环境配置简述

官网下载：python官网，点击download，记得勾选PATH环境变量的配置。检查是否安装完成直接进入cmd窗口（win+r，键入cmd回车）键入py或者python，若成功安装则会弹出python的版本号等信息。
接下来自行安装图形化界面pycharm。

Python编程规范准则

在进入Python学习前，首先简略查看Python编程规范准则,打开Python解释器（或者打开网址：Python之禅）；输入import this,就能看到PEP（Python改进建议书）中的"The Zen of Python"（Python之禅）下面是经过翻译后的“蛇宗三字经”：

> Beautiful is better than ugly.

> 美胜丑

> Explicit is better than implicit.

> 明胜暗

> Simple is better than complex.

> 简胜复

> Complex is better than complicated.

> 复胜杂

> Flat is better than nested.

> 浅胜深

> Sparse is better than dense.

> 疏胜密

> Readability counts.

> 辞达意

> Special cases aren’t special enough to break the rules.

> 不逾矩

> Although practicality beats purity.

> 弃至清（实用性优先）

> Errors should never pass silently.

> 无阴差

> Unless explicitly silenced.

> 有阳错

> In the face of ambiguity, refuse the temptation to guess.

> 拒疑数

> There should be one– and preferably only one –obvious way to do it.

> 求完一

> Although that way may not be obvious at first unless you’re Dutch.

> 虽不至，向往之

> Now is better than never.

> 敏于行

> Although never is often better than right now.

> 戒莽撞

> If the implementation is hard to explain, it’s a bad idea.

> 差难言

> If the implementation is easy to explain, it may be a good idea.

> 好易说

> Namespaces are one honking great idea – let’s do more of those!

> 每师出，多有名

前置

编译类语言（如C/C++,C#等）是是由源程序（以.py结尾的文本文件），经过编译器（compiler）为机器语言或者汇编语言（其中也会经过语法分析，预处理，性能优化），再经过链接器（linker）才能生成可执行程序（以.exe结尾）
Python这类解释类语言不同，它采用解释器，按行读取一行代码就执行一行代码。另外，Python一般指的是CPython，即底层是C语言编写的，不做说明，Python就指CPython，除此之外还有Jython，PyPy等版本。

安装管理Python扩展包（库）

Python3.4后续的版本均自带pip和setuptools库，pip可用于安装管理Python扩展包，setuptools用于发布Python包。 使用时打开终端（cmd窗口）：

```
下载扩展包：pip install 库
卸载扩展包：pip uninstall 库
```

```
查看已安装的扩展包：pip list
更新扩展包：pip install -u 库
查看过时（需要更新的扩展包）：pip list --outdated
```

集成开发环境

Python在下载时自带了一个集成开发环境IDLE，可以编写并执行程序。但一般使用Pycharm来进行集成开发。

程序的打包与发布

在编写完一个Python源文件后，以.py结尾。要变成可执行文件.exe，需要提前安装第三方扩展包PyInstaller，将.py变为.exe。
PyInstaller将源文件生成可执行文件的方式如下：

```
#在终端中：
pyinstaller -F source file.py
```

上述的-F指生成单个可执行文件，若需要所有支撑文件和子目录则可替换为-D。

字面量

类似于C/C++中的常量，python中的数据类型较C++中要多出数字类型的复数类型（complex），无双进度浮点型（double），只有浮点数（float），存在列表（list），元组（tuple），集合（set），字典（dictionary）(也即C++STL中的unordered_map，即无序键值对集合)
另外，字面量在Python中是有它的地址的，用变量去接收字面量时实际上在底层是经过引用的。

注释

单行注释用#，多行注释为三个英文双引号包裹"""。

```
"""
这是一段注释，
这是第二段注释
"""
#这是一行注释
print("hello,世界！")
```

变量

变量定义与C++不同，由于python是一门动态语言，所以无需定义变量的数据类型，直接为变量名=初值
字面量可以用id()函数来查看其唯一标识符，这个标识符表示字面量或者变量所引用的字面量的内存地址（虚拟内存）,而不是其真实的物理地址。

数据类型

python可以使用type(变量)来查看变量类型。返回值为一个类型对象，可通过print来打印显示。

```
index=None
print(type(index))
```

#类型转换 py可以使用数据类型(变量/字面量)来进行类型转换。但是py不支持隐式转换，即不能完成：

```
a=123
b="111"
a+b#报错
```

值得注意的是：py并不能进行类似于C++中将字符转换为对应的Ascll的强制类型转换，如：(int)'A'，这在py中是非法的，要完成这样的操作，对应函数为：

```
#将字符转换为Ascii码
ord('A')
#将Ascii码转换为字符
chr(65)
```

另外，浮点数转换为int会截断。

命名规范

在python中支持以中文来定义变量名，另外py建议使用下划线命名法。

运算符

与C++不同点为：额外定义//为取整除，相当于C++将除法结果类型转换为int，/为普通除法，pow()函数简化为**

```
num=40
num//=20
print(num)
num**=10
print(num)
```

字符串相关

字符串定义

py支持单引号定义，双引号定义，三引号定义。

字符串拼接

用+号可以拼接字符。

字符串格式化

先看一个场景样例：要输出某人的电话号码。

```
str_1="18283626869"
print("小明的电话号码是：%s" % str_1 )
```

这里%s表示(字符串类型)占位符，含义是对应表示有变量要来占位，*多变量的占位符，变量要用括号括起来，按照占位顺序填入*。

```
str_1="18283626869"
str_2="hu"
print("小明的序号是：%s%s"%(str_1,str_2) )
```

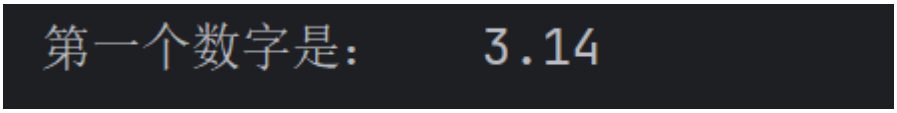
占位符还有%d(整型),%s(字符串),%f(浮点型)

格式化精度控制

这一节和C的printf很类似，如下演示：

```
print("第一个数字是：%7.2f"%3.1415926)
```

m.n其中m代表宽度限制，n表示精度控制。



快速格式化

f"内容{变量}"

```
print(f"圆周率的近似值为：{pi}")
```

注意是在字符串前加上f

输入

使用`input(提示信息)`函数完成。注意：input获取到的值永远是str类型。 若要将多个值输入，可以调用input的split方法。

判断语句

if语句

格式为：

```
if 条件表达式:
    判断体
```

注意，这里py程序是靠缩进来判断判断体的范围的，与C++中使用{}来限定范围不同。

if-else语句

```
if 条件表达式:
    成立的流程
else:
    不成立的流程
```

如：

```
age=int(input("输入年龄: "))#input接受的都是字符串
if age>=18:
    print("you are old enough")
else:
    print("you are so young")
```

if-elif-else语句

与C++中的if-else if-else语句相似。

实例

提示：通过导入random库，调用方法（C++中的接口函数）random.randint(左区间，右区间)，就能随机给出闭区间的整数。

```
import random
guess_num=random.randint(1,10)
print(guess_num)
```

