一、Numpy

为什么使用 numpy & pandas

运算速度快：numpy 和 pandas 都是采用 C 语言编写，pandas 又是基于 numpy，是 numpy 的升级版本。

消耗资源少：采用的是矩阵运算，会比 python 自带的字典或者列表快好多

提到 Numpy，它就是一个 Python 的救星. 能把简单好用的 Python 和高性能的 C 语言合并在一起. 当你调用 Numpy 功能的时候，他其实调用了很多 C 语言而不是纯 Python. 这就是为什么大家都爱用 Numpy 的原因.

1.1 Numpy 的几种属性

Numpy 是基于矩阵的运算

```
array  = np.array([
    [1,2,3],
    [3,5,6]
])
#ndim:维度   shape：行数和列数    size：元素个数
print("number of dim",array.ndim)
print("shape",array.shape)
print("size",array.size)
```

1.2 Numpy 定义数据类型和生成数据

```
# dtype  类型  默认：float64 int32 可以改为 float32
a = np.array([
  [2,23,4],
  [2,6,54]
], dtype=np.int)
# zeros 和 ones  别忘了 s
b = np.zeros((3,4))
c = np.ones((5,5),dtype=np.int)
d = np.empty((4,5))
e = np.arange(10,20,2)
f = np.arange(12).reshape((3,4))
g = np.linspace(1,10,6).reshape((2,3))
```

1.3 基础运算

1.3.1 一维矩阵运算

```
a = np.array([10,20,30,40])
b = np.arange(4)
c = b-a
d = c**2
# np 中的函数 sin 等
e = 10
f = np.sin(a)
print(c<3)
# [ True  True  True  True]
```

1.3.2 高维矩阵计算

```
#逐个相乘
c = a*b
# 矩阵乘法
c_dot1 = np.dot(a,b)
c_dot2 = a.dot(b)

a = np.random.random((2,4))
# axis=0 列求和    axis=1 行求和
np.sum(a)
```

```
np.min(a)
np.max(a)
```

1.3.3 基础运算

```
# 最小值的索引
np.argmin(a)
np.argmax(a)
# 整个矩阵的平均值   中位数
np.mean(a)
np.median(a)
# 累加：后面=前面 1 加前面 2.....    累差
np.cumsum(a)
np.diff(a)
# 非零的数    nonzero(a)
# 排序
np.sort(a)
# 矩阵转置 这个对[1,1,1] 这种行向量是无效的
np.transpose(a)
(a.T).dot(a)
#可以”添加一个维度”,使得行向量变成一维矩阵
a[np.newaxis,:]
#直接变成列的一维矩阵,也可以用 reshape
a[:,np.newaxis]
# 矩阵截取   卡尔曼滤波
np.clip(a,5,9)
```

1.4 Numpy 索引

```
# 第 i 行   索引从零开始
A[3]
B[2][3]   B[2,3]
C[2.:]
D[2,1:3]    #(1,3]
```

1.5 遍历

```
# 遍历每一行  迭代时是对行的迭代，如果要对列，就得进行转置
for row in A:
  print(row)
# 遍历每一项
A.flatten()    #铺平
for item in A.flat;
  print(item)
```

1.6 Numpy 的 array 合并

```
# 上下合并
np.vstack((a,b))
#左右合并
np.hstack((a,b))
# 多个 array 的合并,concatenate 可以在后面进行指定维度
np.concatenate((a,b,b,a),axis = 0)
```

1.7 Numpy array 的分割

```
# 等分割 一个 array 分割为 2 个 array axis=1 对列进行操作
np.split(a,2,axis=1)
# 不等分割
np.array_split(a,3,axis=1)
# 纵向分割
np.vsplit(a,2)
# 横向分割
np.hsplit(a,2)
```

1.8 Numpy copy

```
# a b c d 是同值得
a = [1,2,3,4]
b = a
c = d
d = b
a[0] = 11
```

```
# 不想关联    深度 deep copy
b = a.copy()
```

二、Pandas

如果用 Python 的列表和字典作比较，那么可以说 Num 构建的，让 Numpy 为中心的应用变得 py 是列表形式的，没有数值标签，二 Pandas 是字典形式的。Pandas 是基于 Numpy 为中心的数据变得更加简单。

2.1 Pandas 数据结构

2.1.2 Series

Series 的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据指定索引。于是会自动创建一个 0 到 N-1（N 为长度）的整数型索引。

```
s = pd.Series([1,3,6,np.nan,44,1])
#np.nan 表示空

print(s)

"""
0     1.0
1     3.0
2     6.0
3     NaN
4    44.0
5     1.0
dtype: float64
"""
```

2.1.2 DataFrame

DataFrame 是一个表格型的数据结构，它包含有一组有序的列，每列可以是不同的值类型（数值，字符串，布尔型）。DataFrame 既有行索引又有列索引，他可以被看作是由 Series 组成的大字典。

```
df1 = pd.DataFrame(np.arange(12).reshape((3,4)))
print(df1)

"""
   0  1   2   3
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
"""
# 指定列名称
df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1, index=list(range(4)),
dtype='float32'),
                    'D': np.array([3] * 4, dtype='int32'),
                    'E': pd.Categorical(["test", "train", "test",
"train"]),
                    'F': 'foo'})

print(df2)

"""
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
"""
```

2.1.2.1 查看列序号和行序号

```
# 查看列序号
df.index
# 查看行序号
```

```
df.columns
# 查看所有 df 的值
df.values
# 查看 DataFrame 的数据数字特征
df.describe
# 翻转数据
df.T
```

2.1.2.2 排序

```
# 对数据进行 index 排序 axis=1 对列进行排序 ascending=Flase 逆序
df.sort_index(axis=1,ascending=False)
# 对数据值进行排序输出
df.sort_values(by='B')
```

2.2 Pandas 选择数据（数据筛选）

```
# 按照特定的列进行选择
df.A
# 按照标签进行选择 并打印出序列的数据
df.loc['20130102']
# 保存行中的数据，返回列的项目数据列头和行中的所有信息
df.loc[:,['A','B']]
# select by position:iloc
df.iloc[3]   #第三行
df.iloc[3,1]  #第三行第一列
df.iloc[3:5,1:3]  #切片
df.iloc[[1,2,3],1:3]
# mixed selection:ix   混合选
df.ix[:3,['A','C']]
# Boolean indexing   对比一列中的数据
df.df[A<8]
```

2.3 Pandas 设置值

```
# 用不同的定位方法
df.iloc[2,2] = 111
df.loc['20130102','A'] = 222
df[df.A>0] = 0   #  这里是对 A>0 的每一行进行了改为 0
df.A[df.A>0] = 0 #这里只是对 A 那一列进行改为 0

# 加上空的行
df['F'] = np.nan
```

2.4 处理丢失的数据

```
#按照 axis = 0 水平即为行  how = 'any' 任何一个 nan
# 就丢弃  how="all"  一行或者一列全为 nan 才丢弃
df.dropna(axis = 0,how= 'any')
# fillna() 填上 value=0.0
df.fillna(value = 0)
# 是否缺失数据
df.isnull()
np.any(df.isnull()) == True
```

2.5 数据的导入导出

```
# pandas 可以读取  .cvs  .pickle  .html .sql .excel .json
data = pd.read_csv('student.csv')
```

```
data.to_pickle("stu.pickle")
```

2.6 列表的合并 concat

```
# concatenating
# 列名相同时 concat  axis = 0  水平上下合并
# ignore_index = True  忽略原本的行号，重新进行排序
pd.concat([df1,df2,df3],ignore_index = True)

# 列名不相同，有重合或者不重合
# concat 的 join 功能 join,['inner','outer']
pd.concat([df1,df2])  #默认 join=outer 输出的时候回出现 nan
pd.concat([df1,df2],join='inner')  #只对重复的列进行合并
# 找交集并进行输出
pd.concat([df1,df2],axis=1,join_axes="df1.index")
# 不找交集并进行输出
pf.concat([df1,df2],axis=1)
```

2.7 append

```
# append
df1.append(df2,ignore_index = True)
df1.append([df2,df3],ignore_index = True)
# 一项一项的进行添加
s1 = pd.Series([1,2,3,4],index = ['a','b','c','d'])
df1.append(s1,ignore_index = True)
```

2.8 merge

2.8.1 合并

```
#merge
left = pd.DataFrame({
    'key':['K0','K1','K2','K3'],
     'A':['A0','A1','A2','A3'],
     'B':['B0','B1','B2','B3']
})
right = pd.DataFrame({
    'key': ['K0', 'K1', 'K2', 'K3'],
    'C': ['C0', 'C1', 'C2', 'C3'],
     'D': ['D0', 'D1', 'D2', 'D3']
})
# 基于一组 Key 进行合并
pd.merge(left,right,on = 'key')

# 基于两组 Key 进行合并  how 默认 inner
['left','right','outer','inner']
# 用 key1 和 key2 进行对比，如果没哟 nan 进行填充或直接失去
# outer 没有的进行填充
pd.merge(left,right,on=['key1','key2'],how='inner')

# indicator  indicator=True  默认 False
#定义资料集并打印出
df1 = pd.DataFrame({'col1':[0,1], 'col_left':['a','b']})
df2 = pd.DataFrame({'col1':[1,2,2],'col_right':[2,2,2]})

print(df1)
#   col1 col_left
# 0     0        a
# 1     1        b
```

```
print(df2)
#   col1  col_right
# 0     1          2
# 1     2          2
# 2     2          2

# 依据 col1 进行合并，并启用 indicator=True，最后打印出
res = pd.merge(df1, df2, on='col1', how='outer', indicator=True)
print(res)
#   col1 col_left  col_right      _merge
# 0   0.0        a        NaN   left_only
# 1   1.0        b        2.0        both
# 2   2.0      NaN        2.0  right_only
# 3   2.0      NaN        2.0  right_only

# 自定 indicator column 的名称，并打印出
res = pd.merge(df1, df2, on='col1', how='outer',
indicator='indicator_column')
print(res)
#   col1 col_left  col_right indicator_column
# 0   0.0        a        NaN        left_only
# 1   1.0        b        2.0             both
# 2   2.0      NaN        2.0       right_only
# 3   2.0      NaN        2.0       right_only
```

index

```
#定义资料集并打印出
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                     index=['K0', 'K1', 'K2'])
right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                     index=['K0', 'K2', 'K3'])

print(left)
#     A   B
# K0  A0  B0
# K1  A1  B1
# K2  A2  B2

print(right)
#     C   D
# K0  C0  D0
```

```
# K2  C2  D2
# K3  C3  D3

#依据左右资料集的 index 进行合并，how='outer',并打印出
res = pd.merge(left, right, left_index=True, right_index=True,
how='outer')
print(res)
#      A    B    C    D
# K0   A0   B0   C0   D0
# K1   A1   B1  NaN  NaN
# K2   A2   B2   C2   D2
# K3  NaN  NaN   C3   D3

#依据左右资料集的 index 进行合并，how='inner',并打印出
res = pd.merge(left, right, left_index=True, right_index=True,
how='inner')
print(res)
#     A   B   C   D
# K0  A0  B0  C0  D0
# K2  A2  B2  C2  D2
```

2.8.4 解决 overlapping 的问题

```
#定义资料集
boys = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'age': [1, 2, 3]})
girls = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'age': [4, 5, 6]})

#使用 suffixes 解决 overlapping 的问题
res = pd.merge(boys, girls, on='k', suffixes=['_boy', '_girl'],
how='inner')
print(res)
#    age_boy   k  age_girl
# 0        1  K0         4
# 1        1  K0         5
```

2.9 plot data

2.9.1 Series

```
# Series 线性的数据
data =
pd.Series(np.random.random(1000),index=np.arange(1000))
data = data.cumsum()    #对数据进行累加
data.plot()
plt.show()
```

2.9.2 dataFrame

```
#DataFrame   类似矩阵的数据
data =pd.DataFrame(np.random.randn(1000,4),
                   index = np.arange(1000),
                   columns=list("ABCD"))
data = data.cumsum()
print(data.head())
data.plot()
plt.show()
```

2.9.3 plot method

```
# plot methods:bar hist box kde area scatter hexbin pie
ax = data.plot.scatter(x='A',y='B',color='DarkBule',label='Class 1')
data.plot.scatter(x='A',y='C',color="DarkGreen",label='Class 2',ax
=ax)
```

三、Matplotlib

1. *Matplotlib* 是一个非常强大的 *Python* 画图工具;
2. 手中有很多数据，可是不知道该怎么呈现这些数据.

所以就找到了 *Matplotlib*. 它能帮你画出美丽的:线图;散点图;等高线图;条形图;柱状图;3D 图形,甚至是图形动画等等.

