- **tag:**

# Practical Python Programming

## 1 Introduction to Python

### 1.1 A First Program

在交互模式下,Python提供了下划线变量_,他会保存最后一个表达式的结果,例如:

```
>>> 37*42
1554
>>> _*2
3108
>>> _+50
3158
```

但这只在交互模式下有用,我们并不会在程序中使用.

在Python中,#用于引导单行的注释,多行注释则可以用三个单引号或双引号括起来的字符串来表示.

与C/C++不同,Python并不需要在使用每个变量前声明其类型,其命名规则与C/C++类似,不能以数字开头,不能使用Python的关键字(如if,while等),变量名区分大小写.变量会在首次赋值时创建,并且会根据赋值的值来决定变量的类型.因此,他和C/C++的类型最大的不同在于,他的变量类型其实是可以随着程序需要而改变的,例如:

```
>>> height=442
>>> type(height)
<class 'int'>
>>> height=442.0
>>> type(height)
<class 'float'>
>>> height='Really tall'
>>> type(height)
<class 'str'>
```

与C/C++不同,Python并不通过{}来表示代码块,而是通过代码的缩进来表示代码块,通常缩进4个空格,因此对于Python而言,代码的缩进并不是风格问题,而是语法的一部分.不同代码缩进表示不同的代码块,例如:

```
while num_bills*bill_thickness<sears_height:
    print(day,num_bills,num_bills*bill_thickness)
    day=day+1 #ERROR
    num_bills=num_bills*2
```

这样的话day=day+1就会报错,因为他并不在while循环的代码块内.反之如果day=day+1并不做缩进,那么他就不在while循环内,而是while循环后的第一条语句,因此day=day+1只会执行一次.

Python的条件判断语句和C/C++类似,但是如果我们希望在if判断中检查多个条件,我们需要用elif来添加检查,而不是用else if:

```
if a>b:
    print("a is greater than b")
elif a==b:
    print("a is equal to b")
else:
    print("a is less than b")
```

Python中并不需要引入其他的包文件,就可以用print函数打印输出,并且print函数可以接受多个参数,并且会在参数间自动添加空格:

```
print("Hello","world!")
print("The answer is",42)
```

并且print函数会默认在输出的后面添加一个换行符,如果不希望添加换行符,可以在print函数中添加end参数:

```
print("Hello","world!",end=' ')
print("The answer is",42)
```

Python中可以用input函数向用户打印提示并且以字符串的形式获取用户输入:

```
name=input("What is your name? ")
print("Hello",name)
```

他比较适合于用于调试或者交互,而不适合于正式的程序输入.但这里与C/C++不同的是,Python的输入得到的只能是字符串类型,如果需要其他数据类型,需要用强制类型转换函数进行转换,例如:

```
age = input('Enter your age\n:')
age=int(age)
age=age+1
print('Your age is', age)
```

因为他读取的是一行字符串,所以即使用户在输入的文本中输入了空格,他也会被当做字符串的一部分一起输入,并不会像C/C++那样,遇到空格就结束输入.然而,如果我们需要输入多个值,可以用split方法将字符串拆分为多个部分,例如:

```
name,age=input('Enter your name and age:').split() #默认以空格拆分
name,age=input('Enter your name and age:').split(',') #以逗号拆分
```

如果我们得到的输入多个值是相同的类型,那么我们可以用map函数将拆分后的字符串转换为相同的类型,例如:

```
a,b,c=map(int,input('Enter three integers:').split())
print(a,b,c)
# 如果我们并不知道输入的整数有多少个,可以用一个列表来接收
num_list=list(map(int,input('Enter some integers:').split()))
print(num_list)
```

值得注意的是,这个map函数他只针对于多个值都是相同类型的情况,如果多个值类型不同,那么就需要分别转换.

有时我们需要指定一个空代码块,用于if语句或者函数框架设计等,此时可以用pass关键字用来表示无操作语句.他并不会执行任何操作,只是一个占位符,可能用于后续代码的添加,例如:

```
if a>b:
    pass
else:
    print("a is less than or equal to b")
```

## 1.2 Numbers

Python有三种数字类型:bool,int,float和complex.这里值得注意的是,Python中并没有对浮点数进行进一步分类,而是归为了一类float,因此python中并没有float和double之分.

bool类型可以与int和float类型进行混合运算,其结果类型仍与int和float相同,这相当于在运算过程中出现了隐式类型转换.在运算的过程中,True会被当做1,False会被当做0.

在int和float类型对应的运算中,其有与C/C++类似但结果不同的运算符:

```
5/2 # 其结果为2.5,与C/C++的除法依赖于两侧操作数不同,python的/就是普通除法,无论两端操作数的类型,其结果都是float
5//2 #其结果为2,这其实相当于C/C++的整数除法,但是要求两端操作数均为整数时,其结果类型才为int
5.0//2 #其结果为2.0,两端操作数只要有一个为float时,其结果类型为float
3**2 #其结果为9,其就是C/C++的power函数,只不过power函数要求两个量均为double类型,而**则允许有整数
```

同样与C/C++一样,int也是有位运算的,并且float由于浮点数存储方式的差异并没有位运算.

与C/C++一样,Python也具有与或非三类逻辑运算符,但Python的运算符是and or not,并不是C中的&& || !.

## 1.3 Strings

Python与C/C++不同在于他并没有char类型,因此对于Python的字符串而言,他可以用单引号或者双引号来引导字符串.Python的字符串通常只能传入单行,遇到换行符会报错;但可以通过三引号来表示多行字符串,例如:

```
# Single quote
a='Yeah but no but yeah but...'
print(a)

# Double quote
b="Computer says no"
print(b)

#Triple quotes
c = '''
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes,
don't look around the eyes,
look into my eyes, you're under.
'''
```

```
print(c)
```

但我们前面提到的多行注释也可以用三引号表示,这是因为Python其实并没有多行注释的专用符号,我们通过三引号引导的多行字符串对程序进行解释,但不做赋值操作,程序会阅读后并不保存相关数据,也就认为其可用于注释.其次我们用三引号引导的字符串做注释的时候,如果其出现在函数,模块或者类的开头,那么其会被认作是该代码的文本字符串,我们可以通过.__doc__属性或者help函数来查看:

```
def add(a, b):
    """
    这个函数用于计算两个数的和
    参数:
        a: 第一个数字
        b: 第二个数字
    返回:
        两个数字的和
    """
    return a + b

help(add)  # 查看函数的文档字符串
print(add.__doc__)
```

正是因为Python的字符串可以用单引号或双引号引导,因此我们可以不借助转义符来表示字符串中包含引号的情况,例如:

```
a="He said, 'Hello!'"
b='She replied, "Hi!"'
```

但是如果字符串中既包含单引号又包含双引号,那么我们还是需要用转义符\来表示.

在C/C++中,字符串的每个字符在计算机内部都是以ASCII编码的形式存储的,当然更为现代的版本也是支持了UTF8编码.同样Python的字符串也是以Unicode编码形式存储的,或者说我们可以给程序一个字符的unicode编码,程序会将其转换为对应字符后存储.

```
a='\xf1' # \x表示后面跟的是一个十六进制的数字,其后f1表示两个字节
b='\u2200' # \u表示是unicode编码,其后接的2200是四个十六进制数字,表示其码点为8704
c='\U0001D122' # \u和\U都表示unicode编码,但是\U后面需要接8个十六进制数字,不足8位需要在前面补0,其码点为119074
d='\N{FOR ALL}' # \N{}是通过字符名称来表示字符,其字符名称必须是Unicode标准中定义的名称
```

对于字符串索引,同样字符串也是从0开始索引,但与C/C++不同的是,Python支持负数索引,负数索引表示从字符串的末尾开始向前索引.例如,-1表示字符串的最后一个字符,-2表示导数第二个字符,以此类推:

```
s='Hello world'
print(s[0]) # H
print(s[-1]) # d
print(s[-5]) # w
```

Python也支持利用:指定字符串检索范围来作切片操作或者提取子字符串,若存在缺省指标,则默认为字符串的开头或者结尾,这里值得注意的是s[start:end]会包含start的索引,但不包含end:

```
d = s[:5]     # 'Hello'
e = s[6:]     # 'world'
f = s[3:8]    # 'lo wo'
g = s[-5:]    # 'world'
```

Python的字符串操作有+(链接两个字符串),len(求字符串长度),in¬ in(检查子字符串是否在字符串中出现过)和*(重复字符串):

```
# Concatenation
a='Hello'+'World' # 'HelloWorld'
b='Say'+a # 'SayHelloWorld'
# Length
print(len(a)) # 10
# Membership test(in/not in)
t='e' in a # True,bool类型
f='x' in a # False,bool类型
g='hi' not in a # True,bool类型
# Replication(s*n)
rep=a*2 # 'HelloWorldHelloWorld',他主要是同一个字符串的重复,类似于a+a+...+a(n个a相加)
```

Python的字符串还有很多内置的方法,可以通过.来调用,例如:

```
s.endswith(suffix)      # 检查字符串后缀是否为suffix,如果有则返回True,否则返回False
s.find(t)               # 在字符串s中查找字符串t第一次出现的位置,如果找不到返回-1
s.index(t)              # 在字符串s中查找字符串t第一次出现的位置,如果找不到返回异常ValueError
s.join(slist)           # 以字符串s作为分隔符,将字符列表的所有元素连接成一个新的字符串
s.replace(old,new)      # 替换字符串s中所有old子字符串为new子字符串,返回一个新的字符串
s.rfind(t)              # 从字符串s的末尾开始查找字符串t最后一次出现的位置,如果找不到返回-1
s.rindex(t)             # 从字符串s的末尾开始查找字符串t最后一次出现的位置,如果找不到返回异常ValueError
s.split([delim])        # 以delim为分隔符切片s,如果不指定delim,则以空白字符(空格,换行,制表符等)为分隔符
s.startswith(prefix)    # 检查字符串前缀是否为prefix,如果有则返回True,否则返回False
s.strip()               # 删除字符串s两端的空白字符(空格,换行,制表符等),返回一个新的字符串
```

值得注意的是,Python的字符串是不可改变的,也就是我们不能通过索引赋值和方法来修改已有的字符串,只能通过对整体的重新赋值来改变字符串的值,如:

```
s='  Hello   '
print(s.strip()) # 'Hello'
print(s) # '  Hello   ',原字符串并没有改变
```

对于上面罗列的方法,我们需要指出以下几点:

1. join方法中传入的slist一定要是一个字符串列表,否则会报错.
2. find和index方法的作用其实一样,但是唯一的不同在于find返回的是-1,而index会报出异常.因此在不确定子字符串是否存在的情况下,建议使用index方法.
3. 在replace方法中,如果old子字符串在s中不存在,他并不会报错,返回的值仍然是原字符串s.

Python在普通的字符串之外还提供几类特殊的字符串,如字节字符串,原始字符串和格式化字符串.

字节字符串(byte string)是以b或者B开头的字符串,其是以字节为单位进行存储的,每个元素默认是0-255之间的整数,适合处理二进制数据,网络传输或者文件I/O等操作.他和普通字符串最大的区别在于,在Python3中,普通字符串依靠Unicode编码,每个字符可以是汉字,英文或者特殊符号等;而字节字符串并不依赖编码规则,而是直接存储字节,因为计算机中存储的最小单元就是字节,所以字节字符串可以存储更多类型的内容,如视频,音频等等.

```
bstr=b'Hello' # 字节字符串
print(bstr) # b'Hello'
print(bstr[0]) # 72,字节字符串的每个元素是0-255之间的整数
print(bstr[1:4]) # b'ell',字节字符串的切片操作返回的仍然是字节字符串
```

字节字符串并不能与普通字符串进行链接,否则会报类型错误;可以用索引访问每个字节,但是如果访问的是一个元素,返回的是字节对应的整数值,如果访问的是一个切片,返回的则是字节字符串.字节字符串和普通字符串之间可以通过encode和decode方法进行转换,如:

```
s='Hello'
bstr=s.encode('utf-8') # 普通字符串转换为字节字符串
print(bstr) # b'Hello'
s2=bstr.decode('utf-8') # 字节字符串转换为普通字符串
print(s2) # 'Hello'
```

这里的utf-8表示编码格式,当然也可以使用其他编码格式,如ascii,latin1等.

原始字符串(raw string)是以r或者R开头的字符串,其最大的特点在于字符串中的转义字符并不会被处理,换言之,他不会处理字符串中出现的\和其后面的字符,而是将其作为普通字符进行存储.如:

```
rs=r'C:\newfolder\test.txt' # 原始字符串
print(rs) # C:\newfolder\test.txt
```

字符串是括号内包围的原始文本,与输入完全一致.这在反斜杠具有特殊意义的情况下很有用.例如:文件名,正则表达式等.原始字符串有如下的特点需要注意:

1. 原始字符串不能以奇数个反斜杠结尾,因为这样最后一个反斜杠就不会有任何字符与之配对,会自动匹配引号,从而导致字符串没有结束符而报错.
2. 字符串前的r/R只影响转义,不会影响字符串内容的其他特性.

```
r'C:\path\to\file\\' # 正确,以偶数个反斜杠结尾
r'C:\path\to\file\' # 错误,以奇数个反斜杠结尾,会报错
len(r'Hello\nWorld') # 12,原始字符串中的\n不会被处理为换行符
len('Hello\nWorld') # 11,普通字符串中的\n被处理为换行符
```

格式化字符串(f-string)是以f或者F开头的字符串,其允许在字符串中嵌入表达式,这些表达式会在运行时被计算并替换为其结果.

```
name='Alce'
age=25
print(f"My name is {name}, I am {age} years old.") # My name is Alce, I
am 25 years old.
```

其基本的用法如下:

```
# 表达式
a, b = 5, 10
print(f"{a}+{b}={a+b}") # 5+10=15
# 函数调用
import math
print(f"pi={math.pi:.2f}") # pi=3.14,保留两位小数
# 格式化
value = 1234.56789
print(f"{value:.2f}")   # 保留两位小数 → 1234.57
print(f"{value:10.2f}") # 宽度10，右对齐 → "   1234.57"
print(f"{value:<10.2f}")# 宽度10，左对齐 → "1234.57   "
ratio=0.456
print(f"{value:.1%}")# 百分比格式 → 45.6%
#这里需要注意百分比格式的%和f是不能共存的
num=42
print(f"{num:05x}") # 十六进制格式 → 00042
```

其有几个特点需要注意:

1. f-string中可以嵌入任意的Python表达式,包括函数调用,算术运算,条件表达式等,但是尽量不要太复杂,否则会影响代码的可读性.
2. 大括号本身需要转义,可以使用双大括号{{和}}来表示单个大括号.
3. f-string本身是支持转义字符的,但是在表达式中不支持转义字符.

## 1.4 Regular Expressions

在上一节中提到了Python可以利用字符串的find和index方法来查找字符串,但是这种方法只能查找固定的字符串,也就是精确查找.而我们在正常使用过程中模糊查找其实更为常见,例如查找所有以a开头,以b结尾的字符串,这就需要利用正则表达式来实现搜索.

正则表达式的基本语法如下:

| 符号 | 含义 | 示例 |
| --- | --- | --- |
| . | 匹配除换行符外的任意单个字符 | a.b匹配开头为a,结尾为b的连续三个字符 |
| \^ | 匹配字符串的开头 | ^abc匹配以abc开头的字符串 |
| \$ | 匹配字符串的结尾 | $abc匹配以abc结尾的字符串 |
| [] | 定义字符集合,匹配其中任意一个字符 | [aeiou]匹配任意一个原因字母 |
| [^] | 定义字符集合,匹配不在其中任意的一个字符 | [^0-9]匹配任意非数字字符 |
| - | 字符类中表示范围 | [a-z]匹配任意小写字母 |
| * | 匹配前面的子表达式零次或多次 | a*e匹配开头有且仅有0个或多个a,结尾为e的字符 |
| + | 匹配前面的子表达式一次或多次 | a+e匹配开头有且仅有1个或多个a,结尾为e的字符 |
| ? | 匹配前面的子表达式零次或者一次 | colou?r匹配color和colour |
| {n} | 精确匹配前面的子表达式n次 | da{3}e匹配daaae |
| {n,} | 至少匹配前面的子表达式n次 | da{2,}e匹配daae,daaae等 |
| {n,m} | 匹配前面的子表达式n次到m次 | a{2,4}匹配aa,aaa,aaaa |
| () | 定义子表达式或捕获组 | (ab)+匹配ab,abab等 |

正则表达式里的特殊字符如果需要作为普通字符使用,则需要在其前面加上反斜杠\进行转义,如:

| 符号 | 含义 | 等价符号 |
| --- | --- | --- |
| \d | 匹配任意数字 | 等价于[0-9] |
| \D | 匹配任意非数字 | 等价于[^0-9] |
| \w | 匹配任意单词字符(字母,数字,下划线) | [a-zA-Z0-9_] |
| \W | 匹配任意非单词字符(字母,数字,下划线) | [^a-zA-Z0-9_] |
| \s | 匹配任意空白字符(空格,制表符,换行符) | [ \t\n] |
| \S | 匹配任意非空白字符(空格,制表符,换行符) | [^ \t\n] |
| \b | 边界匹配符,如果在左边出现要求是开头,右边则是结尾,同时出现则精确匹配 | ^,$,"" |

这里我们对前面提到的正则表达式的基本语法进行一些补充说明:

1. *,+,?和{n,m}这些量词都是贪婪的也就是他会尽可能多的匹配字符,如下:

```
import re
text="<div>abc</div><div>def</div>"
re.findall(r"<div>.*</div>",text) # ['<div>abc</div><div>def</div>']
```

这里的.*会尽可能多的匹配字符.如果想让其变为非贪婪模式,则可以在量词后面加上?,如下:

```
# 非贪婪模式
re.findall(r"<div>.*?</div>", text)
# 结果: ['<div>abc</div>', '<div>def</div>']
```

2. 匹配开头或者结尾的时候只写了部分模式,这并不会匹配整个字符串,而是匹配字符串的开头或者结尾部分,如下:

```
re.findall(r"^abc", "abcdef") # ['abc']
re.findall(r"def$", "abcdef") # ['def']
```

3. 忘记了对特殊字符做转义

```
re.findall(r"d{3}.txt","dddktxt") # ['dddktxt']
```

4. 字符集和字符分组混用错误:[abc]可以匹配单个字符'a','b'和'c';而(abc)匹配整个字符串"abc".

re模块是Python标准库中用于正则表达式操作的模块,提供查找,匹配,替换,分组等功能.re模块的基本函数为

| 函数 | 作用 |
| --- | --- |
| re.match(pattern,string) | 从字符串开头匹配 |
| re.search(pattern,string) | 搜索整个字符串,找到第一个匹配 |
| re.findall(pattern,string) | 返回所有匹配的字符串列表 |
| re.finditer(pattern,string) | 返回迭代器,每个元素是Match对象 |
| re.fullmatch(pattern,string) | 整个字符串必须完全匹配 |
| re.split(pattern,string) | 按模式分割字符串 |
| re.sub(pattern,repl,string) | 替换匹配的部分 |
| re.subn(pattern,repl,string) | 返回(替换后字符串,替换次数) |

返回的match对象的常用属性:

```
import re
m=re.search(r"\d+","abc123def")
print(m.group()) # 返回匹配的字符串"123"
print(m.start()) # 返回匹配子串的起始索引,3
print(m.end()) # 返回匹配子串的结束索引,6
print(m.span()) # 返回(start,end),(3,6)
```

re模块提供了一个编译模式,可以提高后续使用相同的正则表达式的运行效率.re.compile的作用会将一个正则表达式编译成一个正则对象,后续可以重复使用这个对象来匹配字符串.这样的好处是

1. 复用性强,不需要每次都写pattern,直接使用正则对象的方法.

