控制结构

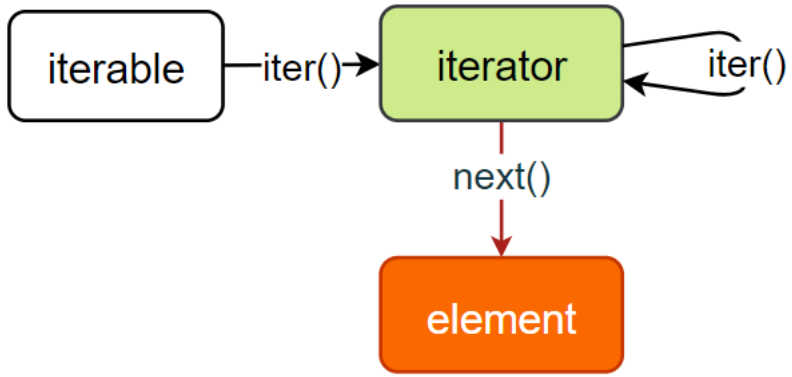
主要内容

- 循环结构：
  - 可迭代对象与for循环
  - while循环
- 异常处理

可迭代对象iterable和迭代器iterator

- 实现了iterable接口的iterable对象(如str,list,range, tuple,dict,set等): 类比于一个魔盒
  - 如何知道是否为iterable对象? dir(iterable)可看到其有方法__iter__()
  - **内置函数 iter(iterable)就是调用该对象的__iter__方法，返回一个迭代器iterator**
- 实现了iterator接口的iterator: 类比于一个打开的魔盒，每次按一下返回下一个元素
  - 如何知道是否为iterator对象？ dir(iterator)可看到有方法__next__()
  - **迭代器一般也是可迭代(iterable)对象**，其__iter__()方法**返回就是自身**
  - **内置函数next(iterator)就是调用该对象的__next__方法，从迭代器返回下一个元素，没有更多的元素时抛出异常StopIteration**
  - 内置函数sorted/reversed()/zip()/enumerate等返回的是一个迭代器对象

- ✓ 一个**一般的魔盒**(iterable)可以打开多次，每次返回一个不同的打开的魔盒(iterator)。但注意如果魔盒(iterable)有变化时，打开的魔盒(iterator)同样也会有影响
- ✓ 一个**打开的魔盒(iterator)**一般也是一个魔盒(iterable)，再打开时返回的就是同一个打开的魔盒(iterator)




```
>>> dir(str)
['__add__',... ,
'__iter__',...]
>>> s='ab'
>>> it = iter(s)
>>> it
<str_iterator object at
0x03A88110>
>>> dir(it)
[..., '__iter__',...
'__next__', ...]
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    next(it)
StopIteration
>>> it is iter(it)
True
>>> it0 = iter(s)
>>> it is it0
False
>>> next(it0)
'a'
```

for循环

```
for target in iterable: # Assign iterable items to target
    循环体
    if 条件表达式1: break #可选
     # Exit loop now, skip else if present
    if 条件表达式2: continue #可选
       #go to top of loop now
else:  #可选
    else语句块 # If we didn't hit a 'break'
```

iterable.py

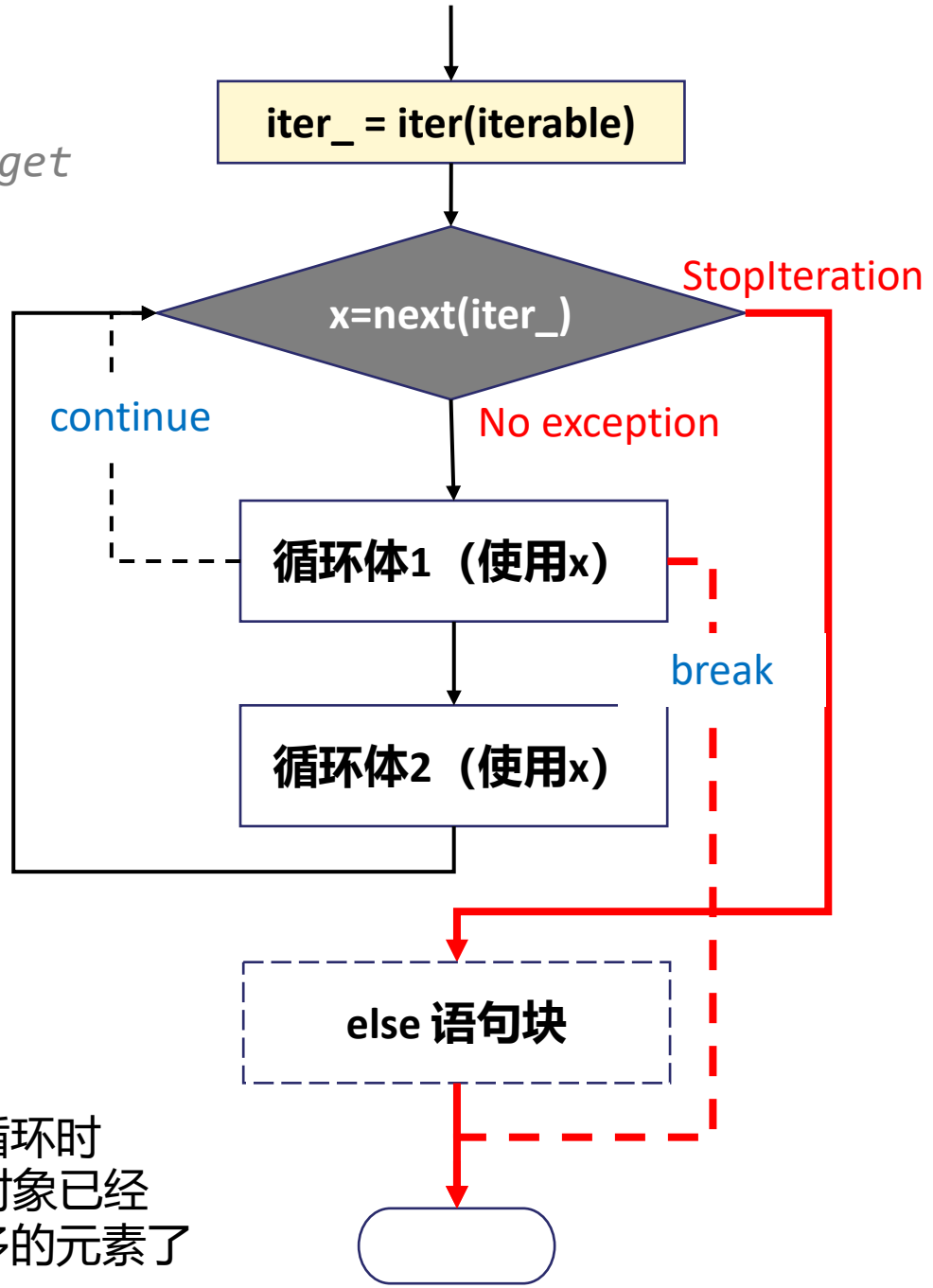
```
s = '12345'
for item in s:
    print(item)
for item in s:
    print(item)

it = iter(s)
for item in it:
    print(item)
print('another loop')
for item in it:
    print(item)
```

第2个例子中**for循环中的容器对象为iterator对象**:

```
1
2
3
4
5
another loop
```

第二个循环时
iterator对象已经
没有更多的元素了



for循环:break/continue/else示例

问题1：检查字符串s中是否有十进制数字

```
found = False
for ch in s:
    if '0' <= ch <= '9':
        print('found digit!')
        found = True
        break

if not found:
    print('no digit!')
```

iterable.py

```
for ch in s:
    if '0' <= ch <= '9':
        print('found digit!')
        break
else:
    print('no digit!')
```

- **break语句**：**结束循环**，顺序执行for循环之后的语句
- **continue语句**：**结束循环的当前轮次**，回到循环开始处取下一个元素继续循环
- 从for循环中退出：
  - 遍历所有元素后退出。如果有**else子句，执行其中的语句块。**注意else子句的缩进，特别是循环体的最后一个语句是if语句时
  - break退出

问题2：检查字符串中是否有十进制数字(5除外)

```
for ch in s:
    if ch == '5':
        continue
    if '0' <= ch <= '9':
        print('found!')
        break
else:
    print('not found!')
```

内置函数range

range(stop)

range(start, stop[, step])

- 返回range对象(**有序**不可变对象），产生一系列的整数，范围[start,stop)，即**半闭半开区间**
- 参数start表示起始值（默认为0）
- 参数stop表示终止值（结果中不包括这个值）
- 参数step可选，表示步长（默认为1）,step可以小于0
- range对象为可迭代对象：可通过for循环访问其中的元素
- 可以用list()函数将range对象转化为列表

range(4)	[0,4)的整数
range(1, 5)	[1,5)的整数
range(1, 10, 2)	1+2*i, i从0开始，且<10，即1,3,5,7,9
range(9, 0, -2)	从9开始，形式为9-2*i, 且>0，即为9, 7, 5, 3, 1
range(0, 0)	空range对象，其长度为0。其他例子如range(3, 1)
range(1, 5, -1)	空range对象

```
>>> range(4)
range(0, 4)
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
>>> s = range(2, 10, 2)
>>> list(s)
[2, 4, 6, 8]
>>> len(s)
4
>>> list(range(9, 0, -2))
[9, 7, 5, 3, 1]
>>> len(range(0, 0))
0
```

range示例

```
for i in range(5):
    print(i)



for i in range(5, 0, -1):
    print(i)
```

```
0
1
2
3
4

5
4
3
2
1
```

