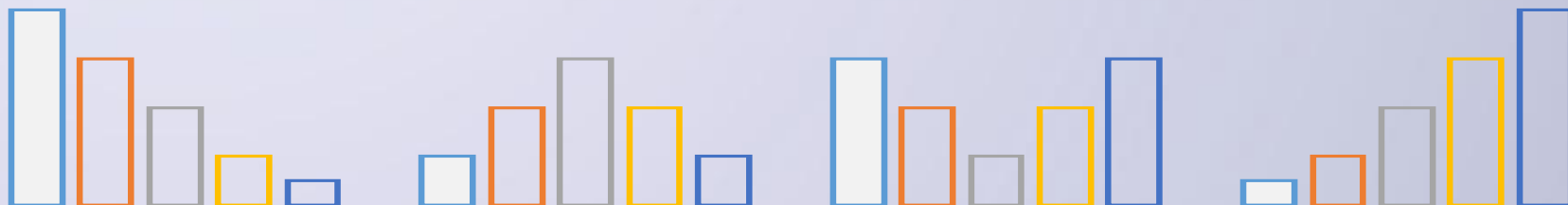
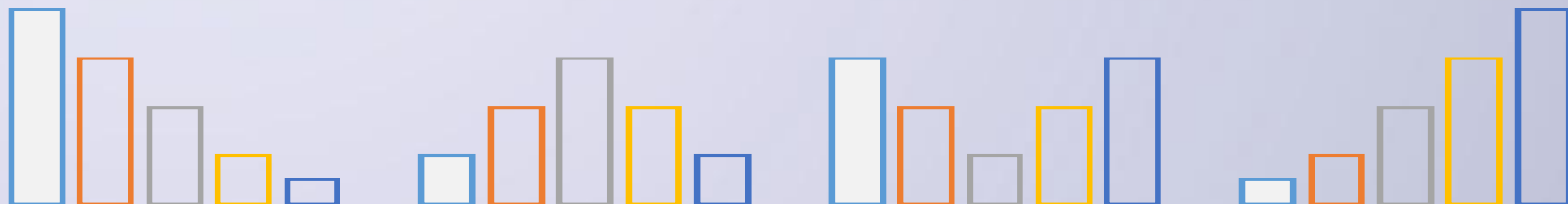