2. 效率更高,编译过的正则对象会缓存,特别是同一个模式需要多次匹配时,具有更高的性能

其基本语法为:

```
import re
# re.compile的基本语法;pattern为正则表达式字符串,flags则是表示匹配模式
regex=re.compile(pattern,flags=0)
pattern=re.compile(r"\d+")
text="Order123, Item456, Code789"
print(pattern.findall(text))
for m in pattern.finditer(text):
    print(m.group(),m.span())
```

上面提到了re.compile的标准语法中存在一个flags变量,实际上re的每个函数都有这个变量,在此介绍flags变量的值及其作用:

| FLAGS | 作用 |
| --- | --- |
| re.I(re.IGNORECASE) | 忽略大小写,匹配时不区分大小写 |
| re.M(re.MULTILINE) | 让^和$作用于每一行的开头和结尾,默认仅匹配整个字符串的开头结尾 |
| re.S(re.DOTALL) | 默认.不匹配换行,开启后可以匹配换行 |
| re.X(re.VERBOSE) | 正则里可以加空格和注释，提高可读性 |

## 1.5 List

列表是Python中存储有序值集合的主要类型,用如下的方式创建:

```
names=['Elwood','Jake','Curtis']
nums=[39,38,42,65,111]
```

我们通过input函数接收到的输入是字符串形式,我们可以用字符串的split方法将其转换为列表形式:

```
text='GOOG,100,490.10'
row=text.split(',')
print(row) # 输出为['GOOG','100','490.10']
```

列表与C语言的数组不同,他允许列表内元素具有不同的数据类型.列表可以通过append方法,在列表末尾加新的元素;列表也可以利用insert方法,在指定索引位置插入新的元素:

```
names.append('Frank')
print(names)# ['Elwood','Jake','Curtis','Frank']
names.insert(2,'Buster')
print(names)# ['Elwood','Jake','Buster','Curtis','Frank']
names.append(10)
print(names)# ['Elwood','Jake','Buster','Curtis','Frank',10]
```

列表的其余操作与字符串的操作基本一致,在此我们不再赘述.但是有两个不同需要我们额外指明:列表与字符串不同,他是可以通过索引赋值的方式修改列表内的元素的.

```
names[1]='Joliet'
print(names) # ['Elwood', 'Joliet', 'Buster', 'Curtis', 'Frank', 10]
```

其次,列表是没有find方法的,如果我们希望在列表中查找某一个元素的索引,只能使用index方法,其如果查找失败返回的是ValueError,而不是find的-1.

因为前面我们提到了列表其实是可以被修改的,故而我们介绍对列表的删除操作.首先,列表自身具有remove方法,我们可以直接利用remove方法来移除我们指定的元素.但是需要注意的是,如果我们指定的元素出现了多次,那么remove方法仅移除第一次出现的;如果我们试图一个不存在的元素,那么他会抛出ValueError异常,因此使用remove方法的时候,尽量保证元素是合法的.除了remove方法以外,python还提供了一个关键字del,del可以直接针对性的删除某个位置的元素或者某段切片的元素.

```
names.remove('Buster')
print(names)
del names[names.index(10)]
print(names)
s=[0,1,2,3,4,5]
del s[2:5]
print(s)
```

python的删除操作并不会产生列表的空位,而是删除了以后,后续的元素不断前移来填充空位.

Python还提供了列表的排序方法.sort()方法可以在原地修改列表并返回None.

```
nums.sort()
print(nums)
```

sort方法则提供了两个比较常用的参数:key和reverse.reverse默认是False,也就是默认的排序方式是升序排序,可以通过给他赋值True来完成倒序排序;虽然列表允许其元素具有不同的数据类型,但是sort方法并不能够应用在不同的数据类型,需要用key来指定比较的方式,比较常见的是str,他会将不同的数据类型转换为字符串,并利用字符串的大小比较.

```
nums.sort(reverse=True)
print(nums)
names.sort()
print(names)
names.append(10)
print(names)
names.sort(key=str)
print(names)
```

sort方法是在原列表的基础上修改,并不会生成新列表.如果希望生成新列表,而不在原列表上修改的话,可以使用sorted函数.

```
s=sorted(names,key=str,reverse=True)
print(s)
print(names)
```

这里我们需要指出的是列表是不支持数学运算的,因为他的加法被定义为拼接,乘法定义为重复.

## 1.6 File Management

打开文件的语法:

```
f=open('foo.txt','rt')
```

open接受两个参数,第一个参数就是需要打开的文件位置,第二个参数表示以什么模式打开文件,r表示以只读方式,如果文件不存在,就会报错;w为以只写的方式打开,如果文件不存在,那就会创建一个新文件,如果文件存在就会覆盖原文件内容;a表示追加,并不会覆盖原文件内容,而是在文件的末尾追加内容;r+和w+都表示同时赋予读写权限,但区别在于r+不会覆盖原文件,而遇到文件不存在就会报错;而w+会覆盖原文件,但文件不存在会创建一个文件.rt中的t表示以文本模式读取文件的内容,默认都是文本模式读取,故可以省略;但如果我们希望以二进制方式读取那么就需要自己显式声明,用b表示.

文件读取的语法:

```
data=f.read(2)
print(data)
data=f.read()
print(data)
```

read方法会一次性读取到文件末尾,而read(n)则是读取n个字符.因此,我们的示例代码的意思是先读取两个字符并输出,然后再从第三个字符开始一直读取到最后再输出.如果我们将这两个读取函数换一下,那么第一次就会返回全部字符,第二次则是返回EOF.

文件的关闭语法: Python提供了一个手动的close方法,可以自己在程序中手动控制文件的关闭.

```
f.close()
```

但是在大项目中,同时打开多个文件,文件的关闭检查是一个十分困难的事情.因此python提供了with语句来简化了文件的关闭操作,他会在进入缩进代码块时打开文件,并在离开缩进代码块时自动关闭文件:

```
with open(filename,'rt') as file:
  statements
```

文件逐行读取的方法:因为read方法他默认了从头读取到尾,如果字符串是多行字符串,我们希望逐行读取,可以利用如下代码实现,

```
with open(filename,'rt') as file:
  for line in file:
    print(line.strip())
```

文件的写入方法:python提供了一个write方法来为文件写入字符串数据.

```
with open('outfile','wt') as file:
  out.write('Hello World\n')
```

同时也可以利用输出重定向的方式来实现.

```
with open('outfile','wt') as out:
  print('Hello World',file=out)
```

## 1.7 Functions

函数是一系列执行任务并返回结果的语句,需要使用return关键词显式指定函数的返回值.

```
def fun_name:
  statements
  return return_result
```

函数通过异常来报告错误,异常会导致函数中止;如果异常没有被处理,那么整个程序就会停止.为了调试目的,异常信息会描述发生了什么,错误发生的位置以及一个回溯信息,显示导致失败的其他函数调用.

异常可以被捕捉和处理.捕捉异常可以使用try-except语句,

```
for line in file:
  fields = line.split(',')
  try:
    shares = int(fields[1])
  except ValueError:
    print("Couldn't parse", line)
```

这里我们用ValueError举例,实际上我们需要与试图捕捉的异常类型相匹配.显然,我们在运行程序时并不知道会发生什么异常,因此异常捕捉一般出现在程序意外崩溃之后才被添加.

同时,我们可以在程序中主动抛出异常,使用raise语句,

```
raise RuntimeError('What a kerfuffle')
```

这个异常同样会导致程序运行中断,也可以用try-except捕捉:

```
try:
    raise RuntimeError('What a kerfuffle')
except RuntimeError:
    print("异常处理完成")
```

# 2 Working with Data

## 2.1 Datatypes and Data structures

None类型: None可选或缺失值的占位符.在条件语句中,认为是False.

元组指的是一组值的组合,其利用如下的方式声明,

```
s=('GOOG',100,49.1)
s='GOOG',100,49.1
```

()的存在对于定义元组并不是重要的,可以舍去.而我们还有如下两种特殊情况,

```
s=() # 0元组
s=('GooG',) # 一元组
s='GOOG',
```

这里需要注意的是一元组的声明是必须要有,在末尾的,否则会被视作基本数据类型.零元组的定义声明则必须要存在(),否则将无法区分语法错误和赋值.

元组通常用于表示简单的记录或者结构.他是一个由多部分组成的单个对象,其包含的多个部分是允许具有不同的数据类型的.和列表一样,他也是一个有序集合,也就是他可以通过下标索引得到对应的值.但不同在于他无法修改元组内容,虽然我们可以通过当前元组去生成一个新元组的方式来覆盖原元组,但其与修改还是有执行逻辑上的差异.

```
s=(s[0],75,s[2])
```

我们可以认为元组是把几个相关对象打包成一个实体对象,这样的话,可以在函数调用之中同时传输几个相关对象.元组解包的方式则是利用左侧变量的赋值来获得,但要求左侧变量的数量与元组结构内的相匹配,至于类型则并不需要,因为Python的变量类型是可以通过程序自动调整的.

```
names,shares,prices=s
```

从我们上面的讨论中,元组可以被认为是只读列表,但一般而言,列表存放多个独立变量对象的数据集合,而元组则是描述一个不会改变的事物的属性.

字典则是对于键与值的映射,所以其是键对的集合,这个与Hash表,关联数组十分类似,都可以通过键来访问对应的值.如

```
d={
    'name': 'GOOG',
    'share':100,
    'prices':470.10
}
print(d['name'])
```

与元组不可修改的性质不同,字典可以根据键名赋值的方式来修改或添加字典值,

```
s['shares']+=100
print(s['shares'])
s['date']='6/6/2024'
print(s)
```

如果希望删除字典中的某个键对,我们可以利用del关键字来完成删除操作.

```
del s['date']
print(s)
```

对于字典,有如下额外的操作

```
l=list(d) # 利用list()函数可以将字典的所有键提取成一个列表
print(l)
```

list函数会默认读取字典的键,换言之,字典其实有点像是某种意义的封装实体,外界访问字典仅可以通过其键来访问他的值,故而其读取会默认读取字典的键.所以从这个角度来看,如果利用for循环迭代访问字典,迭代的结果其实就是字典的key.

字典提供了一个keys()方法来提取字典的键,其结果并不是常见的数据类型,而是dict_keys,他是一个关于字典键的动态视图.

```
keys=d.keys()
print(keys)
del d['account']
print(keys)
```

这个动态视图的动态性指的是他可以同步更新对相关字典的改变,而不需要通过再赋值的方式来修改.同样,字典还提供了提取字典值的方法values,其结果类型是dict_values,他是关于字典值的动态视图.还提供了提取字典键值对的方法items,其结果类型是dict_items,他则是关于字典键值对的动态视图. 如果我们存在一个已知的dict_items类型,可以利用dict函数直接生成一个字典.

```
d_item=d.items()
print(d_item)
dnew=dict(d_item)
print(dnew)
```

值得注意的是,这些东西只是提供了字典的某种动态视图,他并不支持下标访问也不支持修改,如果需要修改,那么需要修改原字典.

## 2.2 Container

Python中提供了存储多个对象的容器,主要为列表,字典或集合.列表一般用于存储有序数据;字典则是用于存储无需数据;集合则是与字典类似,但其用于存储无序且不允许重复元素的数据.

我们先介绍集合的概念,其的赋值方式和字典十分类似,但是不同的是他只存储元素值,而字典存储键值对.

```
s={'IBM','GOOG'} # 集合赋值
d={'IBM':90.1,'GOOG':23.12} # 字典赋值
```

字典和集合存储的都是无序数据,因此他们并不支持利用下标的方式索引,而是利用关键字的方式加以检索.但不同的是,由于字典和集合的赋值方式十分类似,因此我们需要指出二者的空集声明方式是不同的,如

```
s=set() # 空集合
d={} # 空字典
s=set(['a','b','c','a'])
print(s) # {'a','b','c'}
```

从上面的赋值过程,我们发现其实我们是可以给集合赋值重复元素的,但是程序会自动清除重复元素,因此集合可用来处理程序中出现的重复元素.一般来说,字典的基本操作是键值之间的映射运算,而集合的基本操作则是集合运算,如

```
s1={'a','b','c'}
s2={'c','d','e'}
s1|s2 # 集合并运算:a b c d e
s1&s2 # 集合交运算:c
s1-s2 # 集合差运算:a b
s1^s2 # 集合对称差运算:a b
s1.add('f') # 集合添加元素操作
s2.remove('a') # 集合删除元素操作
```

集合和字典一样,底层都是由Hash表实现,因此在字典和集合中查找元素所需的时间复杂度为O(1).

如果考虑的数据对数据顺序十分敏感,那么建议采用列表来存储数据.列表可以包含任何类型的数据对象.列表的构建可以从空列表开始,利用append方法不断延展列表内容.

```
l=[]# 建立空列表
l.append(12)
l.append(12.34)
```

这里我们需要额外声明一下,如果我们想利用append方法往列表内输入元组,必须使用()来显式展示出我们输入的对象是元组,如果我们不加(),会让系统误认为输入了多个参数,从而导致程序报错.

如果考虑的数据需要快速随机查找或者频繁随机查找(随机查找就是按键查找),那么建议采用字典来存储数据.同样,字典的构造也是从空字典开始,而后不断追加字典元素.

```
d={}
d['IBM']=90.2024
d['AA']=10.2
```

对字典的查找,我们可以利用in来完成查找,返回的值是True/False.这一般是用来判断字典中是否存在特定的键.但我们可能需要直接获取键对应的值,那么我们可以采用get方法,他有两个参数,第一个用于输入用于查找的键,第二个则是如果查找不到,则会输出的默认值.

```
print('IBM' in d) # True
print('AA' not in d) # False
print(d.get('IBM',0.0)) # 90.2024
print(d.get('AB',0.0) # 0.0
```

字典的键并不是强制要求是字符串,但要求其是不可变的,如元组;列表,集合和其他的字典都不可以作为字典的键.

```
holiday={
    (1,3):'New York',
    (5,6):'Wuhan'
}
print(holiday[1,3])
```

## 2.3 Sequence

Python中给出了三种不同的序列类型:字符串,列表和元组.序列指有序的数据结构,因此他们可以按整数下标进行索引,同时可以获取其长度.

```
a='Hello' # String
b=[1,4,5] # List
c=('GOOG',100,490.1)

# Indexed order
print(a[0])
print(b[-1])
print(c[1])

# Length of sequence
print(len(a))
print(len(b))
print(len(c))
```

序列的基本操作:可以利用*来重复序列数据;+用来串联两个相同类型的序列数据,一定要是相同类型的,不同类型会报错

```
print(a*3)
print(b*2)
print(c*2)

a=(1,2,3)
b=(2,3,4)
print(a+b)
```

由于序列具有按下标索引的方式,因此序列可以做切片操作,从原序列中提取出子序列,其形式为s[start,end],其从s[start]一直提取到s[end-1].

```
a=list(range(9))

print(a[2:5])
print(a[-5:])
print(a[:3])
```

序列切片的注意点:

1. 索引的开始和结束必须是整数
2. 切片提取的时候并不会提取尾值
3. 如果开始或结束有省略值,那么默认为序列开始或者末尾

序列的切片重赋值操作并不需要提取出的切片长度和赋值长度相同,程序会自己调整;可以利用del关键字,直接对序列的某段切片执行删除操作.

```
b=list(a)
b[2:5]=[10,11,12,13,14]
print(b)
del b[2:5]
print(b)
```

序列的常用函数:sum(对序列元素求和),min(选取序列元素的最小值),max(选取序列元素的最大值).需要注意的是这里的sum并不能对字符串操作,min和max也不能让字符串和数字比较.

```
t = ['Hello', 'World']
print(max(t))
print(max(max(t)))
```

序列迭代,其实是在迭代序列中的每个元素.每个循环会从序列中提取出一个值放入迭代量i中,再对迭代量i操作.每次循环都会对迭代量i进行覆盖,并且与C的for循环不同,Python的迭代量并不会因为循环结束而释放,并保留最后一次迭代值.

```
s=[1,4,9,16]
for i in s:
    print(i)
    print(i)
```

类似C/C++,Python同样具有break和continue.break适用于跳出循环,但是他只能跳出一层循环,如果我们在嵌套循环中使用,那么break只能跳出当前最内层的循环.continue则是直接跳过本次循环进入下一次循环.

range函数可以创建一个可迭代对象,一般是用于for循环中.其语法形式为

```
range(stop)
range(start,stop[,step])
```

start表示计数从start开始,如果不提供start,那么默认从0开始.stop表示计数到stop,但是不包括stop.step表示推进步长,默认是1.

```
for i in range(10):
    print(i*i)
for j in range(10,20):
    print(j)
for k in range(10,51,2):
    print(k)
```

enumerate函数是用于将一个可遍历的数据对象组合为一个索引序列可以同时列出数据和数据下标,一般用于for循环中.他和直接用序列迭代的不同在于提供了额外的计数器来获得对应的数据下标.

```
enumerate(sequence,[start=0])
```

sequence指序列或某种可迭代的对象,start表示下标计数开始的位置,默认是0.

```
names=['Elwood','Jake','Curtis']
for i,name in enumerate(names):
    print('i=',i,'name=',name)
for i,name in enumerate(names,start=1):
    print('i=',i,'name=',name)
```

enumerate有个十分常见的应用场景,就是读取文件的行号.

```
with open(file,'rt') as f:
  for lineno,line in enumerate(f,start=1):
    pass
```

对于元组列表,如果我们直接用一个迭代量进行迭代,那么他会赋值元组,并不是很好操作.我们可以用多个迭代量加以迭代,这样的好处是他会对元组进行解包,每个迭代量对应元组的相应量.这要求迭代量的数量必须与每个元组的项数匹配,列表中每个元组的项数必须相等.

```
points = [
    (1, 4),(10, 40),(23, 14),(5, 6),(7, 8)
]
for x in points:
    print(x)

for x, y in points:
    print(x,y)
```

zip函数通过接受多个序列,将其组合之后创建一个迭代器,其类型为zip类型.如果接收到的多个序列长度不一样,那么zip函数的结果以最短的序列为基准.zip比较常见的应用其实是为了创建字典来构造键值对.

```
columns = ['name', 'shares', 'price']
values = ['GOOG', 100, 490.1 ]
pairs = zip(columns, values)
d=dict(pairs)
```

## 2.4 Collection module

collection模块提供了一些用于数据处理的对象.如Counter计数器,defaultdict和deque等.在此我们只介绍这三个对象.

Counter其实是字典的一个子类,他与普通的字典的区别在于,他的键为待计数的元素,他的值为计数值或其余相关的数据;因此他的值虽然是计数值,但实际上是允许出现0或者负值的.此外,如果我们在字典中查找一个不存在的键,那么会返回一个KeyError异常,而如果对于Counter类查找一个不存在的键,他并不会报错,并且返回0,同时创建一个新键值对,计数值设为0.

