```

```
37  # 操作符使用
38  f3 = f1 + f2
39  print("Fadd=%s" % (f3,))
40  f3 = f1 * 4.5
41  print("Fmul=%s" % (f3,))
42  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
```

类的继承机制：代码复用

- 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

```
class <子类名>(<父类名>):
    def <重定义方法>(self,…):
```

- 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
class Car:
    def __init__(self, name):
        self.name = name
        self.remain_mile = 0

    def fill_fuel(self, miles):  # 加燃料里程
        self.remain_mile = miles

    def run(self, miles):  # 跑miles英里
        print (self.name, end=': ')
        if self.remain_mile >= miles:
            self.remain_mile -= miles
            print("run %d miles!" % (miles,))
        else:
            print("fuel out!")


class GasCar(Car):
    def fill_fuel(self, gas):  # 加汽油gas升
        self.remain_mile = gas * 6.0  # 每升跑6英里


class ElecCar(Car):
    def fill_fuel(self, power):  # 充电power度
        self.remain_mile = power * 3.0  # 每度电3英里
```

```python
gcar=GasCar("BMW")
gcar.fill_fuel(50.0)
gcar.run(200.0)
```

```python
ecar=ElecCar("Tesla")
ecar.fill_fuel(60.0)
ecar.run(200.0)
```

```
BMW: run 200 miles!
Tesla: fuel out!
```