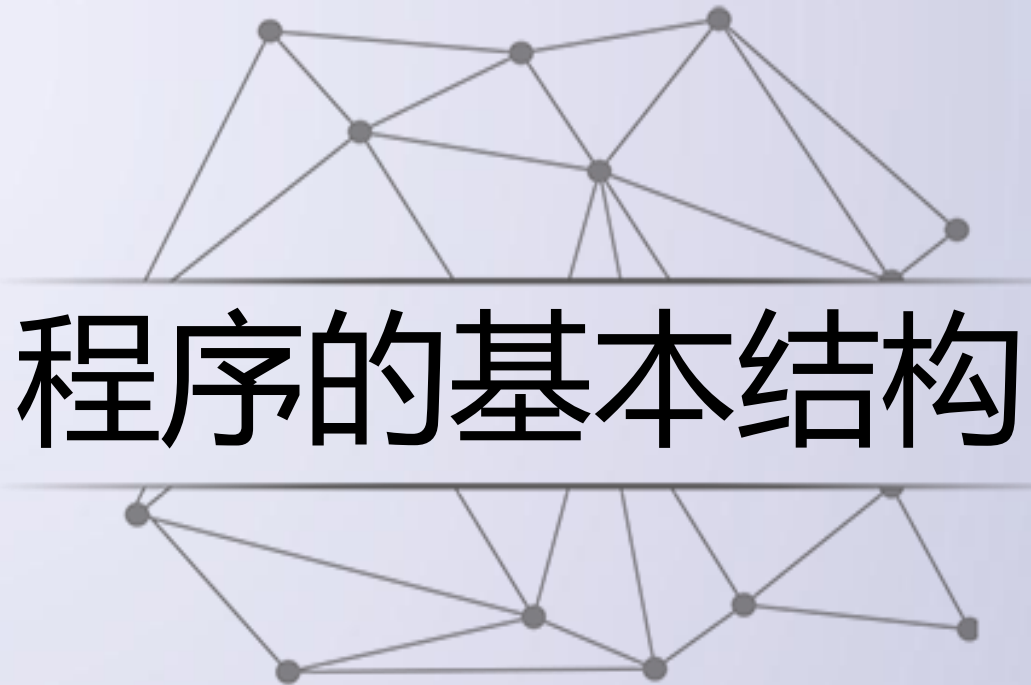
Python语言程序设计

北京理工大学 嵩天



第4章 程序的控制结构



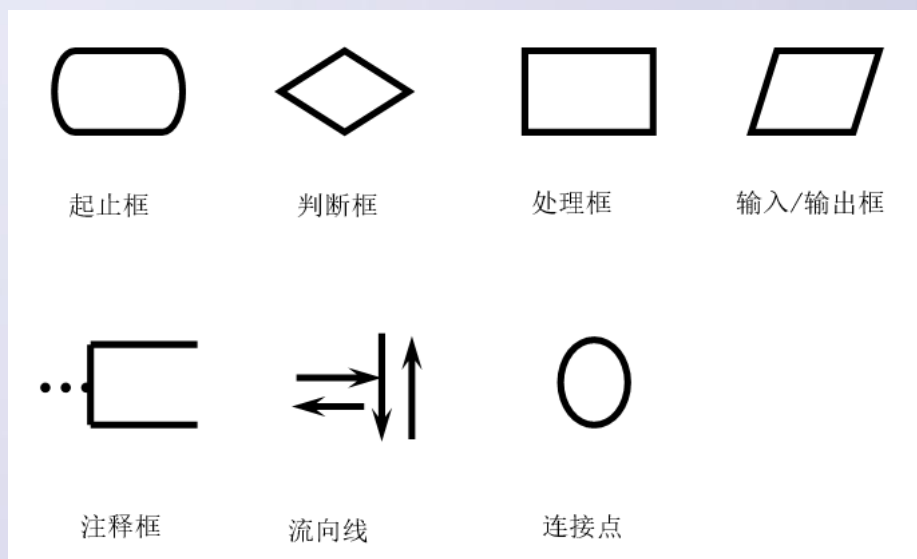


程序的基本结构

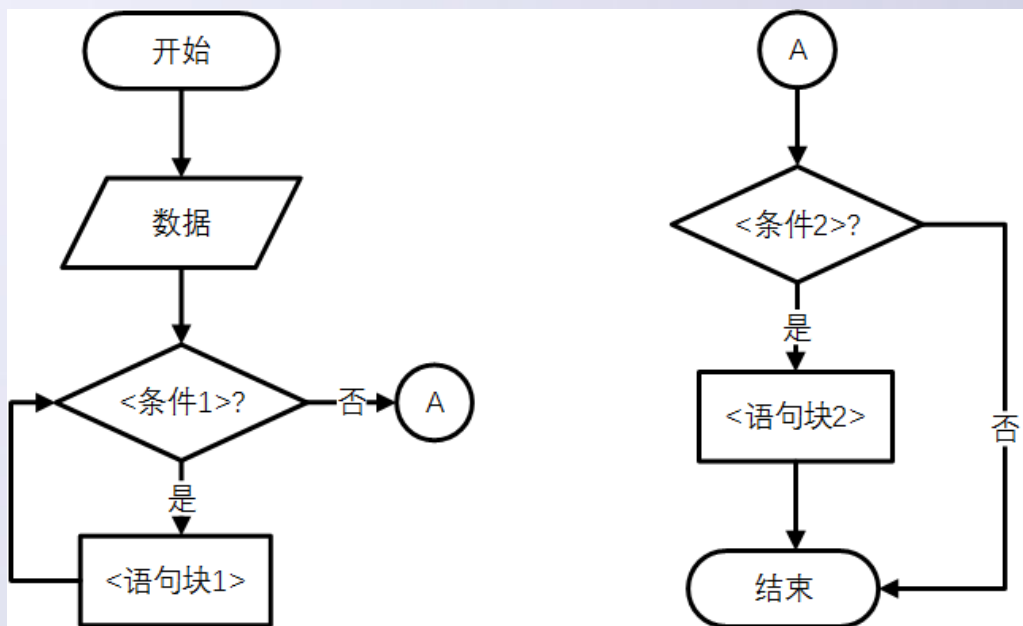
程序的流程图

程序流程图用一系列图形、流程线和文字说明描述程序的基本操作和控制流程，它是程序分析和过程描述的最基本方式。

• 流程图的基本元素包括7种



程序的流程图



程序流程图示例：由连接点A连接的一个程序

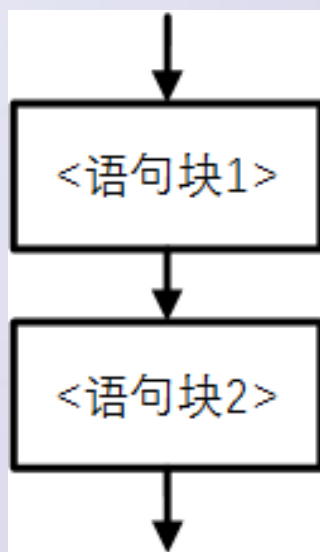


程序的基本结构

- 顺序结构是程序的基础，但单一的顺序结构不可能解决所有问题。
- 程序由三种基本结构组成：
 - 顺序结构
 - 分支结构
 - 循环结构
- 这些基本结构都有一个入口和一个出口。任何程序都由这三种基本结构组合而成