循环语句

while语句

```
while 条件:
    循环体
```

for循环

要注意，这里的的for循环和C++中的for循环不同。

```
for 临时变量 in 可迭代对象（字符串，元组，列表等）:
    循环体
```

如下:

```
str_1='name'#字符串是可迭代对象
for x in str_1:
    print(x)
    #程序功能是将字符串内的单个字符取出打印并换行
```

range语句

`range(num)`表示构建一个[0,num-1]的序列。
`range(num1,num2)`表示构建从[num1,num2-1]的序列 `range(nums1,nums2,step)`第三个参数表示步长。 类似于C++的:

```
for(int i=num1,i<num2,i+step){
    ...
}
```

借助range语句，就能实现类似与C++的for循环。

break语句和continue语句

和C++相似，不再赘述。

函数

```
def 函数名(参数):
    函数体
    return 返回值
```

杂项

1. 在Python解释器中"_"是一个特殊变量，用以表示上次运算的结果。
2. del语句可用来删除变量声明定义。
3. "is"可用来替换"=="
4. Python支持序列解包赋值：`a,b=1,3`，但要注意变量与序列元素个数相同。如果只需要解包部分值，那么可采用特殊变量"_"来占位：`_,c,d,_=1,2,3,4`

数据容器

list列表

定义语法如下方式：

```
字面量
[元素1,元素2...]
赋予给变量
变量=[元素1,元素2...]
定义空列表
变量=[]
变量=list()
```

值得说明的是，py中的list可以存放数据类型不同的元素。

```
 访问列表某一个元素
列表[下标]
```

正向的访问（指从索引0开始向后）和C++相同，但py还支持反向访问，从最后一个元素开始，记作-1，依次往前递减。

```
my_list = [1,2,3,4,5,6,7,8,9,10]
print(my_list[-1])#输出10
```

列表内置的方法

查询指定元素的下标

`列表.index()`
返回元素的下标，若列表中无指定元素，则抛出错误ValueError。

修改指定下标的元素值

`列表[下标]=值`

按位置插入值

`列表.insert(下标，值)`

追加元素

`列表.append(值)`
类似于C++中的`push_back()`。
`列表.extend(其余数据容器)`也能完成将其余容器内的元素进行追加。

删除元素（按照索引删除）

方式1：`del 列表[下标]`
方式2：`列表.pop(下标)`

删除元素（按值删除）

`列表.remove(值)`
这个方法遍历列表，将对应的第一个值删除。

清空

`列表.clear()`

统计某元素的出现次数

`列表.count(值)`

统计列表的长度

`len(列表)`

tuple元组

和C++的键值对元组不同。tuple可视作不可修改的list。
定义和list列表的方式近似，只需要将方括号改为小括号即可。 由于元组的不可修改性，元组支持的常见方法只有index(),count(),len()。

str字符串

支持下标访问（正向反向均可），字符串类型也不支持修改。但count(),index(),len()方法都是可以使用的。 字符串的独特方法为：

字符串替换

`字符串.replace(字符串1, 字符串2)`将字符串1替换为字符串2。

字符串分割

`字符串.split(按照哪一个字符进行分割)`返回值是一个列表对象。

```
str_1 = 'hello world nice to meet you'
print(str_1.split(" "))#按照空格进行分割
```

结果为：`['hello', 'world', 'nice', 'to', 'meet', 'you']`

规整化处理/格式化处理

方法1：`字符串.strip()`将字符串中的空格和换行符去除 方法2：`字符串.strip(需要去除的字符)`将字符串中的指定字符去除

序列与切片操作

序列：内容连续，有序，支持下标索引的数据容器。例如：列表，元组，字符串都是序列。
切片：从序列中去除子序列。
语法：`序列[起始下标:结束下标:步长]`，起始下标和结束下标不写表示从头到尾。步长1表示不跳步取，负数为反向取。默认为1。

```
nums=(0,1,2,3)
nums_1=nums[::2]#输出0 2
nums_2=nums[::-1]#输出3, 2, 1, 0相当于反转元组
```

集合set

无序（不维护插入顺序），去重的数据容器。 不支持下标索引，支持修改操作，是可迭代对象（支持for循环）。 使用大括号定义。

添加新元素（按值）

`集合.add(元素)`

移除元素

`集合.remove(元素)`

随机弹出元素

`集合.pop()`，随机返回一个元素并在集合中移除该元素。

清空

`集合.clear()`

取差集

返回一个新集合，而不修改集合1，2。
`集合1.difference(集合2)`

消除差集

`集合1.difference_update(集合2)`和上一条不同的是，这个方法会修改集合1。

取并集

`集合1.union(集合2)`

统计元素个数

`len(集合)`

dict字典、映射

字典是可迭代的对象（支持循环遍历）
字典定义和集合相似：

```
my_dict={key:value,key:value......}
```

py中的key不允许重复，如果重复，则后面的key会覆盖前面的键值对。不允许使用**索引**。类似于C++中的map。但注意，py中的字典不支持“典中典”的操作，即key不能为dict类型。除此之外，字典可以嵌套。
通过键来访问值的方法为：`变量=字典名[键]`

新增（更新）元素

`字典[key]=value`，若key不存在，则新增，否则修改对应的值。

删除元素

`字典.pop(key)`，删除指定键值对。

统计元素（键值对）个数

`len(字典)`

清空字典

`字典.clear()`

获取全部的key

`字典.keys ()`

总结

	列表	元组	字符串	集合	字典
元素数量	支持多个	支持多个	支持多个	支持多个	支持多个
元素类型	任意	任意	仅字符	任意	Key：Value Key：除字典外任意类型 Value：任意类型
下标索引	支持	支持	支持	不支持	不支持
重复元素	支持	支持	支持	不支持	不支持
可修改性	支持	不支持	不支持	支持	支持
数据有序	是	是	是	否	否
使用场景	可修改、可重复的一批数据记录场景	不可修改、可重复的一批数据记录场景	一串字符的记录场景	不可重复的数据记录场景	以Key检索Value的数据记录场景

数据容器的通用操作

- 均支持循环遍历
- 均支持len()
- 均支持max()，min()
- 均支持数据容器的类型转换函数：list()（注意，字符串将会被拆分为单个字符储存进列表，字典将会被抛弃所有value，仅留下key）,tuple()（和上面相似）,str()（字典进行转换时会保留键值对，set()（所有元素去重，字典仅保留key，所有顺序失去）。**注意，数据容器实际上不支持将其余数据容器转换为dict，即不可以使用dict()函数进行转换**
- 支持指定容器排序:`sorted(容器,[reverse=True/False])`，默认升序排列。注意排序结果是作为list对象的。

函数进阶

多返回值

py支持函数可以多返回值，使用序列解包的方式接收：

```
def return_many(num1,num2):
    return num1,num2
x,y=return_many(1,2)
```