3.1 基本用法

```
# matplotlib
import matplotlib.pyplot as plt
import numpy as np

# -1 到 1 的 50 个点
x = np.linspace(-1,1,50)
y = 2*x +1
plt.plot(x,y)
plt.show()
```

3.2 figure

```
# figure  大窗口
x = np.linspace(-3,3,50)
y1 = 2*x +1
y2 = x**2

plt.figure()
plt.plot(x,y1)

plt.figure(num = 3,figsize=(8,5))
plt.plot(x,y2)
```

```
plt.plot(x,y1,color='red',linewidth=1.0,linestyle = '-')

plt.show()
```

3.3 设置坐标轴

```
# 坐标轴范围
plt.xlim((-1,2))
plt.ylim((-2,3))
# x y 轴名称
plt.xlabel('I am x')
plt.ylabel('I am y')
#  换角标
new_ticks = np.linspace(-1,2,5)
plt.xticks(new_ticks)
# $really\ bad$  这样字体会好看些（数学）
# 特殊符号 alpha: \alpha
plt.yticks([-2,-1.8,-1,1.22,3],
           ['really bad','bad','normal','good','really good'])
```

3.4 移动 x，y 轴位置

```
# gca="get current axis"修改坐标轴的位置
ax = plt.gca()
# 让上面和右边的脊梁消失
ax.splines['right'].set_color('none')
ax.splines['top'].set_color('none')
# 设置 x 为下面，y 为左边  消失 x 轴和 y 轴
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# 开始动位置
# x 轴绑定在 y 轴的-1 的位置
ax.spines['bottom'].set_position(('data',-1))
# y 轴绑定在 x 轴的 0 的位置
ax.spines['left'].set_position(('data',0))
```

3.5 加上标签

```
# 区分每一个坐标轴中的数据
# 对每一条线加上一个标签,即名字
plt.plot(x1,y1,label='up')
```

```
# handles  loc 会自动的找一个数据比较少的地方进行放置
plt.legend(handles=[11,12],label=['aaa','bbb'],loc='')
# 只打印 aaa
plt.legend(handles=[11,],label=['aaa',],loc='')
```

3.6 对坐标轴加注解

```
# 在图片中添加注解
x0 =1
y0 = 2*x0 +1
plt.scatter(x0,y0,s = 50,color='b')  #散点
# scatter 表示点 plot 表示线
#[x0,x0] 表示 x0 一个点，[y0,0]表示一条与 y0 对其的线
# k--表示 黑色 并且虚线 lw 线宽表示 2.5
plt.plot([x0,x0],[y0,0],'k--',lw=2.5)

# method1
plt.annotate(r'$2x+1=%$'%y0,xy=(x0,y0),
             xycoords='data',
             xytext=(+30,-30),textcoords='offset points',
             fontsize = 16,
            arrowprops=dict(arraystyle='->',
                            connectionstyle='arx3',rad=.2))
# method2 (开始位置，)
plt.text(-3.7,3,r'$This\ is\ the\ some\ \alpha_t$')
```

3.7 tick 能见度

```
# 将 x 轴和 y 轴上的数值加深
for label in ax.get_xticklabels+ax.get_yticklabels():
    label.set_fontsize(12)
    label.set_bbox(dict(facecolor='white',edgecolor='None',
                        alpha=0.7))
```

3.8 Scatter 散点图

```
n = 1024
X = np.random.normal(0,1,n)  #平均值为 0 方差为 1
Y = np.random.normal(0,1,n)
```

```
# 颜色(好看而已)
T = np.arctan2(Y,X)
# s : size   c: color  alpha :透明度
plt.scatter(X,Y,s = 75,c=T,alpha=0.5)
plt.xlim((-1.5,1.5))
plt.ylim((-1.5,1.5))
# 隐藏所有的 tick
plt.xticks(())
plt.yticks(())
plt.show()
```

3.9 饼状图

```
# +向上  -向下
plt.bar(X,+Y1,facecolor='#9999ff',edgecolor='white')
plt.bar(X,-Y2)
# 加上数值
for x,y in zip(X,Y1):
    #          位置        ha:harizontal alignment
    plt.text(x+0.4,y+0.05,'%.2f'%y,ha='center',va='bottom')
```

3.10 等高线

```
# 等高线图 contours
def f(x,y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-x**2 -y**2)

n= 256
x = np.linspace(-3,3,n)
y = np.linspace(-3,3,n)
X,Y = np.meshgrid(x,y)
#用 meshgrid 把二维平面中将每一个 x 和每一个 y 分别对应起来，编制成栅格

# use plt.contourf to filling contours
# X, Y and value for (X,Y) point
plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap=plt.cm.hot)
# use plt.contour to add contour lines
C = plt.contour(X, Y, f(X, Y), 8, colors='black', linewidth=.5)
#8 代表分成了 8+2 份、
plt.clabel(C, inline=True, fontsize=10)
plt.xticks(())
plt.yticks(())
```

3.11 subplot 一个 figure 上有几个小图

```
plt.figure()
# 2 行 2 列 位置 1
plt.subplot(2,2,1)
plt.plot([0,1],[0,1])
# 2 行两列 位置 2
plt.subplot(2,2,2)
plt.plot([0,1],[0,1])
# 2 行两列 位置 3
plt.subplot(2,2,3)
plt.plot([0,1],[0,1])
# 2 行两列 位置 4
plt.subplot(2,2,4)
plt.plot([0,1],[0,1])
```

```
#    1
# 4  5  6
plt.subplot(2,1,1)
plt.subplot(2,3.4)
plt.subplot(2,3,5)
plt.subplot(2,3,6)
```

3.12 subplot 分格显示

```
# method1 subplot2grid
plt.figure()
# (3,3) 3 行 3 列    (0,0) 从 (0,0) 开始
# colspan 列跨度   rowspan  行跨度  默认为 1
ax1 = plt.subplot2grid((3,3),(0,0),colspan=3,rowspan=1)
ax1.plot([1,2],[1,2])
ax1.set_title('ax1_title')
```

3.13 主次坐标轴

```
x = np.arange(0, 10, 0.1)
y1 = 0.05 * x**2
y2 = -1 * y1

fig,ax1 = plt.subplots()
# 镜面反射
ax2 = ax1.twinx()
ax1.plot(x,y1,'g-')
ax2.plot(x,y2,'b--')

ax1.set_xlabel('X_data')
ax1.set_xlabel('Y1',color='g')
ax1.set_ylabel('Y1 data', color='g')
ax2.plot(x, y2, 'b-') # blue
ax2.set_ylabel('Y2 data', color='b')
plt.show()
```

3.14 动画

```
from matplotlib import animation
```

四、Python

1. Python 综述

1.1 python 是什么

Python 是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

- Python 的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。
- Python 是一种解释型语言：这意味着开发过程中没有了编译这个环节。类似于 PHP 和 Perl 语言。
- Python 是交互式语言： 这意味着，我们可以在一个 Python 提示符后面直接互动执行写自己的程序。
- Python 是面向对象语言: 这意味着 Python 支持面向对象的风格或代码封装在对象的编程技术。
- Python 是初学者的语言：Python 简单易学，对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。

1.2 python 的发展

Python 是由 Guido van Rossum（龟叔） 在八十年代末和九十年代初，在荷兰国家数学和计算机科学研究所设计出来的。

Python 本身也是由诸多其他语言发展而来的,这包括 ABC、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell 和其他的脚本语言等等。像 Perl 语言一样，Python 源代码同样遵循 GPL(GNU

General Public License)协议。现在 Python 是由一个核心开发团队在维护，Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

1.3 python 的特点

- 易于学习：Python 有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
- 易于阅读：Python 代码定义的更清晰。
- 易于维护：Python 的成功在于它的源代码是相当容易维护的。
- 一个广泛的标准库：Python 的最大的优势之一是丰富的库，跨平台的，在 UNIX，Windows 和 Macintosh 兼容很好。
- 互动模式：互动模式的支持，您可以从终端输入执行代码并获得结果的语言，互动的测试和调试代码片断。
- 可移植：基于其开放源代码的特性，Python 已经被移植（也就是使其工作）到许多平台。
- 可扩展：如果你需要一段运行很快的关键代码，或者是想要编写一些不愿开放的算法，你可以使用 C 或 C++完成那部分程序，然后从你的 Python 程序中调用。
- 数据库：Python 提供所有主要的商业数据库的接口。
- GUI 编程：Python 支持 GUI 可以创建和移植到许多系统调用。
- 可嵌入: 你可以将 Python 嵌入到 C/C++程序，让你的程序的用户获得"脚本化"的能力。

Python3 安装

2.1 python3 和 python2 的区别

Python 的 3.0 版本，常被称为 Python 3000，或简称 Py3k。相对于 Python 的早期版本，这是一个较大的升级。

为了不带入过多的累赘，Python 3.0 在设计的时候没有考虑向下相容。

许多针对早期 Python 版本设计的程式都无法在 Python 3.0 上正常执行。

为了照顾现有程式，Python 2.6 作为一个过渡版本，基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。

新的 Python 程式建议使用 Python 3.0 版本的语法。

除非执行环境无法安装 Python 3.0 或者程式本身使用了不支援 Python 3.0 的第三方库。目前不支援 Python 3.0 的第三方库有 Twisted, py2exe, PIL 等。

Python 3.0 在 print 函数、Unicode 编码、除法运算、数据类型和异常等方面都与 2.X 版本有所变化。

2.2 python3 环境的安装（以 windows 为例）

第一步，下载 python 环境安装包

官方地址：https://www.python.org/

第二步，安装 python 环境

第三步，测试是否安装成功

打开 cmd 命令终端，输入 python，显示版本信息表示安装成功

2.3 Anaconda 安装（可选）

如果是在 windows 系统中安装，为了更简单地使用 python 中丰富的库资源，可以直接安装一个 python“全家桶”——Anaconda。

Anaconda 是一个 python 的发行版，包括了 python 和很多常见

的软件库，和一个包管理器 conda。常见的科学计算类的库都包含在里面，使得安装比常规 python 安装要容易。注意，装了 Anaconda 就不需要再装 python 了。

Anaconda 不仅可以方便地安装、更新、卸载工具包，而且安装时能自动安装相应的依赖包，同时还能使用不同的虚拟环境隔离不同要求的项目；从而大大简化了工作流程。

下载地址：https://www.anaconda.com/distribution/

下载需要的对应版本，安装非常简单，只要跟着引导一步步做就可以了。

3. Python 基本语法

3.1 编码

默认情况下，python3 源文件以 UTF-8 编码，所有字符串都是 unicode 字符串。同时可以指定源文件的不同编码

文件开头加上

```
# -*- coding: UTF-8 -*-
# coding=utf-8(等号两边不能有空格)
```

允许在源文件中使用 utf-8 字符集中的字符编码，对应的适合语言为中文等。

3.2 标识符

- 第一个字符必须是字母表中的字母或下划线_

- 标识符中的其他部分由字母、数字和下划线组成
- 标识符对大小写敏感
- 在 python3 中，非 ASCII 标识符 (如中文字符) 也是允许的

3.3 注释

单行注释：井号 #

多行注释：三个单引号'''，或者三个双引号"""

3.4 关键子和保留字

```
import keyword
print(len(keyword.kwlist))      #33
print(keyword.kwlist)           #打印关键字
```

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

简介：

1.and：表示逻辑‘与’

2.del：用于 list 列表操作，删除一个或者连续几个元素。

3.from：导入相应的模块，用 import 或者 from...import

4.not：表示逻辑‘非’

5.while：while 循环，允许重复执行一块语句，一般无限循环的情况下用它

6.as：as 单独没有意思，是这样使用：with....as 用来代替传统的 try...finally 语法的。

7.elif:和 if 配合使用的，if 语句中的一个分支用 elif 表示。

8.global :定义全局变量

例如：

```
name = 10
age = 20
def test():
    global name
    age = 30
    name = 'aa'
    #print(name)
test()
print(name)          # 'aa'
print(age)       # 20
```

9.or：表示逻辑“或”

10.with：和 as 一起用，使用的方法请看 as，见 with

11.assert：表示断言（断言一个条件就是真的，如果断言出错则

抛出异常）用于声明某个条件为真，如果该条件不是真的，则抛出异常：AssertionError

```
v1 = 10
v2 = 20
assert(v1 > v2)
```

12.else：参考下面 if 的解释

13.if：if 语句用于选择分支，依据条件选择执行那个语句块。(if 语句中最好不要嵌套 if 语句,建议把嵌套的 if 语句写在另一个函数中)

14.pass：pass 的意思就是什么都不做

15.yield：用起来和 return 很像，但它返回的是一个生成器

16.break：作用是终止循环，程序走到 break 的地方就是循环结束的时候。

17.except：和 try 一起使用，用来捕获异常。

18.import：用来导入模块，有时这样用 from....import

19.class：定义类

20.in：查找列表中是否包含某个元素，或者字符串 a 是否包含字符串 b。

21.raise：raise 可以显示地引发异常。一旦执行 raise 语句，后面的代码就不执行了

22.continue：跳过 continue 后面循环块中的语句，继续进行下一

轮循环。

23.finally:看到 finally 语句，必然执行 finally 语句的代码块。

24.is：Python 中的对象包含三要素：id、type、value,用来判断对象是否相等

25.return：用于跳出函数，也可以在跳出的同时返回一个值。

26.def：用于定义方法

27.for：for...in 一起使用：它在一序列的对象上递归，就是遍历队列中的每个项目

28.lambda:即匿名函数

29.try：出现在异常处理中，使用格式为：try...except，try 中放想要执行的语句，except 捕获异常

30.nonlocal：nonlocal 关键字用来在函数或其他作用域中使用外层(非全局)变量

例如：

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

```
def make_counter_test():
    mc = make_counter()
    print(mc())
    print(mc())
    print(mc())
make_counter_test()
```

3.5 行和缩进

学习 Python 与其他语言最大的区别就是，Python 的代码块不使用大括号{}来控制类，函数以及其他逻辑判断。python 最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。