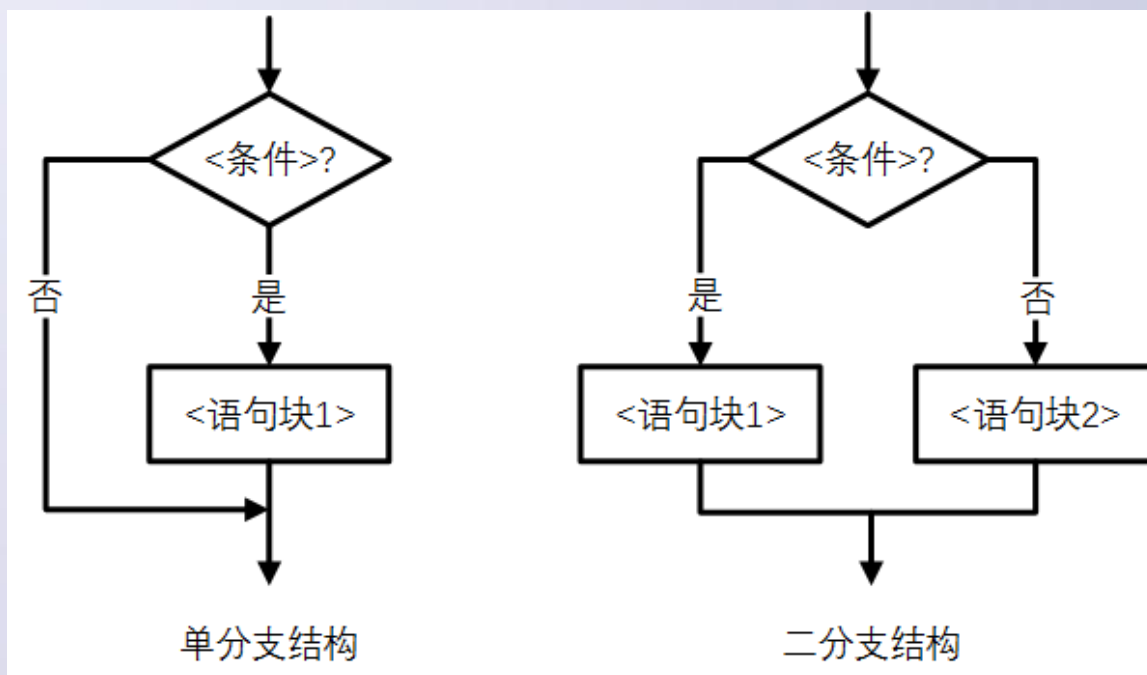
程序的基本结构

- 顺序结构是程序按照线性顺序依次执行的一种运行方式，其中语句块1S1和语句块S2表示一个或一组顺序执行的语句



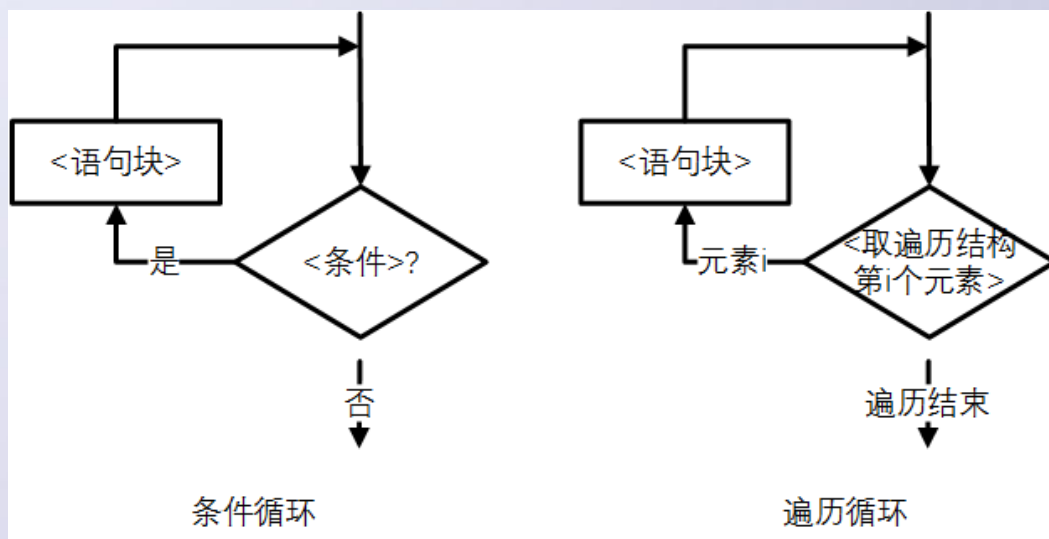
程序的基本结构

分支结构是程序根据条件判断结果而选择不同向前执行路径的一种运行方式，包括单分支结构和二分支结构。由二分支结构会组合形成多分支结构



程序的基本结构

循环结构是程序根据条件判断结果向后反复执行的一种运行方式，根据循环体触发条件不同，包括条件循环和遍历循环结构



程序的基本结构实例

对于一个计算问题，可以用IPO描述、流程图描述或者直接以Python代码方式描述

微实例4.1：圆面积和周长的计算。

输入：圆半径R

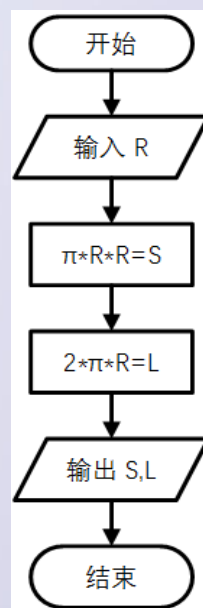
处理：

圆面积： $S = \pi * R * R$

圆周长： $L = 2 * \pi * R$

输出：圆面积S、周长L

问题IPO描述



```
1  R = eval(input("请输入圆半径:"))
2  S = 3.1415*R*R
3  L = 2*3.1415*R
4  print("面积和周长:",S,L)
```

Python代码描述

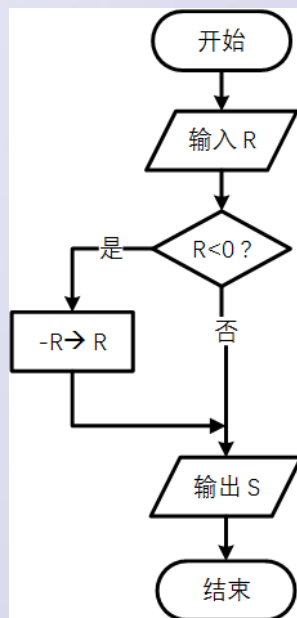
程序的基本结构实例

微实例4.2：实数绝对值的计算。

输入：实数R

$$\text{处理: } |R| = \begin{cases} R & R \geq 0 \\ -R & R < 0 \end{cases}$$

输出：输出|R|



```
1 R = eval(input(" 输入 实  
2 数:"))  
3 if (R < 0):  
4     R = -R  
print("绝对值",R)
```

(a) 问题IPO描述

(b) 流程图描述

(c) Python代码描述

程序的基本结构实例

微实例4.3：整数累加。

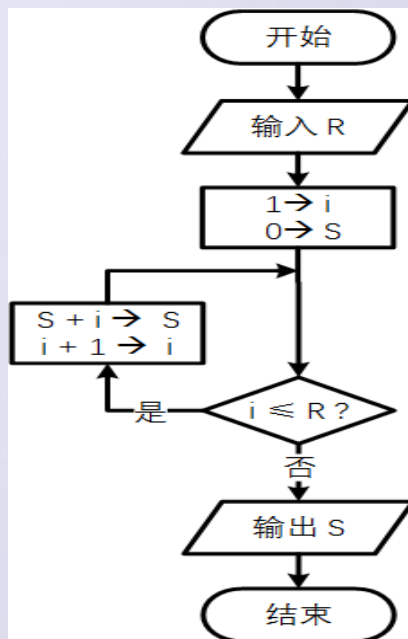
输入：正整数R

处理：

$$S=1+2+3+\cdots+R$$

输出：输出S

(a) 问题IPO描述



(b) 流程图描述

```
1  R = eval(input("请输入正
2  整数:"))
3
4  i, S = 0, 0
5  while (i<=R):
6      S = S + i
7      i = i + 1
8  print("累加求和",S)
```

(c) Python代码描述

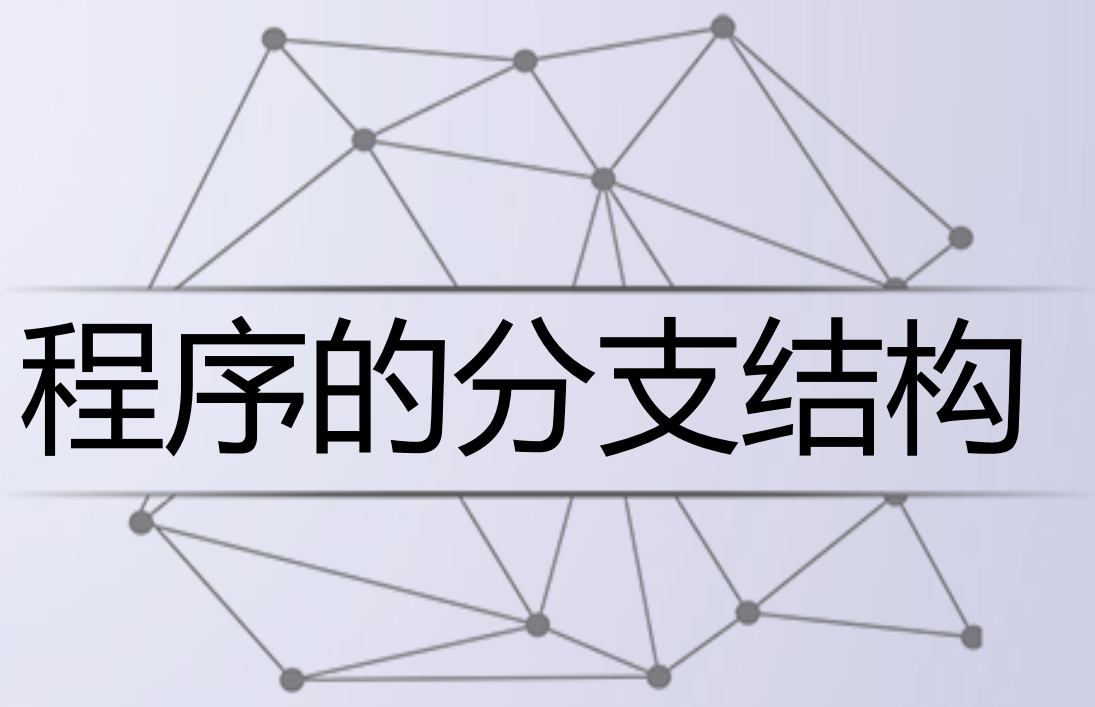


程序的基本结构实例


IPO描述主要用于区分程序的输入输出关系，重点在于结构划分，算法主要采用自然语言描述

流程图描述侧重于描述算法的具体流程关系，流程图的结构化关系相比自然语言描述更进一步，有助于阐述算法的具体操作过程

Python代码描述是最终的程序产出，最为细致。



程序的分支结构



单分支结构: if语句

Python中if语句的语法格式如下：

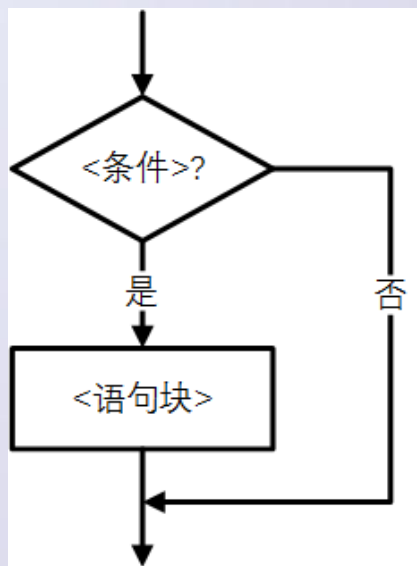
if <条件>:

语句块


- 语句块是if条件满足后执行的一个或多个语句序列
- 语句块中语句通过与if所在行形成缩进表达包含关系
- if语句首先评估<条件>的结果值，如果结果为True，则执行语句块里的语句序列，然后控制转向程序的下一条语句。如果结果为False，语句块里的语句会被跳过。

单分支结构: if语句

if语句中语句块执行与否依赖于条件判断。但无论什么情况，控制都会转到if语句后与该语句同级别的下一条语句




if语句的控制流程图



单分支结构: if语句

- if语中<条件>部分可以使用任何能够产生True或False的语句
- 形成判断条件最常见的方式是采用关系操作符
- Python语言共有6个关系操作符

操作符	数学符号	操作符含义
<	<	小于
<=	≤	小于等于
>=	≥	大于等于
>	>	大于
==	=	等于
!=	≠	不等于



单分支结构: if语句

微实例4.4 : PM 2.5空气质量提醒 (1)

输入 : 接收外部输入PM2.5值

处理 :

if PM2.5值 \geq 75 , 打印空气污染警告

if $35 \leq$ PM2.5值 $<$ 75 , 打印空气污染警告

if PM2.5值 $<$ 35 , 打印空气质量优 , 建议户外运动

输出 : 打印空气质量提醒

微实例4.4

m4.4PM25Warning.py

```
1 PM = eval(input("请输入PM2.5数值: "))
2 if 0<= PM < 35:
3     print("空气优质, 快去户外运动!")
4 if 35 <= PM <75:
5     print("空气良好, 适度户外活动!")
6 if 75 <= PM:
7     print("空气污染, 请小心!")
```



二分支结构: if-else语句

Python中if-else语句用来形成二分支结构，语法格式如下：

```
if <条件>:  
    <语句块1>  
else:  
    <语句块2>
```

- <语句块1>是在if条件满足后执行的一个或多个语句序列
- <语句块2>是if条件不满足后执行的语句序列
- 二分支语句用于区分<条件>的两种可能True或者False，分别形成执行路径



二分支结构: if-else语句

微实例4.5 : PM 2.5空气质量提醒 (2)

微实例4.5

m4.5PM25Warning.py

```
1 PM = eval(input("请输入PM2.5数值: "))
2 if PM >= 75:
3     print("空气存在污染, 请小心!")
4 else:
5     print("空气没有污染, 可以开展户外运动!")
```



二分支结构: if-else语句

二分支结构还有一种更简洁的表达方式，适合通过判断返回特定值，语法格式如下：

<表达式1> if <条件> else <表达式2>

```
1 PM = eval(input("请输入PM2.5数值: "))  
2 print("空气{}污染!".format("存在" if PM >= 75 else "没有"))
```



二分支结构: if-else语句

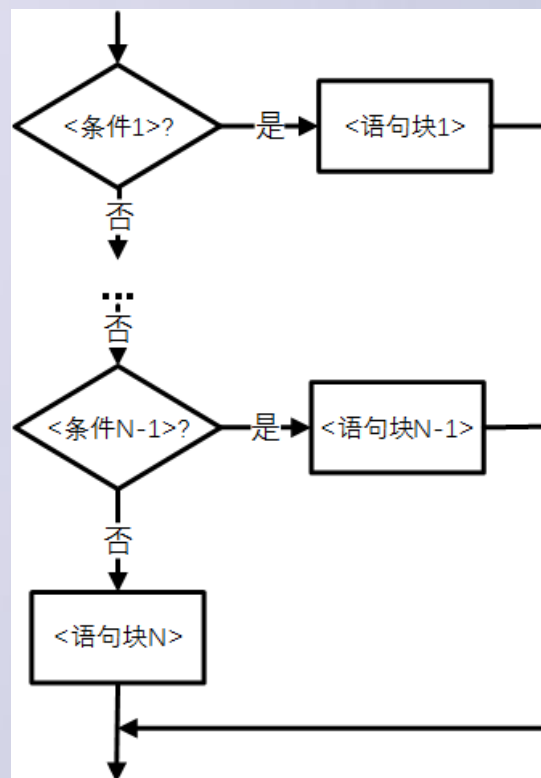
if...else的紧凑结构非常适合对特殊值处理的情况，如下：

```
>>>count = 2
>>>count if count!=0 else "不存在"
2
>>>count = 0
>>>count if count!=0 else "不存在"
"不存在"
```

多分支结构: if-elif-else语句

Python的if-elif-else描述多分支结构，语句格式如下：

```
if <条件1>:  
    <语句块1>  
elif <条件2>:  
    <语句块2>  
...  
else:  
    <语句块N>
```





多分支结构: if-elif-else语句

- 多分支结构是二分支结构的扩展，这种形式通常用于设置同一个判断条件的多条执行路径。
- Python依次评估寻找第一个结果为True的条件，执行该条件下的语句块，同时结束后跳过整个if-elif-else结构，执行后面的语句。如果没有任何条件成立，else下面的语句块被执行。else子句是可选的



多分支结构: if-elif-else语句

微实例4.4通过多条独立的if语句对同一个变量PM进行判断，这种情况更适合多分支结构，改造后的代码如下

```
1 PM = eval(input("请输入PM2.5数值: "))
2 if 0<= PM < 35:
3     print("空气优质，快去户外运动!")
4 elif 35 <= PM <75:
5     print("空气良好，适度户外活动!")
6 else:
7     print("空气污染，请小心!")
```