- range函数的参数要求为整数
- /为真除法，计算结果为浮点数，抛出异常： TypeError

iterable.py

```
def main():
    courses = 5
    students = 25
    class_size = students / courses
    # class_size = students // courses
    for i in range(class_size):
        print(i)
```

```
print('倒计时开始...')
import time
for i in range(9, -1, -1):
    time.sleep(1)
    print('\r%d\a' % i, end='')
print('\rfire!')
```

random模块

```
>>> import random
>>> dir(random)
[..., 'betavariate', 'choice', 'choices', 'expovariate',
'gammavariate', 'gauss', 'getrandbits', 'getstate',
'lognormvariate', 'normalvariate', 'paretovariate', 'randbytes',
'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

```
import random
random.seed(1234567)
r = random.randint(0, 25)
r = random.randrange(0, 26)
# 随机小写英文字母
import string
print("小写字母:", random.choice(string.ascii_lowercase))

# [1, 100)之间随机选择4个不同的整数
print(random.sample(range(1, 100), 4))


s = list(range(20))
random.shuffle(s)
print(s)
```

random_demo.py

主要函数	含义
seed(a=None)	给出随机数生成器的seed
randint(a,b)	返回在[a,b]区间的随机整数
randrange(start, stop=None, step=1)	从range(start, stop[, step])中随机选择一个整数
random()	返回在[0,1)区间的随机实数
uniform(a,b)	返回在[a,b]区间的随机实数
choice(seq)	返回非空序列seq中的随机一个元素
sample(seq, n)	从序列seq中随机(不放回）选择 n个元素组成一个新列表返回
shuffle(list_obj)	原地随机排序列表list_obj，返回None

for循环：图案问题

draw_triangle.py

```
def draw_triangle1():
    for line in range(1, 6):
        print('#' * line)
```

```
#
##
###
####
#####
```

```
def draw_triangle2():
    for line in range(1, 6):
        print('#' * line, end='')
        print('++' * line, end='')
        print('#' * line)
```

```
#+++#
##++++##
###++++###
####++++####
#####++++#####
```

```
def draw_triangle3():
    for line in range(1, 6):
        print(" " * (line - 1), end="")
        print("*" * (11 - 2 * line))
```

```
*********
 *******
  *****
   ***
    *
```

stars = a + b*i = a -2*i

a = 2 * 5 + 1 = 11

```
def draw_triangle4():
    for line in range(1, 6):
        print(("*" * (11 - 2 * line)).center(9))
```

图案问题也可借助对齐和填充机制

等差数列可描述为a + b*i的形式，
其中b为公差(步长)

for循环：图案问题（循环嵌套）

```
print(i, '*', j, '=', i * j, end='\t')
```

问题：打印九九乘法表。

```
def print_multiplication_table():
    ''' 打印九九乘法表 '''
    for i in range(1, 10):
        for j in range(1, i + 1):
            # print('%d * %d = %d' % (i, j, i*j),end='\t')
            print('{} * {} = {}'.format(i, j, i*j),end='\t')
        print()
```

外循环 i

```
1 * 1 = 1
2 * 1 = 2   2 * 2 = 4
3 * 1 = 3   3 * 2 = 6   3 * 3 = 9
4 * 1 = 4   4 * 2 = 8   4 * 3 = 12  4 * 4 = 16
5 * 1 = 5   5 * 2 = 10  5 * 3 = 15  5 * 4 = 20  5 * 5 = 25
6 * 1 = 6   6 * 2 = 12  6 * 3 = 18  6 * 4 = 24  6 * 5 = 30  6 * 6 = 36
7 * 1 = 7   7 * 2 = 14  7 * 3 = 21  7 * 4 = 28  7 * 5 = 35  7 * 6 = 42  7 * 7 = 49
8 * 1 = 8   8 * 2 = 16  8 * 3 = 24  8 * 4 = 32  8 * 5 = 40  8 * 6 = 48  8 * 7 = 56  8 * 8 = 64
9 * 1 = 9   9 * 2 = 18  9 * 3 = 27  9 * 4 = 36  9 * 5 = 45  9 * 6 = 54  9 * 7 = 63  9 * 8 = 72  9 * 9 = 81
```

print_multiplication.py

内循环1<=j<=i

j in [1, i]

i

i
1
2
3
4
5
6
7
8
9

1

1
2

1
2
3

1
2
3
4

for循环：图案问题

- 图案由四部分组成：
  - 第一行：+ 2 * height个 * +
  - 倒立三角形（共height行）：关注中间的数字个数，相邻两行之间相差2
    - 假设循环变量i从1到height，数字个数n = a + b * i = a - 2i
    - 最后一行的数字个数为0，因此 0 = a - 2 * height，即a = 2 * height
    - 每行输出[1, 2 * height - 2 * i]区间内的整数
  - 正立三角形
  - 最后一行

```
+--------+
|\123456/|
| \1234/ |
|  \12/  |
|   \/   |
|   /\   |
|  /21\  |
| /4321\ |
|/654321\|
+--------+
 height=4
```

```
+----------+
|\12345678/|
| \123456/ |
|  \1234/  |
|   \12/   |
|    \/    |
|    /\    |
|   /21\   |
|  /4321\  |
| /654321\ |
|/87654321\|
+----------+
 height=5
```

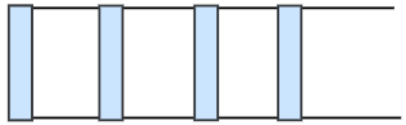
hourglass.py

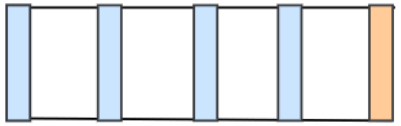
```
def draw_line(height=5):
    print("+", end="")
    print("-" * (2 * height), end="")
    print("+")
```

```
def draw_top(height=5):
    for line in range(1, height + 1):
        print("|" +  " " * (line - 1) + "\\", end="")
        for i in range(1, 2 * height - 2 * line + 1):
            print(i, end="")
        print("/" +  " " * (line - 1) + "|")
```

```
def draw_bottom(height=5):
    for line in range(1, height + 1):
        print("|" +  " " * (height - line) + "/", end="")
        for i in range(2 * line - 2, 0, -1):
            print(i, end="")
        print("\\" +  " " * (height - line) + "|")
```

for循环：栅栏循环(fencepost loop)





- 输出字符串s多次(times)，之间以分号隔开，整个输出采用括号括起来。比如(abc; abc; abc)

fencepost_loop.py

```
def multiprint1(s, times):
    print('(', end='')
    for i in range(times):
        print('%s; ' % s, end='')
    print(')')

def multiprint2(s, times):
    print('(', end='')
    for i in range(times):
        print(s, end='')
        if i < times - 1:
            print('; ', end='')
    print(')')
```

```
def multiprint3(s, times):
    print('(', end='')
    print(s, end='')
    for i in range(times - 1):
        print('; %s' % s, end='')
    print(')')

def multiprint4(s, times):
    if times <= 0:
        print('()')
        return
    print('(', end='')
    print(s, end='')
    for i in range(times - 1):
        print('; %s' % s, end='')
    print(')')
```

- 版本1：参数：'abc', 3时 输出 (abc; abc; abc;)
- **版本2：循环体每次判断是否最后一次**
- 版本3：首先输出s，然后重复输出; s，但是times=0时，应该输出()
- **版本4：版本3的基础上增加了特殊情形的判断**

for循环：更多示例

- **问题：寻找水仙花数。所谓水仙花数是指1个3位的十进制数，其各位数字的立方和等于该数本身。153是水仙花数，因为153 = 1^3 + 5^3 + 3^3**

find_narcissi_few.py

```
def find_narcissi_few():
    '''输出“水仙花数”'''
     for n in range(100, 1000):
        ones = n % 10
        tens = n // 10 % 10
        hundreds = n // 100
        if ones ** 3 + tens ** 3 + hundreds ** 3 == n:
            print(i)
```

```
153
370
371
407
```

```
def find_narcissi_few2():
    for hundreds in range(1, 10):
        for tens in range(10):
            for ones in range(10):
                n = ones + 10*tens + 100*hundreds
                if ones ** 3 + tens ** 3 + hundreds ** 3 == n:
                    print(n)
```

for循环：更多示例

print(f'鸡{chickens}只,兔{rabbits}只')

- **鸡兔同笼问题。假设共有鸡、兔30只，脚90只，求鸡、兔各有多少只。**

```
def chicken_rabbit_cage(heads=30, feet=90):
    found = False
    for chickens in range(0, heads + 1):
        for rabbits in range(heads + 1):
            if (chickens + rabbits) == heads and (2 * chickens + 4 * rabbits) == feet:
                print('鸡%d只,兔%d只' % (chickens, rabbits))
                found = True
                break
        if found:
            break
    if not found:
        print('无解')
```

不建议采用的版本
- 已知鸡的个数，可以计算出兔的个数

chicken_rabbit.py

