```
"""
    复习
    异常处理
        异常:程序不在继续向后执行,而是返回给调用者.
        处理:让异常状态改变正常流程(向后执行)
    迭代:重复(每次参照上次结果)
        可迭代对象
            标志:__iter__(),返回值是迭代器.
            作用:可以参与 for
        迭代器
            标志:__next__(),返回值是聚合对象的元素
            作用:可以迭代(挨个取元素)
        语法:
        class 可迭代对象:
            def __iter__(self):
                return 迭代器(数据)
        class 迭代器:
            def __init__(self,参数):
                self.聚合对象 = 参数
            def __next__(self):
                if 没有元素了:
                    raise StopIteration
                return 元素
        for item in 可迭代对象():
            pass
        iterator = 可迭代对象().__iter__()
        while True:
            try:
                item = iterator.__next__()
            except StopIteration:
                break
```

迭代

每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。例如：循环获取容器中的元素。

可迭代对象 **iterable**

1. 定义：具有__iter__函数的对象，可以返回迭代器对象。
2. 语法
   -- 创建：
   class 可迭代对象名称:
   def __iter__(self):
   return 迭代器
   -- 使用：
   for 变量名 in 可迭代对象:
   语句
3. 原理：
   迭代器 = 可迭代对象.__iter__()
   while True:
   try:
   print(迭代器.__next__())
   except StopIteration:
   break

"""

```
   迭代 --> yield
   练习:exercise02
"""
class MyRange:
   def __init__(self, end):
      self.end = end
   def __iter__(self):
      number = -1
      while number < self.end - 1:
         number += 1
         yield number
      # print("准备数据")
      # yield 0
      # print("准备数据")
      # yield 1
      # print("准备数据")
      # yield 2
      # print("准备数据")
      # yield 3
      # print("准备数据")
      # yield 4
      # yield 在标记,你看见的代码在执行过程中会转换为迭代器.
      # 如何生成迭代器的代码?
my = MyRange(99999999999999999999999999)
```

```
iterator = my.__iter__()
while True:
    try:
        item = iterator.__next__()
        print(item)
    except StopIteration:
        break
# 调用一次  计算一次  返回一次
# 没有将所有结果存储在内存中
# for item in MyRange(5):#[0 1 2 3 4]
#     print(item)
```

"""

```
    yield --> 生成器
    练习:exercise03
    练习:exercise04
    练习:exercise05
"""
'''
class MyRange:
    def __init__(self, end):
        self.end = end
    def __iter__(self):
        number = -1
        while number < self.end - 1:
            number += 1
            yield number
my = MyRange(5)
iterator = my.__iter__()
while True:
    try:
        item = iterator.__next__()
        print(item)
    except StopIteration:
        break
'''
def my_range(end):
    number = -1
    while number < end - 1:
        number += 1
        yield number
```

```
# 调用一次 计算一次  返回一次
# 没有存储所有结果
for item in my_range(999999999999999999999999):
    print(item)

"""
    生成器表达式
    练习:exercise06
    练习:exercise07
"""
list01 = [4,5,"a",67,3.5,"b",True]
'''
def find_str():
    # result = []
    # for item in list01:
    #     if type(item) == str:
    #         result.append(item)
    # return result
    return [item for item in list01 if type(item) ==
str]
re = find_str()
for item in re:
    print(item)
'''
# 生成器函数
# def find_str():
#     for item in list01:
#         if type(item) == str:
#             yield item
# re = find_str()
# for item in re:
#     print(item)
# 生成器表达式
re = (item for item in list01 if type(item) == str)
for item in re:
    print(item)
```

迭代器对象 **iterator**

1. 定义：可以被 next()函数调用并返回下一个值的对象。

2. 语法

```
class 迭代器类名:
  def __init__(self, 聚合对象):
    self.聚合对象= 聚合对象

  def __next__(self):
    if 没有元素:
      raise StopIteration
  return 聚合对象元素
```