传参调用方式

- 位置参数：

```
def fun(num1,num2):
    do something
fun(1,2)
```

- 关键字传参

```
def fun(num1,num2,num3):
    do something
fun(num2=1,num1=2,num3=3)
fun(1,2,num3=3)
```

调用时可以指定关键字参数进行赋值，可以无视顺序，也可与位置参数混用，但位置参数必须在关键字参数之前。

- 缺省参数
  也即C++的函数具有默认参数,注意，缺省参数必须统一放在最后的。

```
def fun(num1,num2=2):
    do something

def(1)"""函数第二个参数默认存在，不用再传入"""
```

- 不定长（可变）参数

```
"""位置传递"""
def fun(*args):
    print(args)
fun(1)
fun(1,2)
fun(1,2,3...)
```

注意这种方式下args是元组类型。参数个数不限。

```
def fun(**kwargs):
    print(kwargs)
fun(nums1="1",nums2="2"...)
```

这种方式强制要求使用关键字传参，所有的“键=值”以字典形式传入kwargs中。传入参数个数也不限。

匿名函数

函数作为参数传递

py支持函数作为参数传递，只是计算逻辑的传递，而非数据的传递，和C++相同，不再赘述。

lambda匿名函数

lambda匿名函数即无名函数，只可临时使用一次：

```
lambda 传入参数：函数体（只能是一行代码）
```

文件操作

文件的打开读写关闭操作

打开操作：`open(name,mode,encoding)`，三个参数依次为文件路径/当前py同一层级的文件名字符串，文件模式（只读r，写入w，追加a等），编码格式（一般用UTF-8）返回一个文件对象（可迭代）

> 注意，win和Linux关于路径表述不相同，若在win下直接复制文件路径，记得在路径引号之前添加r进行适配。

读取文件：`文件对象.read(读取字符数，不填表示全部读取)`,返回字符串。

> 注意，这里和C中的文件读取类似，当调用一次read方法后，指针会停留在上次读取的内容末尾，再次调用read()方法会沿着上次读到的地方继续读取。

另外还有下面几种方法：
`文件对象.readlines()`，结果读取全部行后作为列表储存。
`文件对象.readline()`，一次读取一行
关闭操作：`文件对象.close()`
为了防止忘记关闭文件。py提供一个方法：

```
with open(r"win文件的绝对路径","模式",encoding="UTF-8")as 文件对象名:
    对文件的一系列操作
关闭后的下一步
```

这样的方式就能在完成对文件的一系列操作后自动对文件进行关闭。
写入操作：当以w模式打开文件（不存在时自动创建）后，调用方法：`文件对象.write("内容")`，将写入内容存放在缓存区中，再调用方法：`文件对象.flush()`就能写入内容了。注意w模式是清空并写入。
文件的追加：以a模式打开文件，其余与w模式相同。

异常错误捕获

异常具有可传递性
类似C++的try-catch语句，在py的语法是：

```
try:
    可能报错的代码块
except:
    错误处理
```

这样的方式不管什么错误类型都将捕获并进入错误处理流程。
当然，py也能捕获指定异常：

```
try:
    可能报错的代码块
except NameError as Error_object:
    指定错误处理
```

- 捕获多个指定异常
  当捕获多个异常时，可以将异常名字作为元组放在except之后。如：

```
try:
    可能出错的代码块
except(NameError,ZeroError...)as Error_object:
    错误处理
```

- 捕获所有异常
  除了第一种捕获全部异常的语法，还有如下语法：

```
try:
    可能出错的代码
except Exception as Error_object:
    错误处理
```

除了上面的基本try-except语句，还可在其后补充else语句。
在try-except-else中，else语句块中的内容表示若代码无异常时需要运行的代码。
更进一步，py中还可添加finally语句来表示无论是否报错，都将执行该语句下的代码块。

```
try:
    可能出错的代码
except:
    错误处理
else:
    若不出现异常执行的语句
finally:
    不管是否异常都将执行的语句
```

一般来说，finally语句下放的语句都是资源关闭语句，如文件的close等。
此外，与C++中的`throw`语句相似，python也提供了`raise`语句来主动抛出异常。
基本语法如下：

```
raise [Exception [, args [, traceback]]]
其中，Exception指异常类型，可以是py内置的异常类（如ValueError等），也可以是用户自定义的
```

```
异常类（自定义继承自Exception的异常类）。
args指提示信息
traceback指异常追踪信息，一般不用。
```

```
def add(x: int):
    if type(x) == float:
        raise ValueError("请输入整数! ")
    return x + 10


print(add(10.1))#ValueError: 请输入整数!
```

断言

assert语句用于判断一个表达式，在表达式为False时，抛出AssertionError。

```
assert 表达式 [, 错误信息]
```

```
x=1
assert x==None,"发生错误"
print(x)#AssertionError: 发生错误
```

模块

模块简述

py中的模块类似于C++的库，实际上就是一个py文件，通过`[from 模块名] import 模块名/f方法 [as 模块别名]` 可以引用。
当调用不同模块的同名方法时，最后引入的模块同名方法会覆盖。因此不建议使用同名方法，就算要使用，也要设置别名，或者使用`模块.方法()`
注意：当使用import引入模块时，将会直接执行模块内容，若要在模块内测试，但不想在引入时自动执行模块内容，就应该使用`__main__`变量

__main__变量

py内置一个变量为`__main__`,将测试语句放在这个变量判断之下，就能完成测试语句但又不在import引入时执行。类似于C++中的#if-#endif语句。

```
在自定义的模块中:
if __name__ == '__main__':
    测试语句
```

简单来说，这行语句只有在当前脚本执行时才会执行，在作为模块导入其他文件是不会执行。

__all__变量

也是内置变量，主要用处是，当遇到`from xxx import*`时，只能导入`__all__`中的方法列表的方法。

```
__all__=["方法名列表"]
def 方法1():
    something
def 方法2():
    something
......
```

自定义Python包

Python包是管理自己的创建的模块，是一个文件夹，其中有名为：__init__ .py的文件用来标识这个文件夹是Python包。

实践专题——数据可视化

Json简述

Json是轻量级的文本数据交换格式,Json实际上就是有固定格式的字符串。可视作不同语言数据的中转站/翻译器。另外，py中的字典和列表都是符合json数据格式的。
在py中可以使用`import json`导入json模块，调用`json.dumps(符合json格式的数据)`来将符合json格式的数据转换成json数据。

```
import json
data=[{"name":"小红","age":16},{"name":"小蓝","age":18}]
将字典转换成json数据
data=json.dumps(data)
将json数据转为python数据
data=data.loads(data)
```