身体质量指数BMI

BMI的定义如下：

$$\text{BMI} = \text{体重 (kg)} \div \text{身高}^2 \text{ (m}^2 \text{)}$$

例如，一个人身高1.75米、体重75公斤，他的BMI值为24.49



身体质量指数BMI

编写一个根据体重和身高计算BMI值的程序，并同时输出国际和国内的BMI指标建议值

分类	国际BMI值 (kg/m ²)	国内BMI值 (kg/m ²)
偏瘦	< 18.5	< 18.5
正常	18.5 ~ 25	18.5 ~ 24
偏胖	25 ~ 30	24 ~ 28
肥胖	>= 30	>= 28

```
1  #e5.1CalBMI.py
2  height, weight = eval(input("请输入身高(米)和体重\
   (公斤)[逗号隔开]: "))
3  bmi = weight / pow(height, 2)
4  print("BMI数值为: {:.2f}".format(bmi))
5  wto, dom = "", ""
6  if bmi < 18.5:    # WTO标准
7      wto = "偏瘦"
8  elif bmi < 25:    # 18.5 <= bmi < 25
9      wto = "正常"
10 elif bmi < 30:    # 25 <= bmi < 30
11     wto = "偏胖"
12 else:
13     wto = "肥胖"
14 if bmi < 18.5:    # 我国卫生部标准
15     dom = "偏瘦"
16 elif bmi < 24:    # 18.5 <= bmi < 24
17     dom = "正常"
18 elif bmi < 28:    # 24 <= bmi < 28
19     dom = "偏胖"
20 else:
21     dom = "肥胖"
22 print("BMI指标为:国际'{0}', 国内'{1}'".format(wto, dom))
```

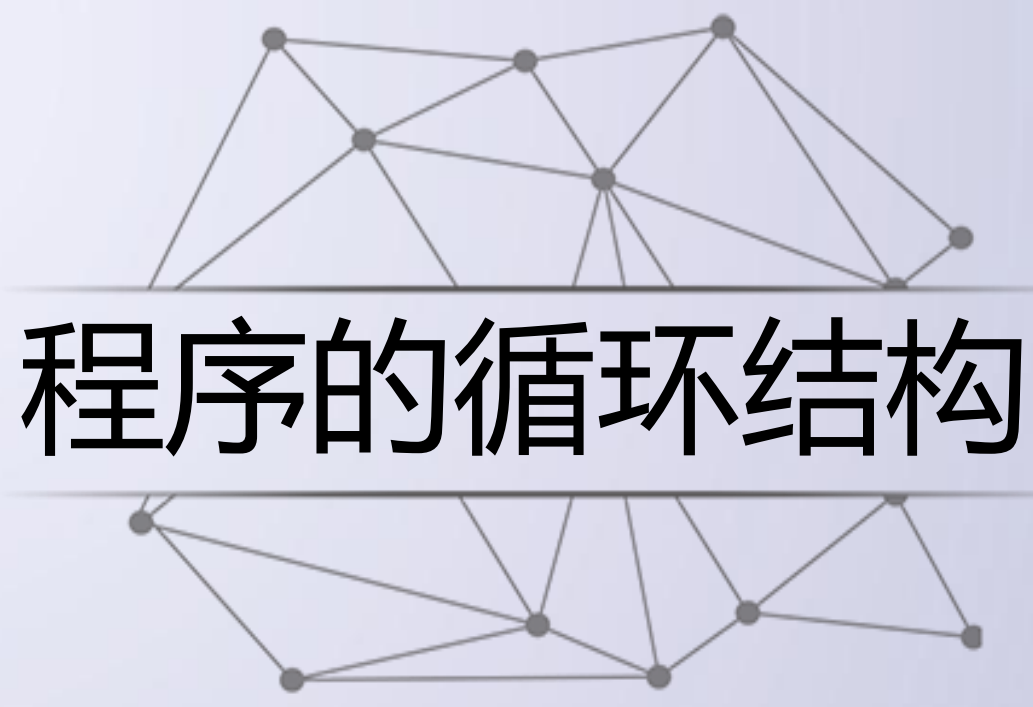
>>>

请输入身高(米)和体重(公斤)[逗号隔开]: **1.75, 75**

BMI数值为: 24.49

BMI指标为:国际'正常', 国内'偏胖'

```
1  #e5.2CalBMI.py
2  height, weight = eval(input("请输入身高(米)和体重\
    (公斤)[逗号隔开]: "))
3  bmi = weight / pow(height, 2)
4  print("BMI数值为: {:.2f}".format(bmi))
5  wto, dom = "", ""
6  if bmi < 18.5:
7      wto, dom = "偏瘦", "偏瘦"
8  elif 18.5 <= bmi < 24:
9      wto, dom = "正常", "正常"
10 elif 24 <= bmi < 25:
11     wto, dom = "正常", "偏胖"
12 elif 25 <= bmi < 28:
13     wto, dom = "偏胖", "偏胖"
14 elif 28 <= bmi < 30:
15     wto, dom = "偏胖", "肥胖"
16 else:
17     wto, dom = "肥胖", "肥胖"
18 print("BMI指标为:国际'{0}', 国内'{1}'".format(wto, dom))
```



程序的循环结构



遍历循环: for语句

遍历循环：

根据循环执行次数的确定性，循环可以分为确定次数循环和非确定次数循环。确定次数循环指循环体对循环次数有明确的定义循环次数采用遍历结构中元素个数来体现

Python通过保留字for实现“遍历循环”：

```
for <循环变量> in <遍历结构>:  
    <语句块>
```



遍历循环: for语句

遍历结构可以是字符串、文件、组合数据类型或range()函数：

循环N次	遍历文件fi的每一行	遍历字符串s	遍历列表ls
for i in range(N):	for line in fi:	for c in s:	for item in ls:
<语句块>	<语句块>	<语句块>	<语句块>

遍历循环还有一种扩展模式，使用方法如下：

for <循环变量> in <遍历结构>:

<语句块1>

else:

<语句块2>



遍历循环: for语句

- 当for循环正常执行之后，程序会继续执行else语句中内容。else语句只在循环正常执行之后才执行并结束，
- 因此，可以在<语句块2>中放置判断循环执行情况的语句。

```
1  for s in "BIT":  
2      print("循环进行中: " + s)  
3  else:  
4      s = "循环正常结束"  
5  print(s)
```

```
>>>
```

```
循环进行中: B
```

```
循环进行中: I
```

```
循环进行中: T
```

```
循环正常结束
```



无限循环: while语句

无限循环：

- 无限循环一直保持循环操作直到特定循环条件不被满足才结束，不需要提前知道确定循环次数。
- Python通过保留字while实现无限循环，使用方法如下：

```
while <条件>:
```

```
    <语句块>语句块
```



无限循环: while语句

- 无限循环也有一种使用保留字else的扩展模式：

while <条件>:

<语句块1>

else:

<语句块2>

```
1 s, idx = "BIT", 0
2 while idx < len(s):
3     print("循环进行中: " + s[idx])
4     idx += 1
5 else:
6     s = "循环正常结束"
7 print(s)
```

>>>

循环进行中: B

循环进行中: I

循环进行中: T

循环正常结束



循环保留字: break和continue

- 循环结构有两个辅助保留字：break和continue，它们用来辅助控制循环执行
- break用来跳出最内层for或while循环，脱离该循环后程序从循环后代吗继续续执行

```
1 for s in "BIT":
2     for i in range(10):
3         print(s, end="")
4         if s=="I":
5             break
```

```
>>>
```

```
BBBBBBBBBBBITTTTTTTTTT
```