Counter常见的实例化方法如下:

```
from collections import Counter
d=Counter() # 实例化一个空对象
d=Counter(iterable objective) # 实例化一个可迭代对象,其元素为可迭代对象的元素,并且对应的count值设定为1
d=Counter(mapping objective) # 实例化一个映射对象,这里的赋值会依赖于映射的值,此处可以将count值赋为其他类型的值
d=Counter(a=1,b=2,c=3) # 利用关键字参数实例化
```

这里虽然Counter类是字典类的一个子类,所以他其实并没有顺序,但print的顺序是依据键值的大小顺序排列,从大到小排序.还有一个地方需要注意的是如果以字典来实例化Counter类,字典的键可以重复,如果出现了多次相同的键,那么他会保存最后一个键值对,因此利用字典实例化Counter类的话,是可以出现重复的键.但是如果选用利用关键字参数实例化,那么并不可以这样,如果出现了多个相同的关键字,那么他就会报SyntaxError异常.

Counter的常用方法是most_common(n).他的作用是输出计数值最大的n个对象.如果n小于Counter类的元素总数,那么输出的结果就是n个Counter类计数值最大的前n个元素;如果n大于等于Counter类的元素个数,那么相当于直接输出Counter类的所有元素;如果没有输入n,那么也是默认输出全部元素;如果输入n=-1,那么返回空列表.这里我们需要强调一点的是Counter的most_common(n)返回的并不是Counter类,而是列表类型.

```
print(d.most_common(3))
```

普通的字典是一对一的映射,也就是一个键只能对应一个值;我们可以利用的defaultdict来完成一对多的映射,其基本语法为

```
from collections import defaultdict
d=defaultdict(default_factory)
```

这里的default_factory是一个可调用对象(比如int,list,set,str),用于生成默认值.如果在defaultdict中查找一个不存在的键,他并不会报错,而是依据可调用对象的方式生成一个默认值.例如int对应的默认值为0,list对应的默认值为[],set对应的默认值为set(),str对应的默认值为".因此他的一对多映射由如下形式定义:

```
from collections import defaultdict
d=default(list)
d=['x'].append(10)
d=['x'].append(20)
print(d)
```

deque是双端队列,因此他的队列两端的插入删除操作时间复杂度为O(1).当然可以把他当做stack或者queue使用,只需要调用输入输出方法的时候控制两端的输入输出.其基本语法为:

```
from collections import deque
d=deque(iterable=None,maxlen=None)
```

其中的iterable表示可以输入一个可迭代对象,用来初始化队列;maxlen表示双端队列的最大长度,如果超过了这个长度,那么就从另一端弹出元素.队列常用的操作如下

| 方法 | 功能 | 示例 |
| --- | --- | --- |
| append(x) | 在右端添加元素 | d.append(i) |
| appendleft(x) | 在左端添加元素 | d.appendleft(i) |
| pop() | 弹出右端元素 | d.pop() |
| popleft() | 弹出左端元素 | d.popleft() |
| extend(iterable) | 在右端批量添加元素 | d.extend([3,4]) |
| extendleft(iterable) | 在左端批量添加元素(顺序反转) | d.extendleft([1,2]) |
| rotate(n) | 向右旋转n步(负数向左) | d.rotate(1) |
| clear() | 清空队列 | d.clear() |

这里需要强调的是关于extendleft和rotate的应用,我们用下面的示例代码来演示:

```
# extendleft
d=deque([1,2,3,4])
d.extendleft([5,6])
print(d) # deque([6,5,1,2,3,4])
# rotate
d = deque([1, 2, 3, 4])
d.rotate(1)
print(d)  # deque([4, 1, 2, 3])
d.rotate(-2)
print(d)  # deque([2, 3, 4, 1])
```

## 2.5 List Comprehensions

列表推导式其实就是循环的一种高效写法,他可以视作将操作应用到序列中的每个元素来创建列表.

```
x=[1,2,3,4,5]
square=[s*s for s in x] # 计算x每个元素的平方
```

不仅如此,还可以通过加if条件判断语句来过滤一些元素,如

```
x=[1,-2,3,-4,5]
square=[s*s for s in x if s>0] # 计算x中每个正元素的平方
```

因此列表推导式的通用格式如下:

```
[<expression> for <variable_name> in <sequence> if <condition>]
```

比较常见的应用如下:

1. 通过列表推导式收集特定的字典的值.

   ```
   names=[stu['name'] for stu in classes]
   ```

2. 可以执行类似数据库的查找操作

   ```
   height_name=[stu['name'] for stu in classes if (stu['height']>170)&&
   (stu['height']<180)]
   ```

3. 可以同时执行列表函数操作

   ```
   total=sum([stu['scores'] for stu in classes])
   ```

类似与列表推导式,其实还存在如集合推导式和字典推导式,在这快速介绍一下,集合推导式的作用其实是可以用来去除一下重复的元素,字典推导式则是在集合推导式的基础上指定键值对映射.

```
names={stu['name'] for stu in classes}
names_score={stu['name']:stu['scores'] for stu in classes}
```

## 2.6 Objects

Python的赋值并非赋实际值,而是创建并赋值引用副本.

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

这里我们设计了三个变量,但其实底层只有一个列表对象[1,2,3],有四个不同的引用指向他,如果我们修改其中任意一个量,都会导致所有引用的值变化.

```
b.append(4)
print(a) # [1,2,3,4]
print(b) # [1,2,3,4]
print(c) # [[1,2,3,4],[1,2,3,4]]
```

因此对于任意一个引用副本的变化都会导致全局引用副本的数值变化.因此这与其他的语言十分不同,需要牢记修改的谨慎性.由于赋值并不是赋实际值,而是赋引用副本.那么对变量的重新赋值并不会修改先前指向的内存结果,而只是修改引用指向的位置.

is运算符可以用来判断两个变量是否对应相同的对象.其是通过比较对象身份的方式进行的,而对象身份则是用id()来获取.

```
a=[1,2,3]
b=a
print(a is b) # true
c=[1,2,3]
print(a is c) # false
print(id(a)) # 2340643625152,每次运行都会不同
```

这里我们发现a和b是指向同一个对象,但是a和c却并不是,尽管a和c指向的对象值完全一样,这是因为尽管他们的指向的对象值一样,但是他们在计算机里面的逻辑存储位置不同,因此他们并不是同一个对象.但是我们可以利用==运算符来判断他们是否值相同,但值得注意的是变量具有相同的值并不一定代表指向相同的对象.

```
print(a==b) # true
print(a==c) # true
```

对于列表和字典,除了赋值的方法获得副本,还可以通过复制的方式获得副本.需要指出赋值和复制的区别在于如果直接赋值,那么两个变量就会指向同一个对象,容易出现在程序其他地方修改导致的不可预测的错误;而复制只是复制对象的值,并不会指向相同的对象,可以保证独立性.

```
# shallow copies
a=[2,3,[100,101],4]
b=list(a)
print(a is b) # false
print(a[1] is b[1]) # false
print(a[2] is b[2]) # true
```

这里我们发现如果我们用list()做浅复制,那列表内的基本数据类型项并不指向相同的对象,但拥有相同的值;而对于内部列表项(实际上,可以延拓到其他的序列类型),却是指向相同的对象,也就是修改任意一个变量的内部列表项是会传递到另一个,而如果修改其他的基本数据类型项则不会影响.

为了进一步完全的通过复制的方式获得全部数据项的副本,而不是某些项指向相同的对象,可以通过调用copy模块来完成这个操作

```
# deep copy
import copy
a=[2,3,[100,101],4]
b=copy.deepcopy(a)
print(a is b) # false
print(a[1] is b[1]) # false
print(a[2] is b[2]) # false
```

deepcopy会将对象及其包含的所有对象一起复制,不会出现其中内嵌的序列类型项指向相同的对象.

类型检查:可以通过利用isinstance()函数来判断一个对象是否为特定类型.

```
isinstance(variable,type)
if isinstance(a,list):
    print('a is a list')
```

我们可以利用元组的方式查找多个可能的类型:

```
if isinstance(a,list):
    print('a is a list')
```

最好不要频繁使用类型检查.

# 3 Program Organization

## 3.1 Functionsx and Script Writing

Python的编程风格更推荐于使用由下向上的代码编写风格. 我们将函数视作程序的构建块,从较小的,较简单的函数开始编写,后面的函数将在先前函数的基础之上继续编写.

在理想情况下,函数只对向函数输入的变量进行操作,避免对全局变量和未知的变量值变化带来的副作用.因此,构建函数的目的是模块化和可预测性.模块化是用来封装程序进程,从而利于程序编写和后期维护;可预测性,是为了避免函数造成一些未知的影响.

函数中的类型注释,其代码如下所示,

```
def func(var_name:var_type) -> return_type:
```

这些提示并不会影响函数的作用,只是起一个注释效果,并且即使实际的输入输出类型与注释的不符,IDE或者编译器可能会警告,但并不会影响程序的执行.

## 3.2 More details on functions

可以在函数参数里面设置一些默认值,那么这些值就是可选参数,使用函数时可以不对其赋值,这样的话,函数会自动调用默认值,但是前提是这些可选参数必须在参数列表的末尾.

```
def fun(var1,var2,var3=init1,var4=init4)
```

由于可选参数的存在,因此推荐参用关键字赋值的方式来传递函数参数,这样可以提高代码的清晰度和可读性.

```
fun(item1,item2,var3=item3)
```

函数如果没有设置返回值或者return空,那么他的返回值为None.

```
def fun():
    return
d=fun() # None
```

函数的返回值并不能同时分开的返回多个值,但可以通过元组的方式实现多值返回.

```
def divide(a,b):
    q=a//b
    r=a%b
    return (q,r)
x=divide(11,3) # (3,2)
x,y=divide(11,3) # x=3,y=2
```

类似于C/C++的变量作用域的讨论,Python同样可以有这样的讨论;Python的外部变量是全局变量,在任意函数中都可以调用;而函数内定义的变量则是局部变量,其生存域仅在定义其的函数内部,出了函数就会被释放.函数虽然可以调用读取全局变量,但并不能在函数中随意修改全局变量的值.如果试图在函数内部修改外部的全局变量,可以用global关键字的方式来强行修改.全局声明要求其必须在使用前出现,并且相应的变量也必须和函数位于同一个文件中,因此这并不适合于多文件编程,并不推荐使用.

```
name='David'
def fun():
    global name
    name='Guide'
    return name
test=fun()
```

值得注意的是,与C不同的是Python函数并不是传递变量值的副本,而是传递变量引用的副本,因此对于可变类型的参数可以直接在函数内修改,至于不可变类型需要利用局部变量的方式来重新赋值修改.

```
def fun(a):
    a.append(1)
```

事实上,如果修改值和重新分配变量名称是存在一些差异的;修改值可以直接利用自带方法就地修改,而如果重新通过设置同名局部变量赋值的方式,他并不会影响外层全局变量.

```
def bar(items):
    items=[4,5,6]
    b=[1,2,3]
    bar(b)
    print(b) # [1,2,3]
```

## 3.3 Error Checking

Python不对函数参数类型或值执行类型检查或类型验证.函数将处理与函数语句兼容的数据类型.

```
def add(a,b):
    return a+b
add(3,4) # 7
add('Hello',' World') # 'Hello World'
add('3','4') # '34'
```

异常用于在程序发生错误的时候将错误信息发送给程序.除了程序本身运行中可能导致异常信息出现,也可以利用raise方式来手动引发异常.

```
def fun(a,b):
    if isinstance(a,str):
        raise RuntimeError(f'{a} is not right type'}
```

类似于C/C++,Python也可以通过try-except代码块来捕获程序抛出的异常,值得注意的是,异常的传播并不是从内层一直传播到外层的,而是如果出现第一个匹配的except,异常的传播就会停止,并不会继续向外传播.

```
def foo():
    raise RuntimeError('Test')
def bar():
    try:
        foo()
    except RuntimeError as e:
        pass
def spark():
    try:
        bar()
    except RuntimeError as e:
        pass
    # 如果调用spark函数,那么他的异常其实是会被bar里面的except捕获后,终止程序
```

我们可以发现这个except捕获异常的时候同时还赋值了变量e,其实这是捕获异常的同时,将异常实例对象赋值给了变量e,但是我们如果将e打印出来,其实和字符串的表示类似.因此我们把try-except语法块的通用形式给出,

```
try:
    statements
except Error as e: # 捕获异常信息
    statements # 处理异常信息的语句
    statements # 完成异常捕捉和处理后继续执行语句
```

如果一个函数或一个语句块中可能抛出多种异常,显然我们可以不断堆叠except代码块的方式来捕获多种异常.

```
try:
    statement
except LookupError as e:
    statement
except RuntimeError as e:
    statement
except IOError as e:
    statement
except KeyboardInterrupt as e:
    statement
```

但是这样的话会将代码的长度毫无意义的增长,因此可以将相同处理方式的异常写成更为紧凑的形式.

```
try:
    statement
except (IOError,LookupError,RuntimeError) as e:
    statement
```

这里利用e来存储抛出的异常实例,如果不用e来存储相应异常实例,容易导致虽然出现异常,但并不能显示得出异常的原因.而且这样的异常元组的方式只能应用于对异常的相同处理,如果对不同的异常有不同的处理方式,则不可以使用这类方法.

如果并不确定程序运行中会出现什么类型的异常,是可以通过使用Exception的方式来捕获全部的异常,但这并不适合于程序中大规模使用,因为他虽然能够捕捉所有的异常但并不知道错误原因,因此不利于程序的后期修改.

```
try:
    statement
except Exception as e: # 虽然他可以捕捉所有的异常信息,但通过赋值e使得异常信息被存储下来,从而有利于后期检查
```

但对于异常捕捉建议遵从以窄捕捉为主,只捕捉自己能够处理的,对于不能处理让外层处理或尝试避免.

由于异常只会被第一次匹配的except捕捉,而不会继续向外传播,我们可以通过raise函数继续引发异常,让外层except继续捕捉异常.

```
def bar():
    try:
        foo()
    except RuntimeError as e:
        raise
```

这里raise后并没有接参数,实际上就是直接将已经实例化的变量e向外层抛出,继续由外层except捕捉.

对于某些重要资源管理(如文件,线程池,CPU资源占用等可能出现死锁现象的资源),即使抛出异常需要仍然利用某些关闭措施,来结束资源占用.故而finally代码块会要求程序无论是否出现异常,无论异常的类型都需要在最后一步执行.

```
try:
    fun()
except Exception as e:
    pass
finally:
    ending statement # 释放系统内部空间
```

## 3.4 Modules

模块可以认为是命名值的集合,换言之,模块其实和C++中的命名空间十分类似.模块中包含模块文件中定义的全局变量和定义的函数.当我们导入模块之后,如果需要调用模块中的命名值,只需要将模块名作为前缀调用即可. 在不同的模块中,是允许存在有相同名称的全局变量或函数的,对其调用如下所示

```
# foo.py
x=1
def foo:
    pass
# bar.py
x=1
def bar:
    pass
# main.py
import foo
import bar
print(foo.x)
print(bar.x)
```

因此,从上面的示例代码中,我们可以发现模块其实是互相独立的,所以模块其实可以做一些代码的命名上的隔离.但是上面我们导入模块的方式是利用import,如果我们采用下面的方式导入x,那他的隔离特性就会被我们破坏:

```
from foo import x
from bar import x
```

如果是这样的话,x只会是bar模块中的值,并不会出现前面的隔离情况.

在程序导入模块的时候,模块的源文件会被完整执行到文件末尾.因此,如果源文件中存在某些全局范围下可执行的语句,在导入模块的同时这些语句会被执行.模块的命名空间中存储的是模块源文件中定义的全局变量以及函数文件.但是这里我们需要明确的一点是,这里存储的全局变量指的是在文件末尾的依然存在的全局变量,对于代码中间释放的全局变量则不会存储.

导入模块的方式如下所示

```
import modules # 导入模块
import modules as nickname # 导入模块并重命名
from modules import func # 导入模块中的特定函数
```

这里有一些注意的点:模块只会在程序中导入一次,重复import相同的模块并不会重复执行模块的源代码,重复导入只会返回对之前加载模块的引用.因此,在jupyter或交互模式中,如果我们先导入了模块,再修改模块的代码,程序并不会做重复导入,我们需要重启解释器内核.

Python中用sys.module来存储已经加载模块的字典,sys.path则是用来存储Python查找模块时的参考路径列表,值得注意的是当前工作目录总是最优先的.

```
import sys
print(sys.modules.keys()) # 存储已经加载模块的键值
print(sys.path) # 存储Python查找模块时的参考路径列表
```

这里的查找参考路径可能没有包含希望的模块,因为sys.path是一个列表,所以我们可以用append的方式把期待的路径添加进去.当然也可以通过环境变量的方式添加到搜索路径.

```
import sys
sys.path.append('/project/foo/pyfiles')
```

但一般来说并不推荐自己手动修改搜索路径,除非某些异常需要手动导入路径.

## 3.5 Main Module

Python和C/C++不同,他没有主函数或方法;但是Python具有主模块,主模块则是第一个运行的源文件.因此可以认为提供给Python解释器的文件就是主模块,在这个主模块文件调用的其他的模块则不是main模块.这里为了验证文件的调用形式,可以__name__来判断:

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

上面的代码表示如果文件以主模块文件的形式那他可以运行if条件的语句.如果不是以主模块文件的形式运行,那他的__name__变量为模块名.

任意的源文件都可以以主模块或者库模块的形式运行或导入调用.模块的常用结构如下所示:

```
import module
variable
def func():
    pass
def foo(x):
    pass
if __name__=='__main__':
    statements
```

python不止可以利用IDE运行,也可以用命令行中调用解释器,所以和C/C++一样,python可以使用命令行参数,如下所示

```
python source.py
python source.py data1.csv
```

python的命令行参数会以文本字符串的形式存储在sys.argv中.并且sys.argv的长度至少为1,因为他的第一个元素是希望运行的python源文件名.其余的参数会存储在sys.argv[i]中,其中i大于等于1.