注意：由于编码问题（utf-8转换为unicode）：中文在转换为json时不会显示原本的字符串，那么可以在dumps()方法的第二个参数加上ensure_ascii=False,这样就能将中文正确显示了。

pyecharts模块

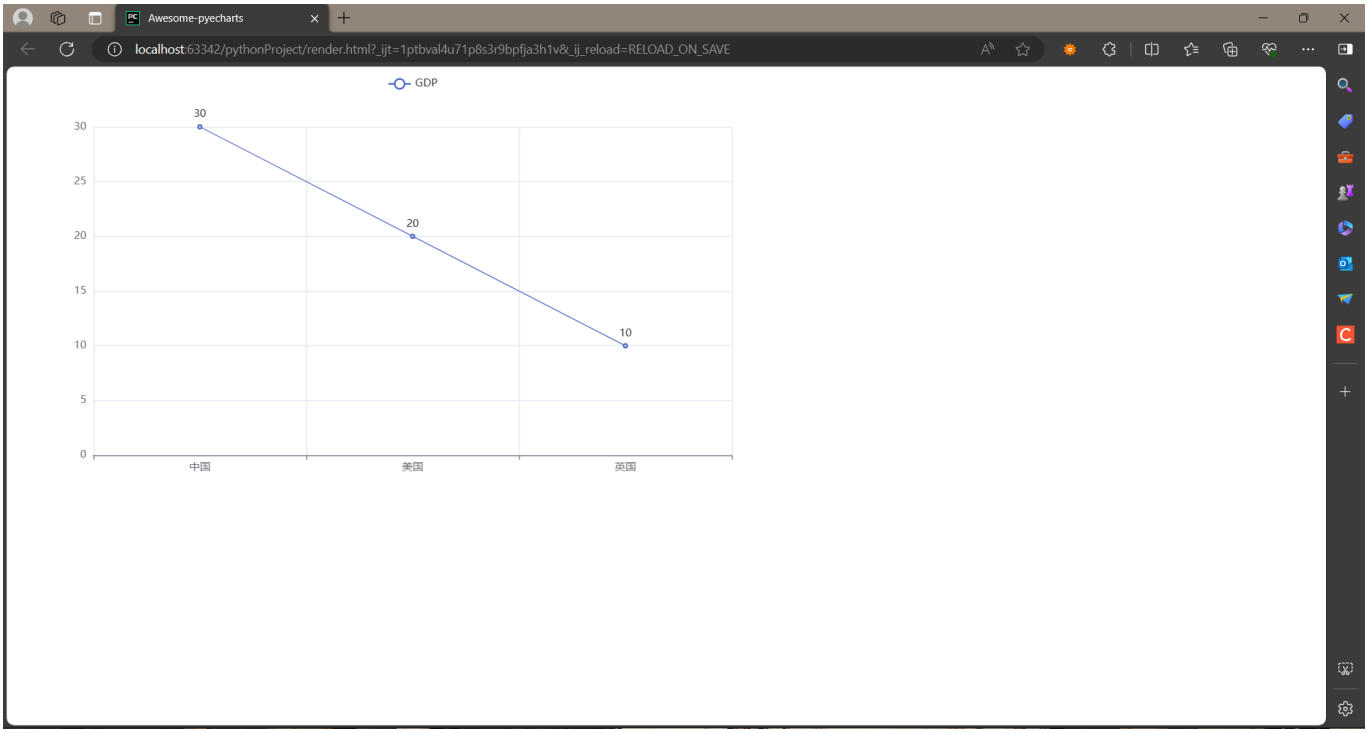
由百度开源的数据可视化。官网如下：pyecharts。其中可详细查看pyecharts的代码帮助。

一个简单的折线图

```
from pyecharts.charts import Line
line=Line()
```

```
line.add_xaxis(["中国","美国","英国"])
line.add_yaxis("GDP",[30,20,10])
line.render()
```

运行后会在当前目录中生成一个html文件，打开文件就能查看图表。

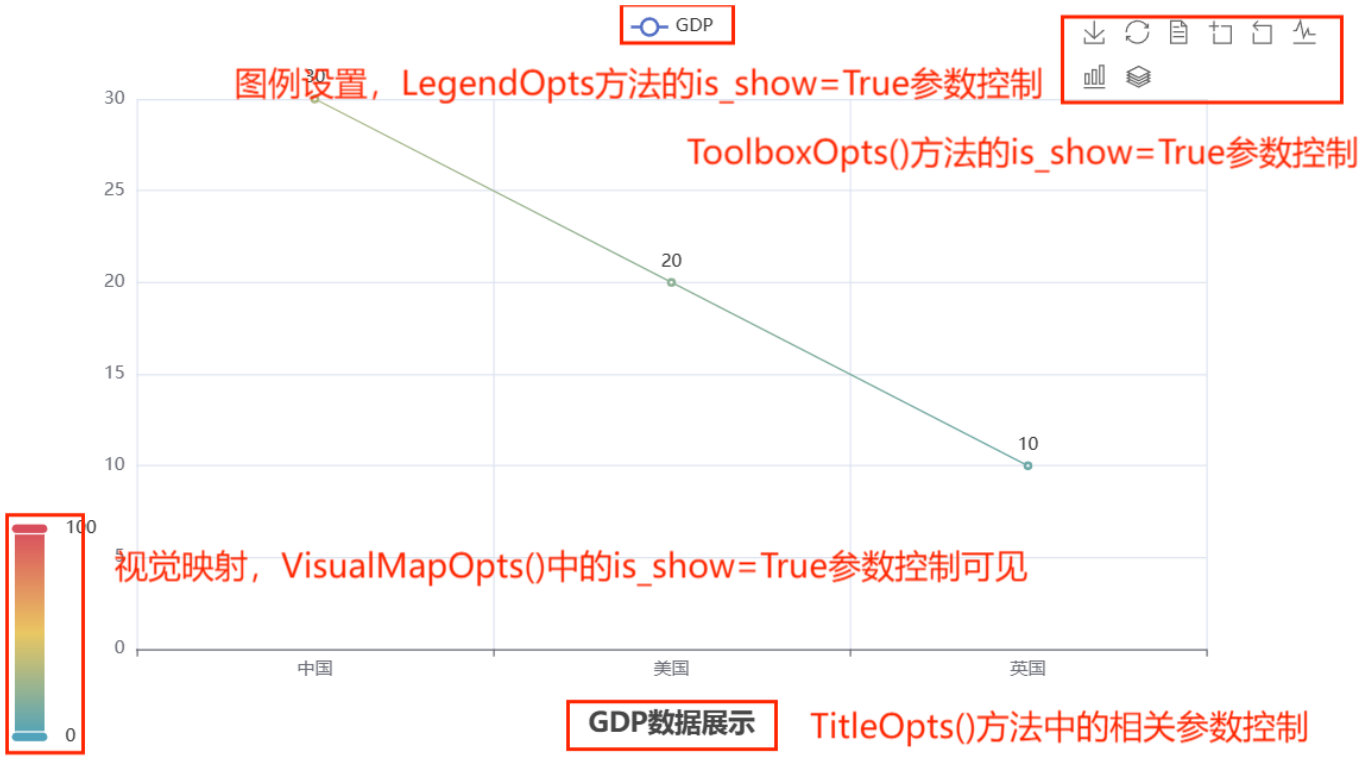


配置选项

pyecharts中提供了多样的配置选项（相当于图表样式）。分为全局配置选项（图表通用）和系统配置选项（部份表可用）。
下面介绍全局配置选项：

```
from pyecharts.charts import Line
from pyecharts.options import TitleOpts,LegendOpts,ToolboxOpts,VisualMapOpts#导入全局设置的部份方法
line=Line()
line.add_xaxis(["中国","美国","英国"])
line.add_yaxis("GDP",[30,20,10])
line.set_global_opts(
    title_opts=TitleOpts(title="GDP数据展示",pos_left="center",pos_bottom="1%"),#标题设置，post_left参数指定距左边距离，post_bottom参数指定距底部距离
    legend_opts=LegendOpts(is_show=True),#图例配置项
    toolbox_opts=ToolboxOpts(is_show=True),#工具箱
    visualmap_opts=VisualMapOpts(is_show=True)#视觉映射
                    )
line.render()
```