例如：

```
if True:
    print "True"
else:
    print "False"
```

3.6 多行语句

Python 语句中一般以新行作为语句的结束符。但是我们可以使用斜杠（ \）将一行的语句分为多行显示，如下所示：

```
total = item_one + \
        item_two + \
        item_three
print('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
      aaaaaaaaaaaaaaaaaaa')
```

语句中包含 [], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

3.7 python 的引号

Python 可以使用引号(')、双引号(")、三引号(''' 或 """)表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用于文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "这是一个句子。"
paragraph = """这是一个段落。
包含了多个语句"""
```

3.8 Python 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。空行与代码缩进不同，空行并不是 Python 语法的一部分。书写时不插入空行，Python 解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

3.9 等待用户输入

输入的类型为 float 类型

执行下面的程序在按回车键后就会等待用户输入：

```
input("\n\n 按下 enter 键后退出。")
```

3.10 同一行显示多条语句

Python 可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
x = 'runoob';print(x + '\n')
```

3.11 多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。像 if、while、def 和 class 这样的复合语句，首行以关键字开始，以冒号(:)结束，该行之后的一行或多行代码构成代码组。

我们将首行及后面的代码组称为一个子句(clause)。

例如：

```
if expression :
    suite
elif expression :
    suite
else :
    suite
```

3.12 Print 输出

同类型才可以相加

print 默认输出是换行的，如果要实现不换行需要在变量末尾加上 end=""：

```
x="a"
y="b"
# 换行输出
print( x )
print( y )

print('---------')
# 不换行输出
```

```
        print( x, end=" " )
        print( y, end=" " )
        print()
```

3.13 import 与 from...import

在 python 用 import 或者 from...import 来导入相应的模块。

a、将整个模块(somemodule)导入，格式为： import somemodule

b、从某个模块中导入某个函数,格式为： from somemodule import somefunction

c、从某个模块中导入多个函数,格式为： from somemodule import firstfunc, secondfunc, thirdfunc

d、将某个模块中的全部函数导入，格式为： from somemodule import *

(1) 导入 sys 模块

```
        import sys
        print('================Python import mode==================');
        print ('命令行参数为:')
        for i in sys.argv:
            print (i)
```

```
print ('\n python  路径为',sys.path)
```

(2) 导入 sys 模块的 argv,path 成员

```
from sys import argv,path    #    导入特定的成员
print('================python                                    from
import====================')
print('path:',path) #  因为已经导入 path 成员，所以引用不需要
```

加 sys.path

3.14 命令行参数

很多程序可以执行一些操作来查看一些基本信息，Python 可以使用-h 参数查看各参数帮助信息：

```
$ python -h
  usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d       : debug output from parser (also PYTHONDEBUG=x)
-E       : ignore environment variables (such as PYTHONPATH)
-h       : print this help message and exit
```

4. 基本数据类型

Type len id

4.1 变量赋值

```
counter = 100       # 整型变量
miles = 1000.0      # 浮点型变量
name = "runoob"       # 字符串

print (counter)
print (miles)
print (name)
```

4.2 多变量赋值

Python 允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为 1，从后向前赋值，三个变量被赋予相同的数值。

可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "runoob"
a,b = b,a              # 变量的交换
```

以上实例，两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象 "runoob" 分配给变量 c。

注意：

```
a = 10
b = 20
a,b = b,a+5
print(a,b)
```

结果：a = 20,b=15

4.3 标准数据类型

Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Set（集合）、Dictionary（字典）

a、不可变数据（3 个）：Number（数字）、String（字符串）、Tuple（元组）

b、可变数据（3 个）：List（列表）、Dictionary（字典）、Set（集合）

4.4 Number

int、float、bool、complex(复数)

例如：

```
a,b,c,d = 20,5.5,True,5+4j
print(type(a),type(b),type(c),type(d))
```

4.1.1 函数

```
type(a)              # 判断数据类型
isinstance(a,int)        # 判断数据是否属于某类型
del var1,var2        # 手动 GC
```

区别:

(1) type()不会认为子类是一种父类类型

(2) isinstance()会认为子类是一种父类类型

例如：

```
class A:
    pass
class B(A):
    pass
print(type(A()) == A)
print(type(B()) == A)
print(isinstance(B(),A))
print(isinstance(B(),A))
```

4.1.2 进制

二进制：使用 0b 开头　例如：0b1010

八进制：使用 0o 开头　例如：0o555

十六进制：0x 开头　　例如：0x52A74（大小写都 OK）

python 中没有数字的大小限制，可以使用任意大的数字；python 可以保证整数运算的精确，但是浮点数运算时可能会得到一个不精确的结果。

4.1.3 数学函数

import math

(1) 基本数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如 abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如 math.ceil(4.1) 返回 5
(x>y)-(x<y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)	返回 e 的 x 次幂(ex), 如 math.exp(1) 返回 2.718281828459045
fabs(x)	返回数字的绝对值，如 math.fabs(-10) 返回 10.0

floor(x)	返回数字的下舍整数，如 math.floor(4.9)返回 4
log(x)	如 math.log(math.e)返回 1.0,math.log(100,10)返回 2.0
log10(x)	返回以 10 为基数的 x 的对数， 如 math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回 x 的整数部分与小数部分， 两部分的数值符号与 x 相同， 整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数 x 的四舍五入值，如给出 n 值， 则代表舍入到小数点后的位数。
sqrt(x)	返回数字 x 的平方根。

(2) 随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

函数	**描述**
choice(seq)	从序列的元素中随机挑选一个元素， 比如 random.choice(range(10))， 从 0 到 9 中随机挑选一个整数。

randrange ([start,] stop [,step]) 从指定范围内，按指定基数递增的集合 中获取一个随机数，基数缺省值为 1 random() 随机生成下一个实数，它在[0,1)范围内。 seed([x]) 改变随机数生成器的种子 seed。 如果你不了解其原理，你不必特别去设定 seed， Python 会帮你选择 seed。 shuffle(lst) 将序列的所有元素随机排序 uniform(x, y) 随机生成下一个实数，它在[x,y]范围内。

(3) 三角函数

函数	描述
acos(x)	返回 x 的反余弦弧度值。
asin(x)	返回 x 的反正弦弧度值。
atan(x)	返回 x 的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回 x 的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)	返回的 x 弧度的正弦值。

tan(x)	返回 x 弧度的正切值。
degrees(x)	将弧度转换为角度,如 degrees(math.pi/2) ， 返回 90.0
radians(x)	将角度转换为弧度

(4) 数学常量

常量	描述
pi	数学常量 pi（圆周率，一般以 π 来表示）
e	数学常量 e，e 即自然常数（自然常数）。

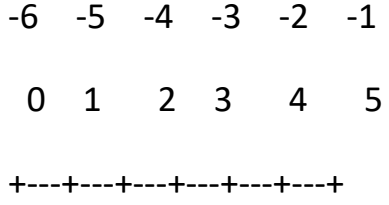
4.5 String

Python 中的字符串用单引号 ' 或双引号 " 括起来，同时使用反斜杠 \ 转义特殊字符。下标从 0 开始。

加号 + 是字符串的连接符， 星号 * 表示复制当前字符串，紧跟的数字为复制的次数。

(1) 字符串截取

变量[头下标:尾下标:步长)



| a | b | c | d | e | f |

+---+---+---+---+---+---+

例如：

```
str = 'hello world'
l = str[0:1:1]
print(l)
```

(2) 字符串打印

Python 使用反斜杠(\)转义特殊字符，如果你不想让反斜杠发生转义，可以在字符串前面添加一个 r，表示原始字符串：

```
print('Ru\noob')
Ru
oob
print(r'Ru\noob')
Ru\noob
```

(3) 字符串获取

```
print(str[0])
```

注意：

1、反斜杠可以用来转义，使用 r 可以让反斜杠不发生转义。

2、字符串可以用+运算符连接在一起，用*运算符重复。

3、Python 中的字符串有两种索引方式，从左往右以 0 开始，从右往左以-1 开始。

4、Python 中的字符串不能改变。

```
str = 'abcdef'
str[0] = 's'  # 报错
```

5、Python 没有单独的字符类型，一个字符就是长度为 1 的字符串。

4.6 List

List（列表）是 Python 中使用最频繁的数据类型。列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表(所谓嵌套)。列表是写在方括号[]之间、用逗号分隔开的元素列表。和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。

定义：

```
list = [0,1,2,3,'c']
list[0]          # 0
len(list)        # 长度 5
list[0:3]        # [0,1,2]
```

注意：

1、List 写在方括号之间，元素用逗号隔开。

2、和字符串一样，list 可以被索引和切片。

3、List 可以使用+操作符进行拼接。

4、List 中的元素是可以改变的。

5、不支持与或非运算

4.7 Tuple

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号 () 里，元素之间用逗号隔开。

定义：

```
t = (0,1,2,3,'c')
t[0]          # 0
len(list)         #  长度 5
t[0:3]            # (0,1,2)
t[0] = 2      #  报错
```

注意：

1、与字符串一样，元组的元素不能修改。虽然 tuple 的元素不可改变，但它可以包含可变的对象，比如 list 列表。

```
list = [0,2,1,3]
t = (0,1,2,3,list)
t[0] = 1          #  报错
t[4][0] = 1
```

```
print(t[4])          # ok
```

2、元组也可以被索引和切片，方法一样。

3、注意构造包含 0 或 1 个元素的元组的特殊语法规则。

```
tup1 = ()            # 空元组
tup2 = (20,)         # 一个元素，需要在元素后添加逗号
```

4、元组也可以使用 + 或 * 操作符进行拼接。

4.8 Set

集合（set）是由一个。合的事物或对象称作元素或是成员。基本功能是进行成员关系测试和删除重复元素。

可以使用大括号 { } 或者 set() 函数创建集合，注意：创建一个空集合必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典。

创建格式：

```
parame = {value01,value02,...} 或者 set(value)
```

```
s = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}
print(s)                    # {'Mary', 'Jack', 'Rose', 'Tom', 'Jim'}

s = set('1b1b2b3b2b')       # {'2', '3', '1', 'b'}
```

set 操作：

```
# set 可以进行集合运算
a = set('abracadabra')
b = set('alacazam')

print(a)
>>> {'b', 'a', 'c', 'r', 'd'}
print(a - b)        # a 和 b 的差集
>>> {'b', 'd', 'r'}
print(a | b)        # a 和 b 的并集
>>> {'l', 'r', 'a', 'c', 'z', 'm', 'b', 'd'}
print(a & b)        # a 和 b 的交集
>>> {'a', 'c'}
print(a ^ b)        # a 和 b 中不同时存在的元素
>>> {'l', 'r', 'z', 'm', 'b', 'd'}
```

4.9 Dictionary（字典）

字典（dictionary）是 Python 中另一个非常有用的内置数据类型。

列表是有序的对象集合，字典是无序的对象集合。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典是一种映射类型，字典用"{ }"标识，它是一个无序的键(key) : 值(value)对集合。

键(key)必须使用不可变类型。

在同一个字典中，键(key)必须是唯一的。

字典创建：

(1) d = {"a":1,"b":2,"c":3}

(2) d = dict([('Runoob', 1), ('Google', 2), ('Taobao', 3)])

(3) d = {x:x**2 for x in (2, 4, 6)}

(4) d = dict(Runoob=1, Google=2, Taobao=3)

字典操作：

```
tinydict = {'name': 'guigu','code':1, 'site': 'www.atguigu.com'}


print (dict['name'])            # 输出键为 'name' 的值
print (dict['code'])          # 输出键为 'code' 的值
print (tinydict)              # 输出完整的字典
print (tinydict.keys())          # 输出所有键dict_keys(['name', 'code', 'site'])
print (tinydict.values())       # 输出所有值 dict_values(['guigu', 1, 'www.atguigu.com'])
```

4.10 python 类型转换

函数	描述
int(x [,base])	将 x 转换为一个整数
float(x)	将 x 转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效 Python 表达式，并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典，d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符（ASCII 码）
ord(x)	将一个字符转换为它的 ASCII 码值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

5. python 解释器

Linux/Unix 的系统上，一般默认的 python 版本为 2.x，我们可以将 python3.x 安装在 /usr/local/python3 目录中。

安装完成后，我们可以将路径 /usr/local/python3/bin 添加到您的 Linux/Unix 操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python3 。

```
$ PATH=$PATH:/usr/local/python3/bin/python3      # 设置环境变量
$ python3 --version
Python 3.4.0
```

5.1 环境变量设置

在 Window 系统下你可以通过以下命令来设置 Python 的环境变量，假设你的 Python 安装在 C:\Python34 下:

```
set path=%path%;C:\python34
```

5.2 交互式编程

我们可以在命令提示符中输入"python"（或者"python3"，具体视安装时的命令名称而定）命令来启动 Python 解释器：

```
$ python3
```

(1) 执行以上命令后，出现如下窗口信息：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(2) 在 python 提示符中输入以下语句，然后按回车键查看运行效果：

```
print ("Hello, Python!");
```

(3) 以上命令执行结果如下：

```
Hello, Python!
```

(4) 当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> flag = True
```

```
>>> if flag :
...         print("flag 条件为 True!")
...
flag 条件为 True!
```

