```
"""
    标准库模块
    时间 time
    练习:exercise01.py
    练习:exercise02.py
"""
import time
# 1. 获取当前时间戳(从1970年1月1日到现在经过的秒数)
# 1563326340.661173
print(time.time())
# 2. 获取当前时间元组(年,月,日,小时,分钟,秒,一周的第几天,一年的第几天,夏令时)
time_tuple = time.localtime()
print(time_tuple)
# 3. 时间戳 -->时间元组
print(time.localtime(15633263))
# 4. 时间元组 -- 时间戳
print(time.mktime(time_tuple))
# 5. 时间元组 -->  字符串
print(time.strftime("%y/%m/%d %H:%M:%S",time_tuple))
print(time.strftime("%Y/%m/%d %H:%M:%S",time_tuple))
# 6. 字符串 --> 时间元组
# "19/07/17 09:36:48"
print(time.strptime("19/07/17 09:36:48","%y/%m/%d %H:%M:%S"))
```

异常处理 Error

异常

1. 定义：运行时检测到的错误。

2. 现象：当异常发生时，程序不会再向下执行，而转到函数的调用语句。
3. 常见异常类型：
   -- 名称异常(NameError)：变量未定义。
   -- 类型异常(TypeError)：不同类型数据进行运算。
   -- 索引异常(IndexError)：超出索引范围。
   -- 属性异常(AttributeError)：对象没有对应名称的属性。
   -- 键异常(KeyError)：没有对应名称的键。
   -- 为实现异常(NotImplementedError)：尚未实现的方法。
   -- 异常基类 Exception。

处理

1. 语法：
   try:
     可能触发异常的语句
   except 错误类型 1 [as 变量 1]:
     处理语句 1
   except 错误类型 2 [as 变量 2]:
     处理语句 2
   except Exception [as 变量 3]:
     不是以上错误类型的处理语句
   else:
     未发生异常的语句
   finally:
       无论是否发生异常的语句

```
"""
    异常处理
"""
def div_apple(apple_count):
    """
        分苹果
    """
    person_count = int(input("请输入人数:"))# ValueError
    result = apple_count / person_count#
ZeroDivisionError
    print("每个人分到了%d 个苹果"%result)
"""
try:
    div_apple(10)
except:
    # 错误的处理逻辑
    print("出错喽")
"""
"""
```

```
try:
    div_apple(10)
except ValueError:
    print("输入的人数必须是整数")
except ZeroDivisionError:
    print("输入的人数不能是零")
except Exception:
    # 错误的处理逻辑
    print("出错喽")
else:
    print("没有错误执行的代码")
"""
try:
    # 如果有错误
    div_apple(10)
finally:
    # 不处理错误,但有一件非常重要的事情,必须执行.
    print("一定执行的代码")
print("后续逻辑.........")
```

2. 作用：将程序由异常状态转为正常流程。
3. 说明：
   as 子句是用于绑定错误对象的变量，可以省略
   except 子句可以有一个或多个，用来捕获某种类型的错误。
   else 子句最多只能有一个。
   finally 子句最多只能有一个，如果没有 except 子句，必须存在。
   如果异常没有被捕获到，会向上层(调用处)继续传递，直到程序终止运行。

raise 语句

1. 作用：抛出一个错误，让程序进入异常状态。
2. 目的：在程序调用层数较深时，向主调函数传递错误信息要层层 return 比较麻烦，所以人为抛出异常，可以直接传递错误信息。。

自定义异常

1. 定义：
   class 类名 Error(Exception):
   def __init__(self,参数):
   super().__init__(参数)
   self.数据 = 参数
2. 调用：