下面是上面涉及到的方法具体控制哪一个部件：



通过json模块对数据进行处理并生成折线图

首先对于标准的json数据，可以放到json在线解析中进行解析，（前提是标准格式的json字符串）。
打开准备好的折线图文件下的美国json数据文件。
去除不符合json数据格式的文件开头结尾。

```
import json
from pyecharts.charts import Line
from pyecharts.options import TitleOpts,ToolboxOpts,LegendOpts,VisualMapOpts
#打开读取文件
file_us=open(r"E:\desktop\资料\可视化案例数据\折线图数据\美国.txt","r",encoding="utf-8")
file_jp=open(r"E:\desktop\资料\可视化案例数据\折线图数据\日本.txt","r",encoding="utf-8")
file_in=open(r"E:\desktop\资料\可视化案例数据\折线图数据\印度.txt","r",encoding="utf-8")
data_us=file_us.read()
data_jp=file_jp.read()
data_in=file_in.read()

# 数据处理（规整为标准的json数据，去除不规范的开头和结尾）
data_us=data_us.replace("jsonp_1629344292311_69436(","")
data_jp=data_jp.replace("jsonp_1629350871167_29498(","")
data_in=data_in.replace("jsonp_1629350745930_63180(","")
data_us=data_us[:-2]
data_jp=data_jp[:-2]
data_in=data_in[:-2]

# 利用json.loads()方法转换为字典
us_dict=json.loads(data_us)
jp_dict=json.loads(data_jp)
```

```
in_dict=json.loads(data_in)

#寻找trend key
trend_us_data=us_dict["data"][0]["trend"]
trend_jp_data=jp_dict["data"][0]["trend"]
trend_in_data=in_dict["data"][0]["trend"]

#获取各自的x轴：日期（只取一年内的数据）
us_x_data=trend_us_data["updateDate"][:314]
jp_x_data=trend_jp_data["updateDate"][:314]
in_x_data=trend_in_data["updateDate"][:314]

#获取y轴数据：确诊人数，仅到314就结束
us_y_data=trend_us_data["list"][0]["data"][:314]
jp_y_data=trend_jp_data["list"][0]["data"][:314]
in_y_data=trend_in_data["list"][0]["data"][:314]

#创建折线对象
line=Line()
#添加x轴
line.add_xaxis(us_x_data)#无论哪个x轴的数据都行，都是同样的日期
#添加y轴
line.add_yaxis("美国确诊人数",us_y_data)
line.add_yaxis("日本确诊人数",jp_y_data)
line.add_yaxis("印度确诊人数",in_y_data)
line.set_global_opts(
    title_opts=TitleOpts(title="新冠确诊人数",pos_left="center",pos_bottom="1%"),
    toolbox_opts=ToolboxOpts(is_show=True),
    visualmap_opts=VisualMapOpts(is_show=True)
)

#关闭文件
line.render()
file_us.close()
file_jp.close()
file_in.close()
```

基础地图的绘制

在pyecharts中导入map，就可调用Map()方法来创建一个地图对象。

```
from pyecharts.charts import Map

ch_map=Map()

test_data=[
    ("北京市",99),
    ("四川省",66),
    ("台湾省",55),
    ("香港特别行政区",78)
]#将测试数据作为列表嵌套元组来储存
```

```
ch_map.add("测试例图",test_data,"china")#add()方法第一个参数表示地图名字，第二个参数表示导入的数据，第三个参数表示适配的哪个地图，默认是china。
ch_map.render()
```

当然，地图对象也能设置全局选项：

```
from pyecharts.charts import Map
from pyecharts.options import TitleOpts,ToolboxOpts,VisualMapOpts
ch_map=Map()

test_data=[
    ("北京市",1000),
    ("四川省",450),
    ("台湾省",10),
    ("香港特别行政区",78)
]
ch_map.add("测试例图",test_data,"china")
ch_map.set_global_opts(
    title_opts=TitleOpts(title="测试例图",pos_left="center",pos_bottom="1%"),
    toolbox_opts=ToolboxOpts(is_show=True),
    visualmap_opts=VisualMapOpts(is_show=True,
                                 is_piecewise=True,
                                 pieces=[
                                     {"min":1,"max":10,"label":"1-10","color":"#CCFFFF"},
                                         {"min":10,"max":100,"label":"10-100","color":"#FF666"},
                                         {"min":100,"max":1000,"label":"100-1000","color":"#990033"}
                                         ]
                                 )
)
ch_map.render()
```

这里主要是对参数方法Visualmap()的参数补充，is_show参数无需多说，后面的is_piecewise=True参数主要指可手动选择视觉映射指示器，即不同范围的数据在地图中呈现不同的色彩，pieces传入一个列表，其中嵌套的字典表示数据范围和使用的色彩，支持16进制和rgb(x,y,z)格式的色彩。

总结：

```
图例对象.set_global_opts(
    title_opts=TitleOpts(Title="标题",pos_left="center",pos_bottom="1%"),
    visualmap_opts=VisualMapOpts(
        is_show="True",
        is_piecewise="True",
        piece=[
            {"min":0,"max":10,"color":"#十六进制颜色或rgb色彩"},
            {"min":10,"max":100,"color":"同上"},
            {"min":100,"max":1000,"color":"同上"},
```

```
            ......
        ]

        )
        toolbox_opts=ToolboxOpts(is_show="True")
)
```

类与对象

这一部分和C++的知识相似，只是具体概念叫法不太同。
类的声明方法如下：

```
class 类名:
    类的属性(变量成员)

    类的行为(成员函数)
```

创建对象的的方式如下：对象=类名称()。值得注意的是，在类内部的函数一般称作方法，其定义方法与普通函数（类外部函数）不太相似，其首个参数是"self"，这在Python中是强制规定的，用以访问类中的变量成员。但在使用方法时，可不用写self参数，也即self参数不占用参数列表。其起到的作用类似于C++的this指针。在方法中若要访问类中的属性（变量成员），需要用到self.变量名的方式。

```
def fun(self,参数1, 参数2,参数3...):
    函数体
```

```
#示例
class Student:
    name=None
    student_id=None
    def test(self,name,student_id):
        self.name=name
        self.student_id=student_id
        print(f"Hello! I am {self.name},my student_id is {self.student_id}")
student_1=Student()
student_1.test("Alex",1)
```