和C一样的,python的sys模块中也存储着输入输出和错误文件的变量,如下所示

```
import sys
sys.stdout # print输出的文件位置
sys.stderr # 错误和traceback输出的文件位置
sys.stdin # 输入文件的位置
```

同样,标准输入输出也可以通过重定向的方式来重新把输入输出的位置移动到期待的位置.

```
python prog.py > text.txt
cmd1|prog.py|cmd2
```

第一个语句会把prog.py的输出重新输出到text.txt上;第二个语句则是cmd1的输出会作为prog.py的输入,而prog.py的输出则会导出给cmd2.这里我们用prog.py直接可以运行,并没有调用解释器.这是因为我们在python文件的开头调用了#!命令,如下所示,

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

系统在读取到这一行指令的时候,会自动搜索用户环境变量中的python3解释器并调用.需要注意的是这个命令在Unix环境下生效,在Windows下效果并不是很好.而且这里如果希望可以让python源文件自动执行,在执行之前需要用chmod命令给他赋予可执行权限

```
chmod +x source.py
```

对于命令行编译python文件,我们还可以手动修改命令行的环境变量.但这个并不是python的语法,而是shell语法.但对于某些需要移植的程序而言,是十分重要的,因此我们在此对他加以介绍.

```
setenv variable value
setenv PATH /usr/local/bin:$PATH
```

第一条式子是设置环境变量的通用式,他只对当前shell进程和子进程生效,如果关闭了当前的shell进程,那么这些环境变量就会恢复默认值.第二个则是一个例子,我们发现他赋的值是并不是一个单独的值,而是在已有的PATH前面加上了一个环境,这里的:是用来分隔环境目录的,我们比较常用的就是在PATH前面加上一个值,如果把:放在$PATH后面那就是在已有的PATH后面加上一个环境.但这里我们不推荐使用

```
setenv PATH /usr/local/bin
```

这个语句会使得在这个shell里的环境变量被完全覆盖可能会有一些意想不到的错误.python中提供了一个字典来存储环境变量,如下所示

```
import os
os.environ # 字典
```

Python的程序退出除了正常运行完成,就是通过异常抛出的方式.给出如下的异常抛出方式

```
raise SystemExit
raise SystemExit(exitcode)
raise SystemExit('Information statements')
```

这里的exitcode只有零值的时候表示程序正常执行,对于非零值,都表示程序运行出错.除了通过raise抛出异常,还可以选择使用python的sys模块的exit方法实现.

```
import sys
sys.exit(code)
```

## 3.6 Design Discussion

本节有个比较有趣的程序的类型推断风格:鸭子类型.他一般用于动态语言或某些静态语言(Golang).静态语言的特点是在程序执行之前,代码编译时就可以知道所有变量的类型和方法返回值类型等.因为静态语言声明变量需要附带类型信息, 其是一块内存区域,

```
#include<stdio.h>
#include<string.h>
int main()
{
  int x=10;
  x="ss";
}
```

例如上面这个代码在代码编译阶段就会报错.静态语言的优点是代码结构非常规范,便于调试,但有时候会显得很罗嗦.

动态语言则是只有程序运行到这一行,程序才知道变量的类型.变量不需要在一开始声明变量的类型,本身也不会携带类型信息,他只与赋值的对象有关.其优点在于方便阅读,不需要写很多类型相关的代码,但其缺点在不方便调试,如果命名不规范容易出现阅读困难,不利于理解. 而鸭子类型指的是如果一个动物走起路来像鸭子,叫声也像鸭子,那么他就是鸭子.这样说来可能很抽象,其实鸭子类型就是动态类型语言的一种设计风格.一个对象的特征不是由父类决定,而是通过对象的方法决定.也就是我们并不关心对象的类型是什么样子的,我们只关心他的方法行为是什么结果.例如如下的代码,

```
def max(a,b):
    if a>b:
        return a
    else:
        return b
    max(2,3)
    max('2','3')
```

上面的代码他都可以正确的比较整型和字符类型,对于这个函数而言,我们不关心输入的类型是什么,我们只关心是否能够运行,结果是否正确.

# 4 Classes and Objects

## 4.1 Introducing Classes

面向对象编程(Object Oriented Programming)指将代码用对象组合的方式拆分.其中对象中包含数据(属性)和行为(应用在对象的方法).Python提供了class语句用来定义新对象,

```
class Point:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def move(self,dx,dy):
        self.x+=dx
        self.y+=dy
```

这里Python作为动态语言,与C++最为不同的一点是,Python的类属性可以在程序运行中随意添加,但是很不推荐这么做,会让程序显得十分杂乱.

在进行后续的讨论之前,我们先介绍一下Python中类的两种属性:实例属性和类属性.

1.实例属性主要是在__init__初始化函数中利用self.name的方式定义.其特点为对象实例的数据独立,也就是不同的实例是不能互相调用对方的数据;本质上实例属性是对象的一个字典项,默认是存储在__dict__里面的;变量类型不固定,可以在程序运行中动态修改.

```
class Player:
    def __init__(self,x,y):
        self.x=x
        self.y=y
        self.health=100

    def move(self,dx,dy):
        self.x+=dx
        self.y+=dy
```

```
    def left(self,amt):
        self.move(-amt,0)

    def damage(self,pts):
        self.health-=pts
```

2.类属性.这种属性是直接写在类体定义里面,这种属性不属于某个特定的实例,而是属于类的本身.在初始化函数中,调用类属性的方式与前面的self.name不同,而是需要用class.name的方式调用.

```
class Player:
    counter=0
    def __init(self,x,y):
        self.x=x
        self.y=y
        Player.counter+=1
```

类属性可以在任何实例化对象中调用.其特点是所有实例对象都可以调用同一个值;类属性可以被任何实例访问,但是实例属性是可以命名与类属性同名的变量,这样的话他就会覆盖类属性,无法在该实例中调用相关类属性.

这里我们简要提到一下C++的类结构,与上述Python的类结构进行对比.

```
class MyClass{
public:
  int a;
  double b;
  static int counter;
}
  int MyClass::counter=0
```

这里的静态成员变量不属于对象,而属于类本身,任何对象都可以调用其值,这个和刚刚介绍的Python的类属性一样.而且在类结构中所使用的static int counter语句只是一个变量声明,并没有分配变量内存空间.因此需要在类外做一个变量定义和初始化语句.值得注意的是对于静态成员变量必须要放在类外做定义初始化.C++类的特点为类型固定,其内部变量类型和数量均在编译时确定;访问受限由private/protected/public控制;静态成员共享.

在上面的C++类结构中,其可利用private做代码封装,类外无法随意访问private成员.而Python中并没有真正意义上的private变量,我们前面定义的类成员都可以在任意情况下被调用.但Python中提供了另一种解决方法

```
class BB():
    def __init__(self,name):
        self.__name=name

    def get_name(self):
        return self.__name
```

这个__开头的变量并不能通过类.__name的方式调用,下面的代码可以展示

```
b=BB('bb')
d=BB('dd')
b.__name='cc'
print(b.__name) # 'cc'
print(b.get_name()) # 'bb'
print(d.get_name())
print(d.__name) # 报错
```

这是因为虽然我们声明初始化的是__开头的变量,但在Python的执行中会将其改变为_类名__属性名的方式,

```
print(d._BB__name)
```

所以其实他还是可以被调用的,只是他不能通过用定义的名字调用而已.

我们为了后续的代码编写的规范,我们提供一个stackoverflow上提及的常用的程序规范.

1. __foo__: 用于Python内部的名字,用来区别其他的用户自定义命名
2. _foo: 约定指定变量是私有的,不应该用from module import 的方式来导入,其余性质与公用变量一致.
3. __foo: 这个是Python程序中具有真实含义的作用,Python会将其变成_类名__name的变量名.

类可以视作一组对实例执行各种操作的函数.实例则指代码中实际操作的对象.实例方法指对对象实例的方法.如果实例方法中需要处理对象中的属性,那么对象本身一般以第一个参数的方式传输.虽然我们常见的传输对象本身使用的是self,不过实际上这个只是约定俗成的约定,可以用其他的.这是因为类只是创建属性的容器,而不是变量的查找域,换言之,虽然我们在类当中定义了某些属性,但是我们并不能在实例函数中直接使用变量名的方式调用,我们应该选择用类.变量名的方式调用.

## 4.2 Inheritance

继承则可以对现有的对象做特殊化修改.

```
class Parent:
  pass
class Child(Parent):
  pass
```

这里我们称Child类为子类,派生类.Parent类为基类,超类.

继承可以认为是对曾经代码的扩展,一般用于添加新方法,重新定义已有方法或者添加新属性.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
    def cost(self):
        return self.shares * self.price
    def sell(self, nshares):
        self.shares -= nshares
class Mystock(Stock):
    def panic(self):
        self.sell(self.shares)  # 添加新方法
    def cost(self):  # 重新定义已有方法
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

上面的代码中提到了super.method().这个super()表示这个子类的父类,因此super.method()实际上就是调用父类的方法,可以用这种方法调用已有方法的不同版本.

我们介绍在子类中定义新属性的方式,

```
class MyStock(Stock):
    def __init__(self,name,share,price,factor):
        super().__init__(name,share,price)
        self.factor=factor
    def cost(self):
        return self.factor*super().cost()
```

首先Python的变量必须初始化,因为他是一个动态语言,如果存在一个变量但他没有被赋值,程序是不可以为他分配内存空间的,所以我们如果希望在子类中添加新属性,我们其实需要重写子类的初始化函数,这里我们不对父类初始化的方式修改,就可以用super().__init()的方式来初始化父类属性.

继承可以认为是组织某类相关对象的架构方式.

```
class Shape:
    pass
class Circle(Shape):
    pass
class Triangle(Shape):
    pass
```

继承的作用一般是用于构建可重用,可扩展的代码框架.如先定义一个类框架,也就是基类,他可以提供一个通用的方法,通过继承基类的方式,我们对子类做特殊化定制的部分.

类似于基本类型,我们也可以用isinstance的函数来判断类的关系.

```
class Shape:
    pass
class Circle(Shape):
    pass
c = Circle()
print(isinstance(c, Circle))   # True
print(isinstance(c, Shape))    # True
print(isinstance(c, object))   # True
print(type(c) is Circle) # True
print(type(c) is Shape) # False
print(type(c) is object) # False
```

从上面的代码中,我们可以发现isinstance(object,Class)判断obj是否为class或其任意子类的实例.type方法则是用于精确判断类的属性,他不会考虑类的包含关系,而isinstance是一种判断类的包含关系.isinstance方法用于判断对象是否可以被视作某个类型.

子类存在如下的Liskov Substitution Principle(LSP,里式替换法则):子类对象必须能在任何需要父类对象的地方被替换使用,而不破坏程序的正确性.

如果这个类没有父类,其实可以认为object为他的父类.

```
class shape(object):
    pass
```

object可以视作所有对象的父类.

## 4.3 Multiple Inheritance and MRO

继承是面向对象编程中用于建立类型层次关系的机制.子类(subclass)继承父类(superclass)的属性和方法;子类对象可以被当做父类对象使用.

```
class Parent: pass
class Child(Parent): pass
```

其在程序设计中的作用:代码复用;抽象公共接口;建立类型约束.

多重继承则指的是一个类可以同时继承自多个父类.

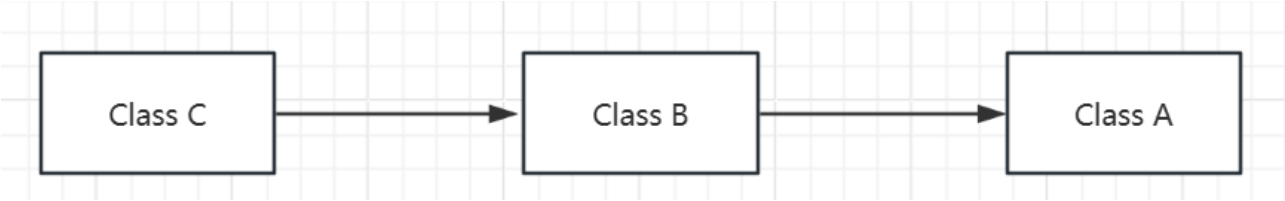
```
class A: pass
class B: pass
class C(A,B): pass
```

上述代码表示类C继承类A和类B.类C同时具有类A和类B的接口和行为.多重继承通常用于组合多个相互独立的功能,而非单一的层级抽象.其父类的特点为结构简单;状态较少;不表达本质类型,而表达附加类型.

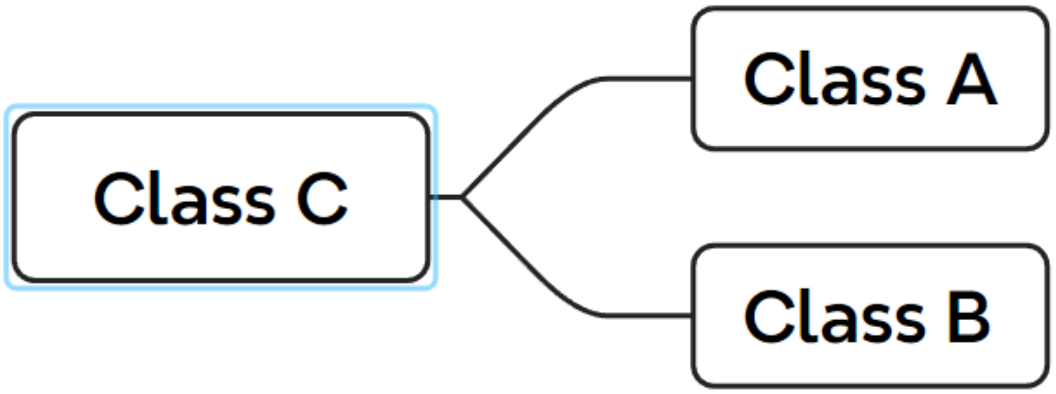
虽然我们前面这么介绍多重继承,但是实际应用中多重继承的继承结构十分之复杂,我们将其简要分为下面三类:线性结构,树状结构,菱形结构.

线性结构,本质上来看其实就是单继承结构.他的优点在于方法查找路径唯一,不存在父类歧义或父类冲突.单继承强调本质类型,结构清晰,语义稳定,行为可预测.代码如下

```
class A: pass
class B(A): pass
class C(B): pass
```



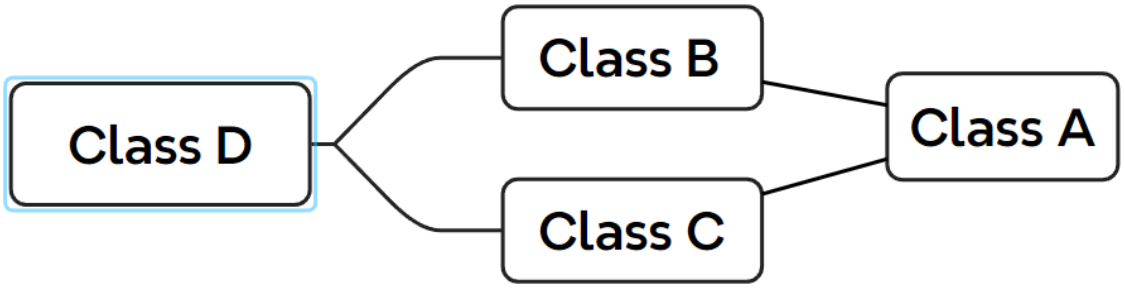

树状结构,与前面的单继承不同,树状结构源自多重继承.Mixin是Python中常用的设计模式,用来给类附加功能,而不依赖于深层次的继承体系. Mixin是一种特殊的类,其只提供某种功能或方法,不单独表示一个独立对象类型,其主要目的是混入其他类中增强功能.其特点为通常不独立实例化;提供可复用的功能;组合多个Mixin类与主类形成新类. Mixin的设计原则为功能单一,一个Mixin只提供一个特定功能;不独立,不能单独实例化;组合使用,与其他Mixin类或主类组合,实现多功能组合.Mixin的实现依赖于Python支持的多继承机制.




我们给出一个Mixin的典型例子,来表示他的作用是增强类的能力.

```
class LogginMixin:
def log(self,msg):
    print(f"[LOG]:{msg}")
class AuthMixin:
def authenticate(self,user):
    print(f"Authenticating {user}")
class User(LogginMixin,AuthMixin):
def __init__(self,name):
    self.name=name
    u=User('Alice')
    u.log("User created")
    u.authenticate("Alice")
```

第三种情况也是多重继承中最为复杂的菱形继承情况,我们先给出他的继承图结构如下:




他面临的问题是类D继承自类B和类C,但是类B和类C都是继承自类A,也就是说在这一个继承体系下类A被继承了两次.所以就会如下几个问题需要解决,如果我们调用A的函数,是否需要调用两次?如果我们调用B和C中的同名函数,是会先调用B的还是C的?D的继承逻辑是否会影响B和C的继承逻辑?

为了方便,我们先给出一套菱形继承的代码.我们将基于这个代码解释上述的三个问题.

```
class A:
def foo(self):
    print("Here is a A")
class B(A):
def foo(self):
    print("Here is a B")
    A.foo(self)
class C(A):
def foo(self):
    print("Here is a C")
    A.foo(self)
class D(B,C):
def foo(self):
    print("Here is a D")
    B.foo(self)
    C.foo(self)
    d=D()
    d.foo()
```

我们希望的是在运行中类A只被调用一次,且类D不会影响类B和类C的继承顺序,并且有一个明确的方法调用顺序.在Python中这一方法解释顺序(MRO),他是存储在类中的__mro__属性中的.我们调用类D中的函数,会先在MRO中依次查找,直到第一次查找到同名函数,就会将其调用,并不会多次调用.这里我们从他的演变开始慢慢介绍其背后的C3线性化算法.

对于继承结构图,我们可以认为其实就是一个有向无环图(DAG),而MRO实际上就是对有向无环图做一个拓扑排序.因此我们很自然的想到使用DFS或者BFS算法,我们在此选用DFS算法作为我们例子,这也是比较早的合乎我们直观的方法解释顺序.其代码如下所示.

```
def old_mro(cls,order):
    order.append(cls)
    for base in cls.__bases__:
        old_mro(base,order)
    return order
```

这里我们用到了一个类的__base__属性,他会存储当前类的父类列表.整个代码结构是通过利用递归的方式,一直走到没有父类了返回.但是这个算法有一个缺陷就是如果我们对上面的菱形结构做这个方法会发现他其实导出的MRO为DBACA,A在这一解释下被调用了两次,这与我们期待的并不一致,所以我们需要对这个方法做一些改动.

我们发现上面的MRO中A出现了两次,所以呢,我们很自然的想到可以先对继承图做一个DFS算法,然后对得到的结果做一个去重操作,只保留最后出现的位置.代码如下.

```
def improve_mro(cls,order):
    mro=old_mro(cls,order)
    dmro=[]
    dmro.append(mro[-1])
    mro.reverse()
    for item in mro:
        if item in dmro:
            continue
        dmro.append(item)
        dmro.reverse()
    return dmro
```

在这个代码下我们发现他是可以把前面的菱形结构的顺序记作DBCA的,这是合乎我们前面希望的,但他其实在保证继承单调性上是存在问题,单调性也就是指的是类B继承的顺序不会因为某个子类继承了以后会发生变化.如下所示,但是这个代码并不能在Python中运行,因为他不满足单调性,但是我们这个算法并没有涉及单调性检验,所以我们只是在形式上给与一个反例:

```
class D: pass
class E: pass
class B(D,E):pass
class C(E,D):pass
class A(B,C):pass
```

这里我们可以看出B的继承顺序为BDE,C的顺序为CED,如果我们用DFS算法计算A的MRO为ABDECED,再对他进行降重发现得到的MRO为ABCED,对C而言他的继承顺序没问题,但是B的继承顺序被改了,所以与单调性矛盾.

在这些基础上,现行的Python选择使用了C3线性算法作为MRO计算的算法. C3线性化算法来自于他实现符合三种重要的性质:一致性扩展优先图(EPG),保留局部优先次序和适合单调性准则.

EPG:在一般意义下,优先图就是一张有向图,节点表示对象,边XY表示一种优先约束:X必须在Y前面,所以优先图其实就是一个偏序关系的图表示.在偏序理论中如果我们已经有了一个偏序关系,我们希望能够获得一个全序关系,且这个全序关系不会破坏原有顺序,这样的全序关系我们称之为一致性扩展,比较典型的例子就是对DAG做拓扑排序.所以EPG其实就是用有向图表示的一组优先约束,并要求寻找一个保持这些约束的一致性扩展.在多重继承中,EPG的节点是类,优先约束就是继承关系,父类要比子类出现早.C3 的目标就是在这个 EPG 上,找到一个一致性扩展作为子类的MRO.