```
def chicken_rabbit_cage2(heads=30, feet=90):
    # 鸡兔同笼问题。假设共有鸡、兔30只，脚90只
    for chickens in range(0, heads+1):
        if 2 * chickens + 4 * (heads - chickens) == feet:
            print('鸡%d只,兔%d只' % (chickens, heads - chickens))
            break
    else:
        print('无解')
```

for循环：更多示例

- **判断一个数是否为素数。**

如果一个正整数除了1和本身外没有其他因子，则称为素数，否则称为合数

$n = f1 * f2$

假设 $f1 \leq f2$

$f1^2 \leq f1 * f2 = n$

$f1 \leq int(\sqrt{n})$

```
def is_prime0(n):
    prime = True
    for factor in range(2, n):
        if n % factor == 0:
            prime = False
            break
    return prime
```

primes.py

```
import math
def is_prime(n):
    ''' 判断一个数是否为素数
    >>> n = int(input("Input a integer:"))
    >>> print(is_prime(n))
    '''
    prime = True
    max_factor = int(math.sqrt(n))
    for factor in range(2, max_factor + 1): # 检查可能的因子
        if n % factor == 0:                 #如果有非1的因子，则n不是素数
            prime = False
            break

    return prime
```

代码也可以采用for … else结构，参见下页slide

for循环：更多示例

```
for condidate in possible_condidates:
    ...
    if satisifed(condidate):
        found(condidate)
        break
```

寻找小于等于100的最大素数

```
def max_prime(limit=100):
    # 寻找100以内的最大素数
    for n in range(limit, 1, -1):
        max_factor = int(math.sqrt(n))
        for factor in range(2, max_factor+1):
            if n % factor == 0:
                break          ← 退出内层循环
        else:
            print(n)
            break              ← 退出外层循环
```

- 循环体中的break跳出当前循环
- 循环语句的else子句中的break跳出循环语句所在的循环
- continue也是类似情形，回到相应的循环继续

```
for condidate in possible_condidates:
    ...
    if satisifed(condidate):
        found(condidate)
```

输出100以内的所有素数

- 相比左边代码，注意最后一个break语句去掉了，**继续寻找其他素数**

primes.py

```
def all_primes(limit=100):
    # 寻找100以内的所有素数

    for n in range(limit, 1, -1):
        max_factor = int(math.sqrt(n))
        for factor in range(2, max_factor + 1):
            if n % factor == 0:
                break
        else:
            print(n, end=' ')
    print()
```

for循环：循环优化

输出由1、2、3、4这四个数字组成的每位数都不相同的所有三位数。

three_digit_numbers.py

```
def three_digit_numbers():
    digits = (1, 2, 3, 4)
    for i in digits:
        for j in digits:
            for k in digits:
                if i != j and j != k and i != k:
                    print(i * 100 + j * 10 + k)
```

```
def three_digit_numbers2():
    digits = (1, 2, 3, 4)
    for i in digits:
        for j in digits:
            if i == j: continue
            for k in digits:
                if j != k and i != k:
                    print(i * 100 + j * 10 + k)
```

部分判断提到外层循环，减少循环次数

为了优化程序以获得更高的效率和运行速度，编写循环语句时，

- 应尽量减少循环内部不必要的计算，将**与循环变量无关的代码**尽可能地提取到循环之外
- 使用多重循环嵌套时，应尽量减少内层循环中不必要的计算，尽可能地将**部分判断和计算**向外提，减少循环次数和计算次数。

```
def three_digit_numbers22():
    digits = (1, 2, 3, 4)
    for i in digits:
        i100 = i * 100
        for j in digits:
            if i == j: continue
            j10 = j * 10
            for k in digits:
                if j != k and i != k:
                    print(i100 + j10 + k)
```

i*100和j*10提到外层循环，减少计算次数

for循环

- **输出由1、2、3、4这四个数字组成的每位数都不相同的所有三位数。**

three_digit_numbers.py

```
def three_digit_numbers2():
    digits = (1, 2, 3, 4)
    for i in digits:
        for j in digits:
            if i == j: continue
            for k in digits:
                if j != k and i != k:
                    print(i * 100 + j * 10 + k)
```

```
>>>
123  124  132  134  142
143  213  214  231  234
241  243  312  314  321
324  341  342  412  413
421  423  431  432
>>>
```

```
def three_digit_numbers3():
    counts = 0
    digits = (1, 2, 3, 4)
    for i in digits:
        for j in digits:
            if i == j: continue
            for k in digits:
                if j != k and i != k:
                    print(i * 100 + j * 10 + k, end=' ')
                    counts += 1
                    if counts % 5 == 0: print()
    if counts % 5:
        print()
```

加入代码，每行输出最多5个数

- counts记录已经输出的数的个数
- 如果输出的数的个数为5的倍数，在后面加上回车
- 所有的数输出结束后也要检查其个数是否为5的倍数，**如果不是，加上回车**

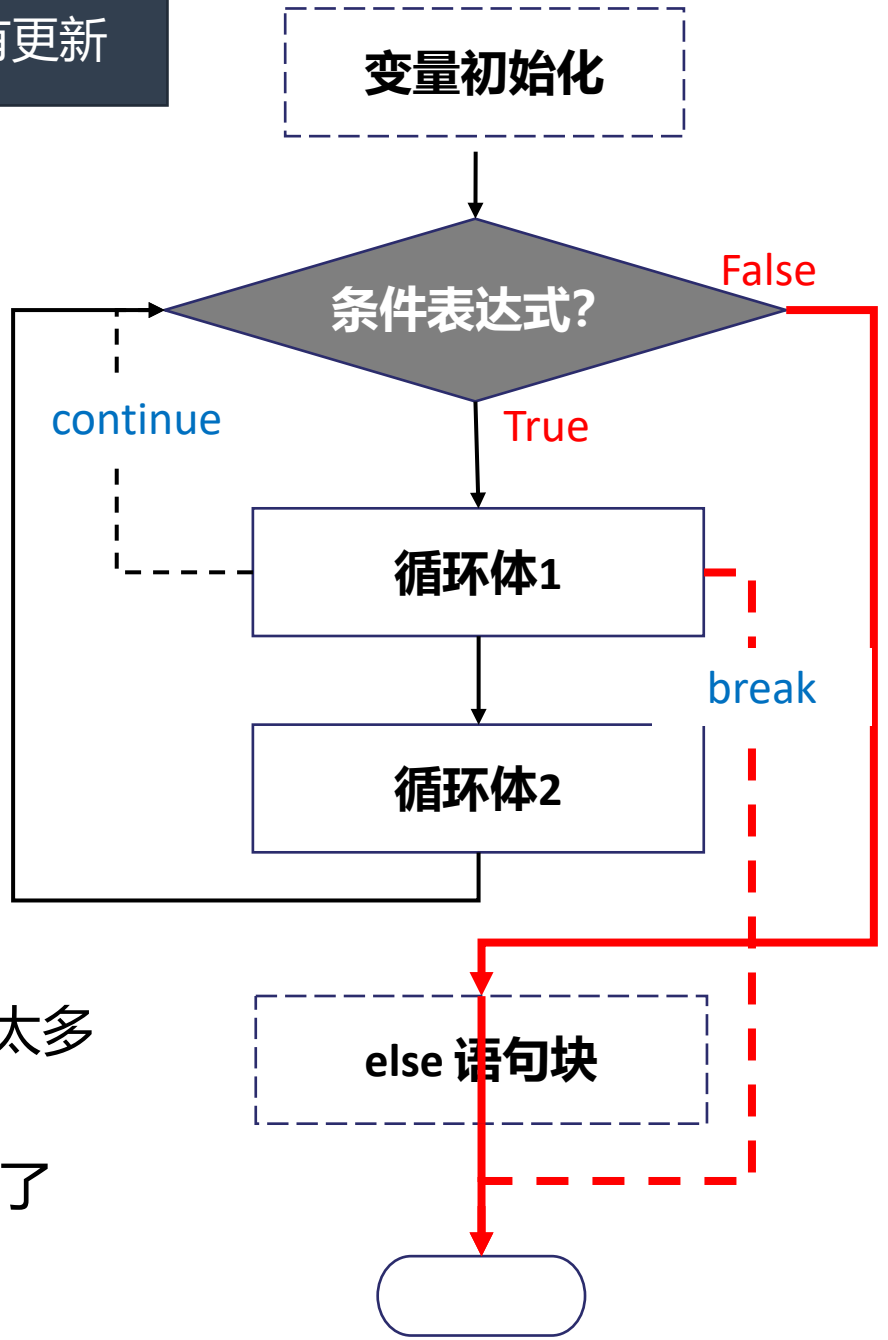
主要内容

- 循环结构：
  - 可迭代对象与for循环
  - **while循环**
- 异常处理

while循环

★在循环体尾部以及continue处应该保证循环变量有更新