构造方法

和C++的构造函数是一类东西，只不过用法不同。

```
#声明方法
__init__(self,参数1,参数2.....):
    self.参数1=参数1
```

```
    self.参数2=参数2
    ......
```

```
class Student:
    name=None
    student_id=None
    def __init__(self,name,student_id):
        self.name=name
        self.student_id=student_id
    def say_hi(self):
        print(f"Hello! I am {self.name}. My student_id is {self.student_id}")
student_1=Student("Alex",1)#类似于构造函数
student_1.say_hi()
```

与C++不同的是，实际上，__init__方法在被声明的时候就可进行创建成员变量。也即，成员变量的声明可以不写，当声明了__init__方法之后就能自动进行变量的声明和赋值。

魔术方法

即Python提供了一系列的内置方法来对类对象提供重载功能。

__str__方法

使用该方法后，就能使用print()语句直接输出类。

```
__str__(self):
    return f"这是一个类对象，{self.参数}，{self.参数}"
```

示例如下:

```
class Student:
    name=None
    student_id=None
    def __init__(self,name,student_id):
        self.name=name
        self.student_id=student_id
    def __str__(self):
        return f"类对象，姓名：{self.name}，学号：{self.student_id}"
    def say_hi(self):
        print(f"Hello! I am {self.name}. My student_id is {self.student_id}")
student_1=Student("Alex",1)
print(student_1)
print(str(student))
```

__it__方法

即可进行类之间的比较大小，类似于C++中的**大于小于**运算符重载。（__it__方法只能进行大于小于号的重载比较）

```
__it__(self,other):
    return self.参数>other.参数
```

示例：

```
class Student:
    name=None
    student_id=None
    def __init__(self,name,student_id):
        self.name=name
        self.student_id=student_id
    def __str__(self):
        return f"类对象，姓名：{self.name}，学号：{self.student_id}"
    def __it__(self, other):
        return self.student_id>other.teacher_id
    def say_hi(self):
        print(f"Hello! I am {self.name}. My student_id is {self.student_id}")
class Teacher:
    name=None
    teacher_id=None
    def __init__(self,name,teacher_id):
        self.name=name
        self.teacher_id=teacher_id
    def __str__(self):
        return f"类对象，姓名：{self.name}，职工号：{self.teacher_id}"
    def __it__(self, other):
        return self.teacher_id>other.Teacher_id
student_1=Student("Alex",1)
print(student_1)
teacher_1=Teacher("Lilei",1)
print(teacher_1)
print(student_1>teacher_1)#返回布尔值
```

__le__方法

和__it__方法相似，只不过重载的是>=和<=运算符。

__eq__方法

类似上面，重载了==运算符。

封装特性

即实现C++中private成员变量和函数。
python只需要在变量和方法名前添加__（两个下划线）即可声明私有成员和方法。

```
class private_class:
    def __init__(self):
        self.__id_private=None#使用__init__方法时就能声明一个私有属性
    def set_id_private(self):
        self.__id_private=32
        print(self.__id_private)
no_1=private_class()
no_1.set_id_private()
```

继承

继承写法如下：

```
class 子类名(父类名1,父类名2...):
    属性

    方法
```

注意，若子类仅是，多个父类的简单组合（不需要额外添加属性和方法）则可以在类内容体中写上pass关键字。

```
class Myclass(Myclass_1,Myclass_2):
    pass
```

- 涉及到多继承的同名成员
  当继承的多个父类中存在同名的成员，Python采用的策略是：先继承的保留，后继承的覆盖。也即，最左边的父类优先级更高。

复写与使用父类成员

复写指在子类中修改更新父类中的成员。（对同名成员修改）

```
class My_class:
    def test(self):
        print("hello")

class Him_class(My_class):
    def test(self):
        print("bye")
```

```
no_1=Him_class()
no_1.test()#输出bye
```

但是，若想要在子类中调用父类的同名对象，有以下两种方式：

1. 父类名.成员属性
   父类名.成员方法(self)

```
class My_class:
    id_no=None
    def test(self):
        print("hello")

class Him_class(My_class):
    id_no=123#复写
    My_class.id_no=None#调用父类成员
    def test(self):
        My_class.test(self)#调用父类方法
        print("bye")
no_1=Him_class()
no_1.test()
```

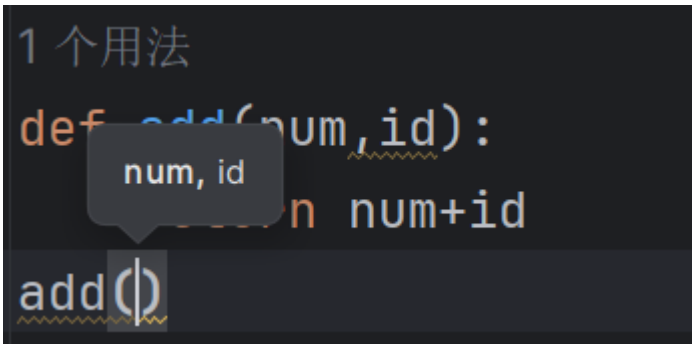
2. super().属性
   super().方法()（这里方法可以不用写self）

```
class My_class:
    id_no=None
    def test(self):
        print("hello")

class Him_class(My_class):
    id_no=123#复写
    def test(self):#复写
        super().test()#调用父类方法
        print("bye")
        print(self.id_no)
        print(super().id_no)#调用父类属性
no_1=Him_class()
no_1.test()
```

类型注解

由于py是一门动态类型的语言，当写了的自定义的函数后，调用函数并不能提示应该输入哪一种类型的数据参数，这样会造成很多麻烦。因此在Python3.5后加入了类型注解的功能。



（并未显示应该输入何种类型的数据）

声明类型注解的语法是：变量:数据类型，当然，也可以在注释中进行类型注解。例如：

```
class Person:
    pass
var_1=random.randint(1,100) # type: int
var_2="hello" # type: str
var_3=Person() # type: Person
var_4=fun()# type: int
var_5=json.loads(data_in) # type: dict[str,int]
var_6=list(range(10)) # type: List[int]
```

即注释如下：#type: 数据类型即可，这样在调用函数时，IDE会提示应该输入什么类型的数据。
而对于函数的形参进行类型注解，则需要在函数定义时进行，例如：

```
def add(a:int,b:int):
    pass
```

对于函数返回值注解也需要在函数定义时进行，例如：

```
def add(a:int,b:int)->int:
    return a+b
```

这样，在调用函数时，IDE会提示返回值的数据类型。

Union类型

Union类型是指多个数据类型组合在一起的类型，用于对混合类型的数据进行类型注解。基本语法如下：
Union[数据类型1,数据类型2...]例如：

```
from typing import Union
def add(a:Union[int,float],b:Union[int,float])->Union[int,float]:
    return a+b #函数参数a,b有可能是int或float类型，返回值也可能是int或float类型。
my_list[Union[int,str]]=[1,"hello"] #列表元素有可能是int或str类型。
```