5.3 脚本式编程

(1) 将如下代码拷贝至 hello.py 文件中：

```
print ("Hello, Python!");
```

(1) 通过以下命令执行该脚本：

```
python3 hello.py
```

(2) 输出结果为：

```
Hello, Python!
```

(3) 在 Linux/Unix 系统中，你可以在脚本顶部添加以下命令让 Python 脚本可以像 SHELL 脚本一样可直接执行：

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

(4) 然后修改脚本权限，使其有执行权限，命令如下：

```
$ chmod +x hello.py
```

(5) 执行以下命令：

```
./hello.py
```

(6) 输出结果为：

```
Hello, Python!
```

6. 运算符

6.1 算术运算符

a = 10 b = 21

运算符	描述	实例
+	加	两个对象相加； a + b 输出结果 31
-	减	得到负数或是一个数减去另一个数； a - b 输出结果 -11

*	乘	两个数相乘或是返回一个被重复若干次的字符串； a * b 输出结果 210
/	除	x 除以 y； b / a 输出结果 2.1
%	取模	返回除法的余数； b % a 输出结果 1
**	幂	返回 x 的 y 次幂 ； a**b 为 10 的 21 次方
//	取整除	向下取接近除数的整数 ； 9//2 # 4 -9//2 #-5

6.2 比较运算符

a = 10 b = 20

运算符	描述	实例
==	等于	比较对象是否相等 (a == b) 返回 False
!=	不等于	比较两个对象是否不相等 (a != b) 返回 True

>	大于	返回 x 是否大于 y (a > b) 返回 False
<	小于	返回 x 是否小于 y (a < b) 返回 True
>=	大于等于	返回 x 是否大于等于 y (a >= b) 返回 False
<=	小于等于	返回 x 是否小于等于 y (a <= b) 返回 True

注意：所有比较运算符返回 1 表示真，返回 0 表示假。这分别与特殊的变量 True 和 False 等价。注意，这些变量名的大写。

6.3 赋值运算符

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为
c+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

6.4 位运算符

a = 60,　b = 13

二进制形式：a = 0011 1100,　b = 0000 1101

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ，二进制解释： 1100 0011
<<	左移动运算符	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ，二进制解释：0000 1111

6.5 逻辑运算符(bool)

a = 10, b = 20

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与"	如果 x 为 False，x and y 返回 False，

		否则它返回 y 的计算值 (a and b) 返回 20
or x or y	布尔"或"	如果 x 是 True，它返回 x 的值， 否则它返回 y 的计算值 (a or b) 返回 10
not not x	布尔"非"	如果 x 为 True，返回 False 。 如果 x 为 False，它返回 True not(a and b) 返回 False

6.6 成员运算符

除了以上的一些运算符之外，Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False	x 在 y 序列中,如果 x 在 y 序列中返回 True
not in	如果在指定的序列中没有找到值返回 True，否则返回 False	x 不在 y 序列中,如果 x 不在 y 序列中返回 True

6.7 身份运算符

身份运算符用于比较两个对象的存储单元

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 类似 id(x) == id(y) , 如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y ， 类似 id(a)!=id(b)。如果引用的不是同一个对象则返回结果 True，否则返回 False。

6.8 Python 运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符

&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
and or not	逻辑运算符

7. 字符串（String）

7.1 字符串创建

```
s = 'abcd'

s = "abcd"

s = """
    abcdefg
    """
```

7.2 字符串访问

```
a = s[0]      # 访问弟 0 个元素

l = len(s)        # 字符串的长度
```

7.3 字符串运算

a = “Hello”, b = “Python”

操作符	描述	实例
+	字符串连接	a + b 输出结果：HelloPython
*	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 e
[:]	截取字符串中的一部分，遵循左闭右开原则	str[0,2] 是不包含第 3 个字符的
in	成员运算符	'H' in a 输出结果 True
not in	成员运算符	'M' not in a 输出结果 True
r/R	原始字符串	print(r'a\nb') # a\nb
%	格式字符串	a=10;print(" 今 年 我%s 岁"%a) # 今年我 10 岁

7.4 Python 字符串格式化

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

```
print ("我叫%s 今年%d 岁!" %('小明', 10))          # 我叫小明今年 10 岁!
```

符号	描述
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f 和%e 的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

7.5 内建函数

方法 描述

capitalize() 将字符串的第一个字符转换为大写

endswith(suffix,beg=0,end=len(string)) 检查字符串是否以 obj 结束

expandtabs(tabsize=8) 把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8 。

find(str, beg=0 end=len(string)) 检测 str 是否包含在字符串中,如果包含返回开始的索引值，否则返回-1

index(str, beg=0, end=len(string)) 跟 find()方法一样，只不过如果 str 不在字符串中会报一个异常.

isalnum() 如果字符串至少有一个字符并且所有字符都是字母或数字则返 回 True,否则返回 False

isdigit() 如果字符串只包含数字则返回 True 否则返回 False..

isnumeric() 如果字符串中只包含数字字符，则返回 True，否则返回 False

isspace() 如果字符串中只包含空白，则返回 True，否则返回 False.

join(seq) 以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串

len(string)　　返回字符串长度

lower()　　转换字符串中所有大写字符为小写.

lstrip()　　截掉字符串左边的空格或指定字符。

max(str)　　返回字符串 str 中最大的字母。

min(str)　　返回字符串 str 中最小的字母。

replace(old, new [, max])　　把 将字符串中的 str1 替换成 str2,如果 max 指定，则替换不超过 max 次。

rfind(str, beg=0,end=len(string)) 类似于 find()函数，不过是从右边开始查找.

rindex(str, beg=0, end=len(string))类似于 index()，不过是从右边开始.

rstrip()　　删除字符串字符串末尾的空格

split(str="",num=string.count(str))　num=string.count(str))以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num 个子字符串

splitlines([keepends])　　按照行('\r', '\r\n', \n')分隔

startswith(str,beg=0,end=len(string))检查字符串是否是以 obj 开头

strip([chars]) 在字符串上执行 lstrip()和 rstrip()

upper() 转换字符串中的小写字母为大写

8. 列表

8.1 列表创建

```
list = [1,2,3,4,5,'atguigu']
[x+1 for x in range(10)]  或  [x+1 for x in (1,1,2,3)]
```

8.2 列表值获取

```
l  = list[0]
list[-1]
l1 = list2[1:5]      [1,5)
len(list)      #长度查看
```

8.3 列表更新

```
list[0] = 'agg'
```

8.4 删除列表元素

```
del list[0]
```

8.5 其它列表操作

表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!']*4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print(x, end=" ")	1 2 3	迭代

8.6 列表嵌套

使用嵌套列表即在列表里创建其它列表，例如：

```
a = ['a', 'b', 'c']

n = [1, 2, 3]

x = [a, n]

# x = [['a', 'b', 'c'], [1, 2, 3]]

# x[0] = ['a', 'b', 'c']

# x[0][1] = 'b'
```

8.7 列表函数&方法

函数	描述
len(list)	列表元素个数
max(list)	返回列表元素最大值

min(list)　　返回列表元素最小值

list(seq)　　将元组转换为列表

list.append(obj)　　在列表末尾添加新的对象

list.count(obj)　　统计某个元素在列表中出现的次数

list.extend(seq)　　在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

list.index(obj)　　从列表中找出某个值**第一个匹配**项的索引位置

list.insert(index, obj)　　将对象插入列表

list.pop([index=-1])　　移除列表中的一个元素（默认最后一个元素），并且返回该元素的值

list.remove(obj)　　移除列表中某个值的**第一个**匹配项

list.reverse()　　反向列表中元素

list.sort(cmp=None, key=None, reverse=False) 对原列表进行排序

list.clear()　　清空列表

list.copy()　　复制列表

9. 元组

9.1 元组创建

```
tup1 = ('Google', 'atguigu', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";     # 不需要括号也可以
```

9.2 元组值获取

```
tup1[1]

tup1[1:5]
```

9.3 元组更新

元组不允许更新

9.4 删除元组元素

```
del tup[0]
```

9.5 元组运算符

表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在

for x in (1, 2, 3): print (x,) 1 2 3　　迭代(生成器)

9.6 元组内置函数

方法	描述	实例
len(tuple)	计算元组元素个数	len(tuple1)
max(tuple)	返回元组中元素最大值	max(tuple2)
min(tuple)	返回元组中元素最小值	min(tuple2)
tuple(seq)	将列表转换为元组	list1= ['Google', 'Taobao', 'Runoob', 'Baidu'] tuple1=tuple(list1) tuple1 = ('Google', 'Taobao', 'Runoob', 'Baidu')

10. 字典

在字典中，键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行。

10.1 字典创建

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
dict = {x:x+1 for x in range(10)}
```

10.2 字典值获取

```
dict['Alice']
```

10.3 更新字典

dict['Alice'] = 10

10.4 删除字典元素

del dict['Alice']

del dict

10.5 字典内置函数&方法

函数	描述	实例
len(dict)	计算字典元素个数，即键的总数。	dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} len(dict)
str(dict)	输出字典，以可打印的字符串表示。	dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} str(dict) "{'Name': 'Runoob', 'Class': 'First', 'Age': 7}"
type(variable)	返回输入的变量类型	dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} type(dict) <class 'dict'>

radiansdict.clear() 删除字典内所有元素

radiansdict.copy() 返回一个字典的浅复制

radiansdict.fromkeys() 创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值

radiansdict.get(key, default=None) 返回指定键的值，如果值不在字典中返回 default 值

key in dict 如果键在字典 dict 里返回 true，否则返回 false

radiansdict.items() 以列表返回可遍历的(键, 值) 元组数组

radiansdict.keys() 返回一个迭代器，可以使用 list() 来转换为列表

radiansdict.setdefault(key, default=None) 和 get()类似，但如果键不存在于字典中，将会添加键并将值设为 default

radiansdict.update(dict2) 把字典 dict2 的键/值对更新到 dict 里

radiansdict.values() 返回一个迭代器，可以使用 list() 来转换为列表

pop(key[,default]) 删除字典给定键 key 所对应的值，返回值为被删除的值。key 值必须给出。 否则，返回 default 值。

popitem() 随机返回并删除字典中的一对键和值(一般删除末尾对)。

扩展：

1. 浅拷贝：

只拷贝引用地址，未拷贝内容：

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

b = a
```

2. 深拷贝：

拷贝引用地址和内容：

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

import copy

b = copy.deepcopy(a)
```

可以递归拷贝；一拷到底

注意：

1、对于不可变类型 Number String Tuple,浅复制仅仅是地址指向，不会开辟新空间。

2、对于可变类型 List、Dictionary、Set，浅复制会开辟新的空间地址(仅仅是最顶层开辟了新的空间，里层的元素地址还是一样的)，进行浅拷贝

3、浅拷贝后，改变原始对象中为可变类型的元素的值，会同时影响拷贝对象的；改变原始对象中为不可变类型的元素的值，只有原始类型受影响。

11. Set 集合

集合不支持切片操作。

11.1 Set 集合创建

```
s = {'name','aa','bb'}
s = set(序列)        # dict 序列，值添加 key
s = {x for x in range(10) if x not in range(5,10)}
```

11.2 Set 集合添加元素

```
s.add(x)        #  添加单个元素
s.update(x)     #  添加序列元素
```

11.3 移除元素

```
s.remove(x)     #  移除单个元素
s.discard(x)    #  移除集合(不存在不报错)
s.pop()         #  随机删除集合中的一个元素
```

11.4 集合操作方法

方法	描述
len(s)	查看集合的长度
s.clear()	清空集合
x in s	判断元素是否在集合中
add()	为集合添加元素

clear() 移除集合中的所有元素

copy() 拷贝一个集合

difference() 返回多个集合的差集

difference_update() 移除集合中的元素，该元素在指定的集合也存在。

discard() 删除集合中指定的元素

intersection() 返回集合的交集

intersection_update() 删除集合中的元素，该元素在指定的集合中不存在。

isdisjoint() 判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。

issubset() 判断指定集合是否为该方法参数集合的子集。

issuperset() 判断该方法的参数集合是否为指定集合的子集

pop() 随机移除元素

remove() 移除指定元素

symmetric_difference() 返回两个集合中不重复的元素集合。

symmetric_difference_update() 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。

union()	返回两个集合的并集
update()	给集合添加元素

12. 条件判断

12.1 基本语法

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

12.2 注意要点

1、每个条件后面要使用冒号 :，表示接下来是满足条件后要执行的语句块。

2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。

3、在 Python 中没有 switch – case 语句。

13. 循环语句

13.1 while 循环

```
while bool：
```

```
    pass
else:
    pass
```

练习：1-100 求和；输出 9*9 乘法表

13.2 for 循环

```
for <variable> in <sequence>:
    <statements>
```

13.3 range()函数

如果你需要遍历数字序列，可以使用内置 range()函数。它会生成数列：

```
range(start,end,step)
```

13.4 break 和 continue 语句及循环中的 else 子句

(1) break 语句可以跳出 for 和 while 的循环体。

如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。

(2) continue 语句被用来告诉 Python 跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

13.5 pass 语句

Python pass 是空语句，是为了保持程序结构的完整性。pass 不做任何事情，一般用做占位语句。

14. 迭代器和生成器

迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

14.1 迭代器生成

字符串，列表或元组对象都可用于创建迭代器

```
list=[1,2,3,4]
it = iter(list)          # 创建迭代器对象
print(next(it))          # 输出迭代器的下一个元素
print(next(it))
```

14.2 迭代器遍历