```
while 条件表达式:
    循环体
    if 条件表达式1: break  #可选
    # Exit loop now, skip else if present
    if 条件表达式2: continue  # 可选
    # Go to top of loop now
else:  #可选
    # Run if we didn't hit a 'break'
    else语句块
```



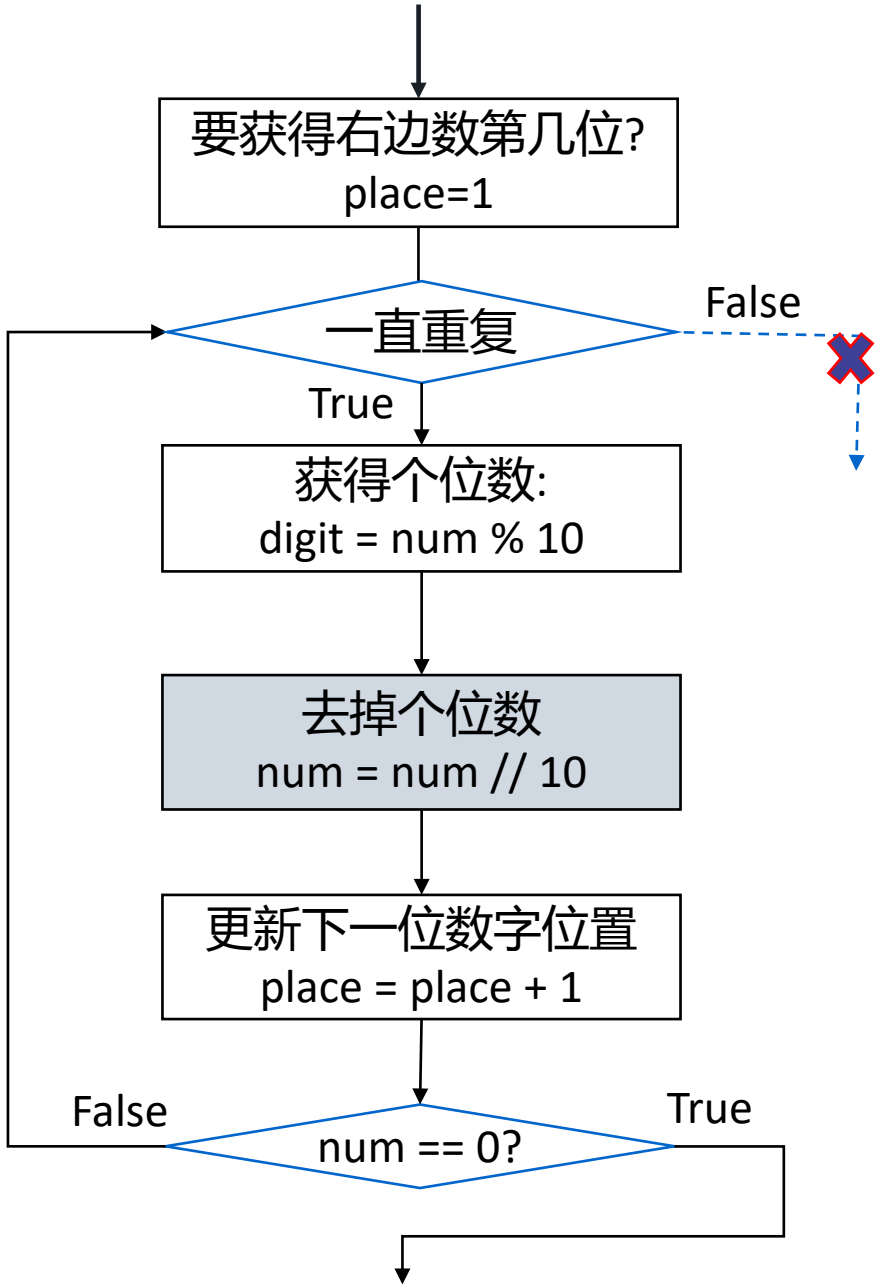
- 循环变量：在循环过程中改变且作为循环条件的变量
- 循环体中：
  - break语句，循环提前结束
  - continue语句，当前轮次结束，提前进入下轮
  - 适当地使用break语句可让代码更简单或更清晰，但不可有太多的break或continue!!!
- while支持可选的else子句，当循环自然结束时（不是因为执行了break而结束）执行else结构中的语句

while循环

- **问题：给出一个正整数，获得该整数的各位数字**

```
def extract_digits_noloop(num):
    """ 采用//和%运算来得到从右边开始数起的各个数字"""
    place = 1  # 右边数起第几位
    digit = num % 10
    print('右边数起第%d位数字为%d' % (place,digit))
    place = place + 1
    num = num // 10
    if num == 0:
        return
    digit = num % 10
    print('右边数起第%d位数字为%d' % (place,digit))
    place = place + 1
    num = num // 10
    if num == 0:
        return
    digit = num % 10
    print('右边数起第%d位数字为%d' % (place,digit))
    place = place + 1
    num = num // 10
    if num == 0:
        return
```

extract_digits.py



while循环

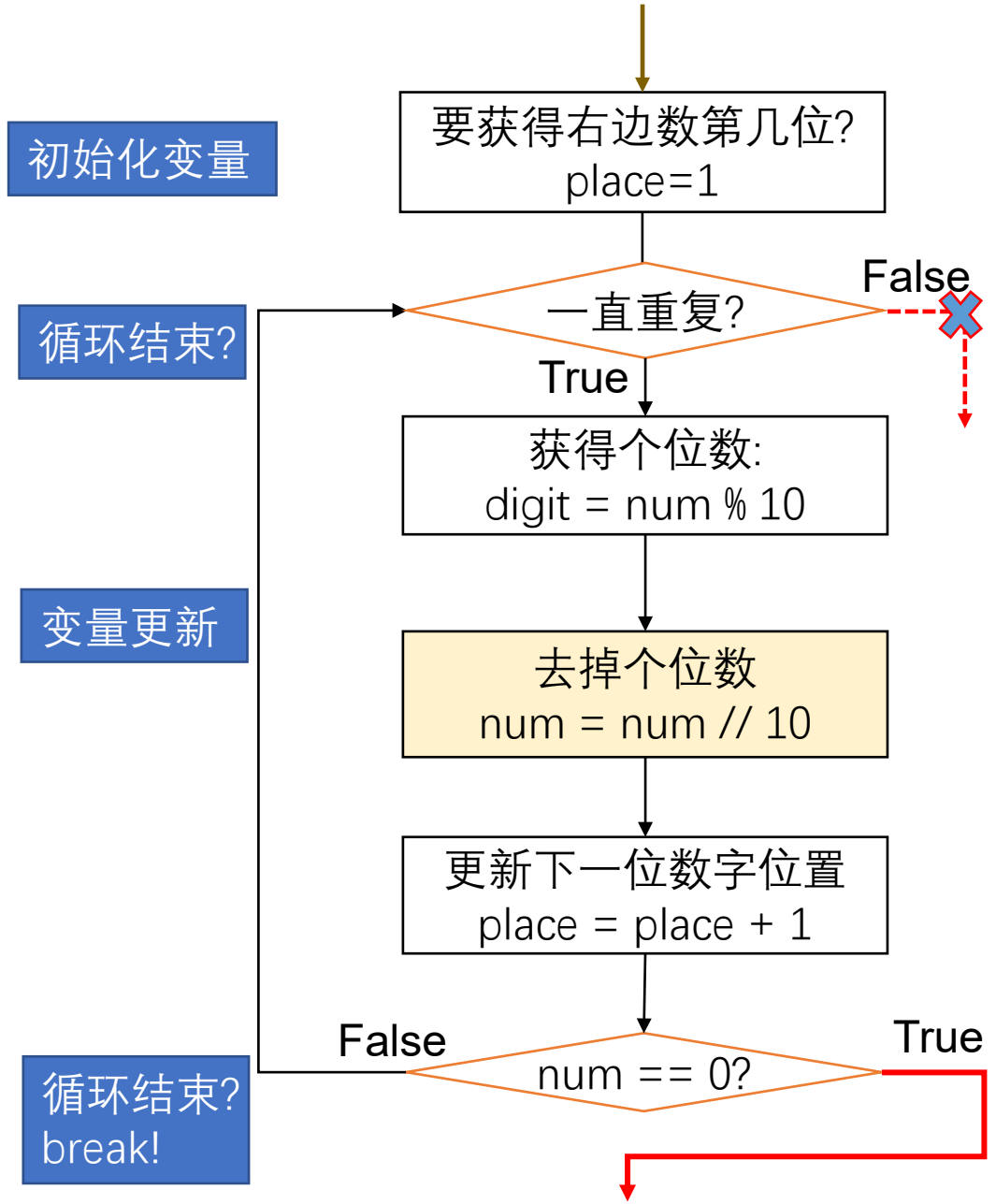
while语句：当expr真值判断为真时重复
- while True: 永远不可能为假，因此表示一直重复
- 循环体中，改变的变量是？

num = 当前的整数
place = 当前第几个数字

```
while expr :
    dosomething()
```

- 循环变量：在循环过程中改变且作为循环条件的变量
- 循环的结束条件？ num == 0
- 循环开始前一般会对这些改变的变量进行初始化
- 循环体中应该对这些变量进行更新