这样，在调用函数时，IDE会提示输入的数据类型，并提示返回值的数据类型。

多态

多态是指不同子类对象对同一消息作出不同的响应。在Python中，多态是通过方法重写来实现的。例如：

```
class Animal:
    def run(self):
        pass#父类提供方法，但不实现具体功能，其功能由继承的子类具体自行实现

class Dog(Animal):
    def run(self):
        print("Dog is running")

class Cat(Animal):
    def run(self):
        print("Cat is running")

def run_animal(animal:Animal):
    animal.run()#类型建议传入Animal类型

dog=Dog()
cat=Cat()
run_animal(dog) #输出Dog is running
run_animal(cat) #输出Cat is running
#传入子类对象依旧可以调用父类的方法
```

在上面的代码中，Animal类是父类，Dog和Cat是子类。run_animal()函数接收Animal对象作为参数，并调用其run()方法。由于Dog和Cat都继承了Animal类，因此，run_animal()函数可以接收Dog和Cat对象，并调用其run()方法。这就是多态的实现。
对于父类这种，并不实现具体功能，而是提供一个接口的类的这种写法，就称为抽象类。抽象类不能实例化，只能作为父类被继承。其中的接口方法就称为抽象方法。抽象方法的具体实现由子类完成。

进阶与补充

以下的章节是一些进阶的知识点和对前面没有提到的的内容的补充，主要参考资料是《Python程序设计算法基础教程》。

格式化输入输出

py提供了风格类似于C的printf()函数来格式化输入输出。如下：

```
print("hello, %s, %d, %f.2f"%("world",123,3.1415926))
```

输出：

```
hello, world, 123, 3.14
```

其中，%s表示字符串，%d表示整数，%f表示浮点数。 与C不同的是，printf其中的逗号换成了%占位。

另外，py还提供了format()函数来格式化输入输出。如下：

```
print("hello, {} {}, {:.2f}".format("world",123,3.1415926))
```

输出：

```
hello, world 123 3.14
```

其中，{}表示占位符，第一个{}表示第一个参数，第二个{}表示第二个参数，第三个{}表示第三个参数。
不过，py更常用的还是前面基础提到的f-string格式化输入输出。

列表推导式

列表推导式是一种简洁的创建列表的方式。其语法如下：

```
[表达式 for 变量 in 可迭代对象 if 条件]
```

- 表达式：用于生成元素的表达式。
- 变量：用于遍历可迭代对象中的元素。
- 可迭代对象：用于生成元素的可迭代对象。
- 条件：用于过滤元素的条件。

循环

enumerate()函数

enumerate()`函数用于将一个可迭代对象（如列表、元组、字符串）组合为索引-元素对，并返回一个可迭代对象。

```
seasons=['spring','summer','autumn','winter']
for index,season in enumerate(seasons,start=1):#若不指定start则默认从0开始
    print(index,season)
```

输出：

```
1 spring
2 summer
3 autumn
4 winter
```

zip()函数

zip()`函数用于将多个可迭代对象（如列表、元组、字符串）组合为一个键值对的元组，并返回一个可迭代对象。若对应的元素个数不一致，则返回最短的可迭代对象长度的元组。

```
seasons=['spring','summer','autumn','winter']
days=['Monday','Tuesday','Wednesday','Thursday']
for i in zip(seasons,days):
    print(i)
```

输出：

```
('spring', 'Monday')
('summer', 'Tuesday')
('autumn', 'Wednesday')
('winter', 'Thursday')
```

map()函数

map()函数用于将一个函数应用到一个可迭代对象（如列表、元组、字符串）的每个元素，并返回一个新的可迭代对象。

```
def square(x):
    return x**2

numbers=[1,2,3,4,5]
result=list(map(square,numbers))
print(result)
```

输出：

```
[1, 4, 9, 16, 25]
```

```
char_1,char_2=map(int,input("请输入两个数字：".split(" "))

print(char_1+char_2)
```

输出：

```
请输入两个数字：1 2
输出：3
```

filter()函数

filter()`函数用于从一个可迭代对象（如列表、元组、字符串）中过滤出符合条件的元素，并返回一个新的可迭代对象。

```
def is_odd(x):
    return x%2!=0


numbers=[1,2,3,4,5,6,7,8,9]
result=list(filter(is_odd,numbers))
print(result)
```

输出：

```
[1, 3, 5, 7, 9]
```

数据类型

int对象

int(字符串,base=10)方法可选参数base用于指定进制，默认为10。若字符串以0x或0X开头，则以16进制解析；若以0b或0B开头，则以2进制解析；否则，以10进制解析。

```
print(int("100"))
print(int("0x100"))
print(int("0b100"))
```

输出：

```
100
256
4
```

而int对象还存在bit_length()方法，用于返回整数的二进制数的位数。

```
num=input("Enter a number:")
intnum=int(num,base=2)
print(intnum,intnum.bit_length(),end=" ")
```

输出：

```
Enter a number:101
5 3
```

注：bin()函数可将整数转换为二进制字符串。

```
print(bin(5))
```

输出：

```
0b101 //0b表示这是一个二进制数
```

复数类型

Python中提供了`complex()`函数来创建复数，其语法如下：

```
complex(real, imag)
```

- real：实部，必选参数。
- imag：虚部，可选参数，默认为0。

```
c1=complex(1,2)
c2=complex(3)
print(c1)
print(c2)
```

输出：

```
1+2j
3+0j
```

序列数据

序列的链接与重复操作

对于序列，Python提供了+运算符来链接序列，*运算符来重复序列。

```
a=[1,2,3]
b=[4,5,6]
c=a+b
print(c)
d=a*3
print(d)
```

输出：

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

序列的成员关系操作

- in运算符：用于判断元素是否在序列中。
- not in运算符：用于判断元素是否不在序列中。
- s.count(x)方法：用于统计序列s中元素x出现的次数。
- s.index(x)方法：用于查找序列s中元素x的索引位置。

```
a=[1,2,3,4,5]
print(3 in a)    #True
print(6 in a)    #False
print(a.count(3))#1
print(a.index(3))#2
```

输出：

```
True
False
1
2
```

序列的排序操作

- s.sort()方法：用于对序列s进行排序，默认升序。
- s.reverse()方法：用于反转序列s。
- sorted(s)函数：用于返回一个新的排序后的序列。
  sorted()函数的语法如下：`sorted(iterable, key=None, reverse=False)`其中key参数用于指定排序的依据（key=None时，默认按元素的升序排序），reverse参数用于指定排序顺序，默认为False。（正序）

```
a=[3,1,4,2]
a.sort()
print(a)
a.reverse()
print(a)
b=sorted(a)
print(b)
```

输出：

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

内置函数all()和any()

all()函数用于判断一个可迭代对象（如列表、元组、字符串）中的所有元素是否都为真，any()函数用于判断一个可迭代对象（如列表、元组、字符串）中的至少有一个元素为真。

```
a=[True,True,True]
print(all(a))#True
a=[True,False,True]
print(all(a))#False
a=[False,False,False]
print(all(a))#False
a=[True,True,False]
print(any(a))#True
a=[False,False,False]
print(any(a))#False
```