```
list=[1,2,3,4]
it = iter(list)
```

(1) for 循环

```
for i in it:
    print(i)
```

(2) while 循环

```
import sys
while True:
    try:
        print(next(its))
    except StopIteration:
        sys.exit()
```

14.3 创建一个迭代器

把一个类作为一个迭代器使用需要在类中实现两个方法 __iter__() 与 __next__() 。

例如：

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self
    def __next__(self):
        if self.a < 20:
            x = self.a
            self.a += 1
```

```
            return x
        else:
            raise StopIteration
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
for x in myiter:
    print(x)
```

14.4 生成器

在 Python 中，使用了 yield 的函数被称为生成器(generator)。跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值, 并在下一次执行 next()方法时从当前位置继续运行。

调用一个生成器函数，返回的是一个迭代器对象。

```
import sys
def fibonacci(n):                    # 生成器函数 - 斐波那契
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
```

```
                return
            yield a
            a, b = b, a + b
            counter += 1
    f = fibonacci(10)                # f 是一个迭代器，由生成器返回
生成
    while True:
        try:
            print (next(f), end=" ")
        except StopIteration:
            sys.exit()
```

15. 函数

15.1 基本语法

```
def 函数名(参数列表):
    函数体
```

15.2 函数分类

1. 有参数

(1) 有几个参数，就得传入几个参数

(2) 在函数调用中输入参数时，参数名称必须对应

```
def aa(x):
```

```
    print(x)
aa(x=5)
```

(3) 当调用函数时，必须全部定义名称,且名称对应，顺序可以不同

```
def aa(x,y):
    print(x)
aa(y=2,x=5)
```

(4) 函数中可以定义默认值

```
def aa(x=30,y):
    print(x)
aa(y=2,x=5)
```

(5) 不定长度参数

```
def aa(x,y,*args,**kwargs):
    print(x)
    print(args)                # 元组
    print(kwargs)              # 字典
aa(1,2,3,4,5,6,7,a = 8,b=9)
```

2. 有返回值

(1) 单个返回值

```
def aa(x):
    return x
a = aa(10)
```

(2) 多个返回值

```
def aa(x):
    return x,10
a = aa(10)      # a 是一个元组
a,b = aa(10)    #  多个参数接收
```

15.3 匿名函数

基本语法：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

```
sum = lambda arg1, arg2: arg1 + arg2
#  调用 sum 函数
print ("相加后的值为 : ", sum( 10, 20 ))
print ("相加后的值为 : ", sum( 20, 20 ))
```

15.4 变量作用域

L （Local） 局部作用域

E （Enclosing） 闭包函数外的函数中

G （Global） 全局作用域

B （Built-in） 内建作用域

L –> E –> G –>B(在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内建中找)

```
x = int(2.9)                # 内建作用域
g_count = 0                 # 全局作用域
def outer():
    o_count = 1             # 闭包函数外的函数中
    def inner():
        i_count = 2         # 局部作用域
        o_count += 1
    inner()
outer()
```

15.5 全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。

```
total = 0 # 这是一个全局变量
a = None
# 可写函数说明
def sum( arg1, arg2 ):
```

```
    global a #
    #返回 2 个参数的和."
    total = arg1 + arg2                    # total 在这里是局部变量.
    print("函数内是局部变量 : ", total)
    return total
#调用 sum 函数
sum( 10, 20 )
print ("函数外是全局变量 : ", total)


print(sum(1,20))
print('a now= ',a)
```

16、文件的读写

16.1 文件写入

```
text = "This id my first test.\n This is next line\n"
# w:write   r :only read
my_file =open('my_file.txt','w')
```

```
my_file.write(text)
my_file.close()
```

16.2 文件追加

```
append_text = "This id my first test.\n This is next line\n"
# w:write   r :only read  a:append
my_file =open('my_file.txt','a')
my_file.write(append_text)
my_file.close()
```

16.3 读文件

```
file =open('my_file.txt','r')
content = file.read()
#   readline 逐行输出 和  readlines
print(content)
```

17、类

17.1 class

```
class Calculator:
    name = 'lin'
    # self
    def add(self,x,y):
        print(self.name)
        result = x + y
        print(result)
    def minus(self,x,y):
        result = x-y
        print(result)
    def times(self,x,y):
        result = x- y
        print(result)

calcul = Calculator()
calcul.name
calcul.add(10,11)
```

17.2 __init__

```
class Calculator:
    name = 'lin'
    price = 18
    # 在初始化 class 时，得传进__init__中的参数，或者先默认
    def __init__(self,name,price,high,width,weight):
        self.name = name
        self.price = price
        self.h = high
        self.wi =width
        self.we = weight
        self.add(1,2)
    # self
    def add(self,x,y):
        print(self.name)
        result = x + y
        print(result)
    def minus(self,x,y):
```

```
        result = x-y
        print(result)
    def times(self,x,y):
        result = x- y
        print(result)

calcul = Calculator()
calcul.name
calcul.add(10,11)
```

18、try

```
# try
try:
    file = open('eeee','r+')
except Exception as e:
    print(e)
else:
    file.write('ssss')
file.close()
```

19、zip lambda map 定义函数

```
a = [1,2,3]
b = [4,5,6]
zip(a,b)
list(zip(a,b))
list(zip(a,a,b))
# [(1,4),(2,5),(3,6)]
for i,j in zip(a,b):
    print(i/2,j*2)

#     def fun(x,y)
#           return(x+y)
fun =lambda x,y:x+y
# map
map(fun1,[1],[2])
list(map(fun1,[1],[2]))
list(map(fun1,[1,3],[2,5]))
```

20、copy 和 deepcopy

```
import copy
# a b 等价 改变其中一个，另外一个也变
a = [1,2,[2,3]]
b = a
# 浅复制  外边缘不会改变，但是内部会变
c = copy.copy(a)
a[2][0] = 111
# 这是就会出现 c 也变了，
# 但是 a[0] = 111 是不会让 c 变的

# deepcopy
e = copy.deepcopy(a)
```

21、pickle

```
import pickle

a_dict = {'da': 111, 2: [23,1,4], '23': {1:2,'d':'sad'}}

# pickle a variable to a file
file = open('pickle_example.pickle', 'wb')
pickle.dump(a_dict, file)
file.close()

# reload a file to a variable  with 自动打开和关闭
with open('pickle_example.pickle', 'rb') as file:
    a_dict1 =pickle.load(file)

print(a_dict1)
```

21、set 找不同

Set 最主要的功能就是寻找一个句子或者一个 list 当中不同的元素

```
char_list = ['a', 'b', 'c', 'c', 'd', 'd', 'd']

sentence = 'Welcome Back to This Tutorial'
```

```
print(set(char_list))
# {'b', 'd', 'a', 'c'}

print(set(sentence))
# {'l', 'm', 'a', 'c', 't', 'r', 's',
# ' ', 'o', 'W', 'T', 'B', 'i', 'e', 'u',
# 'h', 'k'}

print(set(char_list+ list(sentence)))
# {'l', 'm', 'a', 'c', 't', 'r', 's', ' ',
# 'd', 'o', 'W', 'T', 'B', 'i', 'e', 'k', 'h', 'u', 'b'}
```

```
# add
unique_char = set(char_list)
unique_char.add('x')
# unique_char.add(['y', 'z']) this is wrong
print(unique_char)
# {'x', 'b', 'd', 'c', 'a'}

# 清除
# 清除不存在的 remove 会报错，但是 discard 不会
unique_char.remove('x')  #print 为 null
print(unique_char)
# {'b', 'd', 'c', 'a'}

unique_char.discard('d')
print(unique_char)
# {'b', 'c', 'a'}

unique_char.clear()
print(unique_char)
# set()

# 交集或者并集
unique_char = set(char_list)
print(unique_char.difference({'a', 'e', 'i'}))
# {'b', 'd', 'c'}

print(unique_char.intersection({'a', 'e', 'i'}))
# {'a'}
```

五、python 多线程

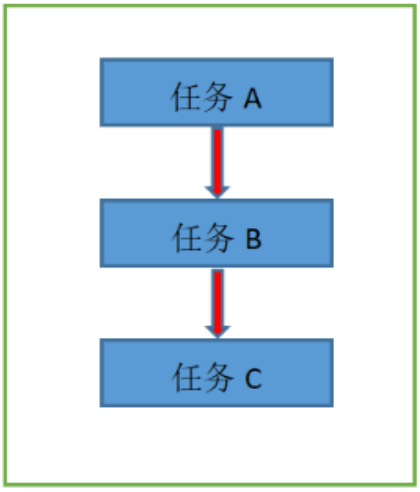
5.1 什么是进程？什么是线程？

电脑中时会有很多单独运行的程序，每个程序有一个独立的进程，而进程之间是相互独立存在的。比如下图中的 QQ、酷狗播放器、电脑管家等等。

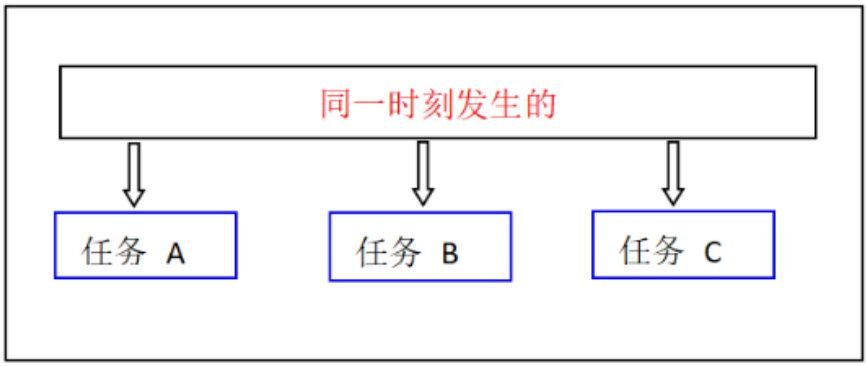
进程想要执行任务就需要依赖线程。换句话说，就是进程中的最小执行单位就是线程，并且一个进程中至少有一个线程。

那什么是多线程？提到多线程这里要说两个概念，就是串行和并行，搞清楚这个，我们才能更好地理解多线程。

所谓串行，其实是相对于单条线程来执行多个任务来说的，我们就拿下载文件来举个例子：当我们下载多个文件时，在串行中它是按照一定的顺序去进行下载的，也就是说，必须等下载完 A 之后才能开始下载 B，它们在时间上是不可能发生重叠的。



并行：下载多个文件，开启多条线程，多个文件同时进行下载，这里是严格意义上的，在同一时刻发生的，并行在时间上是重叠的



了解了这两个概念之后，我们再来说说什么是多线程。举个例子，我们打开腾讯管家，腾讯管家本身就是一个程序，也就是说它就是一个进程，它里面有很多的功能，我们可以看下图，能查杀病毒、清理垃圾、电脑加速等众多功能。

按照单线程来说，无论你想要清理垃圾、还是要病毒查杀，那么你必须先做完其中的一件事，才能做下一件事，这里面是有一个执行顺序的。

如果是多线程的话，我们其实在清理垃圾的时候，还可以进行查杀病毒、电脑加速等等其他的操作，这个是严格意义上的同一时刻发生的，没有执行上的先后顺序。

5.2 python 多线程实现

```
def thread_job():
    print("second thread")

# 看看有多少激活了的线程
def main():
    # 添加一个线程  ,  并执行
    added_thread = threading.Thread(target = thread_job)
    added_thread.start()
    print(threading.active_count())
    print(threading.enumerate())    #看看哪些线程
```

```
    print(threading.current_thread())

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

5.3 join 功能

```
import threading
import time

def thread_job():
    print("T1 start\n")
    for i in range(10):
        time.sleep(0.1)
    print("T1 finish\n")

# 看看有多少激活了的线程
def main():
    # 添加一个线程 ， 并执行
    added_thread = threading.Thread(target = thread_job,name='T1')
    added_thread.start()
    # 在没加上 join 之前 T1 start--> all done -->T1 finish
    # added_thread.join() 后面的命令得等待 join 之前的结束
    print("all done")

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

5.4 queue

```
from queue import Queue
# 代码实现功能，将数据列表中的数据传入，
# 使用四个线程处理，将结果保存在 Queue 中，
# 线程执行完后，从 Queue 中获取存储的结果
def job(l,q):
    for i in range(len(l)):
        l[i] = l[i]**2
    q.put(l)

def multithreading(date):
```

```
    q = Queue()
    threds = []
    data = [[1,2,3],[3,4,5],[4,4,4],[5,5,5]]
    for i in range(4):
        t = threading.Thread(target = job,args=(data[i],q))
        t.start()
        threds.append(t)
    for thread in threds:
        threds.join()
    results  = []
    for _ in range(4):
        results.append(q.get())
    print(results)
```