```
def extract_digits(num):
    """ 采用 //和 % 运算来得到从右边开始数起的各个数字"""
    place = 1  # 右边数起第几位
    while True:
        digit = num % 10
        print('右边数起第%d位数字为%d' % (place, digit))
        num = num // 10
        place = place + 1
        if num == 0:
            break
```



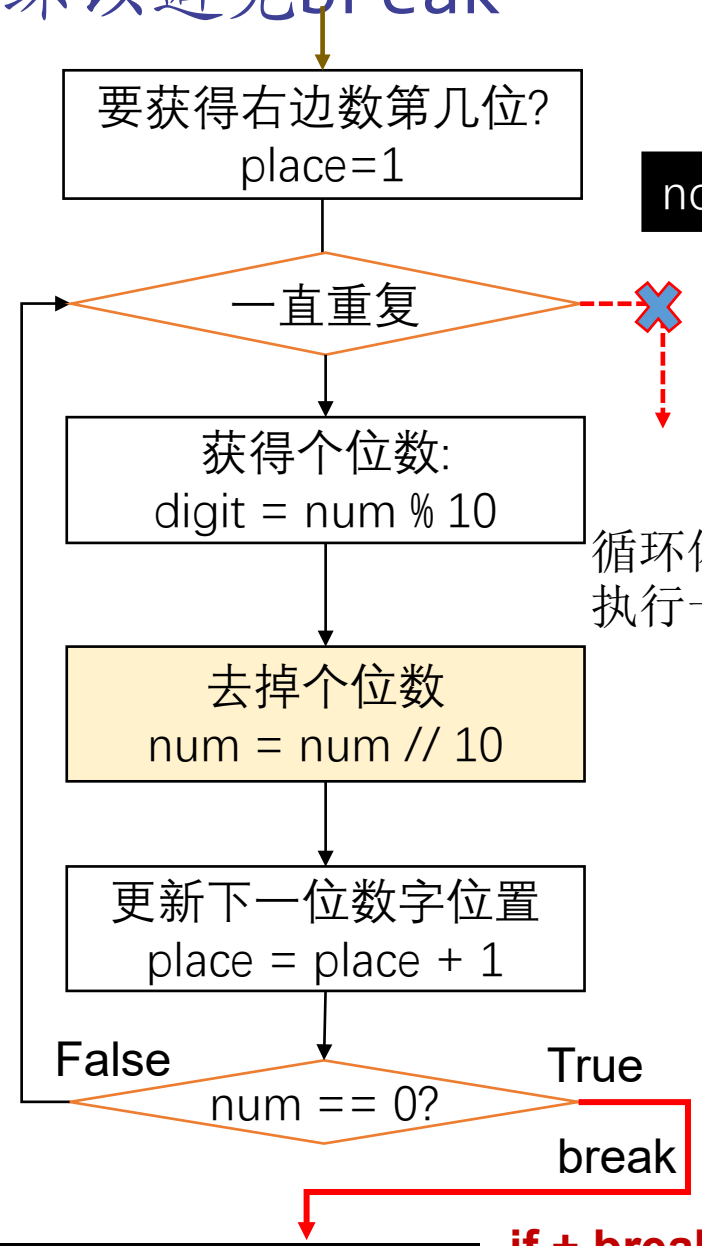
改写循环以避免break

初始化变量

循环结束?

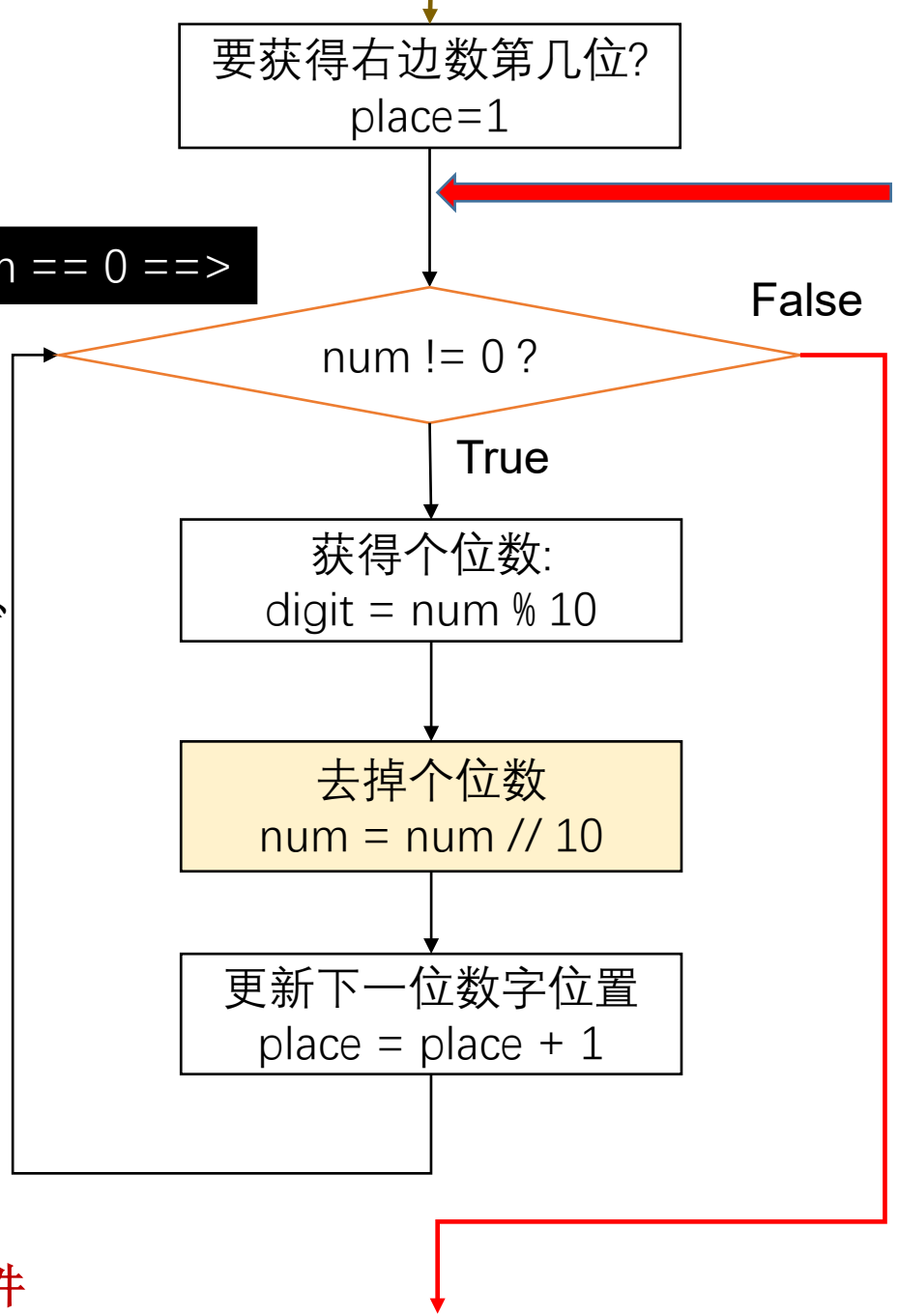
变量更新

循环结束?
break!



循环体至少执行一次

not num == 0 ==>



要考虑num == 0时循环体一次也不会执行的情况

条件满足时继续循环

★如果num初始为0，左边框图中循环体至少执行了一次，但是右边框图中循环体不会执行。

在循环前面加上单分支结构

```
if num ==0:
    print('右边…')
```

while True: 循环体一定会执行一次

if + break: 条件满足时退出循环

改写循环以避免break

```
def extract_digits(num):
    """ 采用 //和 % 运算来得到从右边开始数起的各个数字"""
    place = 1  # 右边数起第几位

    while True:
        digit = num % 10
        print('右边数起第%d位数字为%d' % (place,digit))
        num = num // 10
        place = place + 1
        if num == 0:
            break
```

```
def extract_digits2(num):
    """ 采用 //和 % 运算来得到从右边开始数起的各个数字"""
    place = 1  # 右边数起第几位
    if num == 0:  print('右边数起第1位数字为0‘)
    while num != 0 :    #也可以 while num :
        digit = num % 10
        print('右边数起第%d位数字为%d' % (place,digit))
        num = num // 10
        place = place + 1
```

extract_digits.py

```
def extract_digits3(num):
    place = 1  # 右边数起第几位
    done = False
    while not done:
        digit = num % 10
        print('右边数起第%d位数字为%d' %
(place, digit))
        place = place + 1
        num = num // 10
        if num == 0:
            done = True
```

只在循环条件处判断是否结束循环，避免使用break

版本3：

- 引入一个变量，比如done/found等
- 初始 done=False
- 循环条件中包含not done
- 循环体中发现要退出循环时，设置done=True，然后(可能使用continue语句)回到循环条件判断处

while循环：另一个示例

- **问题1. 查找一个最小正整数, 要求满足: 被3除余2, 被5除余3, 被7除余4**

```python
def smallest_n1():
    i = 1
    while True:
        if i % 3 == 2 and i % 5 == 3 and i % 7 == 4:
            break
        i += 1
    print(i)

def smallest_n2():
    i = 1
    while not (i % 3 == 2 and i % 5 == 3 and i % 7 == 4):
   # while i % 3 != 2 or i % 5 != 3 or i % 7 != 4:
        i += 1
    print(i)

def smallest_n3():
    i = 1
    done = False
    while not done:
        if i % 3 == 2 and i % 5 == 3 and i % 7 == 4:
            done = True
        else:
            i += 1
    print(i)
```

remainder_theorem.py

- 版本1：从1开始循环，直到满足条件时使用break跳出循环
- 版本2： 不使用break，将循环结束条件归并到while后面的条件表达式中
- 版本3： 不使用break，引入done，在循环体发现循环结束时，设置done=True

✓ 中国剩余定理：由于3/5/7两两互素，则问题1一定有解
✓ 如果改成6/7/9，由于不是两两互素，问题可能有解，也有可能无解
✓ 有可能找不到满足条件的解，这样循环一直无法结束

while循环：另一个示例

问题1. 查找一个最小正整数, 要求满足: 被3除余2, 被5除余3, 被7除余4

问题2. 要求满足: 被6除余2, 被7除余4, 被9除余5, 有解

问题3. 要求满足: 被6除余2, 被7除余4, 被9除余4, 无解