其中，break语句跳出了最内层for循环，但仍然继续执行外层循环。每个break语句只有能力跳出当前层次循环



循环保留字: break和continue

- continue用来结束当前当次循环，即跳出循环体中下面尚未执行的语句，但不跳出当前循环。
- 对于while循环，继续求解循环条件。而对于for循环，程序流程接着遍历循环列表
- 对比continue和break语句，如下

```
1 for s in "PYTHON":  
2     if s=="T":  
3         continue  
4     print(s, end="")
```

```
>>>  
PYHON
```

```
1 for s in "PYTHON":  
2     if s=="T":  
3         break  
4     print(s, end="")
```

```
>>>  
PY
```



循环保留字: break和continue

continue语句和break语句的区别是：

- continue语句只结束本次循环，而不终止整个循环的执行。
- break语句则是结束整个循环过程，不再判断执行循环的条件是否成立

```
1 for s in "PYTHON":  
2     if s=="T":  
3         continue  
4     print(s, end="")
```

```
>>>  
PYHON
```

```
1 for s in "PYTHON":  
2     if s=="T":  
3         break  
4     print(s, end="")
```

```
>>>  
PY
```



循环保留字: break和continue

- for循环和while循环中都存在一个else扩展用法。
- else中的语句块只在一种条件下执行，即for循环正常遍历了所有内容没有因为break或return而退出。
- continue保留字对else没有影响。看下面两个例子

```
1  for s in "PYTHON":
2      if s=="T":
3          continue
4  print(s, end="")
5  else:
6      print("正常退出")
```


```
>>>
```

```
PYTHON正常退出
```

```
1  for s in "PYTHON":
2      if s=="T":
3          break
4  print(s, end="")
5  else:
6      print("正常退出")
```

```
>>>
```

```
PY
```



random库的使用



random库概述

- 随机数在计算机应用中十分常见，Python内置的random库主要用于产生各种分布的伪随机数序列。random库采用梅森旋转算法（Mersenne twister）生成伪随机数序列，可用于除随机性要求更高的加解密算法外的大多数工程应用。
- 使用random库主要目的是生成随机数，因此，读者只需要查阅该库的随机数生成函数，找到符合使用场景的函数使用即可。这个库提供了不同类型的随机数函数，所有函数都是基于最基本的random.random()函数扩展而来。



random库解析

函数	描述
<code>seed(a=None)</code>	初始化随机数种子，默认值为当前系统时间
<code>random()</code>	生成一个[0.0, 1.0)之间的随机小数
<code>randint(a, b)</code>	生成一个[a,b]之间的整数
<code>getrandbits(k)</code>	生成一个k比特长度的随机整数
<code>randrange(start, stop[, step])</code>	生成一个[start, stop)之间以step为步数的随机整数
<code>uniform(a, b)</code>	生成一个[a, b]之间的随机小数
<code>choice(seq)</code>	从序列类型(例如：列表)中随机返回一个元素
<code>shuffle(seq)</code>	将序列类型中元素随机排列，返回打乱后的序列
<code>sample(pop, k)</code>	从pop类型中随机选取k个元素，以列表类型返回



random库解析

对random库的引用方法与
math库一样，采用下面两种
方式实现：

import random 或

from random import *

```
>>>from random import *
>>>random()
0.2922089114412476
>>>uniform(1,10)
1.5913082783598524
>>>uniform(1,20)
7
>>>randrange(0,100,4) #从0开始到100以4递增的
元素中随机返回
96
>>>choice(range(100))
97
>>>ls = list(range(10))
>>>print(ls)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>shuffle(ls)
>>>print(ls)
[5, 8, 4, 7, 6, 9, 3, 0, 2, 10, 1, 2, 3, 4, 5, 6, 7,
8, 9]
```



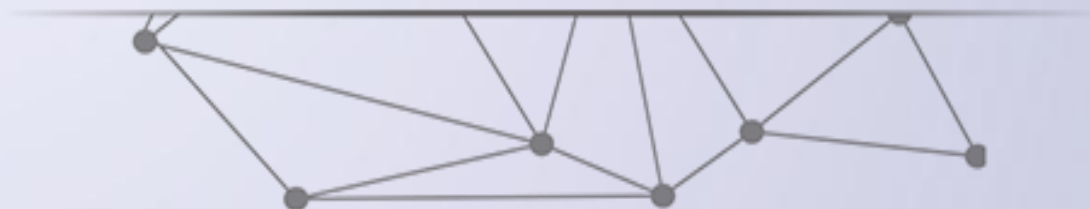
random库解析

生成随机数之前可以通过seed()函数指定随机数种子，随机种子一般是一个整数，只要种子相同，每次生成的随机数序列也相同。这种情况便于测试和同步数据

```
>>>seed(125) # 随机种子赋值125
>>>"{}.{}.{}".format(randint(1,10),randint(1,10),randint(1,10))
'4.4.10'
>>>"{}.{}.{}".format(randint(1,10),randint(1,10),randint(1,10))
'5.10.3'
>>>seed(125) # 再次给随机种子赋值125
>>>"{}.{}.{}".format(randint(1,10),randint(1,10),randint(1,10))
'4.4.10'
```



π 的计算





π 的计算

- π （圆周率）是一个无理数，即无限不循环小数。精确求解圆周率 π 是几何学、物理学和很多工程学科的关键。
- 对 π 的精确求解曾经是数学历史上一一直难以解决的问题之一，因为 π 无法用任何精确公式表示，在电子计算机出现以前， π 只能通过一些近似公式的求解得到，直到1948年，人类才以人工计算方式得到 π 的808位精确小数。



π 的计算

随着计算机的出现，数学家找到了另类求解 π 的另类方法：蒙特卡罗（Monte Carlo）方法，又称随机抽样或统计试验方法。当所要求解的问题是某种事件出现的概率，或者是某个随机变量的期望值时，它们可以通过某种“试验”的方法，得到这种事件出现的频率，或者这个随机变数的平均值，并用它们作为问题的解。这就是蒙特卡罗方法的基本思想。