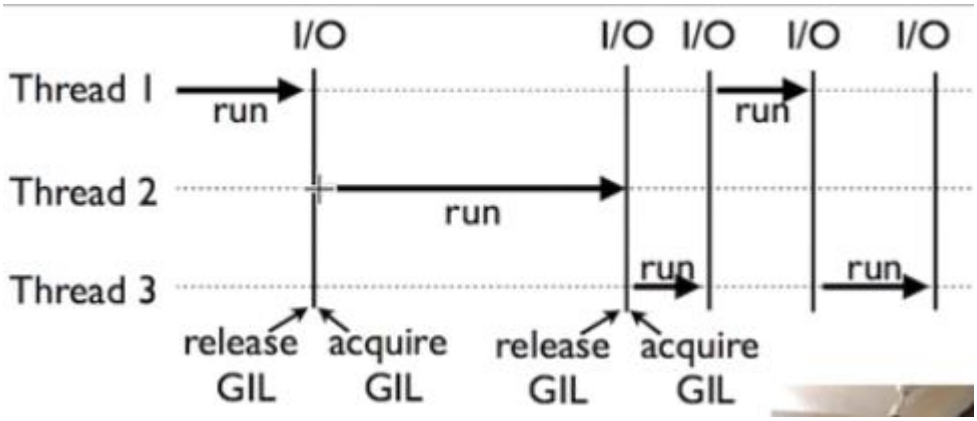
5.4 GIL

Python Threading 并不是特别理想，最主要的原因是，Python 的设计上，有一个必要的缓解，就是 Global Interpret Lock（GIL），这个东西让 Python 还是一次性智能处理一个东西。

尽管 Python 完全支持多线程编程，但是解释器的 C 语言实现部分在完全并行执行时并不是线程安全的。实际上，解释器被一个全局解释器锁保护着，它确保任何时候都只有一个 Python 线程执行。GIL 最大的问题就是 Python 的多线程程序并不能利用多核 CPU 的优势（比如一个 shi8yong 了多线程的计算密集型程序只会在一个单 CPU 上面运行）。

GIL 只会影响到那些严重依赖 CPU 的程序（比如计算型）。如果你的程序大部分只会涉及到 I/O，比如网络交互，那么使用多线程就很合适，因为它们大部分时间都在等待。实际上你完全可以放心的创建几千个 Python 线程，现代操作系统运行这么多线程完全没有压力。

多线程有效率，但是在 python 不一定，有一个 GIL，不停的进行切换。



python 多线程对 I/O 密集型任务有用，对计算密集型任务没用。对于像聊天的发送和接收用多线程是十分好的，但是对于计算进行分批多线程对效率并没有很大的提升。这个时候就要用到多核的运算，每一个 cpu 有自己的计算逻辑空间，不会受到 GIL 的影响。

5.5 线程锁 Lock

```
def job1():
    global A,lock
    lock.acquire()  #
    for i in range(10):
        A+=1
        print('job1',A)
    lock.release()

def job2():
    global A,lock
    lock.acquire()
    for i in range(10):
        A+=10
        print("job2",A)
    lock.release()
```

```
if __name__=='__main__':
    # 如果没有 lock，就会 job1 和 job2 打印的比较乱
    lock = threading.Lock()
    A = 0
    t1 = threading.Thread(target=job1)
    t2 = threading.Thread(target=job2)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

六、 多核运算 python Multiprocessing

和 threading 比较，多进程的 Multiprocessing 和多线程的 threading 类似，它们都是在 python 中并行运算的，不过既然有了 threading，为什么 Python 还要出一个 multiprocessing 呢，原因很简单，就是用来弥补 threading 的一些劣势，比如在 threading 中提及的 GIL。

使用 multiprocessing 也非常简单，可以充分的发挥计算机的多核系统的优势。

6.1 multiprocessing 和 thread 类似

```
import multiprocessing as mp
import threading as td

def job(a,b):
    print("aaa")
t1 = td.Thread(target = job,args=(1,2))
p1 = mp.Process(target=job,args=(1,2))
t1.start()
p1.start()
t1.join()
p1.join()
```

```
import multiprocessing as mp
import threading as td

def job(a,b):
    print("aaa")

if __name__ == '__main__':
    p1 = mp.Process(target=job,args=(1,2))
    p1.start()
    p1.join()
```

6.2 Queue

Queue 的功能是将每个核或线程的运算结果放在队里中， 等到每个线程或核运行完毕后再从队列中取出结果， 继续加载运算。原因很简单, 多线程调用的函数不能有返回值, 所以使用 Queue 存储多个线程运算的结果。

```
import multiprocessing as mp

def job(q):
    res = 0
    for i in range(1000):
        res += i +i**2
    q.put()
#     不能有 return，放置在一个 queue 中

if __name__ == '__main__':
    q = mp.Queue()
    # 不可以 args=(q),这样可能会报错，必须要加上一个,
    p1 = mp.Process(target=job,args=(q,))
    p2 = mp.Process(target=job,args=(q,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    res1 = q.get()
    res2 = q.get()
    print(res1+res2)
```

6.3 对比 threading

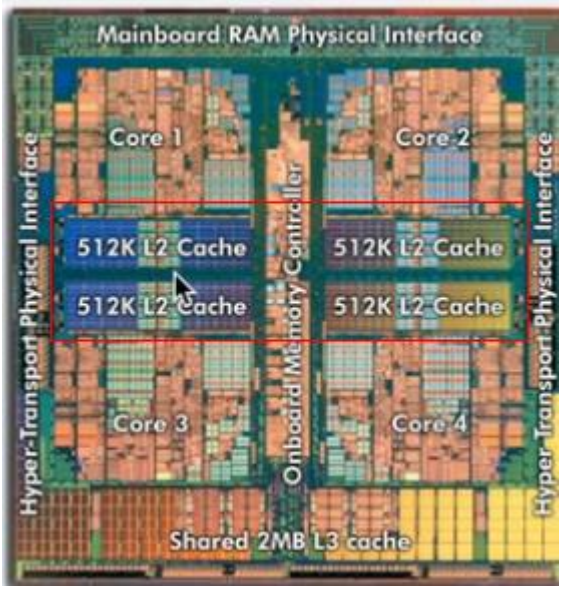
https://morvanzhou.github.io/tutorials/python-basic/multiprocessing/4-comparison/

6.4 进程池

进程池就是我们将所要运行的东西，放到池子里，Python 会自行解决多进程的问题

```
import multiprocessing as mp

def job(x):
    return x*x

def multicore():
    # method1  输出多个结果
    # 默认会将所有的核都用上，但是可以用 processes 定义个数
    pool = mp.Pool(processes=3)
    res = pool.map(job,range(10))
    print(res)
    # method2.1 输出一个结果
    res = pool.apply_async(job,(2,))
    print(res.get())
    # method2.2 加上迭代器
    multi_res = [pool.apply_async(job,(i,)) for i in range(10)]
    print([res.get() for res in multi_res])

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

6.5 共享内存




```
import multiprocessing as mp

# d 小数  i  int 整数   f float
value =mp.Value('d',1)
# 一维列表
array = mp.Array('d',[1,2,3])
```

6.5 进程锁

6.5.1 无进程锁

```
import multiprocessing as mp
import time

def job(v, num):
    for _ in range(5):
        time.sleep(0.1)  # 暂停 0.1 秒，让输出效果更明显
        v.value += num  # v.value 获取共享变量值
        print(v.value, end="")

def multicore():
    v = mp.Value('i', 0)  # 定义共享变量
    p1 = mp.Process(target=job, args=(v, 1))
    p2 = mp.Process(target=job, args=(v, 3))  # 设定不同的 number 看如何抢夺内存
    p1.start()
    p2.start()
    p1.join()
```

```
    p2.join()


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

6.5.2 加上进程锁

```
import multiprocessing as mp
import time

def job(v, num,l):
    l.acquire()
    for _ in range(5):
        time.sleep(0.1)  # 暂停 0.1 秒，让输出效果更明显
        v.value += num  # v.value 获取共享变量值
        print(v.value, end="")
    l.release()

def multicore():
    # 避免抢夺
    l = mp.Lock()
    v = mp.Value('i', 0)  # 定义共享变量
    p1 = mp.Process(target=job, args=(v, 1,l))
    p2 = mp.Process(target=job, args=(v, 3,l))  # 设定不同的 number 看
如何抢夺内存
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

七、Tkinter GUI 界面化

Tkinter 是使用 python 进行窗口视窗设计的模块。Tkinter 模块("Tk 接口")是 Python 的标准 Tk GUI 工具包的接口。作为 python 特定的 GUI 界面，是一个图像的窗口，tkinter 是 python 自带的，可以编

辑的 GUI 界面，我们可以用 GUI 实现很多直观的功能，比如想开发一个计算器，如果只是一个程序输入，输出窗口的话，是没用用户体验的。所有开发一个图像化的小窗口，就是必要的。

对于稍有 GUI 编程经验的人来说，Python 的 Tkinter 界面库是非常简单的。python 的 GUI 库非常多，选择 Tkinter，一是最为简单，二是自带库，不需下载安装，随时使用，三则是从需求出发，Python 作为一种脚本语言，一种胶水语言，一般不会用它来开发复杂的桌面应用，它并不具备这方面的优势，使用 Python，可以把它作为一个灵活的工具，而不是作为主要开发语言，那么在工作中，需要制作一个小工具，肯定是需要有界面的，不仅自己用，也能分享别人使用，在这种需求下，Tkinter 是足够胜任的！

对于 Tkinter 编程，可以用两个比喻来理解：

第一个，作画。我们都见过美术生写生的情景，先支一个画架，放上画板，蒙上画布，构思内容，用铅笔画草图，组织结构和比例，调色板调色，最后画笔勾勒。相应的，对应到 tkinter 编程，那么我们的显示屏就是支起来的画架，根窗体就是画板，在 tkinter 中则是 Toplevel，画布就是 tkinter 中的容器（Frame），画板上可以放很多张画布（Convas），tkinter 中的容器中也可以放很多个容器，绘画中的构图布局则是 tkinter 中的**布局管理器**（几何管理器），绘画的内容就是 tkinter 中的一个个小组件，一幅画由许多元素构成，而我们的 GUI 界面，就是有一个个组件拼装起来的，它们就是 widget。

第二个，我们小时候都玩过**积木**，只要发挥创意，相同的积木可

以堆出各种造型。tkinter 的组件也可以看做一个个积木，形状或许不同，其本质都是一样的，就是一个积木，不管它长什么样子，它始终就是积木！所以这些小组件都有许多共性，另外，个人认为，学习界面编程，最重要的不是一开始学习每个积木的样子，不是学习每个组件怎么用，而是这些组件该怎么放。初始学习中，怎么放远远比怎么用重要的多。网上有大量的文章资料，基本全是介绍组件怎么用的，对于怎么放，也就是 tkinter 中的布局管理器，都是一笔带过，这对初学者有点本末倒置，或许绝大部分是转载的原因吧，极少是自己真正写的。组件怎么用不是最迫切的，用到的时候再去了解也不迟，边用边学反而更好。因此我将专门写一章，详细介绍布局管理器的使用。

7.1 Tkinter 模块元素简要说明

tkinter 类	元素	简要说明
Button	按钮	类似标签,但提供额外的功能，点击时执行一个动作,例如鼠标掠过、按下、释放以及键盘操作/事件
Canvas	画布	提供绘图功能(直线、椭圆、多边形、矩形) 可以包含图形或位图
Checkbutton	复选框	允许用户选择或反选一个选项，一组方框,可以选择其中的任意个(类似 HTML 中的 checkbox)
Entry	单行文本框	单行文字域，显示一行文本，用来收集键盘输入(类似 HTML 中的 text)
Frame	框架	用来承载放置其他GUI元素，就是一个容器
Label	标签	用于显示不可编辑的文本或图标
LabelFrame	容器控件	是一个简单的容器控件。常用与复杂的窗口布局
Listbox	列表框	一个选项列表,用户可以从中选择
Menu	菜单	点下菜单按钮后弹出的一个选项列表,用户可以从中选择
Menubutton	菜单按钮	用来包含菜单的组件(有下拉式、层叠式等等)
Message	消息框	类似于标签,但可以显示多行文本
OptionMenu	选择菜单	下拉菜单的一个改版，弥补了Listbox无法下拉列表框的遗憾
PanedWindow	窗口布局管理	是一个窗口布局管理的插件，可以包含一个或者多个子控件。
Radiobuttion	单选框	允许用户从多个选项中选取一个，一组按钮,其中只有一个可被“按下” (类似 HTML 中的 radio)
Scale	进度条	线性“滑块”组件,可设定起始值和结束值,会显示当前位置的精确值
Scrollbar	滚动条	对其支持的组件(文本域、画布、列表框、文本框)提供滚动功能
Spinbox	输入控件	与Entry类似，但是可以指定输入范围值
Text	多行文本框	多行文字区域，显示多行文本，可用来收集(或显示)用户输入的文字(类似 HTML 中的 textarea)
Toplevel	顶层	类似框架,为其他的控件提供单独的容器
messageBox	消息框	用于显示你应用程序的消息框。(Python2中为tkMessagebox)