```python
def smallest_n4(limit=20000):
    i = 1
    found = False
    while i < limit:
        if i % 6 == 2 and i % 7 == 4 and i % 9 == 5:
            print(i)
            found = True
            break
        i += 1
    if not found:
        print('没有解')
```

remainder_theorem.py

```python
def smallest_n7(limit=20000):
    i = 1     #问题3. 要求满足: 被6除余2, 被7除余4, 被9除余4, 无解

    found = False
    while not found and i < limit:
        if i % 6 == 2 and i % 7 == 4 and i % 9 == 4:
            print(i)
            found = True
        else:
            i += 1
    if not found:
        print('没有解')
```

考虑问题有可能无解的情形

版本1：

- 从1开始循环，直到**足够大**时循环结束
- 如果找到，输出并设置found标志，**跳出循环**
- 如果结束后found仍然为False，说明没有解

版本2：

- 从1开始循环，直到**找到解或足够大**时循环结束
- 如果找到，输出并设置found标志
- 如果结束后found仍然为False，说明没有解

while循环：另一个示例

```
def smallest_n4(limit=20000):
    i = 1
    found = False
    while i < limit:
        if i % 6 == 2 and i % 7 == 4 and i % 9 == 5:
            print(i)
            found = True
            break
        i += 1
    if not found:
        print('没有解')
```

版本1：

- 如果找到解，break退出循环
- 如果没有找到解，while后面的条件表达式不满足时正常退出循环
- while后面的if语句，如果是正常退出循环时执行相应语句块

```
def smallest_n8(limit=20000):
    i = 1
    while i < limit:
        if i % 6 == 2 and i % 7 == 4 and i % 9 == 5:
            print(i)
            break
        i += 1
    else:
        print('没有解')
```

while循环支持**可选的else子句**，在循环体中**没有通过break结束（即正常退出循环）时**会执行该else子句的语句块

while循环：另一个示例

```
def smallest_n5(limit=20000):
    i = 1
    while i < limit:
        if i % 6 == 2 and i % 7 == 4 and i % 9 == 5:
            print(i)
            break
        i += 1

    if i >= limit:
        print('没有解')
```

循环结束有两种情形：

- break语句退出循环
- 循环条件不满足退出循环

在之后的if语句中：

- 如果循环条件仍然满足，则说明为break退出
- 否则为循环条件不满足而结束循环

while循环：用户输入问题

average_score.py

```
def average_score1():
    total = n = 0
    text = input("请输入成绩[0, 100]: ")
    while text:
        score = float(text)
        if 0 <= score <= 100:
            total += score
            n += 1
        else:
            print('请输入[0, 100]之间的成绩:')
        text = input("请输入成绩[0, 100]: ")

    if n > 0:
        print('平均成绩 = %.2f' % (total / n))
```

- **用户输入若干成绩，直到用户直接回车，不输入任何信息(空字符串）时结束，求平均分。**
- **一个典型的栅栏循环问题：input + do + input + do + … + input。可以有两种实现方式**
  - **版本1：(input + do) *，input后检查是否为最后**
  - **版本2：input + (do + input) ***

哨兵(sentinel value)：在处理过程中一旦出现就结束循环的值

- 循环变量：用户输入的字符串text
- 循环变量初始化：text=input(...)
- 循环变量更新：text = input(...)

```
def shortest_phrase():
    phrase = input('type a phrase (Enter to quit)? ')
    shortest = phrase
    while phrase:
        if len(phrase) < len(shortest):
            shortest = phrase
        phrase = input('type a phrase (Enter to quit)? ')
    print('shortest phrase was:', shortest)
```

用户输入若干个字符串，直到用户直接回车时结束，寻找其中最短(或最长)的字符串

思考：如果采用栅栏循环的第1种实现方式呢?

while循环：级数求和

- **计算0! + 1!+2!+3!+...+n!**

求和/求积...问题： item1+ item2 + item3 + ...
循环结构：
total = item = item1
while ....:
  item = 根据上一项来计算当前项
  total += item

- 循环变量 i
- 求解空间：[1, n]
  - 循环变量i初始化为1
  - 循环条件 i <= n
  - 循环变量更新： i = i + 1
- 检查循环变量初始值和最后一个满足循环条件的终值，避免**"偏1"错误**

sum_factorial.py

```
def sum_factorial(n):
    '''计算0! + 1!+2!+3!+...+n! '''
    i = 1
    total = item = 1
    while i <= n:
        item *= i
        total += item
        i += 1

    return total
```

```
def sum_factorial_forloop(n):
    total = item = 1
    for i in range(1, n+1):
        item *= i
        total += item
        # i += 1    # 这一行如果执行也不会带来别的问题

    return total
```

while循环

- 蒙特卡罗方法计算pi:
  - 如图所示，圆的面积为pi，而正方形的面积为4,在正方形的区域随机产生一个点，该点在圆内的概率为pi/4
  - frequency = hits/tries = pi/4 ➔ pi = 4*hits/tries

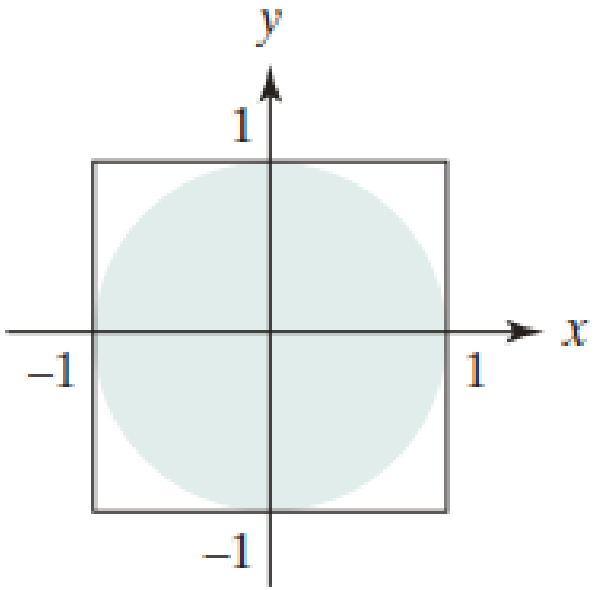
random.uniform(-1, 1)

montecarlo.py

```
import random
def montecarlo_pi(tries = 1000000):
    hits = 0
    for i in range(tries):
        x = random.random() * 2 - 1
        y = random.random() * 2 - 1

        if x * x + y * y <= 1:
            hits += 1

    pi = 4 * hits / tries
    return pi                    # 3.141392
```



while循环：更新循环变量

- for循环一般用于循环次数可以提前确定的情况，尤其是用于遍历可迭代对象中的元素
- while循环一般用于循环次数难以提前确定的情况，也可以用于循环次数确定的情况
- 一般优先考虑使用for循环

问题：输出11以内[0, 10]的奇数

```
for i in range(11): #[0,10]
    if i % 2 == 0:
        continue
    print(i, end=' ')
```

```
for i in range(11):
    if i % 2 == 0:
        i += 1
        continue
    print(i, end=' ')
```

i更新不起作用，for循环每次重新赋值

```
i=0
while i < 11: #[0,10]
    if i % 2==0:
        continue
    print(i, end=' ')
    i += 1
```

continue时，i没有更新，死循环

```
i= 0
while i < 11:  # [0,10]
    i += 1   # [1,11]
    if i % 2 ==0:
        continue
    print(i, end=' ')
```

这个版本求的是[1,11]范围内的奇数

loop_continue_demo.py

```
i=0
while i < 11: #[0,10]
    if i % 2==0:
        i += 1
        continue
    print(i, end=' ')
    i += 1
```

```
i= -1
while i < 10:  #[-1, 9]
    i += 1     #[0, 10]
    if i % 2==0:
        continue
    print(i, end=' ')
```


32

- 循环结构：
  - 可迭代对象与for循环
  - while循环
- **异常处理**

错误和异常

- 语法错误：解释器在将代码转换为bytecode时发现不符合python语法时报SyntaxError
- 逻辑错误(bug)：程序能够运行，但是逻辑上有错误
- 运行时错误(runtime error)：解释器在执行过程中检测到一个不合理的情况出现，即出现了异常(exception)，**如果不捕获异常**，解释器就会指出当前代码已无法继续执行下去而退出。但要注意导致异常出现的真正原因可能在之前的代码中
- Python的异常处理分为两个阶段：
  - 第一个阶段了解可能出现哪些异常
  - 第二个阶段是检测并处理