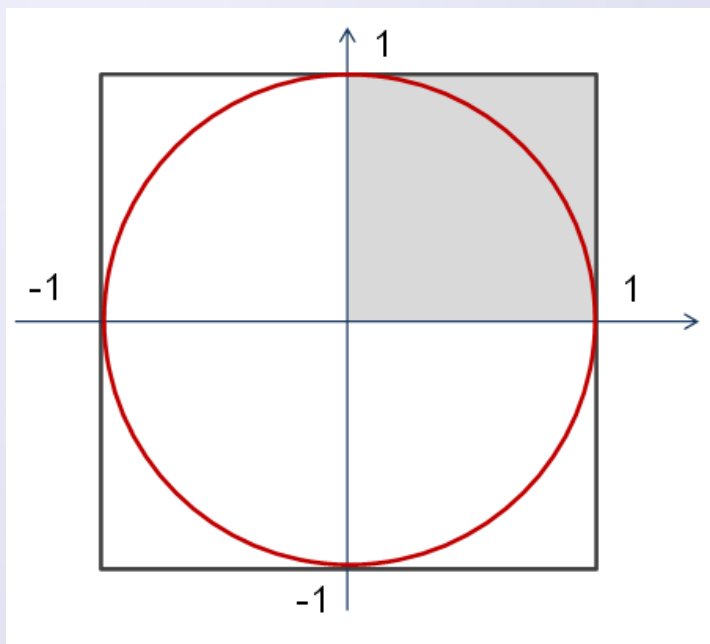
π 的计算

应用蒙特卡罗方法求解 π 的基本步骤如下：

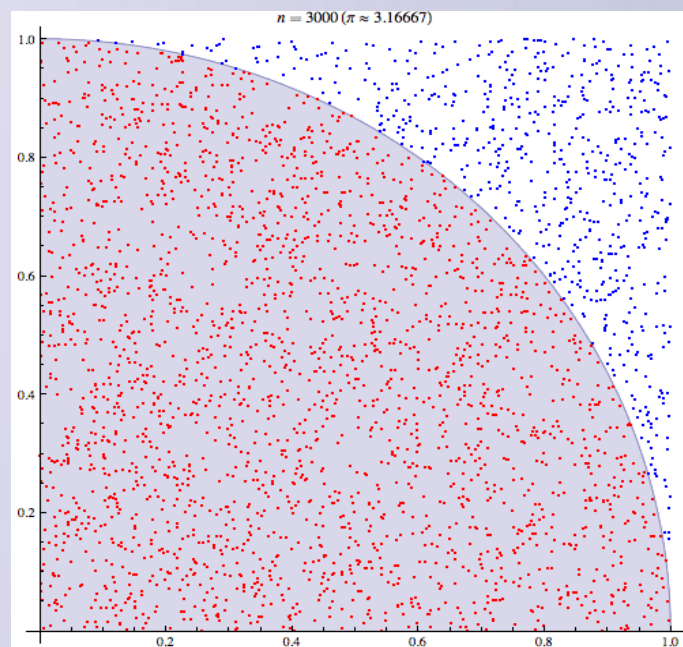
- 随机向单位正方形和圆结构，抛洒大量“飞镖”点
- 计算每个点到圆心的距离从而判断该点在圆内或者圆外
- 用圆内的点数除以总点数就是 $\pi/4$ 值。

随机点数量越大，越充分覆盖整个图形，计算得到的 π 值越精确。实际上，这个方法的思想是利用离散点值表示图形的面积，通过面积比例来求解 π 值。

π 的计算



计算 π 使用的正方形和圆结构



计算 π 使用的1/4区域和抛点过程



π 的计算

```
>>>
```

```
Pi值是3.144.
```

```
运行时间是: 0.016477s
```

实例代码6.1

e6.1CalPi.py

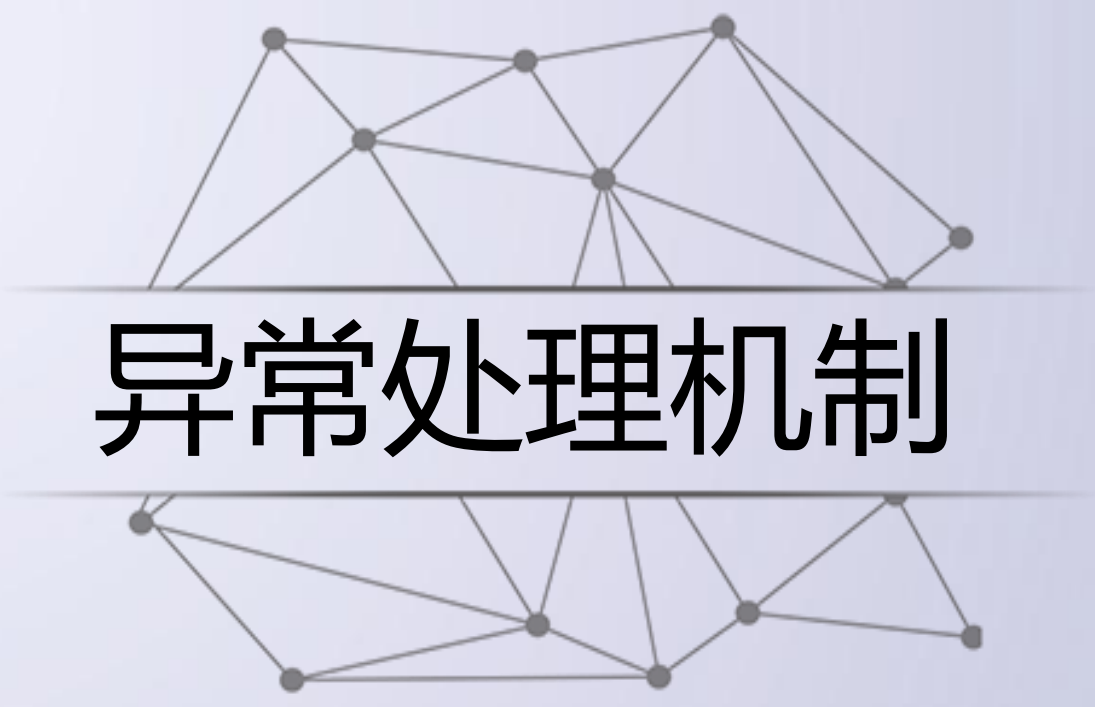
```
1  #e6.1CalPi.py
2  from random import random
3  from math import sqrt
4  from time import clock
5  DARTS = 10000
6  hits = 0.0
7  clock()
8  for i in range(1, DARTS+1):
9      x, y = random(), random()
10     dist = sqrt(x ** 2 + y ** 2)
11     if dist <= 1.0:
12         hits = hits + 1
13 pi = 4 * (hits/DARTS)
14 print("Pi值是{}".format(pi))
15 print("运行时间是: {:.5}s".format(clock()))
```



π 的计算

DARTS	π	运行时间
2^{10}	3. 109375	0. 011s
2^{11}	3. 138671	0. 012s
2^{12}	3. 150390	0. 014s
2^{13}	3. 143554	0. 018s
2^{14}	3. 141357	0. 030s
2^{15}	3. 147827	0. 049s
2^{16}	3. 141967	0. 116s
2^{18}	3. 144577	0. 363s
2^{20}	3. 1426696777	1. 255s
2^{25}	3. 1416978836	40. 13s