try:
….
raise 自定义异常类名(参数)
….
except 定义异常类 as 变量名:
变量名.数据

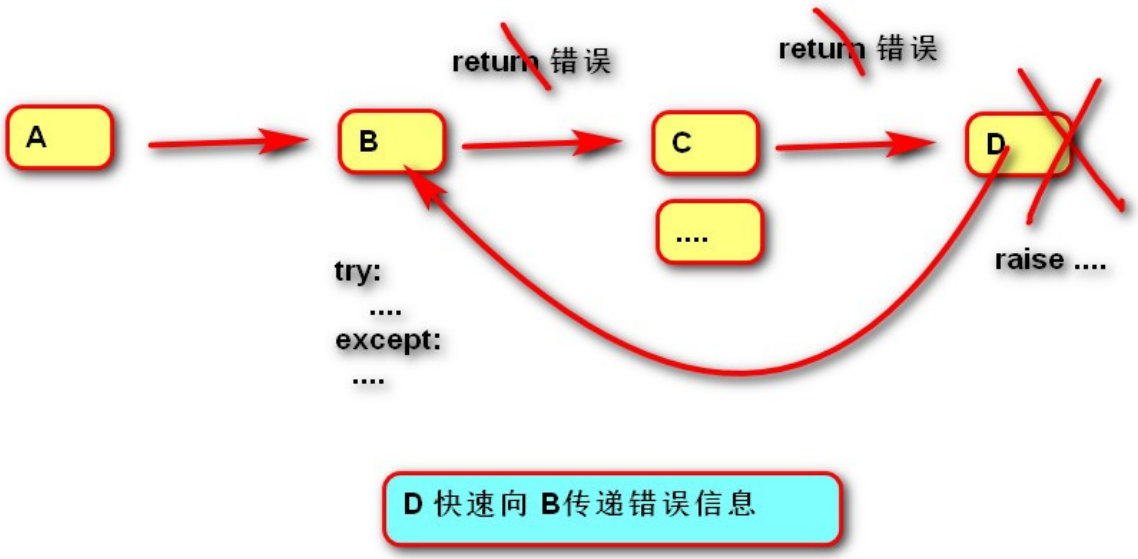
3. 作用：封装错误信息

```
class Wife:
   def __init__(self, age=0):
      self.age = age
   @property
   def age(self):
      return self.__age
   @age.setter
   def age(self,value):
      if 20 <=value <= 30:
         self.__age = value
      else:
         # 抛出/发送  错误信息(异常对象)
         raise ValueError("我不要")
# 接收错误信息
try:
   w01 = Wife(85)
   print(w01.age)
except ValueError as e:
   print(e.args)# 信息
```

```
"""
   自定义异常类
   15:20 上课
"""
class AgeError(Exception):
   def __init__(self, message="", code="", id=0):
      # 消息/错误代码/错误编号....
      self.message = message
      self.code = code
      self.id = id
class Wife:
   def __init__(self, age=0):
      self.age = age
   @property
   def age(self):
```

```
        return self.__age
    @age.setter
    def age(self, value):
        if 20 <= value <= 30:
            self.__age = value
        else:
            # 抛出/发送  错误信息(异常对象)
            raise AgeError("我不要", "if 20 <=value <=
30", 101)
            # 需要传递的信息:消息/错误代码/错误编号....
# 接收错误信息
try:
    w01 = Wife(25)
    print(w01.age)
except AgeError as e:
    print(e.id)  # 信息
    print(e.message)  # 信息
    print(e.code)  # 信息
```

4.



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

"""

```
   使用可迭代对象,迭代器
"""
list01 = [43,45,54,5,67]
# for item in list01:
#     print(item)
# 原理:
# 1. 调用iter方法获取迭代器对象
iterator = list01.__iter__()
while True:
   try:
      # 2. 获取下一个元素
      item = iterator.__next__()# Stop Iteration
      print(item)
      # 3. 遇到"停止迭代"异常,则停止循环.
   except StopIteration:
      break
```

```
# 面试题:
# 能够参与for循环的对象,必须具备什么条件?
# 答:该对象必须具有__iter__()方法
```

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。

"""

```
    做迭代器/可迭代对象
    17:13
"""
# 需求:让技能管理器对象可以参与for循环.
class SkillManager:
    """
        技能管理器
        具有__iter__方法,可迭代对象.
    """
    def __init__(self):
        self.__all_skill = []
    def add_skill(self, skill):
        self.__all_skill.append(skill)
    def __iter__(self):
        # 创建技能迭代器对象(传递需要迭代的数据)
        return SkillIterator(self.__all_skill)
class SkillIterator:
    """
        技能迭代器
    """
```

```
    def __init__(self, data):
        self.__target = data
        self.__index = -1# 返回数据时,会先自增1
    def __next__(self):
        self.__index += 1
        # 当前需要获取的索引 比最大索引还大
        if self.__index  >  len(self.__target) - 1:
            raise StopIteration
        return self.__target[self.__index]
manager = SkillManager()
manager.add_skill("九阳神功")
manager.add_skill("乾坤大挪移")
manager.add_skill("九阴白骨爪")
# for item in manager:
#     print(item)
iterator = manager.__iter__()
while True:
    try:
        # 2. 获取下一个元素
        item = iterator.__next__()  # Stop Iteration
        print(item)
        # 3. 遇到"停止迭代"异常,则停止循环.
    except StopIteration:
        break
```