```
>>> 10 / 0
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    10 / 0
ZeroDivisionError: division by zero
>>> namee + 1
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    namee + 1
NameError: name 'namee' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    '2' + 2
TypeError: can only concatenate str (not "int")
to str
>>> int('abc')
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    int('abc')
ValueError: invalid literal for int() with base
10: 'abc'
```

异常类型

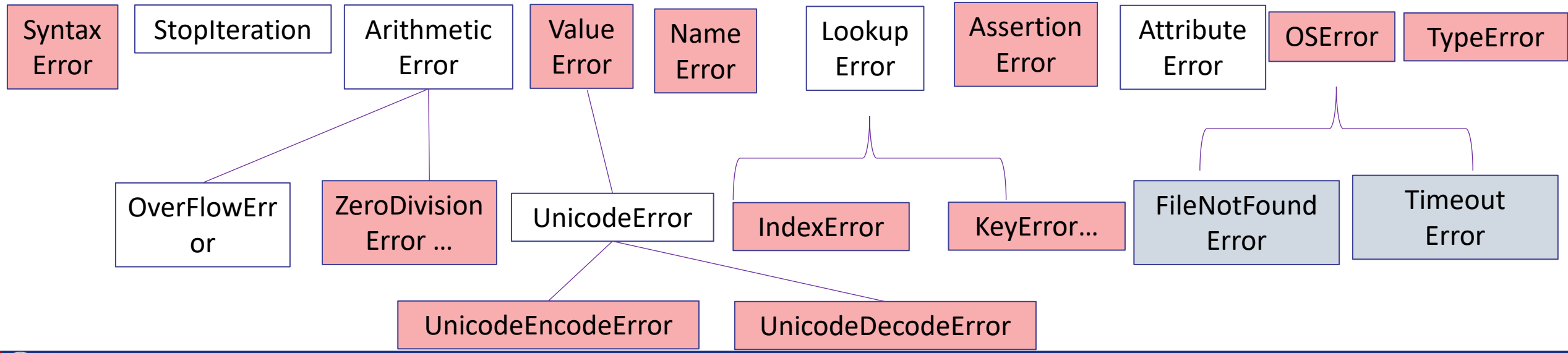
- 出现异常时构造一个异常对象
- try语句用于捕获指定类型的异常
- 如果没有捕获最终导致解释器结束执行时，输出异常对象以及调用栈(traceback对象）信息

```
>>> v = ValueError('xxx')
>>> type(v)
<class 'ValueError'>
>>> isinstance(v, Exception)
True
```

BaseException

- GeneratorExit：生成器退出
- SystemExit：调用sys.exit()时会抛出该异常，缺省情况下会退出解释器
- KeyboardInterrupt：比如在调用input()时用户输入Ctrl-C，缺省不捕获，抛出异常
- **Exception**

用户自定义的异常应该继承Exception或者其某个子类。异常处理时有时会捕获Exception



常用异常类型

异常	描述
Exception	几乎所有异常的基类(base class)
SyntaxError	语法错误
NameError	访问一个没有定义的变量
AttributeError	访问对象的属性时出错
IndexError	下标不存在
KeyError	字典的key不存在
TypeError	内置运算符或者某个函数作用的对象的类型不符
ValueError	内置运算符或某个函数作用的对象类型相符，但值不合适
ZeroDivisionError	除法类运算中除数为0
OSError	执行操作系统操作时出现错误，比如文件不存在等
AssertionError	断言异常

- 异常类：异常的类型
- 异常对象：某个具体的异常，其类型为某个异常类
  - 属性args给出了创建该异常对象时传递的参数

```
>>> ValueError()
ValueError()
>>> ValueError('wrong')
ValueError('wrong')
>>> t = ValueError('wrong', 1)
>>> t
ValueError('wrong', 1)
>>> t.args
('wrong', 1)
```

`dir(__builtins__)` 查看内置函数以及内置的异常类型

执行<body>中的代码，如果**没有异常**，

- 如果有可选的else子句，则**执行else块**
- 如果有可选的finally子句，则**执行finally块**
- **顺序执行**try语句之后的代码

在执行<body>期间**出现异常**时：

- 逐个按照顺序匹配except块中的异常，找到**第一个匹配的异常**为止：
  - 如果找到匹配，执行该块的代码，**跳过后面的其他except子句**
  - 如果有可选的finally子句，则**执行finally块**
  - 如果前面**匹配(处理)了异常**，则**顺序执行try结构后面的代码**
- 如果没有找到匹配的异常，即未处理异常（**异常没有捕获**），如果有finally子句，则**执行该块**；**往外层抛出异常**（不会顺序执行try之后的代码）
- 在body/else/except语句块中，如果**有合法的break/continue/return时，会执行对应的跳转逻辑。但是如果有finally块，在跳转前会执行finallly块**

except关键字后面的expression应该为：

✓ 异常类，比如ValueError，KeyError等

✓ 异常类的元组，表示其中任一异常出现，如(ValueError, TypeError)

isinstance(obj, class_or_tuple)

```
try:
    <body>                          执行<body>中的代码
except [expression [as identifier]]:
    <exceptionBody>

except [expression [as identifier]]:
# 可以有多个except子句，出现异常时按序匹配找到对应的exception为止。不带表达式的except等价于except BaseException，应该是最后一个
    <exceptionBody>
else:    # 可选的，<body>顺序执行完且没有异常出现时会执行该块的代码
    <elseBody>
finally:  # 可选的，不管有没有异常（甚至是从body中break/continue/return），异常是否捕获都要执行
    <finallyBody>
```

顺序执行try结构之后的代码

如果前面没有抛出异常(没有异常，有异常但已捕获），除非执行了break/continue/return，都会顺序执行之后的代码

✓ 不带表达式的except等价于except BaseException，应该是最后一个

✓ 如果要查看异常的具体信息：except expression as instance可以获得异常对象instance

示例1：输入整数

- 使用Python的异常处理机制
  - 允许错误发生，在错误发生时进行处理.
  - **抓大放小**：关注主要流程，较少发生的情形在except部分处理
  - 代码更易读写，也方便纠正错误
- 增加额外的逻辑(复杂性):
  - 要考虑用户输入
    - 要求为非空字符串，且全部为数字，或者第一个为符号，后面全为数字
  - 随着必须考虑的错误越来越多，复杂性也随之增加→可能会掩盖程序的本来作用.

调用input()时： 用户输入Ctrl-C，异常KeyboardInterrupt
用户输入Ctrl-D/Ctrl-Z: 异常EOFError

input_int.py

```
def input_int():
    while True:
        try:
            x = int(input("Please enter a number: "))
            break
        except ValueError as e:
            print(type(e), e)
            print("That was not a valid number. Try again...")
    return x
```

```
def input_int2():
    while True:
        x = input("Please enter a number: ")
        if not x :
            continue
        if x.isdigit() or (x[0] in '+-' and x[1:].isdigit()):
            x = int(x)
            break
        print("That was not a valid number. Try again...")
    return x
```

示例2：输入多个值+异常处理

- **用户输入若干成绩，直到用户直接回车，不输入任何信息(空字符串）时结束，求平均分。**

average_score.py

```
def average_score2():
    total = n = 0
    text = input("请输入成绩[0, 100]: ")
    while text:
        try:
            score = float(text)
            if 0 <= score <= 100:
                total += score
                n += 1
            else:
                print('请输入[0, 100]之间的成绩:')
        except ValueError as e:
            print('非法的浮点数', e)

        text = input("请输入成绩[0, 100]: ")

    if n > 0:
        print('平均成绩 = %.2f' % (total / n))
```

raise语句：主动抛出异常

```
raise [expression [from orig_exception]]
```

- expression必须是：
  - 一个异常类的实例对象
  - 一个异常类，此时系统会首先创建一个该类的实例对象(参数为空)，然后抛出该异常对象
- from orig_exception:一般出现在except块中，给出了将两个异常连接起来的方法，表示异常是由于另一个异常(当前正在匹配处理的异常orig_exception)引起的
- 单独的raise一般出现在except块中，表示重新抛出正在匹配处理的异常，如果当前没有，则抛出RuntimeError异常(No active exception to reraise)

```
raise Exception('foo occurred!')
raise Exception    # raise Exception()
raise Exception('foo occurred!') from Exception('an error occured!')
raise
```

raise_exceptions.py

```
def compute_circle_area(radius):
    if not isinstance(radius, (float, int)) or radius < 0:
        raise ValueError('radius should be a number larger than 0')
    return math.pi * radius * radius
```

raise语句

- **raise** expression **from orig_exception**
- 一般出现在except块中，给出了将两个异常连接起来的方法，表示现在抛出的异常是由于另一个异常((当前正在匹配处理的异常orig_exception)引起的，后面的异常orig_exception保存在前一个异常的__context__属性中
- 在处理异常exception1的except语句块中：
  - 出现异常raise exception2，相当于raise exception2 from exception1
  - raise exception2 from None 避免与exception1关联

```
raise Exception('foo occurred!') from Exception('an error occured!')
```

```
Exception: an error occured!
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    raise Exception('foo occurred!') from Exception('an error occured!')
Exception: foo occurred!
```

raise语句

```
raise Exception
raise Exception('foo occurred!')
raise Exception('foo occurred!') from Exception('an error occured!')
raise
```

- **单独的raise一般出现在except块中**，表示重新抛出正在匹配的异常。如果当前没有异常时进行raise，则抛出RuntimeError异常(No active exception to reraise)

```
>>> logging.warning('This is a warning message!')
WARNING:root:This is a warning message!
```

raise_exceptions.py