保留局部优先次序:在一般系统设计中,表示在一个局部上下文中显式给出的顺序约定在更大的系统中必须得到尊重.这是语义一致性问题.在类定义中我们有A(B,C)这并不是一个可以随意更改的顺序,而是表示在继承关系中B的优先级高于C.因此,我们期待合法的MRO可以满足B在C前面.

适合单调性准则:在数学中,单调性指的是在某个序结构下,扩展或演化并不会破坏原有的相对顺序.在多重继承中,单调性表示子类的MRO必须是父类MRO的一致性扩展,也就是父类已经确定的顺序在子类中仍然能够成立.

综上来,多重继承的C3线性化本质上是在一个由局部优先次序和父类顺序共同诱导的优先图上,寻找一个满足单调性的一致性扩展.这个一致性扩展也就是我们想要的MRO.

C3的核心结构其实就是一个递归结构和merge函数设计.对于任意类C,他的线性化表示C+merge(父类线性化,父类列表),我们用公式的形式给出这一表示,
$L(C) = [C] + merge(L(P_1), L(P_2), \cdots, [P_1, P_2, \cdots])$ 其中$P_1, P_2, \cdots$是C的直接父类.merge函数则是一个受约束的拓扑排序过程,他试图在多个已排序的序列中,逐步选出一个合法的下一个元素.其关键步骤在于每一步中先选取第一个元素的首元素作为候选元素,判断这个元素不出现在任何其他序列的尾部,如果成立,那么他就将其加入结果并且在所有序列中将其删除,如果不成立,就看下一个元素的首元素作为候选元素,直到元素都被遍历,若仍没有合法的MRO,那么抛出异常.

```
def merge(seq):
    result=[]
    while True:
        # 把空列表去掉
        seq=[seqs for seqs in seq if seqs]
        # 如果seq序列空了,也就是没有问题完全输出了
        if not seq:
            return result
        for seqs in seq:
            flag=seqs[0]
            if not any(flag in item[1:] for item in seq):
                break
        else:
            raise  TypeError("Inconsistent hierarchy, no C3 MRO
possible")
        result.append(flag)
        # 在序列里面把他删掉
        for item in seq:
            if item and item[0] == flag:
                item.pop(0)
def C3Mro(cls):
    # 父类序列
    Base=cls.__bases__
    # 父类线性化
    BaseLinearSeq=[list(C3Mro(base)) for base in Base]
    BaseLinearSeq.append(list(Base))
    return [cls]+merge(BaseLinearSeq)
```

这里用到了一个尚未提到的代码结构for…else…他的意思是如果在for循环中出现了break,那么跳过else继续运行,如果没有break,那就进入else运行.any函数则表示如果可迭代对象iterables中任意存在一个元素为True,则返回True;如果迭代对象是空那么返回False.与之相对的是all函数,如果可迭代对象iterables中所有元素都为True则返回True;若迭代对象为空,则返回True.

检测C3算法的例子

```
class D: pass
class E: pass
class F: pass

class B(D,E):pass
class C(D,F):pass
class A(B,C):pass

print(Mro.C3Mro(A))
```

## 4.4 Special Methods

类可以定义特殊方法,这些方法对于Python是具有特殊含义的.他们一般在开头和结尾都会用__,例如我们前面常用的__init__初始化函数.在此,我们只关注几个例子,作为范例.

对象有两种字符串表示形式,一种是我们常用的str()函数,一种则是将要介绍的repr()函数

```
from datetime import date
d=date(2025,12,18)
print(str(d)) # 2025-12-18
print(repr(d)) # datetime.date(2025,12,18)
```

str()函数我们前面已经介绍过了,在此我们只是提及一下其作用是输出一些适合展示的结果形式,他主要是在Python脚本的print中调用.repr()则是用于交互模式下的输出,他会生成更详细的表达形式.repr返回对象的官方字符串表示,其设计目标在于无歧义,尽量包含重建对象所需的信息,面向开发者/调试.在理想情况下,

```
eval(repr(obj))==obj # True
```

这是一个约定,但并不是强制要求,repr是可以无法重建对象的,只需要保证对象描述的清晰即可.例如,如下的代码repr结果就是无法重建的,

```
class A:
    pass
a = A()
print(repr(a))
# <__main__.A object at 0x0000019E611CD3D0>
```

上面提到了eval函数,eval函数的完整语法是

```
eval(expression[,global[,local]])
```

这里eval执行的是表达式而不是语句,因此如果输入x=1,会报错,因为他是赋值语句.eval会将字符串做Python表达式来解析执行,并且返回表达式的值.eval中表达式的变量依赖于命名空间,其中的global和local以字典的形式提供,分别表示全局命名空间和局部命名空间.其先在局部命名空间搜索,再在全局命名空间搜索,如果二者都没有找到,就会抛出NameError异常.

```
x=10
print(eval("x+1"))
print(eval("x+1",{"x":10}))
print(eval("a + b", {"b": 2}, {"a": 1}))
```

数学运算符实际上存在如下的方法调用

| 运算符 | 特殊函数 |
|---|---|
| a+b | a.__add__(b) |
| a-b | a.__sub__(b) |
| a*b | a.__mul__(b) |
| a/b | a.__truediv__(b) |
| a//b | a.__floordiv__(b) |
| a%b | a.__mod__(b) |
| a<<b | a.__lshift__(b) |
| a >> b | a.__rshift__(b) |
| a & b | a.__and__(b) |
| a ^ b | a.__xor__(b) |
| a ** b | a.__pow__(b) |
| ~a | a.__invert__() |

容器的特殊方法:

| 运算符 | 特殊函数 |
|---|---|
| len(x) | x.__len__() |
| x[a] | x.__getitem__(a) |
| x[a]=v | x.__setitem__(a,v) |
| del x[a] | x.__delitem__(a) |

其方法调用其实是分成两步:先是通过类.方法返回绑定方法,然后再调用这个绑定方法计算结果.我们用下面的代码将其拆分开

```
s=Stock.cost # 绑定方法
s()# 方法调用
```

绑定方法其实就是调用类方法的时候忘记加上了尾随括号.因此绑定方法通常是粗心且不明显错误的常见来源或者某些难以调试的错误行为.

从上面的特殊函数中,我们其实可以知道存在一些如getattr,setattr等的函数,其中hasattr(item,iterable)是可以判断item是否在可迭代对象中. getattr的代码结构如下所示

```
getattr(object, name[, default])
```

如果object中存在名为name的属性返回该属性的值;如果不存在且提供了default返回default;如果不存在且没有提供default,抛出AttributeError.

### 4.5 Defining new Exception

前面提到的异常其实也是通过类来定义的,一切的异常都是继承于Exception,

```
class NetworkError(Exception):
    pass
```

异常一般是空类,用pass作为异常主体.因此我们可以基于此,创建用户自定义的异常层次结构

```
class AuthenticationError(NetworkError):
    pass
class ProtocolError(NetworkError):
    pass
```

## 5 Inner Workings of Python Objects

### 5.1 Objects and Dictionaries

Python的对象在系统底层很大依赖于字典构建.换言之,Python对象本质上存放在一个字典里.

在模块内部,字典会存放模块中所有的全局变量和函数.

```
# foo.py
x=42
def bar():
    pass
def spam():
    pass
print(globals())
# main.py
import foo
print(foo.__dict__)
```

我们前面提到的类的数据实际上也是存放在字典里的,定义的类其实可以看成对底层的一个封装.实例属性是存放在实例的__dict__字典中,他存放的是通过__init__函数初始化的实例属性或者代码运行中动态维护的实例名.变量名的方式.

```
class A:
    x = 1
    def __init__(self,num):
        self.num=num
                a = A(10)
                a.x = 100      # 实例属性，遮蔽类属性
                print(a.__dict__)   # 'x':100 'num':10
```

这里我们会发现实例属性里面没有初始化函数__init__,这是因为初始化函数其实是类的方法,实例是类的实例化,因此可以调用其,但其自身是没有函数的.所以从这里我们就可以知道Python为什么可以在程序运行中动态增加实例属性了,因为他的实例属性的存储实际上就是维护实例字典的过程,动态增加实例属性,只不过是修改实例字典的过程.并且每个实例都会具有自己独特的实例字典,有多少个实例就会存储多少个实例字典.

类属性和类方法都会存储在类字典里.

```
class A:
    x=1
    def func(self):
        return A.x
    print(A.__dict__)
```

其结果比较多,我们稍微对每个结果加以解释.__module__映射的是类定义在的模块名,如果直接运行得到的就是__main__;类属性的对应关系,函数会返回一个对象,后续我们会提到,不再赘述;__dict__他的返回不是一个字典,而是一个对象,我们会在下面详细解释,他的作用是定义A的实例如何拥有并访问实例字典;__weakref__则表示一个弱引用机制.

描述器(Descriptor)是Python对象模型里的一个协议,他可以控制属性访问行为,换言之,他其实就是类属性级别的访问控制器,决定你访问属性时发生什么.如果一个对象实现了以下三个方法的任意一个就可以认为其是描述器:

```
__get__(self,instance,owner)
__set__(self,instance,value)
__delete__(self,instance)
```

如果只实现了__get__,那么他是non-data descriptor;如果实现了__set__或__delete__,那么他是data descriptor.non-data descriptor可以被实例属性覆盖,data descriptor不允许实例属性覆盖.

非数据描述器的例子比较常见的就是类中的函数:

```
class A:
    def f(self):
        return "method called"
    a=A()
    # a.f调用function的 __get__,返回bound method
print(a.f())   # "method called"
# 实例属性遮蔽non-datadescriptor
a.f = "instance attribute"
print(a.f)     # "instance attribute"(覆盖了 class的f)
```

上述的例子我们可以看出他会被实例属性覆盖.

数据描述器的例子:property实现了__get__和__set__,我们会在下一节内容进一步说明.

```
class B:
    def __init__(self):
        self._x=10
        @property
    def x(self):
        return self._x
    b=B()
    print(b.x) # 10

b.__dict__['x'] = 999
print(b.x)  # 10(data descriptor优先,实例字典被遮蔽)
```

我们虽然在b中添加了实例属性x,但是Python会先查找类字典的data descriptor,关于其的访问永远优先于实例字典.

我们在此解释一下上面提到的两个对象为什么会是解释器.我们从函数出发,其代码如下,

```
class A:
    def f(self):
        return 1
    a=A()
    print(A.__dict__['f']) # function 对象
    print(A.__dict__['f'].__get__(a,A) # bound method
```

这里的A.__dict__['f]是一个function对象,function对象在底层实现了一个__get__方法,他返回的会是一个绑定对象,因此前面说的a.f其实和如下代码一样

```
A.__dict__['f'].__get__(a,A) # 绑定方法
```

这个function对象其实是一个非数据描述器,他不会覆盖实例属性.

特殊属性__dict__是描述器,他的作用是用来访问实例字典.

```
class A:
    pass
a=A()
print(A.__dict__['__dict__']) # attribute对象
```

这个attribute属性在底层实现了__get__函数,不允许随便__set__或__delete__.我们访问a.__dict__时,底层实现如下

```
print(A.__dict__['__dict__'].__get__(a,A))
```

他会返回一个实例字典,这个优先级高于实例字典,他可以控制字典的访问.他提供了安全访问实例字典的方法,可以防止破坏对象内部结构,是对象模型里最基础的描述器之一.

综上,我们可以给出基于字典的查找属性方法的代码.为了判断描述器具体为数据描述器还是非数据描述器,我们先给出如下的判断函数

```
def is_data_descriptor(attr):
    return hasattr(attr,"__set__") or hasattr(attr,"__delete__")
def is_non_data_descriptor(attr):
    return hasattr(attr,"__get__")
```

这里的hasattr函数表示如果对象有该属性则返回true否则返回false.加入描述器判断的getattr函数实现如下

```
def getattr(obj,name):
    cls=obj.__class__
    # 查找MRO顺序的数据描述器
    for base in cls.__mro__:
        if name in base.__dict__:
            attr=base.__dict__[name]
```

```
            if is_data_descriptor(attr):
                return attr.__get__(obj,cls)
            # 查实例字典
    if name in obj.__dict__:
        return obj.__dict__[name]
    # 查非数据描述器或者普通类属性
    for base in cls.__mro__:
        if name in base.__dict__:
            attr = base.__dict__[name]
            if hasattr(attr, "__get__"):
                return attr.__get__(obj, cls)
            return attr
    raise AttributeError(name)
```

可以用一些简单的例子加以测试,不再赘述.

最后,我们介绍一下__slots__属性,在前面中我们提到了对象的属性在底层是利用字典维护的,并且这个字典可以动态增加,并且每个实例都有自己的实例字典,但这样的维护在大量实例的情况下十分消耗计算资源.__slots__属性则是提供了一种静态属性表,实例并不会创建实例字典,而是只能存放制定字段的结果.

```
class A:
    __slots__ = ('x',)
    print(A.__dict__)
    print('__dict__' in A.__dict__)   # False
    print('x' in A.__dict__)          # True
    a=A()
    print(a.__slots__)
```

这样的A.__dict__返回中不会有__dict__,因为他的实例并不会创建实例字典,其通过__slots__维护一个静态属性表.

## 5.2 Encapsulation Techniques

类的使用会试图封装内部实现细节.因为类的主要作用之一是封装对象数据和内部实现细节,但是会提供一些public接口供外部调用.这一特点在C++体现的尤为明显,他通过严格的访问控制机制来实现代码的封装.但是Python中并没有提供严格的封装机制以及访问控制机制.

1. Python可以轻松地检查对象内部结构
2. Python可以任意修改对象内部数据
3. Python并没有严格的访问控制概念

Python基于变量命名形成某种约定俗成的规定,_name认为他是私有变量,虽然我们还是可以直接访问并修改这些数据.但是一般来说我们认为当我们开始在外部操作_name变量的时候,程序可能已经发生了错误,_name默认为类的内部实现,通过程序员的约定俗成维护.

```
class Person(object):
    def __init__(self,name):
        self._name=name
        p=Person('Guido')
        print(p._name)
        p._name='Dave'
        print(p._name)
```

在类中其实有一个比较大的问题,因为Python是一个动态语言,因此我们可以将属性修改为任意类型的值,但不会报错.然而,类型的变化可能影响我们后续代码的运行,因此我们需要控制这个风险.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares=shares
        self.price = price
        s=Stock('IBM',50,91.1)
        print(s.shares)
        s.shares="hundred"
        print(s.shares)
        s.shares=[1,0,0]
        print(s.shares)
```

第一种方式其实十分简单,就是在赋值之前我们做一个类型检测,并且将变量做成内部变量.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price
    def get_shares(self):
        return self._shares
    def set_shares(self,value):
        if not isinstance(value,int):
            raise TypeError("Expected an int")
        self._shares=value
```

但是他有一个问题,就是原代码中我们提取shares属性都是通过.shares来实现的,他则需要将其改成使用.get_shares()的方法来实现.如果代码量较大的时候,修改将会十分复杂.因此我们给出另一种方法来对他做实现.

在给出方法之前,我们先介绍Python中一个十分重要的机制:装饰器机制.装饰器本质上其实就是Python中的一个函数或者类,他可以在不改动原有代码的基础上使得现有函数增加某些额外功能.以函数装饰器为例,他可以应用在日志插入,性能测试等具有切面需求的场景.切面指的是如函数的进入和退出,称之为一个横切面,这类编程方式则是面向切面的编程.

我们先给出不使用装饰器的结果

```
import logging # 导入日志模块
logging.basicConfig(level=logging.INFO)# 把info放行,因为正常的是warning
def f():
    print('Here is function f')
    logging.info('function f is running')
```

但这样的话,我们在大量函数中会出现雷同的logging.info函数,因此我们可以选择写一个函数专门完成这样的重复工作,即

```
def log(func):
    logging.info('function %s is running' % func.__name__)
    func()
def f():
    print('Here is function f')
    log(f)
```

这样虽然可以避免写大量重复的logging.info函数,但是他将函数调用修改成了log(f),修改量又太多了.因此,我们其实很自然可以想到将一个函数传入之后,返回的时候也是一个函数,这其实就是装饰器的思想,通过对现有函数的包装,实现功能的添加组合.

```
def log(func):
    def deco(*args,**kwargs):
        logging.info('function %s is running' % func.__name__)
        return func(*args,**kwargs)
    return deco
def f():
    print('Here is function f')
    f=log(f)
    f()
```

这里的log其实就是一个装饰器,他将真正的函数f包裹在了函数deco中并且返回了deco.所以实际上虽然我们用的是f=log(f),这个赋值后的f他的__name__应该是deco.换言之,f其实就是函数deco,他会丢失原有的名字和文档,后面我们将给出一个方法来保留原有的内容.Python为了避免二次赋值,提供了@符号,其为装饰器的语法糖,他在定义函数时直接使用

```
def log(func):
    def deco(*args,**kwargs):
        logging.info('function %s is running' % func.__name__)
        return func(*args,**kwargs)
    return deco
@log
def f():
    print('Here is function f')
    f()
```

装饰器在Python使用极其便利,是因为Python的高度自由,如Python中的函数可以像普通参数一样传入,赋值和返回,并且允许函数的嵌套定义.

在上面的装饰器的基础上,我们还可以对装饰器再做一次封装,这次封装,可以给装饰器加上参数以扩展能力.

```
def log(level):
    def deco(func):
        def wrap(*args,**kwargs):# 保证被装饰函数 func 可以是任意参数形式
            if level=='info':
                logging.info('function %s is running' % func.__name__)
        return func(*args,**kwargs)
        return wrap
    return deco
@log('info')
def f():
    print('Here is function f')
    f()
```

除了函数装饰器,我们还可以写一个类装饰器,其具有灵活度大,高内聚,封装性等优点.同时如果利用类装饰器封装函数,我们还可以用__call__内部调用函数来使用,自动调用函数.

```
class F(object):
    def __init__(self,func):
        self._func=func
    def __call__(self):
        print('class decorator running')
        self._func()
        print('class decorator ending')
        @Foo
def bar():
    print('bar')
    bar()
    print(type(bar))
```

这里的bar已经不再是函数,而是类对象.

上面我们提到过通过装饰器包装后的函数会丢失函数的原名和注释.Python为此提供了functools.wraps,wraps其实也是一个装饰器,他可以将原函数的元信息复制到装饰器函数中,从而避免赋值后丢失信息的问题,但其实本质上还是另一个函数.

```
def logged(func):
    @wraps(func)
    def with_logging(*args,**kwargs):
        print(func.__name__+' was called')
        return func(*args,**kwargs)
    return with_logging
@logged
def f(x):
    '''does some math'''
    return x+x*x
print(f.__name__)
print(f.__doc__)
```

装饰器是允许叠用的,如

```
@a
@b
@c
def f:
    pass
f()
```

他相当于f=a(b(c(f))).Python中也提供了几种较为常用的装饰器,@staticmethod,@classmethod和@property.我们提到的第二种方法将会基于@property装饰器展开.