3. 说明：

-- 聚合对象通常是容器对象。

4. 作用：使用者只需通过一种方式，便可简洁明了的获取聚合对象中各个元素，而又无需了解其内部结构。

生成器 generator

1. 定义：能够动态(循环一次计算一次返回一次)提供数据的可迭代对象。
2. 作用：在循环过程中，按照某种算法推算数据，不必创建容器存储完整的结果，从而节省内存空间。数据量越大，优势越明显。
3. 以上作用也称之为延迟操作或惰性操作，通俗的讲就是在需要的时候才计算结果，而不是一次构建出所有结果。

生成器函数

1. 定义：含有 yield 语句的函数，返回值为生成器对象。
2. 语法

```
-- 创建：
  def 函数名():
      …
      yield 数据
      …
    -- 调用：
     for 变量名 in 函数名():
         语句
```

3. 说明：

-- 调用生成器函数将返回一个生成器对象，不执行函数体。

-- yield 翻译为”产生”或”生成”

4. 执行过程：

(1) 调用生成器函数会自动创建迭代器对象。

(2) 调用迭代器对象的__next__()方法时才执行生成器函数。

(3) 每次执行到 yield 语句时返回数据，暂时离开。

(4) 待下次调用__next__()方法时继续从离开处继续执行。

5. 原理：生成迭代器对象的大致规则如下
   -- 将 yield 关键字以前的代码放在 next 方法中。
   -- 将 yield 关键字后面的数据作为 next 方法的返回值。

内置生成器

枚举函数 **enumerate**

1. 语法：
   for 变量 in enumerate(可迭代对象):
     语句

   for 索引, 元素 in enumerate(可迭代对象):
     语句
2. 作用：遍历可迭代对象时，可以将索引与元素组合为一个元组。

zip

1. 语法：
   for item in zip(可迭代对象 1, 可迭代对象 2….):
     语句
2. 作用：将多个可迭代对象中对应的元素组合成一个个元组，生成的元组个数由最小的可迭代对象决定。

生成器表达式

1. 定义：用推导式形式创建生成器对象。
2. 语法：变量 = (表达式 for 变量 in 可迭代对象 [if 真值表达式])

函数式编程

1. 定义：用一系列函数解决问题。
   -- 函数可以赋值给变量，赋值后变量绑定函数。
   -- 允许将函数作为参数传入另一个函数。
   -- 允许函数返回一个函数。
2. 高阶函数：将函数作为参数或返回值的函数。

```
"""
    函数式编程 -- 语法
"""
def fun01():
    print("fun01 执行喽")
```

```
# 1. 函数可以赋值给变量
a = fun01
# 通过变量调用函数
a()# 活的
fun01()# 死的
# 2. 将函数作为参数传递
# 如果传入基本数据类型(整数/小数/字符串..),称之为传入数据
# 如果传入函数,称之为传入行为/逻辑/算法
def fun02(func):
    print("fun02执行喽")
    func()
fun02(fun01)
```

```
"""
    函数式编程 -- 思想
    17:10
"""
list01 = [4,5,5,6,767,8,10]
"""
# 1. 找出所有偶数
def find01():
    for item in list01:
        if item % 2 == 0:
            yield item

# 2. 找出所有奇数
def find02():
    for item in list01:
        if item % 2:
            yield item
# 3. 找出所有大于10
def find03():
    for item in list01:
        if item  > 10:
            yield item
"""
# "封装":提取变化
def condition01(item):
    return item % 2 == 0
def condition02(item):
    return item % 2
```

```python
def condition03(item):
    return item  > 10
# "继承":隔离变化
# 根据任何条件,在任何可迭代对象中查找多个元素.
def find(target,func):
    for item in target:
        # if item > 10:
        # if condition03(item):
        if func(item):
            yield item
# "多态":执行变化
for item in find(list01,condition03):
    print(item)
```