7.2 常用窗口部件及简要说明：

Tkinter 支持 16 个核心的窗口部件，这个 16 个核心窗口部件类简要描述如下：

Button：一个简单的按钮，用来执行一个命令或别的操作。

Canvas：组织图形。这个部件可以用来绘制图表和图，创建图形编辑器，实现定制窗口部件。

Checkbutton：代表一个变量，它有两个不同的值。点击这个按钮

将会在这两个值间切换。

Entry：文本输入域。

Frame：一个容器窗口部件。帧可以有边框和背景，当创建一个应用程序或 dialog(对话）版面时，帧被用来组织其它的窗口部件。

Label：显示一个文本或图象。

Listbox：显示供选方案的一个列表。listbox 能够被配置来得到 radiobutton 或 checklist 的行为。

Menu：菜单条。用来实现下拉和弹出式菜单。

Menubutton：菜单按钮。用来实现下拉式菜单。

Message：显示一文本。类似 label 窗口部件，但是能够自动地调整文本到给定的宽度或比率。

Radiobutton：代表一个变量，它可以有多个值中的一个。点击它将为这个变量设置值，并且清除与这同一变量相关的其它 radiobutton。

Scale：允许你通过滑块来设置一数字值。

Scrollbar：为配合使用 canvas, entry, listbox, and text 窗口部件的标准滚动条。

Text：格式化文本显示。允许你用不同的样式和属性来显示和编辑文本。同时支持内嵌图象和窗口。

Toplevel：一个容器窗口部件，作为一个单独的、最上面的窗口显示。

messageBox：消息框，用于显示你应用程序的消息框。(Python2

中为 tkMessagebox)

注意在 Tkinter 中窗口部件类没有分级；所有的窗口部件类在树中都是兄弟关系。

所有这些窗口部件提供了 Misc 和几何管理方法、配置管理方法和部件自己定义的另外的方法。此外，Toplevel 类也提供窗口管理接口。这意味一个典型的窗口部件类提供了大约 150 种方法。

https://blog.csdn.net/ahilll/article/details/81531587

八、Python 正则表达式

正则表达式（Regular Expression）又被称为 RegEx，是用来匹配字符的一种工具，在一大串字符中虚招你需要的内容，它常被用在很多方面，比如网页爬虫、文档整理、数据筛选等等。

在网络爬虫方面，用一种简单的匹配方法，一次性选取出成千上万网页的信息。正则表达式的内容非常多。

8.1 简单的 python 匹配

```
# 简单的 Python 匹配
pattern1 = "bird"
pattern2 = "dog"
string = "dog runs to cat"
```

```
print(pattern1 in string)
print(pattern2 in string)
```

8.2 用正则寻找匹配

```
import re
pattern1 = "bird"
pattern2 = "dog"
string = "dog runs to cat"
print(re.search(pattern1,string))
print(re.search(pattern2,string))
```

8.3 灵活匹配

```
# 灵活匹配
ptn = r'r[au]n'
print(re.search(ptn,"dog runs to cat"))
# <_sre.SRE_Match object; span=(4, 7), match='run'>

print(re.search(r"r[A-Z]n","dog runs to cat"))
# None
print(re.search(r"r[a-z]n","dog runs to cat"))
print(re.search(r'r[0-9]n',"dog runs to cat"))
print(re.search(r"r[0-9a-z]n", "dog runs to cat"))
```

8.4 按类型匹配

除了自己定义的规则外，还有很多匹配的规则是提前就已经定义行的了

\d:任何数字

\D:不是数字

\s:任何 white space 包括[\t \n \r \f \v]

\S：任何不是 white space

\w:任何大小写字母，数字和”” [a-z] [A-Z] [0-9]

\W:不是\w

\b:空白字符（只在某个字的开头或结尾）

\B:空白字符（不在某个字的开图或结尾）

\\:匹配\

.:匹配任何字符（除了\n）

^:匹配开头

$:匹配结尾

？:前面的字符可有可无

```
# \d : decimal digit
print(re.search(r"r\dn", "run r4n"))          # <_sre.SRE_Match object;
span=(4, 7), match='r4n'>
# \D : any non-decimal digit
print(re.search(r"r\Dn", "run r4n"))          # <_sre.SRE_Match object;
span=(0, 3), match='run'>
# \s : any white space [\t\n\r\f\v]
print(re.search(r"r\sn", "r\nn r4n"))         # <_sre.SRE_Match object;
span=(0, 3), match='r\nn'>
# \S : opposite to \s, any non-white space
print(re.search(r"r\Sn", "r\nn r4n"))         # <_sre.SRE_Match object;
span=(4, 7), match='r4n'>
# \w : [a-zA-Z0-9_]
print(re.search(r"r\wn", "r\nn r4n"))         # <_sre.SRE_Match object;
span=(4, 7), match='r4n'>
# \W : opposite to \w
print(re.search(r"r\Wn", "r\nn r4n"))         # <_sre.SRE_Match object;
span=(0, 3), match='r\nn'>
# \b : empty string (only at the start or end of the word)
print(re.search(r"\bruns\b", "dog runs to cat"))    # <_sre.SRE_Match
object; span=(4, 8), match='runs'>
# \B : empty string (but not at the start or end of a word)
print(re.search(r"\B runs \B", "dog   runs  to cat"))  #
<_sre.SRE_Match object; span=(8, 14), match=' runs '>
# \\ : match \
print(re.search(r"runs\\", "runs\ to me"))     # <_sre.SRE_Match object;
```

```
span=(0, 5), match='runs\\'>
# . : match anything (except \n)
print(re.search(r"r.n", "r[ns to me"))          # <_sre.SRE_Match object;
span=(0, 3), match='r[n'>
# ^ : match line beginning
print(re.search(r"^dog", "dog runs to cat"))    # <_sre.SRE_Match object;
span=(0, 3), match='dog'>
# $ : match line ending
print(re.search(r"cat$", "dog runs to cat"))    # <_sre.SRE_Match object;
span=(12, 15), match='cat'>
# ? : may or may not occur
print(re.search(r"Mon(day)?", "Monday"))        # <_sre.SRE_Match
object; span=(0, 6), match='Monday'>
print(re.search(r"Mon(day)?", "Mon"))           # <_sre.SRE_Match
object; span=(0, 3), match='Mon'>
```

8.5 换行匹配

```
string = """
dog runs to cat.
I run to dog.
"""
# 换行了也可以匹配
print(re.search(r"^I", string))                 # None
print(re.search(r"^I", string, flags=re.M))
```

8.6 重复匹配

- `*`：重复零次或多次
- `+`：重复一次或多次
- `{n,m}`：重复 n 至 m 次
- `{n}`：重复 n 次

```
# * : occur 0 or more times
print(re.search(r"ab*", "a"))            # <_sre.SRE_Match object;
```

```
span=(0, 1), match='a'>
print(re.search(r"ab*", "abbbbb"))        # <_sre.SRE_Match object;
span=(0, 6), match='abbbbb'>

# + : occur 1 or more times
print(re.search(r"ab+", "a"))             # None
print(re.search(r"ab+", "abbbbb"))        # <_sre.SRE_Match object;
span=(0, 6), match='abbbbb'>

# {n, m} : occur n to m times
print(re.search(r"ab{2,10}", "a"))        # None
print(re.search(r"ab{2,10}", "abbbbb"))   # <_sre.SRE_Match object;
span=(0, 6), match='abbbbb'>
```

8.7 分组匹配

```
match = re.search(r"(\d+), Date: (.+)", "ID: 021523, Date: Feb/12/2017")
print(match.group())
# 021523, Date: Feb/12/2017
print(match.group(1))                    # 021523
print(match.group(2))
 # Date: Feb/12/2017

match = re.search(r"(?P<id>\d+), Date: (?P<date>.+)", "ID: 021523, Date:
Feb/12/2017")
print(match.group('id'))                  # 021523
print(match.group('date'))                # Date: Feb/12/2017
```

8.8 findall

```
# findall
print(re.findall(r"r[ua]n", "run ran ren"))    # ['run', 'ran']
```

8.9 |

```
# | : or
print(re.findall(r"(run|ran)", "run ran ren")) # ['run', 'ran']
```

8.10 replace

```
print(re.sub(r"r[au]ns", "catches", "dog runs to cat"))     # dog
catches to cat
```

8.11 split

```
print(re.split(r"[,;\.]", "a;b,c.d;e"))              # ['a', 'b', 'c',
'd', 'e']
```

8.12 compile

```
compiled_re = re.compile(r"r[ua]n")
print(compiled_re.search("dog ran to cat"))  # <_sre.SRE_Match object;
span=(4, 7), match='ran'>
```

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c
数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	
边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc
逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(abc){2} a(123|456)c	abcabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用'|'或后接数量词。	(?:abc){2}	abcabc
(?iLmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc1 abcabc

九、爬虫

学习爬虫，首先要懂的是网页．支撑起各种光鲜亮丽的网页的不是别的，全都是一些代码．这种代码我们称之为 HTML，HTML 是一种浏览器(Chrome，Safari，IE，Firefox 等)看得懂的语言，浏览器能将这种语言转换成我们用肉眼看到的网页．所以 HTML 里面必定存在着很多规律，我们的爬虫就能按照这样的规律来爬取你需要的信息．

其实除了 HTML，一同构建多彩/多功能网页的组件还有 CSS 和 JavaScript．但是这个简单的爬虫教程，大部分时间会将会使用 HTML．CSS 和 JavaScript 会在后期简单介绍一下．因为爬网页的时候多多少少还是要和 CSS JavaScript 打交道的．

8.0 中文文档

https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/

8.1 BeautifulSoup 解析网页

流程:

选着要爬的网址 (url)

使用 python 登录上这个网址 (urlopen 等)

读取网页信息 (read() 出来)

将读取的信息放入 BeautifulSoup

使用 BeautifulSoup 选取 tag 信息等 (代替正则表达式)

安装:

```
# Python 2+

pip install beautifulsoup4


# Python 3+

pip3 install beautifulsoup4
```

8.1.1 简单应用

```
from bs4 import BeautifulSoup
from urllib.request import urlopen

# if has Chinese, apply decode()
html =
urlopen("https://morvanzhou.github.io/static/scraping/basic-structure
.html").read().decode('utf-8')
print(html)
soup = BeautifulSoup(html, features='lxml')
print(soup.h1)
# <h1>爬虫测试 1</h1>

print('\n', soup.p)
# <p>
#  这是一个在 <a href="https://morvanzhou.github.io/">莫烦 Python</a>
# <a href="https://morvanzhou.github.io/tutorials/scraping">爬虫教程
</a> 中的简单测试.
# </p>
# <a href="https://morvanzhou.github.io/tutorials/scraping">爬虫教程
</a>

all_href = soup.find_all('a')
all_href = [l['href'] for l in all_href]
print('\n', all_href)
```

```
# ['https://morvanzhou.github.io/',
'https://morvanzhou.github.io/tutorials/scraping']
```

8.1.2 根据 class

```
soup = BeautifulSoup(html, features='lxml')

# use class to narrow search
month = soup.find_all('li', {"class": "month"})
for m in month:
    print(m.get_text())

"""
一月
二月
三月
四月
五月
"""

jan = soup.find('ul', {"class": 'jan'})
d_jan = jan.find_all('li')                # use jan as a parent
for d in d_jan:
    print(d.get_text())

"""
一月一号
一月二号
一月三号
"""
```

8.1.3 加上正则

```
soup = BeautifulSoup(html, features='lxml')

img_links = soup.find_all("img", {"src": re.compile('.*?\.jpg')})
```

```
for link in img_links:
    print(link['src'])

course_links = soup.find_all('a', {'href':
re.compile('https://morvan.*')})
for link in course_links:
    print(link['href'])
```

8.2 Requests

8.2.1 获取页面的方式 post get

详情见 Restful API

8.2.2 get

```
import requests
import webbrowser
param = {"wd":"莫烦 Python"}  #搜索的信息
r = requests.get('http://www/baidu.com/s',params=param)
print(r.url)
webbrowser.open(r.url)
```

8.2.3 post

```
data = {'firstname': '莫烦', 'lastname': '周'}
r = requests.post('http://pythonscraping.com/files/processing.php',
data=data)
print(r.text)

# Hello there, 莫烦 周!
```

8.2.3 cookie 登录

```
payload = {'username': 'Morvan', 'password': 'password'}
r =
requests.post('http://pythonscraping.com/pages/cookies/welcome.php',
data=payload)
```

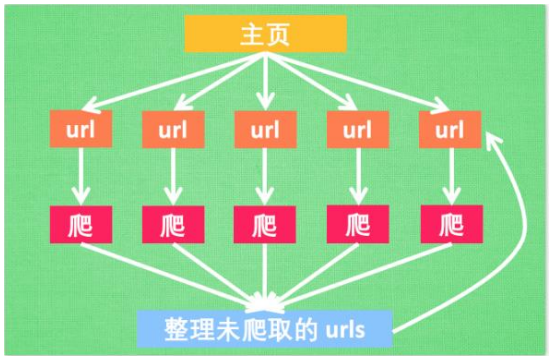
```
print(r.cookies.get_dict())

# {'username': 'Morvan', 'loggedin': '1'}


r = requests.get('http://pythonscraping.com/pages/cookies/profile.php',
cookies=r.cookies)
print(r.text)

# Hey Morvan! Looks like you're still logged into the site!
```

8.2.4 session 登录

```
Session = requests.Session()
payload = {'username': 'Morvan', 'password': 'password'}
r = session.post('http://pythonscraping.com/pages/cookies/welcome.php',
data=payload)
print(r.cookies.get_dict())

# {'username': 'Morvan', 'loggedin': '1'}


r = session.get("http://pythonscraping.com/pages/cookies/profile.php")
print(r.text)

# Hey Morvan! Looks like you're still logged into the site!
```