我们先给出第二种方法的代码实现

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
```

```
        if not isinstance(value, int):
            raise TypeError('Expected int')
        self._shares = value
```

这个代码的作用是将shares属性设置成了一个受控属性,在不改变属性访问语法的情况下,多了对赋值进行类型校验的过程.他对外表示的形式是通过shares属性,对内则是用_shares完成私有化.第一个property表示定义一个名为shares的property并且shares(self)的方法作为这个属性的取出方法,对外的表现其实仍然是一个属性.第二个@shares.setter,则是给已存在的property对象添加写入逻辑,他不会拓展新的属性,而是增强原有property的能力.这里我们需要注意的是他内部用_shares实现存储,其实不仅仅是为了封装,而是为了防止在setter装饰器中出现死循环.如果仍然使用shares,那他会在self.shares=value里不断循环并无法跳出.我们可以用如下方式验证shares其实不是一个属性

```
print(Stock.__dict__['shares'])
```

同样property装饰器也可以作用到方法上,这样可以允许我们省略括号,使得方法和属性在形式上一致.

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def cost(self):
        return self.shares * self.price
    s=Stock('GOOG', 100, 490.1)
    print(s.shares)
    print(s.cost)
```

# 6 Generators

## 6.1 Iteration protocol

在前面的讨论中,我们多次使用了容器迭代的程序,但是我们并没有深入了解过的迭代循环的底层实现,因此我们在此先介绍他的底层实现.

```
_iter=obj.__iter__()
while True:
    try:
        x=_iter.__next__()
        # statements
    except StopIteration:
        break
```

这里的obj表示一个可迭代对象,而_iter则表示迭代器对象.根据上面的底层实现来看,循环迭代的行为是基于迭代器对象.

迭代器在Python中并非一个模糊的概念,而是具有严格技术定义的对象类型.一个对象只要同时具备__iter__()和__next__()方法,并遵守停止约定,那么就是一个合法的迭代器.对于__iter__()方法而言,他会返回一个迭代器对象,一般来说都是他自己,如果不是他自己,那么其实并不是传统意义上的迭代器,但本质上没有影响.

```
def __iter__(self):
    return self
```

for会先调用iter(obj),iter(obj)则会在内部查找obj.__iter__().

__next__()返回下一个元素或迭代终止.

```
def __next__(self):
    if EndCondition:
        raise StopIteration
    return NextValue
```

其每被调用一次,都会推进一次内部状态;到迭代终止状态时,那么结束时必须抛出异常StopIteration;不能返回特殊值表示结束.

迭代器是存在一个读取状态的.

```
it=iter([1,2,3])
next(it) #1
next(it) #2
next(it) #3
next(it) #StopIteration
```

状态是会随着next读取不断前进,但是不能自动回退的.

迭代器一般来说是一次性的.

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

这其实就是iterator和iterable的关键区别

## 6.2 Customizing Iteration with Generators

生成器generator指的是一个定义迭代的函数.

```
def countdown(n):
    while n>0:
        yield n
        n-=1
```

从上面的代码中,我们意识到生成器其实是任意使用yield语句的函数.与普通的函数不同,调用生成器函数会创建一个生成器对象,但并不会立刻执行函数.

```
x=countdown(10)
print(x) # <generator object countdown at 0x0000020956043850>
```

函数只在__next__()方法调用下不断推进函数内部状态.yield产生一个值之后,会挂起函数执行,直到函数下一次调用__next__()方法恢复执行;同样在函数迭代的结束时会引发一个异常.

进一步介绍一些Generator的概念.Generator是一种惰性求值的迭代器,用于按需产生序列中的元素,并不是一次性的把所有结果存入内存,换言之,生成器其实是一个带状态的可暂停执行的函数,每次恢复时都从上次yield的位置继续执行.

生成器本身就是一个迭代器,他满足我们之前提及的迭代器协议,即__iter__()函数返回生成器本身做迭代器;__next__()函数返回下一个值或者抛出StopIteration异常.

```
gen = (x*x for x in range(3))
iter(gen) is gen        # True
next(gen)               # 调用 __next__()
```

这里我们用到了一个类似于列表表达式的东西,他其实是生成器表达式,他返回的会是一个生成器对象,我们会在后续的笔记中介绍这一用法,在此不再赘述.

我们开始介绍Generator的运行流程.首次调用next()方法,会先从函数的第一行开始执行,一直遇到yield value的语句,他会返回value,然后就是生成器的十分重要的性质了,他会冻结当前执行现场(包括栈帧,局部变量和指令指针等);后续调用next()方法,会从上次yield的下一行开始执行,直到再次遇到yield返回新值;若函数结束,则会抛出StopIteration.

```
def demo():
    x=1
    yield x
    x+=1
    yield x
    # 执行流程 next->yield 1(x=1被保存)
    # 执行流程 next->yield 2(x=2被保存)
    # 执行流程 next->StopIteration
```

yield的语义表示他会产生一个值并暂停函数执行,保存当前的执行状态.这使得Generator本质上是一个可恢复执行的协程原型.

协程原型通常指一种尚未具备完整并发调度机制但已经体现协程核心语义的语言结构.换言之,他可以实现暂停和恢复执行的能力,但是他的调度并不是通过系统完成的,而是需要用户或者库函数显式控制.协程的本质能力有三点:可挂起,可恢复以及保持执行状态,而Generator已经具备了这三个能力,因此理论上生成器已经是一个完整的协程交互模型.但是其与真正的协程还差一个调度器来实现他的自动调度.

这里我们就开始基于这个生成器,我们逐步解释python中的协程原型的实现.从生成器开始,代码如下:

```
def gen():
    print("start")
    yield 1
    print("resume")
    yield 2
    print("stop")
    g=gen()
    print(next(g))
    print(next(g))
```

这里函数可以被暂停,执行状态(局部变量和指令指针)被保存,再次调用next()时从原地恢复;但他还不是协程,他不能实现向函数的传值以及只能单向涉及从yield向外部传值.

利用send()函数给生成器提供了双向传值的能力,使得生成器具有一定协程原型的能力.

```
def gen():
    print("coroutine started")
    x = yield          # 接收外部 send 的值
    print("got:", x)
    y = yield x + 1
    print("got again:", y)
    g=gen()
    next(c) #启动协程
    print(c.send(10))
```

这一过程中我们将yield返回的值改成了一个表达式,并且提供了值双向传播的方式:外部到协程是send函数,协程到外部是yield语句.这里的send函数会传入值之后继续向后执行,类似于赋值后执行next函数.但这里我们距离真正的协程还差几个问题,没有调度器,没有并发,需要用户手动send/next.

利用协程调用协程,给出一个可组合协程的demo.

```
def sub():
    x=yield
    yield x*2
    return "done"
def main():
    result=yield from sub()
    print("sub returned:",result)
    m=main()
    next(m)
    print(m.send(10))
```

这里的第一个next在main()函数中会运行到yield from语句,并进入sub函数在x=yield语句停止;紧接着,send(10),会给x赋上10,并返回20且冻结函数状态.如果我们再用一个next,首先他会执行return done,并且在main中打印结果同时抛出终止迭代的异常.这里其实我们实现了协程的自动转发,具有了函数调用级组合性.

现代的协程原型代码如下所示:

```
import asyncio
async def coro(name):
    print(name, "start")
    await asyncio.sleep(1)
    print(name, "end")
async def main():
    await asyncio.gather(
        coro("A"),
        coro("B")
    )
    asyncio.run(main())
```

我们逐行解释上述代码,async def定义的是原生协程函数,调用他的时候并不会执行函数体,而是生成一个coroutine object,这个和我们前面介绍的Generator一样只是前者生成的是可迭代对象,而async def生成的则是可await对象.但值得注意的是coroutine object只能await一次,如果希望多次await需要进一步做封装成Task,不在此介绍.

coro函数体的第一个print函数并不会在coro("A")的时候调用,而是会在事件循环调度开始的时候才执行.asyncio.sleep(1)返回一个awaitable对象,这个对象会注册一个1s后的回调,并且将当前协程挂起,把控制权交还给事件循环.await的简单理解像是工作停止后告诉事件循环先去处理其他的任务,1s后再回来运行.下一个print则是在sleep(1)完成后,事件循环在1s后重新调度该协程的时候继续执行.

原生协程函数main()中的asyncio.gather()函数的作用,其中的coro()函数用于创建两个协程对象但并不会执行协程对象.asyncio.gather则是用于接受多个awaitable对象,并且把他们注册到事件循环,并发调度,并且返回一个新的awaitable对象.这里是用的并发调度而不是并行调度,因此本质上还是单线程协作式并发.

最后的asyncio.run(main())则是创建了事件循环,执行main()协程,并且关闭事件循环清理资源.

| 语句 | 作用 |
| --- | --- |
| async def | 定义协程 |
| await | 让出控制权 |
| asyncio.gather | 管理并发 |
| asyncio.run | 驱动事件循环 |

## 6.3 Coroutine

协程函数是使用async def定义的函数.

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

其在调用不会执行函数体,仅返回一个coroutine object,只创建了一个可等待对象(awaitable).协程对象是协程函数的返回值,本质上是一种可等待对象,他的内部保存函数体,局部变量,当前执行位置(尚未开始/挂起点).他的调度状态一般为
CREATED/RUNNING/SUSPENDED/FINISHED,用于调度分析.

Awaitable表示可等待对象,表示任意可以被await挂起的对象:coroutine object;Task;Future.await X的行为取决于X是否已经完成.await表示协程挂起点/调度点,这是协程调度中唯一的合法的协程挂起点.await X的不同行为:X结束则立刻结束,X不结束则挂起当前协程,把控制权交还给事件循环.await则是协程调度的关键. Event loop是一个持续运行的调度器,负责管理任务,调度协程执行,在协程挂起/恢复切换.他是单线程协程式调度系统,他并不是抢占式调度系统,因为asyncio只在await处挂起,并不会打断协程的运行.

```
import asyncio
#get_event_loop是在函数中读取事件循环的,如果没有则新建并绑定事件循环
loop=asyncio.get_event_loop()
#new_event_loop是新建事件循环,set_event_loop绑定事件循环
loop=asyncio.new_event_loop()
asyncio.set_event_loop(loop)
```

任务是被事件循环管理的协程包装器,

```
task=asyncio.create_task(coro)
```

他的作用是coroutine object注册到事件循环,让事件循环意识到协程的存在.事件循环只调度任务,不直接调度协程对象.Ready表示一个task当前可以继续执行,这些任务一般由未完成或不在sleep,不在I/O阻塞中的.准备队列则是事件循环内部会维护一个可运行的任务队列,如果协程运行到某个await语句时,挂起后从其中挑选一个任务执行.suspended状态表示协程执行到await时,任务被挂起,等待某个未完成对象的状态.

```
await asyncio.sleep(1)
await some_future
```

他会将当前任务从准备队列中移除当前任务,待条件满足后重新写入准备队列.对于协程调度来说,事件循环会在准备队列中选取一个准备任务将其推进执行,不抢占资源,但不保证公平分配,也不一定按照先进先出的顺序.asyncio.gather()函数则是接受多个awaitable对象,并将这些注册成任务,将其并发完成.并发并不是并行,他们仍是单线程的.

简单的调度流程:当多个Task被创建,事件循环依次推进任务,直到遇到第一个await语句.当协程遇到await处主动暂停,并将这个任务挂起,把控制权交还事件循环.我们开始逐步加深调度流程的复杂度.

```
async def f():
    print("A")
    print("B")
    print("C")
    asyncio.run(f())
    #A B C
```

这个程序中没有await挂起任务,因此代码不可以被打断,事件循环无法介入.这里用了asyncio.run()函数接受协程,只能一次性使用,并在最后会将创建的事件循环清理关闭.

单协程并只有一个await语句,其运行顺序仍然确定.

```
async def f():
    print("A")
    await asyncio.sleep(1)
    print("B")
    asyncio.run(f())
    #A (wait 1s) B
```

上面的程序中除了协程函数f外,没有其他的任务.因此await就算挂起,也无法在准备队列中找到另一个任务运行,故而他需要暂停1s后再打印B.

顺序await必然串行

```
async def f():
    print("f start")
    await asyncio.sleep(1)
    print("f end")
async def g():
    print("g start")
    await asyncio.sleep(1)
    print("g end")
async def main():
    await f()
    await g()
    asyncio.run(main())
    # f start (1s) f end
    # g start (1s) g end
```

main函数中的两个await是一个串行逻辑,f还没有结束,g根本还不会运行,以协程的方式进入事件循环,因此他的结果会是一个顺序输出.

使用gather()函数并发执行

```
async def f():
    print("f start")
    await asyncio.sleep(1)
    print("f end")
async def g():
    print("g start")
    await asyncio.sleep(1)
    print("g end")
async def main():
    await asyncio.gather(f(), g())
    asyncio.run(main())
    # fstart gstart fend gend
```

gather会同时注册多个任务,一般来说是从左到右推进,依次将其推进到第一个await,触发第一个await才会进入真正的协程调度阶段.

create_task函数在无await语句的情况下

```
async def child():
    print("child start")
async def main():
    print("main start")
    asyncio.create_task(child())
    print("main end")
    asyncio.run(main())
    # mainstart mainend childstart
```

main函数在输出start后将child()协程转换为任务后由于没有遇到挂起,因此会继续执行main,并将child放入准备队列;等main()协程运行结束,才会运行child任务.

create_task()函数加上await语句,

```
async def child(name):
    print(name, "child start")
    await asyncio.sleep(0)
    print(name, "child end")
async def parent(name):
    print(name, "parent start")
    asyncio.create_task(child(name))
    await asyncio.sleep(0)
    print(name, "parent end")
async def main():
    A=asyncio.create_task(parent("A"))
    B=asyncio.create_task(parent("B"))
    await asyncio.gather(A,B)
    asyncio.run(main())
    # Astart Bstart Achildstart Aend Bchildstart
    # Bend Achildend Bchildend
```

他的调用本质上其实是一个队列的先进先出原则,因此所以我们只需要在推演过程中记录所谓的准备队列即可.值得注意的是,如果我们这里不在main函数里面加上await,那么run只会运行完main协程直接释放掉整个事件调度,可能多跑一下A,B到第一个await.这是因为我们的run是基于main构建的,而main中没有await将其挂起,那么只会将其执行完后释放资源.因此执行顺序其实基于同一任务内的确定顺序以及await导致的不确定性运行.

## 6.4 Producer/Consumer Problems and Workflows

生成器generator是用来设置各种生产者/消费者问题和管道工具的解决方案.生产者-消费者问题是经典的并发问题之一,其基本思想是生产者负责产生数据;消费者则负责使用数据;两者之间通过某种缓冲区或者管道的方式进行数据交换.Python中的生成器则很适合实现这种模式,他们具有延迟求值的特性:生成器不会一次性返回所有数据,而是每次yield产生一个值给消费者.

```
#producer
def follow(f):
    '''生产者:从文件逐行读取数据'''
    while True:
        line=f.readline()
        if not line:
            break
        yield line # 生产数据
        #consumer
with open('data.txt') as f:
    for line in follow(f): # 消费数据
        print(line.strip())
```

这里的follow(f)是生成器,每次for循环迭代的时候,他才读取文件的一行并且返回.for line in follow(f)是消费者,每次从生成器获取数据并进行处理.yield是连接生产者与消费者的桥梁,生成器每次暂停,等待消费者获取数据.这里我们可以看出生成器避免了一次性读取文件导致的大量内存占用的问题.

生成器的另一个重要的应用是数据处理流水线(pipeline).

```
producer->processing->processing->consumer
```

生产者用于生产原始数据;加工阶段对数据进行处理或过滤,可以有多个阶段;消费者使用处理后的数据.中间阶段既是消费者(消费前一阶段的数据),又是生产者(产生下一阶段的数据).

```
# Producer
def producer():
    for i in range(10):
        yield i  # 生成 0~9 的整数
        # Processing stage: square the number
def square(numbers):
    for n in numbers:
        yield n ** 2  # 消费并生成新的数据
        # Processing stage: filter even numbers
def even_filter(numbers):
    for n in numbers:
        if n % 2 == 0:
```

```
            yield n  # 只输出偶数
            # Consumer
def consumer(numbers):
    for n in numbers:
        print("Consumed:", n)
        # 设置流水线
a = producer()
b = square(a)
c = even_filter(b)
consumer(c)
```

数据在各阶段之间以惰性迭代的方式传递,而不是一次性全部生成.

生成器的特点与优势:

1. 延迟计算: 数据在需要时才生成,适合大数据流或无限序列
2. 低内存占用: 不需要一次性存储所有数据,只保留当前迭代状态
3. 组合灵活: 可以轻松将多个生成器函数串联形成流水线;中间阶段可以修改,过滤或者扩展数据流
4. 简洁的生产者-消费者实现: 不需要额外的队列,锁等机制即可处理顺序数据流;如果需要多线程/多进程,可以在生成器基础上增加同步.

无限流处理:

```
# 无限整数生成器
def naturals():
    n = 0
    while True:
        yield n
        n += 1
        # 偶数过滤
def evens(nums):
    for n in nums:
        if n % 2 == 0:
            yield n
            # 平方计算
def squares(nums):
    for n in nums:
        yield n ** 2
        # Consumer: 取前10个平方偶数
from itertools import islice
a = naturals()
b = evens(a)
c = squares(b)
```

```
for n in islice(c, 10):
    print(n)
```

## 6.5 Generator Expressions

生成器表达式可以认为是列表解析式的惰性版本,他们的形式上基本上完全一致只是用圆括号代替方括号,不立即生成结果,只在迭代时才计算.

```
a=[1,2,3,4]
b=(2*x for x in a)
```

此时b不是列表,是一个Generator object,没有数据存储,只有生成规则.他与普通列表解析的区别为

| 方面 | 列表解析 | 生成器表达式 |
| --- | --- | --- |
| 是否立刻计算 | 是 | 否(惰性) |
| 是否占用内存 | 保存完整列表 | 几乎不占用内存 |
| 是否可以重复使用 | 是 | 否(一次性) |
| 主要用途 | 多次访问数据 | 流式计算,一次计算 |

因此这里需要提醒的是,生成器表达式只能使用一次,一旦消耗,不可以重复使用.

```
nums=[1,2,3,4,5]
squares=(x*x for x in nums)
for x in squares:
    print(x)
for x in squares:
    print(x) # 什么都没有
```

这是因为生成器内部维护一个当前位置;迭代结束后,状态耗尽;不会自动重置状态.生成器表达式的通用语法结构为

```
(expr for x in iterable if condition)
```

生成器还可以用于函数参数,

```
sum(x*x for x in nums)
sum([x*x for x in nums])
```

这里的区别在于第二个做法会先生成一个完整列表,而第一鞯是边算边加.如果出现如下的条件,建议先使用生成器表达式作为函数参数:

1. 数据只遍历一次
2. 中间结果不需要保存

3. 函数本身是消费型(如sum,max,any,all)

生成器表达式适用于任意可迭代对象,

```
a = [1, 2, 3, 4]
b = (x*x for x in a)
c = (-x for x in b)
for i in c:
    print(i)
```

他其实和前面的生产者/消费者问题一样.本质上是一样的,数据以"流"的形式经过一系列变换.