```
import logging
logging.basicConfig(filename='main.log',
                    format='%(asctime)s [%(levelname)s] %(message)s',
                    level=logging.DEBUG)
logging.info('logging started.')
def get_threshold(prompt=None) :
    try:
        thresh = int(input(prompt))
        return thresh
    except ValueError as e:
        # print(type(e), e)
        logging.error('get_threshold failed')
        logging.exception(e)
        raise
```

日志模块：logging

- 记录哪些级别(缺省WARNING)之上(从低到高分别为DEBUG,INFO,WARNING,ERROR,FATAL)的消息
- 记录在哪个文件，缺省标准输出
- 日志格式如何，缺省'%(levelname)s:%(name)s:%(message)s'

主要方法：

- debug(msg)
- info(msg)
- warning(msg)
- error(msg)
- fatal(msg)
- exception(e)

内置函数eval和exec

eval(source, globals=None, locals=None)

- **执行source中的代码**，source一般为字符串，**包含的代码必须为表达式**。该表达式在globals和locals指出的环境（**缺省为当前环境**）下执行，**运算后的对象作为返回值**。如果指定了globals和locals，则只会查找指定名字空间的名字

exec(source, globals=None, locals=None)

- 与eval类似，执行source对应的代码，返回None

```
text = input('请输入一个数...')
x = eval(text)

text = input('请输入两个整数，以逗号隔开...')
x, y = eval(text)
```

```
a = eval('4, 5')        # a = 4, 5
a = eval('4*5 ')        # a = 4*5
a = eval('"abc"')       # a = 'abc'
a = eval('a + (6,)')    # a = a + (6,)
a = eval('aa')          # a = aa，抛出异常名字aa未定义
a = eval('a = 3')       # 语法错
exec('a=4')             # a = 4
```

eval函数是不安全的

```
>>> cmd = "__import__('os').startfile(r'C:\\Windows\\notepad.exe')"
>>> print(cmd)
__import__('os').startfile(r'C:\Windows\notepad.exe')
>>> eval(cmd)
>>> import os
>>> eval("os.startfile(r'C:\\Windows\\notepad.exe')")
```

- 上述代码可以看到eval传递的参数来自于用户的输入时，调用eval函数是不安全的！！
- __import__为模块builtins对象（__builtins__）中的内置函数，eval可以传递第二个参数(名字空间)，这样在计算表达式时会查看第二个参数给出的名字空间中的名字

```
>>> safe_dict = {"__builtins__": {}, "os": None }
>>> eval("os.startfile(r'C:\\Windows\\notepad.exe')", safe_dict)
```

内置函数eval

- 建议采用ast模块

import ast
ast.literal_eval(cmd)

test_eval.py

```
import os
import traceback
import ast
def test_eval():
    cmd = "__import__('os').startfile(r'C:\\Windows\\notepad.exe')"
    input('eval %s...'% cmd)
    eval(cmd)
    cmd2 = "os.startfile(r'C:\\Windows\\notepad.exe')"
    input('eval %s...'% cmd2)
    eval(cmd2)
    try:
        safe_dict = {"__builtins__": {}, "os": None }
        input('eval with safe_dict...')
        eval(cmd2,safe_dict)
    except :
        traceback.print_exc()

    try:
        input('ast.literal_eval %s...' % cmd2)
        ast.literal_eval( cmd )
    except ValueError :
        traceback.print_exc()
```

ValueError: malformed node or string: <_ast.Call object at 0x02DA6210>

except子句

- 在异常出现时，会按照顺序匹配异常，一旦匹配执行其中的异常块，后面的except块不再执行
- 建议尽量显式捕捉可能会出现的异常，并编写具有针对性的代码
- 如果要捕获所有异常(不建议)，最后一个except为：
  - **except BaseException:**
  - **except:** 等价于except BaseException
- **except Exception as e 捕获绝大部分异常。**除Exception外还包括KeyboardInterrupt，SystemExit等异常，这些异常一般不建议捕获
- else子句是可选的，在没有异常出现时执行
  - else子句中的代码在执行过程中也可能出现异常，如果没有捕获，将抛出异常
  - 也可不使用else子句，将其放在try结构包含的语句块的最后，顺序执行。但这些代码执行过程中出现的异常可能被当前try结果的except子句捕获

division.py

```
try:
    x = eval(input('请输入被除数: '))
    y = eval(input('请输入除数: '))
    z = x / y
except ZeroDivisionError:
    print('除数不能为零')
except TypeError:
    print('被除数和除数应为数值类型')
except NameError as e:
    print('变量不存在')
except Exception as e:
# except 等价于 except BaseException
    print(type(e), e)
else:
    print(x, '/', y, '=', z)
```

except子句

- 当有多个except块而且处理相同时，可以使用**元组的形式**，表示出现的异常匹配元组中的**任意一种异常**时，执行相应的except语句块

```
def division():
    try:
        x = eval(input('请输入被除数: '))
        y = eval(input('请输入除数: '))
        z = x / y
    except (ZeroDivisionError, TypeError, NameError) as e:
        print(type(e), e)
        print('您的输入有误')
        # t = e
    except Exception as e:
        print(type(e), e)
    else:
        print(x, '/', y, '=', z)
```

```
# 相当于
except Exception as e:
 try:
    pass
 finally:
    del e
```

expression as instance：instance为异常对象
在exception block引入的变量在执行完该block仍然可用，但是instance仅在该block中可用

finally子句

- 不管有没有异常，异常是否捕获都会执行finallyBody：
  - 可进行清理工作，以便释放资源

```
try:
    <body>
finally:
    <finallyBody>
```

- try语句也可仅包含finally，表示不捕获任何异常，但是不管异常是否出现会执行finallyBody

```
try:
    3/0
except:
    print(3)
finally:
    print(5)
# 3
# 5
```

```
try:
    print('....')
    main()
finally:
    print('good luck!')
```

```
try:
    <body>
except [expression [as identifier]]:
    <exceptionBody>

except [expression [as identifier]]:
# 可以有多个except子句，出现异常时按序匹配找到对应的exception为止。不带表达式的except等价于except BaseException，应该是最后一个
    <exceptionBody>

else:    # 可选的，没有异常出现时会执行该块的代码
    <elseBody>

finally:   # 可选的，不管有没有异常，异常是否捕获都要执行
    <finallyBody>
```

finally子句

- 不管有没有异常出现，有没有捕获，finally代码都会执行
- 如果异常没有被捕获（记录该异常），finally代码执行完后**重新抛出异常**，但是：
  - 如果**finally代码中出现return以及break，则异常被取消**，不会再抛出
  - python3.7之前，finally代码中不能使用continue语句， python3.8取消了这一限制，显然finally部分有continue也会导致异常被取消
  - (try语句在循环中才可使用break和continue，在函数体内才可使用return)
- 一些特殊的情形的总结：
  - 如果在try body中执行到break/continue/return时，不会执行else block的内容
  - **finally一定会执行，如果finally块执行到break/continue/return时，该语句起作用，之前本来要执行的动作不再执行**
  - 只有在没有异常，或有异常且已捕获，且没有执行到break/continue/return时，才会顺序执行try语句后的语句

division2.py

```
def demo_div(a, b):
  try:
    return a/b
  except ZeroDivisionError:
    print("division by zero!")
  finally:
    return -1
```

```
>>> demo_div(1, 0)
division by zero!"
-1
>>> demo_div(1, 2)
-1
>>> demo_div(1, '2')
#虽然有异常，但是被取消了
-1
```

```
while True:
    try:
        1/0
    finally:
        break
```

assert语句

- 断言语句: **assert** expression[,reason]
- 对于必须满足的条件expression进行验证
- 要求expression真值判断的结果为真，这样才会顺序执行assert语句之后的语句
  - 如果表达式expression判断为假，则**抛出异常AssertionError**，可选的reason给出了原因

```
def assert_divzero( ):
    a = int(input('请输入被除数:'))
    b = int(input('请输入除数:'))
    assert b!=0, '除数不能为0!'
    c = a/b
    print(a,'/',b,'=',c)
```

assert_divzero.py

```
>>> assert_divzero()
请输入被除数:15
请输入除数:0
Traceback (most recent call last):
assert_divzero(), line 190, in assert_divzero
    assert b!=0, '除数不能为0!'
AssertionError: 除数不能为0!
```

```
if __debug__:
    if not expression:
        raise AssertionError
      # raise AssertionError(reason)
```

什么时候__debug__为True?

- 目前的实现中，正常情况下，__debug__为True
- 当Python脚本以-O选项执行时，__debug__为False，assert语句不会产生任何代码，相当于pass语句

```
>>> b = 0
>>> assert(b != 0)   # assert b != 0
Traceback (most recent call last):
AssertionError
>>> assert(b != 0, 'b不能等于0')
<stdin>:1: SyntaxWarning: assertion is always
true, perhaps remove parentheses?
```

回溯traceback

发生异常时，Python可以回溯异常，给出大量的提示

- 如果已经捕获了异常，但是要输出调用栈信息，可使用traceback模块

- print_exc()会输出最近异常（包括调用栈在内）信息
- format_exc与print_exc()类似，只是返回的是字符串

traceback_demo.py

```
import traceback
def traceback_demo():
    try:
        C()
    except:
        traceback.print_exc()
```

```
def A():
    1/0

def B():
    A()

def C():
    B()
```

```
>>> traceback_demo()
Traceback (most recent call last):
  File "C:\Users\dlmao\Documents\src\PythonClass\chap8\traceback_demo.py", line 19, in traceback_demo
    C()
  File "C:\Users\dlmao\Documents\src\PythonClass\chap8\traceback_demo.py", line 14, in C
    B()
  File "C:\Users\dlmao\Documents\src\PythonClass\chap8\traceback_demo.py", line 11, in B
    A()
  File "C:\Users\dlmao\Documents\src\PythonClass\chap8\traceback_demo.py", line 8, in A
    1/0
ZeroDivisionError: division by zero
```