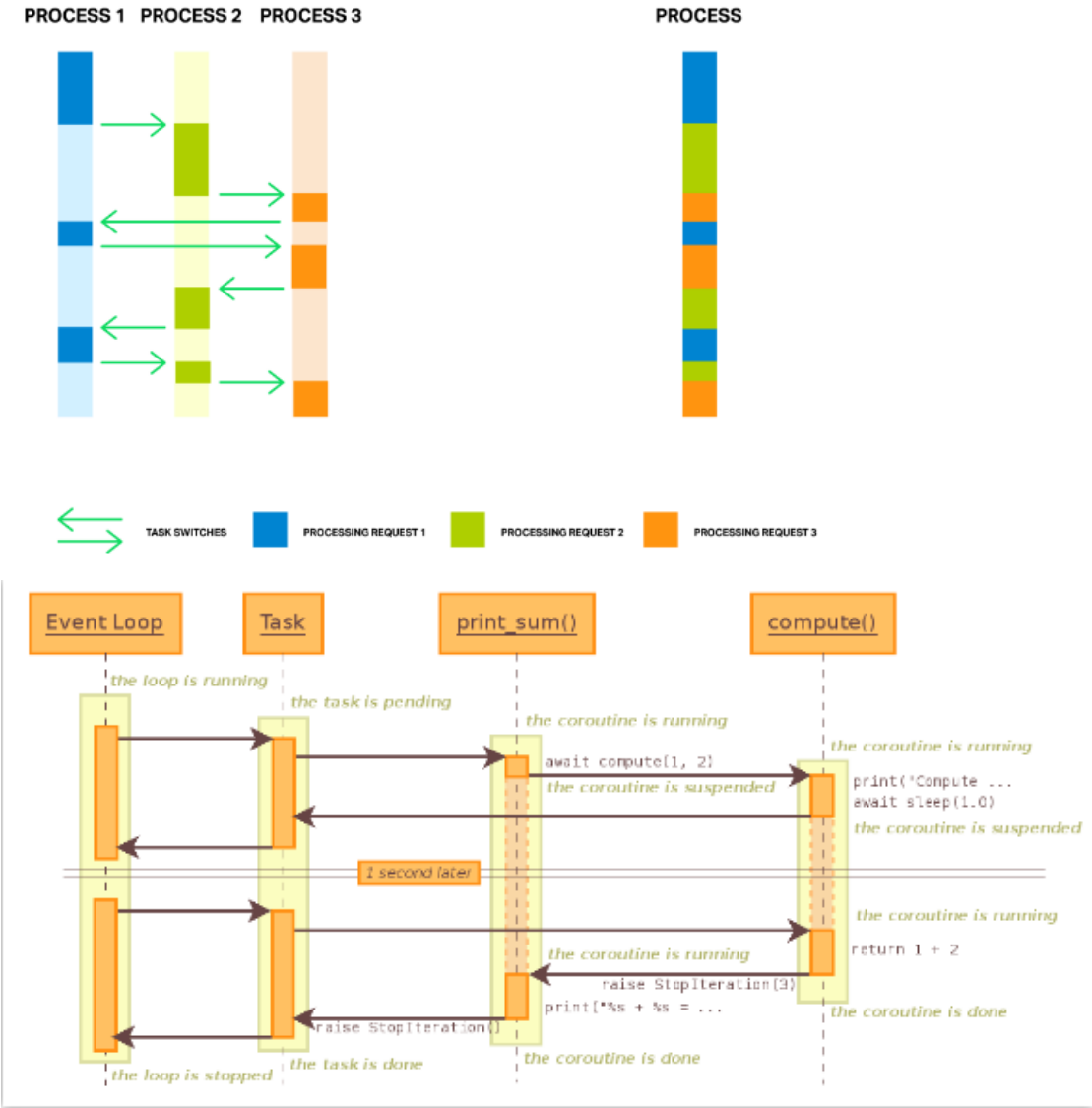
8.3 分布式爬虫

我们想搭建一个爬虫，就是同时下载，同时分析的这一类型的分布式爬虫，运用多线程进行爬取



8.4 异步加载 Asyncio

8.4.1 是单线程



8.4.2 基本原理代码：

```
import asyncio

async def job(t):                    # async 形式的功能
    print('Start job ', t)
    await asyncio.sleep(t)           # 等待 "t" 秒，期间切换其他任务
    print('Job ', t, ' takes ', t, ' s')

async def main(loop):                          # async 形式的功能
    tasks = [
    loop.create_task(job(t)) for t in range(1, 3)
    ]                                          # 创建任务，但是不执行
    await asyncio.wait(tasks)                  # 执行并等待所有任务完成

t1 = time.time()
loop = asyncio.get_event_loop()                # 建立 loop
loop.run_until_complete(main(loop))            # 执行 loop
loop.close()                                   # 关闭 loop
print("Async total time : ", time.time() - t1)

"""
Start job  1
Start job  2
Job  1  takes  1  s
Job  2  takes  2  s
Async total time :  2.001495838165283
"""
```

8.4.3 aiohttp

8.5 Selenium+火狐 Katalon Recorder 控制浏览器

geckdriver 安装：

https://blog.csdn.net/only_Tokimeki/article/details/81390083

8.6 Scrapy 爬虫库

https://www.jianshu.com/p/a8aad3bf4dc4
https://blog.csdn.net/u012150179/article/details/32343635

十、RestAPI

RESTful是什么

本质

一种软件架构风格

核心

面向资源

解决的问题

- 降低开发的复杂性
- 提高系统的可伸缩性

RESTful是什么

设计概念和准则

- 网络上的所有事物都可以被抽象为资源。
- 每一个资源都有唯一的资源标识，对资源的操作不会改变这些标识。
- 所有的操作都是无状态的。

HTTP协议-URL

HTTP是一个属于应用层的协议，特点是简捷、快速。

schema://host[:port]/path [?query-string][#anchor]

- scheme　　指定低层使用的协议(例如：http, https, ftp)
- host　　服务器的IP地址或者域名

HTTP协议-URL

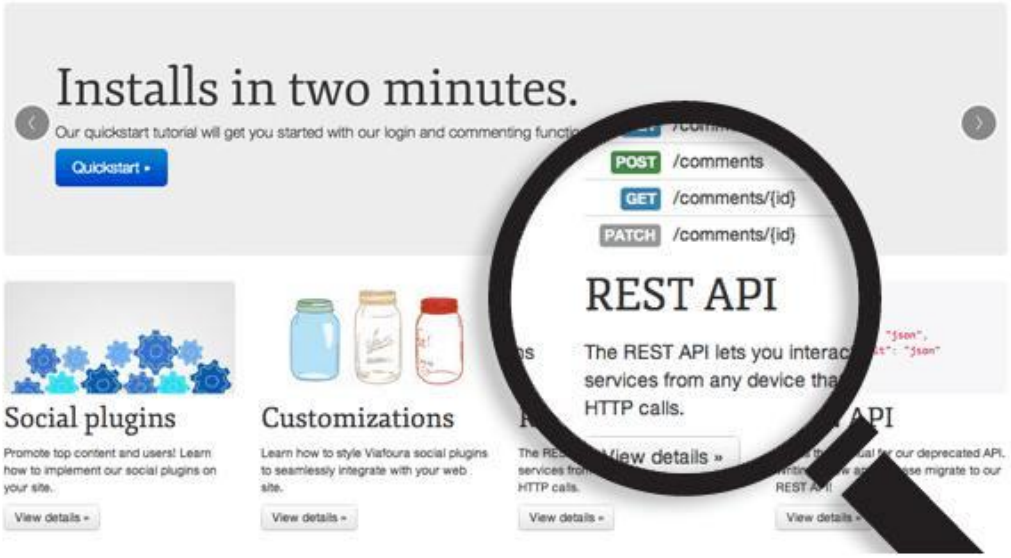
- port　　服务器端口，默认为80
- path　　访问资源的路径
- query-string　　发送给http服务器的数据
- anchor　　锚

http　80　　https 443

网络应用程序，分为前端和后端两个部分。当前的发展趋势，就是前端设备层出不穷（手机、平板、桌面电脑、其他专用设备.....）。

因此，必须有一种统一的机制，方便不同的前端设备与后端进行通信。这导致 API 构架的流行，甚至出现"API First"的设计思想。RESTful API 是目前比较成熟的一套互联网应用程序的 API 设计理论。我以前写过一篇《理解 RESTful 架构》，探讨如何理解这个概念。

今天，我将介绍 RESTful API 的设计细节，探讨如何设计一套合理、好用的 API。我的主要参考了两篇文章（1，2）。



一、协议

API 与用户的通信协议，总是使用 HTTPs 协议。

二、域名

应该尽量将 API 部署在专用域名之下。

```
https://api.example.com
```

如果确定 API 很简单，不会有进一步扩展，可以考虑放在主域名下。

```
https://example.org/api/
```

三、版本（Versioning）

应该将 API 的版本号放入 URL。

```
https://api.example.com/v1/
```

另一种做法是，将版本号放在 HTTP 头信息中，但不如放入 URL 方便

和直观。Github 采用这种做法。

四、路径（Endpoint）

路径又称"终点"（endpoint），表示 API 的具体网址。

在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以 API 中的名词也应该使用复数。

举例来说，有一个 API 提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

- `https://api.example.com/v1/zoos`
- `https://api.example.com/v1/animals`
- `https://api.example.com/v1/employees`

五、HTTP 动词

对于资源的具体操作类型，由 HTTP 动词（谓词）表示。

常用的 HTTP 动词有下面五个（括号里是对应的 SQL 命令）。

- `GET（SELECT）：从服务器取出资源（一项或多项）。`
- `POST（CREATE）：在服务器新建一个资源。`

- PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。
- PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。
- DELETE（DELETE）：从服务器删除资源。

还有两个不常用的 HTTP 动词。

- HEAD：获取资源的元数据。
- OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

- GET /zoos：列出所有动物园
- POST /zoos：新建一个动物园
- GET /zoos/ID：获取某个指定动物园的信息
- PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
- PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
- DELETE /zoos/ID：删除某个动物园
- GET /zoos/ID/animals：列出某个指定动物园的所有动物

- `DELETE /zoos/ID/animals/ID`：删除某个指定动物园的指定动物

六、过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API 应该提供参数，过滤返回结果。

下面是一些常见的参数。

- `?limit=10`：指定返回记录的数量
- `?offset=10`：指定返回记录的开始位置。
- `?page=2&per_page=100`：指定第几页，以及每页的记录数。
- `?sortby=name&order=asc`：指定返回结果按照哪个属性排序，以及排序顺序。
- `?animal_type_id=1`：指定筛选条件

参数的设计允许存在冗余，即允许 API 路径和 URL 参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

七、状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的 HTTP 动词）。

- 200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
- 201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
- 202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
- 204 NO CONTENT - [DELETE]：用户删除数据成功。
- 400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
- 401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
- 403 Forbidden - [*] 表示用户得到授权（与 401 错误相对），但是访问是被禁止的。
- 404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

- `406 Not Acceptable - [GET]`：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
- `410 Gone -[GET]`：用户请求的资源被永久删除，且不会再得到的。
- `422 Unprocesable entity - [POST/PUT/PATCH]` 当创建一个对象时，发生一个验证错误。
- `500 INTERNAL SERVER ERROR - [*]`：服务器发生错误，用户将无法判断发出的请求是否成功。

八、错误处理（Error handling）

如果状态码是4xx或者5xx，就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

```
{
    error: "Invalid API key"
}
```

九、返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

- `GET /collection`：返回资源对象的列表（数组）

- `GET /collection/resource`：返回单个资源对象
- `POST /collection`：返回新生成的资源对象
- `PUT /collection/resource`：返回完整的资源对象
- `PATCH /collection/resource`：返回完整的资源对象
- `DELETE /collection/resource`：返回一个空文档

十、Hypermedia API

RESTful API 最好做到 Hypermedia，即返回结果中提供链接，连向其他 API 方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向 api.example.com 的根目录发出请求，会得到这样一个文档。

```
{"link": {
  "rel":   "collection https://www.example.com/zoos",
  "href":  "https://api.example.com/zoos",
  "title": "List of zoos",
  "type":  "application/vnd.yourformat+json"}}
```

上面代码表示，文档中有一个 link 属性，用户读取这个属性就知道下一步该调用什么 API 了。rel 表示这个 API 与当前网址的关系（collection 关系，并给出该 collection 的网址），href 表示 API 的路径，title 表示 API 的标题，type 表示返回类型。

Hypermedia API 的设计被称为 HATEOAS。Github 的 API 就是这种设计，访问 api.github.com 会得到一个所有可用 API 的网址列表。

```
{
  "current_user_url":
"https://api.github.com/user",
  "authorizations_url":
"https://api.github.com/authorizations",
  // ...}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问 api.github.com/user，然后就得到了下面结果。

```
{
  "message": "Requires authentication",
  "documentation_url":
"https://developer.github.com/v3"}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

十一、其他

（1）API 的身份认证应该使用 OAuth 2.0 框架。

（2）服务器返回的数据格式，应该尽量使用 JSON，避免使用 XML。

（完）

实战：


项目需求
• 用户登录、注册
• 文章发表、编辑、管理、列表
确认设计要素
资源路径
/users、/articles
HTTP动词
GET、POST、DELETE、PUT
过滤信息
文章的分页筛选
状态码
200、404、422、403

确认设计要素

错误处理

输出JSON格式错误信息

返回结果

输出JSON数组或JSON对象

数据库设计

用户表

ID、用户名、密码、注册时间

文章表

文章ID、标题、内容、发表时间、用户ID