生成器的核心使用场景:

1. 用迭代而不是数据结构思考问题:生成器表示如何一步步处理数据,这对处理文件,日志分析,网络流,大规模数据十分重要.
2. 流式处理:经典例子,过滤文件的数据行

   ```
   f = open('somefile.txt')
   lines = (line for line in f if not line.startswith('#'))
   for line in lines:
       ...
       f.close()
   ```

   文件不需要一次性读取入内存,每行只处理一次,非常适合大文件.
3. 内存效率和性能:生成器的优势不是"更快算法".而是更低峰值内存,更好的可组合性,更清晰的数据流结构.

itertools模块提供的是通用的迭代模式,零内存或极低内存开销,可以无限生成数据.本质上是把常见的for-loop模板封装成函数.

| 函数 | 核心用途 | 基本用法示例 | 典型使用场景 | 重要注意事项 |
|---|---|---|---|---|
| chain(a,b,…) | 顺序拼接多个可迭代对象 | chain(a, b) | 多个数据源合并为一条数据流 | 不创建新列表,按顺序消费 |
| count(start=0,step=1) | 无限整数序列 | count(1) | 生成索引,时间步,编号 | 无限生成,必须搭配终止条件 |
| cycle(s) | 无限循环迭代 | cycle([1,2,3]) | 轮询资源,周期性模式 | 会缓存整个 s |
| dropwhile(p,s) | 丢弃开头满足条件的元素 | dropwhile(x<0,data) | 跳过文件头,前导无效数据 | 一旦失败,后续不再检查条件 |
| groupby(s,key) | 按相邻键分组 | groupby(data,key=f) | 日志按字段聚类 | 数据必须先排序 |
| repeat(x,n) | 重复常量值 | repeat(0,5) | 填充,占位,参数广播 | n=None时为无限 |

对于一些功能单一的生成器函数其实可以直接写成生成器表达式.如原函数为

```
def filter_symbols(rows, names):
    for row in rows:
        if row['name'] in names:
            yield row
```

生成器表达式为

```
rows = (row for row in rows if row['name'] in names)
```

生成器表达式和 itertools对于管道搭建而言,是这个架构的"轻量级零件",用来消除不必要的中间函数,可以提升代码可读性和组合性.

# 7 Advanced Topics

## 7.1 Variable argument functions

在我们前面介绍的函数模型中,函数的调用参数是固定的:

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

这表示函数的参数数量固定,参数名称固定,外部调用必须要严格匹配.一旦程序需求变化,那么接口就会破坏兼容性.因此为了解决这一问题,Python引入可变参数,为了满足如下三个目标:接口数量稳定,调用方式自然以及函数内部能统一处理参数集合.

位置可变参数*args解决数量不确定的位置参数问题.

```
def f(x,*args):
    print("x =", x)
    print("args =", args)
    print(type(args))
    f(1,2,3,4)
    f(1)
```

所有未被显式命名的位置参数会被按顺序收集为一个元组,这样的*args是一种剩余参数捕获机制.这里我们要解释一下元组在这里的意义,因为参数在语义上是不可变输入,tuple强化了只读,结构化输入的概念,避免调用者误以为可以在函数内修改调用参数.这样的参数绑定发生在函数调用阶段,而不是函数体执行阶段.换言之,在进入函数体之前,解释器已经完成了参数数量校验,参数分流并构造出tuple.

由于位置可变参数只能表达顺序,但是实际上有很多参数本质上是行为开关,模式选择,可选策略等关键字参数.因此他们需要的是命名快+可选参数,故而python提供了关键字可变参数.

```
def g(x,t,**kwargs):
    print("x =", x)
    print("t =", t)
    print("kwargs =", kwargs)
    print(type(kwargs))
    g(2,3,flag=True,mode='fast',header='debug')
    g(5,6)
```

所有没有被显式接收的关键字参数都被打包成一个字典.字典是因为关键字其实纯天然就有键值对.

可以将上面两个可变参数综合在同一个函数内

```
def f(*args,**kwargs):
    print("args =", args)
    print("kwargs =", kwargs)
    f(1,2,3,flag=True,mode='fast')
    f()
```

这种一般出现在我们在写一个封装函数,我们并不关心具体参数,但是我们必须要把参数完整的转移给底层函数,如果我们不利用*arg/**kwargs,那么我们就要在外层函数中复制底层函数的参数列表,那么他的适用性就受到了极大限制.

```
def wrapper(*args, **kwargs):
    preprocess()
    return target(*args, **kwargs)
```

上面的函数结构表示他接受任何合法调用形式,并且保证不丢失信息.这样一来,对于不同函数输入,他具有统一的结构,从而使得结果更具普适性.

我们经常使用的是元组和字典,但是函数需要的是单独变量,需要做一个解包.

```
# Passing Tuples and Dicts
numbers=(1,2,3,4)
f(*numbers)
options={'flag':True,'mode':'fast'}
f(**options)
```

这样的数据结构在语义上等价与该函数的参数.数据结构可以直接映射为调用接口.元组args利用*args扩展为变量参数;字典kwargs也可以利用**kwargs扩展为关键字参数.

## 7.2 Anonymous functions and lambda

匿名函数是没有名字的函数.在Python中,通过lambda关键字定义.其核心特征就是用一个表达式,临时定义一个函数.Python的基本语法:

```
lambda parameter: expression
f=lambda x: x**2
f(3) # 9
```

这里需要注意的是lambda只能写一个表达式,表达式的值会自动返回,且不能写语句,只允许表达式.

lambda作为高阶函数的参数,这其实是lambda存在的根本理由.我们以列表的原地排序函数为例,对于字典列表而言,我们关键字参数函数来提供比较方式.

```
def stock_name(s):
    return s['name']
portfolio.sort(key=stock_name)
pprint(portfolio)
```

这里的关键字参数key用来表示使用比较的关键字函数.由于上面的关键字参数函数,其实只是返回一个表达式的值,因此可以使用lambda函数将上面的逻辑更加清晰.

```
portfolio.sort(key=lambda s:s['name'])
```

给几个额外的例子

```
list(map(lambda x:x**2,[1,2,3,4]))
list(filter(lambda x:x>0,[-2,-1,0,1,2]))
data=[(1,3),(4,1),(2,2)]
sorted(data,key=lambda x:x[1])
```

lambda函数的作用还可以覆盖一些简短的一次性代码逻辑.这里的一次性代码逻辑指的是只涉及输入和输出的关系,不引入中间变量.

```
sign=lambda x:-1 if x<0 else 1
sign(-3)
```

这里的if…else…是表达式,并不是语句,因此可以应用lambda函数.

lambda函数还可以作为回调函数或者临时行为的方式.这类的lambda的关键词是把行为当做参数传递.回调的思维常见的结构为

```
def apply(f,x):
    return f(x)
```

这里的f并不是一组数据,而是是一个行为.换言之,f应该是一个函数,而lambda函数在这里的角色应该是临时定义角色,然后立刻使用.

```
apply(lambda x:x+1,10)
apply(lambda x:x**2,10)
```

这里是因为回调行为并不需要关心函数的名字,只关心函数的行为.

lambda的限制在于lambda函数是需要被刻意设计的.lambda的不能做的场景如下:

1. lambda不能写多行表达式
2. lambda的表达式中不能出现变量赋值
3. lambda函数中不能出现如for/while/try/with关键字
4. lambda函数计算完表达式的值并且返回值,这里不能使用return关键字
5. lambda函数中不能包含注释和文档字符串.

非法实例:

```
lambda x:
    y=x+1 #y=x+1
    return y
```

这里的判断标准其实很简单,如果函数逻辑比较复杂,那么就不应该选择使用lambda.

## 7.3 Returning function and closures

在Python中,函数和整数,字符串一样都是对象.因此函数可以作为另一个函数的返回值,函数可以赋值给变量,函数可以作为参数传递.如

```
def add(x,y):
    def do_add():
        print('Adding',x,y)
        return x+y
    return do_add
a=add(3,4)
```

这里的add函数并没有执行加法,而是返回了一个执行加法的新函数do_add.这里的变量a指向的是函数do_add,因此可以用a()的方式来调用内部函数.

上面中我们发现do_add函数可以调用函数体以外的变量,这是内部函数引用由外部函数定义的变量.在此我们需要介绍一下python的变量作用域.Python读取一个变量名,会按照如下顺序查找:L(Local)-E(Enclosing)-G(Global)-B(Built-in).这些分别表示如下作用域

1. L(Local):最内层,包含局部变量,如一个函数/方法的内部

2. E(Enclosing): 包含非局部也非全局的变量,例如两个嵌套函数,一个函数A中包含了一个函数B,那么对于B来说A的内部变量就是Enclosing作用域.
3. G(Global):当前脚本的最外层,比如当前模块的全局变量
4. B(Built-in):包含了内建的变量/关键字等,最后被搜索


```
g_count=0 #Global
def outer():
    o_count=1 #Enclosing
    def inner():
        i_count=2 #Local
```

最后的内建变量/关键字是通过一个名为builtin的标准模块来实现,我们可以用如下代码来查看预定义的变量,

```
import builtins
dir(builtins)
```

Python中只有模块(module),类(class)以及函数(def,lambda)才可以引入新的作用域,其他的代码如if/elif/else,try/except,for/while是不会引入新的作用域,也就是这些语句定义的变量,代码块外也是可以访问的.

```
if True:
    x=1
print(x) #1
def test():
    y=10
    return y
print(y) #error
```

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,全局变量可以在整个程序范围内访问.在函数内部声明的变量只在函数内部的作用域中有效，调用函数时，这些内部变量会被加入到函数内部的作用域中，并且不会影响到函数外部的同名变量.

```
total=0
def sum(arg1,arg2):
    total=arg1+arg2
    print('Local variable is ',total)
    return total
sum(10,20)
print('Global variable is ',total)
```

从此我们可看出,虽然全局变量可以在整个文件都可以被访问,但如果函数内部定义了同名的局部变量,那么就会把他给覆盖掉.

上面我们提到了同名的局部变量会把全局变量给覆盖掉,如果我们真的想要修改全局变量,那么可以使用global关键字,但是在使用global关键字之前一定要保证这个变量出现并存在过.

```
num = 1
def fun1():
    global num  # 需要使用 global 关键字声明
    print(num)
    num = 123
    print(num)
fun1()
print(num)
```

同样如果想要修改嵌套作用域(enclosing作用域)的变量,那么就需要使用nonlocal关键字,如下所示

```
def outer():
    num = 10
    def inner():
        nonlocal num   # nonlocal关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()
```

当一个内部函数引用了外部函数的局部变量，并且该内部函数在外部函数返回后仍然存在，这个内部函数就称为闭包.所以闭包可以认为就是一个函数被封装在一个局部变量环境中.这样可以延长局部变量的生存时间,如上面的add函数,当add函数执行结束,理论上局部变量应该被释放掉,但是由于do_add还需要使用这些变量,因此Python仍然会保留这些变量在闭包环境中.我们可以用如下方式来查看局部变量在闭包中的生存方式,

```
a = add(3, 4)
print(a.__closure__)
#(<cell at 0x...: int object at ...>,
#<cell at 0x...: int object at ...>)
print([c.cell_contents for c in a.__closure__])
# [3, 4]
```

这里的局部变量并不是直接复制到函数参数里面,而是通过存放在闭包单元中等待访问,do_add通过cell来间接访问它们.闭包的关键特性为闭包保留了函数正常运行所需的所有变量的值.可以将闭包视为一个函数加上一个额外的环境,该环境保存它所依赖的变量的值.

闭包的经典用途一:延迟求值,其代码如下:

```
def after(seconds, func):
    import time
    time.sleep(seconds)
    func()
def greeting():
    print('Hello Guido')
after(30, greeting)
after(30,add(2,3))
```

他的执行逻辑是先执行add函数,将参数绑定到闭包单元并且返回闭包do_add,30s后after再执行do_add函数.所以其实是先绑定函数参数到闭包单元,然后再执行闭包函数逻辑.

闭包的经典用途二:避免代码重复.闭包本质上是生成函数的函数.

```
def make_adder(n):
    def add(x):
        return x+n
    return add
add10 = make_adder(10)
add100 = make_adder(100)
print(add10(5))    # 15
print(add100(5))   # 105
```

他的优势在于不需要写多个基本上一样的函数,参数n成为函数的内置配置.

对于闭包有一个十分细节的问题是晚绑定(Late Binding),代码如下

```
funcs = []
for i in range(3):
    def f():
        return i
    funcs.append(f)
print([f() for f in funcs])
# [2, 2, 2]
```

因为闭包捕获的是变量i的值,而不是i当时的值.因为其实这里的i并没有立刻求值,而是后续调用的时候才会去找i,这个时候循环已经结束了,所以i已经是2了.如果我们要调用i的当前值可以做如下改动

```
funcs = []
for i in range(3):
    def f(i=i):
        return i
    funcs.append(f)
print([f() for f in funcs])
```

我们给几个闭包比较经典的示例:计数器,缓存计算和模拟私有变量.

```
#计数器
def make_counter():
    count=0
    def counter():
        nonlocal count
        count+=1
        return count
    return counter
c=make_counter()
print(c()) #1
print(c()) #2
print(c()) #3
```

这里的count定义在make_counter函数中,counter闭包会调用count这个变量,并利用nonlocal关键字修改enclosing环境变量.每调用一次c(),count都会自增一次.

```
def memorized_power():
    cache={}
    def power(base,exp):
        if (base,exp) in cache:
            print('Get from cache:')
            return cache[(base,exp)]
        print('Compute now')
        result=base**exp
        cache[(base,exp)]=result
        return result
    return power
f = memoized_power()
print(f(2, 10))  # 计算并缓存
print(f(2, 10))  # 从缓存中取出
print(f(3, 3))   # 计算并缓存
```

这里利用闭包来维护一个私有的cache字典,只有通过闭包函数来访问cache字典.

```
def make_account(initial_balance):
    balance = initial_balance
    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return "余额不足"
        balance -= amount
```

```
        return balance
    def get_balance():
        return balance
    return deposit, withdraw, get_balance
deposit, withdraw, get_balance = make_account(100)
print(deposit(50))      # 150
print(withdraw(30))     # 120
print(get_balance())    # 120
print(withdraw(200))    # 余额不足
```

这里的balance作为enclosing环境变量的方式存储,他被完全封装在闭包函数里面,因此通过闭包函数返回的函数来操作其值.

## 7.4 Static and class methods

我们之前介绍了装饰器和一些Python预定义的装饰器.这些预定义装饰器是用于在类中定义中指定特殊类型的方法.一般有staticmethod,classmethod和property.其中property,我们前面已经介绍过了property,他将类中的方法伪装成属性.通过property,可以在保持对象接口简洁的同时,对属性访问进行控制.

Python中,静态方法是定义在类内部,但是不依赖实例状态或类状态的方法.与普通示例方法不同,静态方法在调用的时候不会自动接受实例状态参数self;与类方法不同,他也不会接收类状态参数cls.从实现机制来看,@staticmethod的作用是阻止函数在类属性访问时发生方法绑定,因此无论通过类还是实例访问,得到的都会是同一个普通函数对象.

```
class A:
    def normal_method(self):
        print("normal_method", self)
    @staticmethod
    def static_method(x):
        print("static_method", x)
a = A()
# 访问类属性
print(A.normal_method)       # <function A.normal_method at ...>
print(A.static_method)       # <function A.static_method at ...> ???
# 访问实例属性
print(a.normal_method)       # <bound method A.normal_method of
<__main__.A object ...>>
print(a.static_method)       # <function A.static_method at ...>
print(A.static_method is a.static_method)   # True
```

我们可以从输出中发现,类的实例化并不会将函数绑定.实际上,staticmethod可以认为就只是普通函数,只不过他在类内部定义,体现一种逻辑归属.

在工程上,静态方法经常用于实现类的内部支撑性代码,其特点是服务于类的整体行为,常以工具性或基础设施存在,不应暴露为实例的公共接口,他们往往负责实例创建,资源管理或内部协作机制.在此我们给出一些简短的示例代码以解释其应用.

```
import weakref
class Solver:
    _instance=weakref.WeakSet()
    def __init__(self,name):
        self.name=name
        Solver._register(self)
    @staticmethod
    def _register(instance):
        Solver._instance.add(instance)
    @staticmethod
    def active_instances():
        return list(Solver._instance)
s1=Solver("A")
s2=Solver("B")
print([s.name for s in Solver.active_instances()])  # ['A', 'B']
del s1
import gc
gc.collect()  # 强制垃圾回收
print([s.name for s in Solver.active_instances()])  # ['B']
```

这里我们简要介绍一下Python的weakref机制.weakref是用于创建对象的弱引用的模块,其核心特征是不增加对象的引用计数,因此不会阻止垃圾回收.

1. 强引用: 普通变量名,容器中的对象
2. 弱引用: 只用于观察对象是否存在,不延长对象的生命周期

如缓存注册表,对象生命周期跟踪等场景,我们希望的是检测对象,但不希望影响对象的生命周期,所以需要用weakref.常用代码例子

```
import weakref
class A:
    pass
a = A()
r = weakref.ref(a)
r()        # 返回 a
del a
r()        # 返回 None

class DataStore:
    _open_files = {}
    def __init__(self, path):
```

```
        self.path = path
    def open(self):
        self.handle = DataStore._open_file(self.path)
    def close(self):
        DataStore._close_file(self.path)
    @staticmethod
    def _open_file(path):
        if path not in DataStore._open_files:
            DataStore._open_files[path] = open(path, "w")
        return DataStore._open_files[path]
    @staticmethod
    def _close_file(path):
        f = DataStore._open_files.pop(path, None)
        if f:
            f.close()
```

这个类的设计意图是多个实例共享同一个底层文件对象,避免出现重复open(path),集中管理文件的打开与关闭,这是一个较为典型的类级资源池.如果类考虑的文件是全局唯一资源,明确规定谁关闭,谁负责所有使用者,例如单例文件/日志文件,但如果多实例并发使用或生命周期独立的文件读取就不适用.静态方法在此的作用是管理全局类资源,实例只请求资源,但不涉及管理,并且不会污染实例接口.

```
import threading
class Cache:
    _lock = threading.Lock()
    _data = {}
    def get(self, key):
        with Cache._acquire():
            return Cache._data.get(key)
    def set(self, key, value):
        with Cache._acquire():
            Cache._data[key] = value
    @staticmethod
    def _acquire():
        return Cache._lock
```

threading是一种用于多线程并发编程的模块,提供了在同一进程内并发执行多个控制流的能力.线程可以简单的认为是程序中的一条执行路线,一个进程里可以有多个线程,共享同一块内存空间,并同时推进代码执行.在多线程环境中,多个线程会同时访问共享数据.Lock则是一种互斥锁,是一种保证同一时间只有一个线程进入临界区的同步工具.他的基本行为:

```
lock=threading.Lock()
lock.acquire() # 请求锁
lock.release() # 释放锁
with lock:
    pass
# 进入with获得锁,退出with释放锁
```

这里可能会出现死锁的地方在于threading.Lock不可重入,也就是同一个线程不能重复获得同一把锁.也就是如果代码形式如下:

```
with Cache._acquire():
    r=Cache._acquire()
    Cache._data[key] = value
```

这样就会发生死锁现象.与Lock相反的是,RLock允许同一线程多次进入临界区,这样的话就不会发生上述的死锁现象.

```
class Connection:
    _pool = []
    def __init__(self, id):
        self.id = id
    @staticmethod
    def create(id):
        if Connection._pool:
            obj = Connection._pool.pop()
            obj.id = id
            return obj
        return Connection(id)
    def release(self):
        Connection._pool.append(self)
```

这个类的目标是维护一个可复用的对象池,优先从池中取已有对象,用完后把对象放回池中,降低创建对象的成本.用回收再利用代替反复new/delete.例如如下的代码

```
c1 = Connection.create(1)
c2 = Connection.create(2)
c1.release()
c3 = Connection.create(3)
c3 is c1   # True
```

这里的c3就是复用了c1对象的结果.他适合用于创建成本高的对象,可重复使用的资源,链接,缓冲区和临时计算对象.但是上面只是一个简单的demo,并不涉及对象的状态重置.