输出：

```
True
False
False
True
False
```

序列拆分操作

在Python中，存在一种变量个数和序列长度不等的情况，即序列元素个数可以是可变的。这种情况下，可以使用*运算符来拆分序列。

```
a,b,*c=range(5)
print(a,b,c)
```

输出：

```
0 1 [2, 3, 4]
```

序列推导式

序列推导式是一种简洁数据处理方式，用于生成符合条件的序列。

列表推导式

语法如下:

```
[表达式 for 变量 in 列表]
或者:
[表达式 for 变量 in 列表 if 条件]
```

- 表达式：用于生成元素的表达式。可以是有返回值的函数。
- 变量：用于遍历可迭代对象中的元素。类似于i的作用。
- 条件：用于过滤元素的条件。

```
#列表推导筛选姓名长度大于3的元素并且转为大写
name=["Alice","Bob","Charlie"]
names_choose=[name.upper() for i in name if len(i)>3]
print(names_choose)
```

输出：

```
['ALICE', 'CHARLIE']
```

```
nums=[i for i in range(101) if i%2==0]
```

输出：

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80,
82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
```

小结：对于上述语法，可以简单将*表达式*视作第一层筛选，如果与后面的迭代临时变量相同即不进行第一次筛选，if表达式则是第二层筛选

元组推导式

和列表推导式基本相同，只是列表的[]改成()即可，不再赘述。

集合推导式

语法相同，下面是一个例子：

```
new_set={i**2 for i in range(11) if i != 0}
print(sorted(new_set))#由于集合的无序性，用sorted()函数进行排序方便展示
```

输出：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

字典推导式

语法如下：

```
{key表达式:value表达式 for 变量 in 可迭代对象}
或者:
{key表达式:value表达式 for 变量 in 可迭代对象 if 条件}
```

语法意思是将可迭代对象各元素作为键（key），将value表达式计算后作为值（value），组成字典。

```
my_list=[1,2,3,4,5]
my_dict={i:i**3 for i in my_list if i!=4}#将list中的元素作为键，值的立方作为值，并且去除4为键的键值对。
print(my_dict)
```

输出：

```
{1: 1, 2: 8, 3: 27, 5: 125}
```

字符串操作

字符串的链接与重复操作

对于字符串，Python提供了+运算符来链接字符串，*运算符来重复字符串。

```
#去掉一个最高分和一个最高分，求平均分
score=[90,85,95,80,92]
 first,*middle,last=sorted(score)
 print(sum(middle)/len(middle))
```

输出：

```
89.0
```

命令行参数

sys.argv

py中可以通过导入sys包，然后使用sys.argv来获取命令行参数。其中，sys.argv[0]是程序本身的名称，sys.argv[1]是第一个命令行参数，以此类推。
假如脚本文件为test.py，内容如下：

```
import sys
n=sys.argv[1]
print(n)
```

在命令行中运行：

```
python test.py 10
```

输出：

```
10
```

其中main.py是程序本身的名称，10是第一个命令行参数。

argparse模块

argparse模块可以用来解析命令行参数。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("n", type=int, help="an integer for the accumulator")
args = parser.parse_args()
print(args.n)
```

```
python test.py 10
```

输出：

```
10
```

其中，`parser.add_argument()`方法用于添加命令行参数，第一个参数是参数名称，第二个参数是参数类型，第三个参数是参数帮助信息。`parser.parse_args()`方法用于解析命令行参数，返回一个命名空间对象，可以通过属性来获取参数值。

迭代器与生成器

与C++相似，Python也提供了迭代器和生成器。

迭代器

迭代器（Iterator）是一种特殊的对象，可以用于遍历可迭代对象并记住遍历位置。

Python的迭代器有两个基本方法：

```
it=iter(可迭代对象)#创建迭代器对象
print(next(it))#获取下一个元素
```

迭代器对象还能使用常规的for循环进行遍历。

```
import sys

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
it = iter(my_list)
try:
    while 1:
        print(next(it), end=" ")
except StopIteration:
    sys.exit()
```

输出：

```
1 2 3 4 5 6 7 8 9
```

当然，对于类也可实现__next__和__iter__方法，使其成为迭代器。
StopIteration 异常用于表示迭代完成，以防出现死循环。

生成器

生成器（Generator）是一种特殊的迭代器，在Python中，使用了yield语句的的函数就可被称为生成器。也即，只要使用了yield语句的函数即可作为一个迭代器。
首先，先解释一下yield语句的作用，可以粗略（且不太正确）地理解为return语句，如下:

```
def increase(num):
    x = 0
    while x < num:
        yield x
        x += 1
```

与return不同的是，一旦声明了yield，该函数立刻成为一个生成器（这意味着不能使用函数名+参数列表直接调用函数），在函数调用时进行到yield是返回保存后面的表达式，接着函数暂停执行，直到下一次继续调用。
因为生成器是一类特殊的迭代器，所以也包括迭代器的next()方法。
需要注意：生成器的生命周期与yield语句的个数相关，若生成器内部无循环且只有一个yield语句，那么生成器只能调用一次next()

```
def next_num(n: int):
    n += 1
    yield n
```

```
it = next_num(10)
print(next(it))
print(next(it))#报错，生成器内没有更多值
```

则会输出：

```
Traceback (most recent call last):
  File "E:\desktop\py_file\pythonProject\day2.py", line 8, in <module>
    print(next(it))
          ^^^^^^^^
StopIteration
11
```

生成器主要是为了解决当需要一长段序列时，不想要占用过多空间，提高内存利用率。

装饰器

前置

首先，需要回顾和更新一下函数的知识点，在C/C++中，是不允许函数的嵌套定义的，即，函数内部再次声明定义一个函数。但在Python中，允许函数的嵌套定义。而且，在主函数调用时就会调用其中的所有嵌套函数，但不能在主函数外使用这些嵌套函数。
另外，变量也能存储函数，即，变量可以指向函数，函数也可以作为参数传递给其他函数。

```
def fun():
    def fun1():
        print('hello world')

a=fun
print(type(fun))#<class 'function'>
print(type(a))#<class 'function'>
a()#调用fun函数
```

输出：

```
hello world
```

装饰器语法

装饰器（修饰器）是Python独特的特性，它可以用来修改其他函数的行为，但又不改变函数本身的定义。装饰器其本身也是一个函数，它接收一个函数作为参数，并返回一个修改后的函数。
如下示例：

```
import time

#count_time函数作为装饰器，innerfun函数作为装饰器的具体装饰内容（对其余函数要进行什么操作）
def count_time(otherfun):
    def innerfun():#定义装饰器的具体内容
        start = time.time()
        otherfun()  # 调用被装饰的函数
        ends = time.time()
        print(ends - start)
    return innerfun

@count_time
def myfunc():
    time.sleep(1)
    print("hello world")

myfunc()
```

输出：

```
hello world
1.0001144313812256
```