不同抛点数产生的精度和运行时间



异常处理机制



异常处理: try-except语句

观察下面这段小程序：

```
1 num = eval(input("请输入一个整数："))
2 print(num**2)
```

当用户输入的不是数字呢？

```
>>>
请输入一个整数： 100
10000
>>>
请输入一个整数： NO
Traceback (most recent call last):
  File "D:/PythonPL/echoInt.py", line 1, in <module>
    num = eval(input("请输入一个整数： "))
  File "<string>", line 1, in <module>
NameError: name 'No' is not defined
```

异常处理: try-except语句

Python解释器返回了异常信息，同时程序退出

```
Traceback (most recent call last):  
File "D:/PythonPL/echoInt.py", line 1, in <module>  
    num = eval(input("请输入一个整数: "))  
File "<string>", line 1, in <module>  
NameError: name 'No' is not defined
```

异常回溯标记

异常文件路径

异常发生的代码行数

异常类型

异常内容提示



异常处理: **try-except**语句


- Python异常信息中最重要的部分是异常类型，它表明了发生异常的原因，也是程序处理异常的依据。
- Python使用try-except语句实现异常处理，基本的语法格式如下：

```
try:
```

```
    <语句块1>
```

```
except <异常类型>:
```

```
    <语句块2>
```



异常处理: try-except语句

```
1 try:
2     num = eval(input("请输入一个整数: "))
    print(num**2)
except NameError:
    print("输入错误, 请输入一个整数!")
```

该程序执行效果如下：

```
>>>
请输入一个整数: NO
输入错误, 请输入一个整数!
```




异常的高级用法

- try-except语句可以支持多个except语句，语法格式如下：

try:

 <语句块1>

except <异常类型1>:

 <语句块2>

....

except <异常类型N>:

 <语句块N+1>

except:

 <语句块N+2>



异常的高级用法

- 最后一个except语句没有指定任何类型，表示它对应的语句块可以处理所有其他异常。这个过程与if-elif-else语句类似，是分支结构的一种表达方式，一段代码如下

```
1  try:
2      alp = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
3      idx = eval(input("请输入一个整数: "))
4      print(alp[idx])
5  except NameError:
6      print("输入错误, 请输入一个整数!")
7  except:
8      print("其他错误")
```



异常的高级用法

该程序将用户输入的数字作为索引从字符串alp中返回一个字符，当用户输入非整数字符时，`except NameError`异常被捕获到，提示升用户输入类型错误，当用户输入数字不在01到256之间时，异常被`except`捕获，程序打印其他错误信息，执行过程和结果如下：

```
>>>
```

```
请输入一个整数： NO
```

```
输入错误，请输入一个整数！
```

```
>>>
```

```
请输入一个整数： 100
```

```
其他错误
```



异常的高级用法

除了try和except保留字外，异常语句还可以与else和finally保留字配合使用，语法格式如下：

try:

 <语句块1>

except <异常类型1>:

 <语句块2>

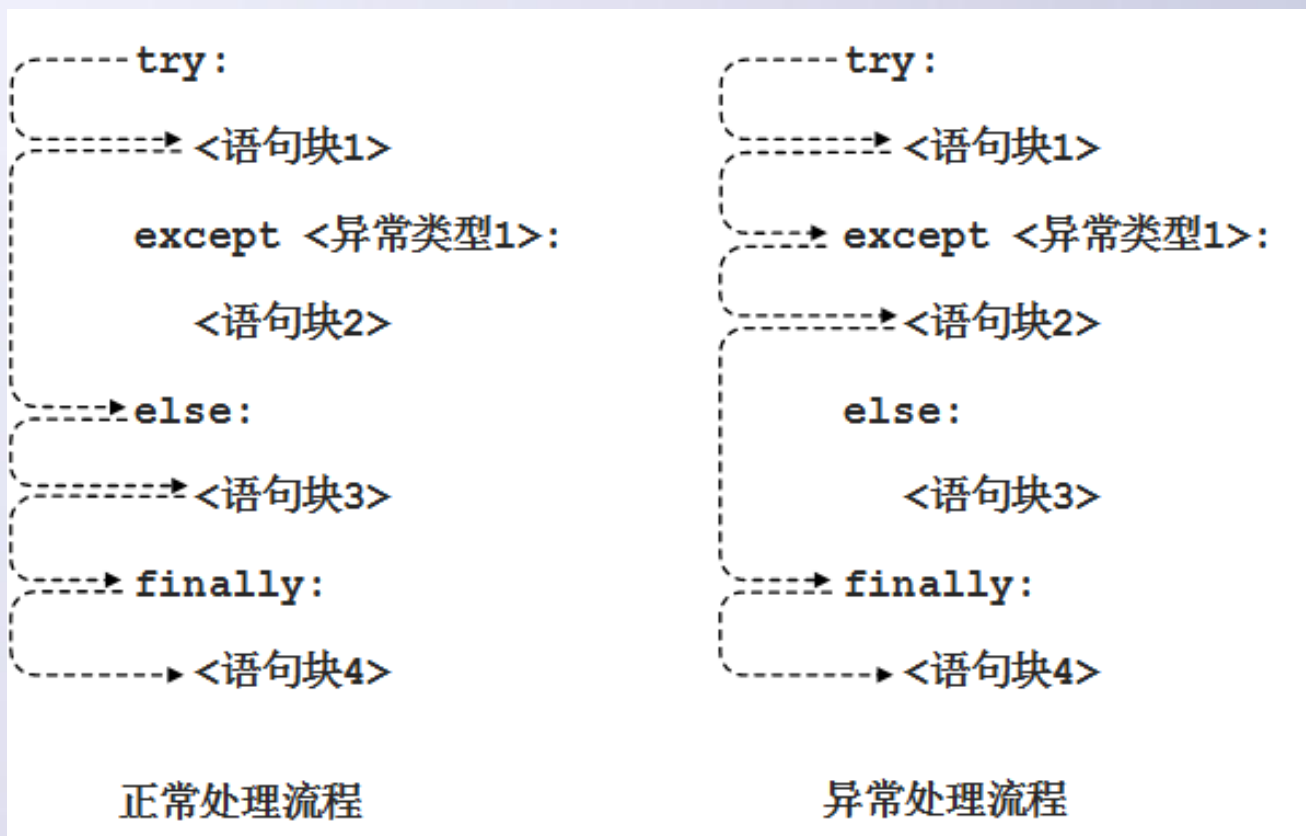
else:

 <语句块3>

finally:

 <语句块4>

异常的高级用法





异常的高级用法

采用else和finally修改代码如下：

```
1  try:
2      alp = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
3      idx = eval(input("请输入一个整数: "))
4      print(alp[idx])
5  except NameError:
6      print("输入错误, 请输入一个整数!")
7  else:
8      print("没有发生异常")
9  finally:
10     print("程序执行完毕, 不知道是否发生了异常")
```

执行过程和结果如下：

```
>>>
```

请输入一个整数： 5

```
F
```

没有发生异常

程序执行完毕，不知道是否发生了异常

```
>>>
```

请输入一个整数： NO

输入错误，请输入一个整数！

程序执行完毕，不知道是否发生了异常