类方法是一种绑定到类对象本身的方法,而不是绑定到实例.他的第一个参数约定为cls,表示当前类.

```
class A:
    @classmethod
    def func(cls,x):
        print(cls,x)
A.func(x)
a=A()
a.func(x)
```

两种方法都可以,返回的类名称都是A.静态方法的本质是阻止实例化的时候方法被绑定,而类方法则是强制绑定到类对象.

类方法可以用于设计一些操作或查询类级状态的函数

```
class Counter:
    count=0
    def __init__(self):
        Counter.count+=1
    @classmethod
    def how_many(cls):
        return cls.count
```

这里的count是类属性,因此访问他的函数how_many应该自然的属于类的行为,而不归属于特定的实例,故而利用classmethod将其绑定到类上.

由于他是绑定在类上的,所以他在继承上具有较为优秀的表现,从下面的代码,

```
class Point:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    @classmethod
    def from_tuple(cls,t):
        print(cls,t)
        return cls(t[0],t[1])
class SubPoint(Point):
    pass
a=Point.from_tuple(2,3)
b=SubPoint.from_tuple(2,3)
```

此时a返回的类名是Point,而b的返回类名是SubPoint,这是因为b的类是SubPoint,他是继承父类的方法,但还是靠SubPoint去调用的.而且这样并不会硬编码类名,硬编码类名会导致程序灵活性不够.

# 8 Testing and debugging

## 8.1 Testing

Python动态特性使得测试对大多数程序至关重要.编译器的静态编译并不能帮你查到程序错误,唯一发现错误的方式就是运行代码,并确保测试了他的所有功能.

assert语句是程序的内部检查.如果表达式不为真,他就会引发一个AssertError异常.他的常用的语法是

```
assert <expression> [, 'Diagnostic message']
```

一般来说,我们不会使用他来做用户输入检测.他的作用一般是用于程序内部检查和不变式(应该始终是真的条件)

Contract Programming(契约编程)或者也称之为Design by Contract(设计契约),广泛使用断言是设计软件的一种方法,他规定软件设计者应该为软件的组件定义精确的接口规范.如

```
def add(x, y):
    assert isinstance(x, int), 'Expected int'
    assert isinstance(y, int), 'Expected int'
    return x + y
print(add(2,3))
print(add('hello',3))
```

这样的代码检查可以快速发现那些没有使用适当参数的参数.断言也可以做一些简单的代码测试,

```
def add(x,y):
    return x+y
assert add(2,2)==4
```

这样我们就可以将测试和代码放在同一个模块中.这样的优点是如果代码有错误,那么导入模块就会直接报错崩溃.这其实有点像是常见的冒烟测试,冒烟测试是一种最基础,快速的验证性测试,用于确认系统或软件在经历一次构建,部署或重大修改之后,核心功能是否处于可用状态.它的目标不是发现细节错误,而是尽早暴露致命问题,以判断是否值得继续进行后续,更深入的测试.其主要特点在于

1. 覆盖面小但关键:只测试最核心,最重要的功能路径
2. 测试深度浅:不涉及复杂逻辑,边界条件或异常场景
3. 执行迅速:通常几分钟内完成
4. 结论明确:只有通过/不通过,用于决定是否继续后续工作

除了内置的assert检查,Python还有提供了unittest模块.

```
# test_simple.py
import simple
import unittest
# Notice that it inherits from unittest.TestCase
class TestAdd(unittest.TestCase):
    def test_simple(self):
        # Test with simple integer arguments
        r = simple.add(2, 2)
        self.assertEqual(r, 5)
    def test_str(self):
        # Test with strings
        r = simple.add('hello', 'world')
        self.assertEqual(r, 'helloworld')
```

测试类必须要继承unittest.TestCase.每个测试函数都要以test开头.unittest也内置了一些断言,每个断言都对应着不同的情况.

| 断言语句 | 作用 |
| --- | --- |
| self.assertTrue(expr) | 断言expr为真 |
| self.assertEqual(x,y) | 断言x等于y |
| self.assertNotEqual(x,y) | 断言x不等于y |
| self.assertAlmostEqual(x,y,places) | 断言x和y在小数点后places位内相等 |
| self.assertRaises(exc,callable,arg1,..) | 断言调用 callable(arg1, arg2, …)时会抛出exc异常 |

除了unittest模块外,其实还有一个pytest标准测试框架,他强调简洁,可扩展,可读性强.他的特点在于

1. 零样本:测试函数可以直接用test_开头,不需要继承基类
2. 断言重写:使用原生的assert,失败给出结构化可读性极强的错误信息
3. Fixture机制:用以来注入方式管理测试资源
4. 高度可扩展: 丰富的插件生态
5. 与unittest兼容:可以运行基于unittest.TestCase的测试

pytest 会自动发现:文件名为test_*.py或*_test.py的自动执行或函数名为test_*自动调用测试.

## 8.2 Logging,error handling and diagnostics

logging模块解决的是一个工程级别的诊断信息管理问题:如何在不污染业务逻辑的前提下,记录那些对程序分析有用的信息,并且允许用户自行决定是否查看这些信息.这个之前我们其实也有一些做法,我们将其罗列在下面的表格

| 做法 | 问题 |
| --- | --- |
| print() | 无级别,无法关闭,无法重定向 |
| pass | 丢失执行信息,丢失诊断信息 |
| logging | 可分级,可配置,可集中管理 |

所以对于日志模块而言,强调的是行为(记录日志)和策略(如何输出)分离.因此logging的设计是分层的,其中有Logger,Handler,Formatter和Filter.

我们从Logger出发,讨论日志模块.logging提供了getLogger函数来返回一个logging.Logger对象,

```
import logging
log=logging.getLogger(__name__)
```

这里的__name__是模块名,每个模块都有自己的log,如果getLogger同名,那么他其实是返回的同一个对象,因为Logger是按照名字全局缓存,并且可以通过名字共享配置.一个Logger实例其实代表日志命名空间的一个节点,他并非一个临时对象,而是一个长期存在且可复用的单例式对象.这些Logger会构成一个日志的命名树.Logger的日志等级的显示方式并不一定是打印在屏幕,而是受其内部配置决定.常见的日志等级如下所示

| 级别 | 用途 |
| --- | --- |
| CRITICAL | 程序无法继续 |
| ERROR | 操作失败 |
| WARNING | 异常但可以继续 |
| INFO | 关键运行信息 |
| DEBUG | 细节诊断信息 |

一般来说,默认输出是WARNING以上的警告信息.其常见用法为

```
import logging
name=__name__
log=logging.getLogger(name)
log.debug("Debug message from %s", name)
log.info("Info message from %s", name)
log.warning("Warning message from %s", name)
log.error("Error message from %s", name)
log.critical("Critical message from %s", name)
```

上面的这些日志代码并不负责调整日志输出的形式,如果我们需要自定义一些日志配置,可以利用basicConifg函数来修改.

```
# main.py
import logging
logging.basicConfig(
    filename='app.log',
    level=logging.WARNING,
    format='%(levelname)s:%(name)s:%(message)s'
)
```

上面的代码一般出现在程序开头,用于告诉程序把日志写到哪里,写多少,怎么写.如果我们不做任何操作,那么默认的level是WARNING,输出的地方是stderr.我们也可以精确的控制某个特定的版块,

```
logging.getLogger('fileparse').setLevel(logging.DEBUG)
```

只打开fileparse日志的debug播报等级.

Logger用于接受日志事件,那么Handler决定往哪里写,Formatter决定写入的形式,Filter决定要不要写.Handler是日志的输出通道.他负责回答这个日志写到哪里,如终端,文件,缓冲区等.常见的Handler类型如下所示

| Handler | 作用 |
| --- | --- |
| StreamHandler | 输出到流(默认是stderr) |
| FileHandler | 输出到文件 |
| RotatingFileHandler | 文件滚动 |
| TimeRotatingFileHandler | 按时间滚动 |

```
import logging
logger = logging.getLogger("demo")
logger.setLevel(logging.DEBUG)
# Handler 1: 终端
h1 = logging.StreamHandler()
h1.setLevel(logging.INFO)
# Handler 2: 文件
h2 = logging.FileHandler("app.log")
h2.setLevel(logging.WARNING)
logger.addHandler(h1)
logger.addHandler(h2)
logger.debug("debug")
logger.info("info")
logger.warning("warning")
```

如果我们同时给logger配置了多个Handler.同一条日志被赋值并且依次发送给每一个满足条件的Handler.其余的RotatingFileHandler表示当日志文件满足某个条件,当前文件被封存并重命名,新的日志写入一个全新的文件.

```
from logging.handlers import RotatingFileHandler
handler = RotatingFileHandler(
    "app.log",
    maxBytes=1024,      # 单个文件最大字节数
    backupCount=3       # 最多保留多少个旧文件
)
```

其可以限制单个日志文件的大小,同时保留历史日志,他能够保存一共n+1个日志,RotatingFileHandler主要是基于文件大小做一个轮转.而对于TimeRotatingFileHandler则是基于时间做轮转.

```
from logging.handlers import TimedRotatingFileHandler
handler = TimedRotatingFileHandler(
    "app.log",
    when="D",         # 每天
    interval=1,
    backupCount=7
)
```

这里的when关键字决定轮转的时间单位,上面的when="D",interval=1,表示每天轮转一次,如果考虑when='H',interval=6,那么就是每六小时轮转一次.如果超过了backupCount,那么最旧的文件就会被自动删除.TimeRotating并不是实时记录器,他只发生在当前时间越过轮转点.

Formatter则是决定记录的日志应该是以什么形式写入.其主要的调用方式如下

```
formatter = logging.Formatter(
    "%(levelname)s:%(name)s:%(message)s"
)
```

这里面常用的字段为

| 字段 | 含义 |
| --- | --- |
| %(levelname)s | 日志级别 |
| %(name)s | logger 名 |
| %(message)s | 日志内容 |
| %(asctime)s | 时间 |
| %(filename)s | 文件名 |
| %(lineno)d | 行号 |

Formatter并不是挂载在Logger上,而是挂载在Handler上.

```
handler.setFormatter(formatter)
```

Filter则是对日志做一个筛选,考虑是否需要将这个信息写入日志,和前面提到的播报level不同的是,Filter可以基于任意的条件.

```
class MyFilter(logging.Filter):
    def filter(self, record):
        return "password" not in record.getMessage()
```

这里返回值True表示通过,False表示拦截.Filter则是既可以挂载在Logger上,也可以挂载在Handler上.

```
logger.addFilter(...)
handler.addFilter(...)
```

如果挂载在Logger上,那他会自动作用在所有的Handler上.如果挂载在Handler上,那他只会作用在这个Handler.

给一个完整的代码

```
import logging
import sys
# 基本配置
logging.basicConfig(
    level=logging.WARNING,  # 兜底级别
    format="%(levelname)s:%(name)s:%(message)s"
)
# 构造过滤器类
class IgnoreKeywordFilter(logging.Filter):
    #->表示设计上应该返回什么类型,提高可读性
    #record:logging.LogRecord表示参数类型注解
    #表示record这个参数在设计和语义上应当是一个logging.LogRecord对象
    def filter(self, record: logging.LogRecord) -> bool:
        return "ignore" not in record.getMessage()
# 创建Logger对象
logger = logging.getLogger("demo.app")
logger.setLevel(logging.DEBUG)  # Logger 总闸门
# 如果True,会调用logger的handler之后,再去调用root logger
logger.propagate = False        # 避免重复输出到 root logger
# 创建新的formatter
console_formatter = logging.Formatter(
    fmt="%(levelname)s - %(message)s"
)
file_formatter = logging.Formatter(
    fmt="%(asctime)s [%(levelname)s] "
        "%(name)s %(filename)s:%(lineno)d - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S"
```

```
)
# 创建不同的Handler
# 控制台 Handler
console_handler = logging.StreamHandler(sys.stderr)
console_handler.setLevel(logging.INFO)
console_handler.setFormatter(console_formatter)
console_handler.addFilter(IgnoreKeywordFilter())
# 文件 Handler
file_handler = logging.FileHandler("app.log", mode="w", encoding="utf-
8")
file_handler.setLevel(logging.DEBUG)
file_handler.setFormatter(file_formatter)
file_handler.addFilter(IgnoreKeywordFilter())
# 绑定 Handler 到 Logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)
def main():
    logger.debug("This is a DEBUG message")
    logger.info("Application started")
    logger.warning("Low disk space")
    logger.error("Something went wrong")
    # 这条会被 Filter 丢弃
    logger.warning("This message should be ignored")
    try:
        1 / 0
    except ZeroDivisionError:
        logger.exception("Unhandled exception occurred")
if __name__ == "__main__":
    main()
```

## 8.3 Debugging

调试的开始都是从Traceback开始.当Python程序方程未补货异常的时候,解释器就会打印Traceback(错误回溯),他包含调用栈(函数是如何一层层调用到出错位置的),精确的出错行号,异常类型和异常信息.

```
Traceback (most recent call last):
  File
"C:\Users\Liyaoda\Nutstore\1\Computer\Learned\Pratical_Python\Chapter8\demo\Ch8_Sec3.py", line 21, in <module>
    process()
  File
"C:\Users\Liyaoda\Nutstore\1\Computer\Learned\Pratical_Python\Chapter8\demo\Ch8_Sec3.py", line 19, in process
    result = normalize(data)
             ^^^^^^^^^^^^^^^
  File
"C:\Users\Liyaoda\Nutstore\1\Computer\Learned\Pratical_Python\Chapter8\demo\Ch8_Sec3.py", line 5, in normalize
    total = sum(values)
            ^^^^^^^^^^^
TypeError: 'int' object is not iterable
```

阅读Traceback的方式其实是从最后一行开始,因为最后一行是异常的直接原因,然后在慢慢向上回溯,从而可以了解错误是如何被逐层调用传播出来的.值得注意的是,异常点往往并不是错误发生的地方,而是错误暴露的地方.

如果我们直接用python解释器去处理含错误的程序,他遇到错误之后会崩溃并退出.但我们仍可以通过python -i的方式保留解释器运行.

```
python -i Ch8_Sec3.py
```

他的含义是脚本执行完,即使是因为异常退出,解释器也不会退出,当前的作用域,变量,对象仍然存在.我们可以直接现场调用一些变量来检查程序内部状态.

```
>>> x
>>> type(x)
>>> locals()
>>> globals()
```

本质上这等价于在C/C++崩溃后还能冻结现场,是一种轻量级事后调试.但是值得注意的是,-i他只能保留全局变量和定义的函数,类和模块,对于函数内部的局部变量是不能检查的.

虽然上面提到了一些调试方式,但实际上print仍然是最简单的调试方式.但是对于调试,更为建议的是选择使用repr()函数,因为repr函数返回的字符串会更为符合调试分析的精确表达.因为在调试过程中,我们更期待的是返回变量的类型,精度以及是否被隐式转换,如果直接print可能会被解释器做一些隐匿的美化,导致信息被隐藏.

上面的python -i只能检查全局变量的正确与否,我们可以用pdb来做交互式调试.在python3.7以上的版本,可以通过breakpoint()来直接调用pdb

```
def some_function():
    ...
    breakpoint()
    ...
```

这里推荐使用breakpoint()的原因是可以通过环境变量切换调试器,不强耦合pdb.加了断点之后,程序运行到这个断点,运行停止,进入REPL和调用栈上下文,然后可以逐行执行,检查变量和修改状态.我们也可以直接调用调试器来运行程序,

```
python3 -m pdb Ch8_Sec3.py
```

程序从第一行就进入调试器.我们可以在程序中提前设断点,控制执行路径,检查初始化阶段的状态.pdb常见的命令如下所示

| 命令 | 含义 |
|---|---|
| w | 查看当前调用栈（极其重要） |
| u / d | 在调用栈中上下移动 |
| l | 查看当前代码 |
| a | 查看当前函数参数 |
| p expr | 打印表达式 |
| s | 单步进入 |
| n | 单步执行（不进入函数） |
| c | 继续运行 |
| b | 设置断点 |

调试器实际上就是在时间轴,调用栈和函数局部状态三个调试维度上调试程序.设置断点的常用方式如下所示

| 命令 | 断点设置方式 |
|---|---|
| b 45 | 当前文件第45行 |
| b file.py | 指定文件的第45行 |
| b foo | 当前文件的函数foo() |
| b module.foo | 模块中的foo() |

实际上,按函数名下断点通常比行号更稳定.

# 9 Packages

## 9.1 Packages

任意一个python文件就是一个模块,这里的模块解决的是代码复用与命名空间隔离.

```
# foo.py
def grok(x):
    pass
def spam(x):
    pass
# main.py
import foo
foo.grok(2)
```

但是在大型程序中,具有大量的模块文件,如果仍然通过模块管理代码,会使得主程序被大量的模块文件隐藏,因此对于大量代码组织架构的程序中,包代替模块成为管理代码的好方式.

包实际上就是模块的集合加上层级命名空间,形式上来看,包就是一个目录以及__init__.py文件.包解决的问题是大规模代码的组织问题,模块名冲突问题,以及应用与库的结构分离问题.但如果我们只是简单的将模块文件放入包目录里面,那么程序的运行会立刻崩溃.这是因为包内模块之间的导入方式不同以及包内模块不能直接当做脚本运行.

我们常见的导入模块的方式为

```
import fileparse
```

如果fileparse模块在包目录内,那么他就会报错,这是因为此时fileparse并不是顶层模块,而应该用pack.modules的方式调用.正确的导入方式可以简单的分为两种:绝对导入,不推荐使用在包内模块的互相调用

```
from pack import module
import pack.module
from pack.module import func
```

他的缺点是将包名写死,如果我们在后续程序运行中修改了包名,那么就要逐一修改包名,比较麻烦.因此Python官方更为推荐的包内导入方式为相对导入

```
#.表示当前包
#..表示上一级包
from . import module
from .module import func
```

相对导入只能用于包内模块,不能应用到顶层脚本.

包内模块不能直接当脚本运行,

```
python porty/pcost.py
```

这是因为Python将porty/pcost.py当做孤立文件,sys.path中没有porty的父目录,所以相对导入和包结构就会全部失效.我们可以用python -m的方式来运行模块

```
python -m porty.pcost
```

Python先把当前目录加入sys.path再以模块的方式执行,包结构完整可见.或者我们也可以用顶层脚本的方式调用,此时我们需要在和包目录同级的地方创建一个python文件并用如下代码调用,

```
# print-report.py
import sys
from porty.report import main
main(sys.argv)
```

这里需要牢记脚本永远在包外,包内只放库代码.

包里面我们一开始创建了一个__init__.py,他并非一个占位文件,他的核心作用是声明这是一个包并且组织对外的API接口.

```
# porty/__init__.py
from .pcost import portfolio_cost
from .report import portfolio_report
#main.py
from porty import portfolio_cost
```

这一个顶层文件的工程意义是控制用户应该看到什么,隐藏内部实现细节和提供稳定接口