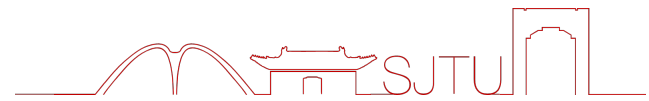




上海交通大学
SHANGHAI JIAO TONG UNIVERSITY



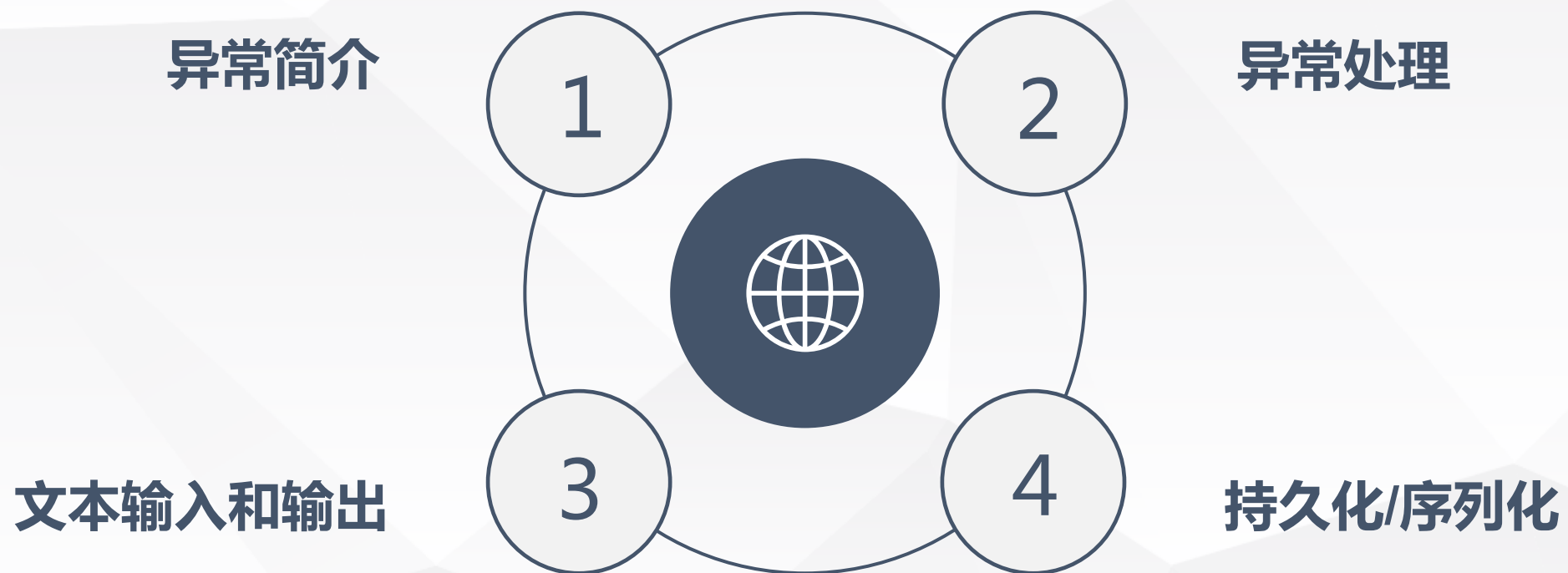
Python基础：异常处理和文件

课程组：叶南阳，郑冠杰，温颖

饮水思源 · 爱国荣校



课程结构



1. 异常简介

① 什么是异常？

- 计算机程序运行时由于外部问题（如硬件错误、输入错误）发生的错误
- 异常（Exception）都是运行时的。
编译时产生的不是异常，而是错误（Error）

② 为什么要进行异常处理？

- 异常机制可以使程序中的异常处理代码和正常业务代码分离开，保证程序代码更加优雅，并可以提高程序的健壮性（robustness）





1. 异常简介

即便语法是正确的，编译没有任何问题，在运行中依然存在出现异常的可能。

即便他们没办法正常运行，他们只是异常而不是错误。

- `>>> 10 * (1/0)`

- Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero

- `>>> 4 + spam*3`

- Traceback (most recent call last):File "<stdin>", line 1, in <module> NameError: name 'spam' is not defined

- `>>> '2' + 2`

- Traceback (most recent call last):File "<stdin>", line 1, in <module>TypeError: Can't convert 'int' object to str implicitly





1. 异常简介



异常小结：

- 即使一个语句或表达式在语法上是正确的，当试图执行它时也可能会导致错误。在执行过程中检测到的错误称为异常，并不是无条件致命的。
- 然而，大多数异常都不会被程序处理，并且会导致如之前所示的错误消息。
- 错误消息的最后一行表示发生了什么。异常有不同的类型，类型将作为消息的一部分打印出来。
- 上一张ppt示例中的异常类型为ZeroDivisionError、NameError和TypeError。



2. 异常处理——Try Except

异常处理机制最核心的两个关键字是Try和Except

Try语句的形式及执行流程：

- try:

可能产生异常的代码块

except [(Error1, Error2, ...) [as e]]:

处理异常的代码块1

except [(Error3, Error4, ...) [as e]]:

处理异常的代码块2

except [Exception]:

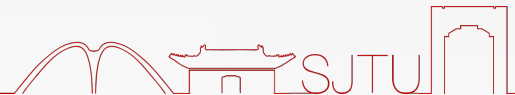
处理其它异常

try 块有且仅有一个

except 代码块可以有多个

每个 except 块都可以同时处理多种异常

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```





2. 异常处理——Try Except



Try语句执行过程：

- 首先执行 try 中的代码块（try 和 except 关键字），如果执行过程中出现异常，系统会自动生成一个异常类型，并将该异常提交给 Python 解释器，此过程称为捕获异常。
- 当 Python 解释器收到异常对象时，会寻找能处理该异常对象的 except 块，如果找到合适的 except 块，则把该异常对象交给该 except 块处理。如果 Python 解释器找不到处理异常的 except 块，则程序运行终止，Python 解释器也将退出。



2. 异常处理——Raise

Raise

-  raise语句允许程序员强制发生指定的异常。
-  如果您需要确定是否引发了异常但不打算处理它可以使用更简单的raise语句形式重新引发异常:

```
In [2]: raise NameError('HiThere')
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-2-72c183edb298> in <module>  
----> 1 raise NameError('HiThere')  
  
NameError: HiThere
```

```
In [3]: raise ZeroDivisionError()  
raise ZeroDivisionError  
raise TypeError  
raise TypeError()  
raise ValueError
```

```
-----  
ZeroDivisionError                        Traceback (most recent call last)  
<ipython-input-3-dd0a82cd4bef> in <module>  
----> 1 raise ZeroDivisionError()  
      2 raise ZeroDivisionError  
      3 raise TypeError  
      4 raise TypeError()  
      5 raise ValueError
```

ZeroDivisionError:





2. 异常处理——try ... finally

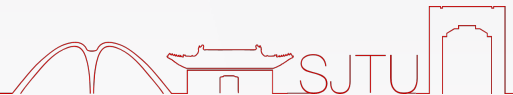
- ④ try语句还有另一个可选子句，用于定义在任何情况下都必须执行的清理操作。
- ④ 无论是否发生异常，finally子句总是在离开try语句之前执行。
- ④ 当一个异常在try子句中发生并且没有被except子句处理(或者它在except或else子句中发生)时，它会在finally子句执行后被重新引发。
- ④ 当try语句的任何其他子句通过break、continue或return语句离开时，finally子句也会被执行。
- ④ 如您所见，finally子句在任何情况下都会执行。通过分隔两个字符串引发的TypeError不会由except子句处理，因此在finally子句执行后重新引发。
- ④ 在实际应用程序中，finally子句用于释放外部资源(如文件或网络连接)，而不管资源的使用是否成功。

```
In [4]: try:
        raise KeyboardInterrupt
        finally:
            print('Goodbye, world !')
```

Goodbye, world !

```
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-4-ee8884cd7395> in <module>
      1 try:
----> 2     raise KeyboardInterrupt
      3 finally:
      4     print('Goodbye, world !')
```

KeyboardInterrupt:



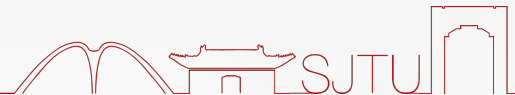


2. 异常处理

- try...except语句有一个可选的else子句，当它出现时，必须跟在所有except子句后面。对于在try子句没有引发异常时必须执行的代码，它是有用的。例如：

```
import sys
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

- 使用else子句要比向try子句中添加额外代码更好，因为它避免了意外捕获到由try...except语句保护的代码所没有引发的异常。
- 只有当try子句没有引发异常时，才会执行else.





2. 异常处理——更多关于Except

- ❶ 一个try语句可以有多个except子句，用于指定不同异常的处理程序。最多将执行一个处理程序。处理程序只处理相应的try子句中发生的异常，而不处理同一try语句的其他处理程序中发生的异常。except子句可以将多个异常命名为带括号的元组，例如：
 - ❶ ... except (RuntimeError, TypeError, NameError):
 - ❶ ... pass #do nothing
- ❷ pass: 当一个语句在语法上是必需的，但你不执行任何命令或代码时使用。pass语句是一个空操作; 当它执行时，什么也不会发生。在代码最终要用到但还没有编写好的地方，pass也很有用。
- ❸ 最后一个except子句可以省略异常名称，作为通配符。使用时要格外小心，因为用这种方法很容易掩盖真正的编程错误! 它还可以用于打印错误消息，然后重新引发异常(允许调用者也处理异常):





2. 异常处理——更多关于Except



- ❶ 程序可以通过创建一个新的异常类来命名自己的异常(关于Python类的更多信息，请参阅类)。异常通常应该直接或间接地派生自Exception类。
- ❷ 可以定义异常类，它们可以做任何其他类可以做的事情，但通常保持简单，通常只提供一些属性，这些属性允许异常处理程序提取错误信息。
- ❸ 当创建一个可能引发几个不同错误的模块时，一个常见的实践是为该模块定义的异常创建一个基类，并为不同的错误条件创建特定的异常类的子类。



2. 异常处理——更多关于Except

❶ Error checks vs exception handling

```
#with checks

n = None
while n is None:
    s = input("Please enter an integer: ")
    if s.lstrip('-').isdigit():
        n = int(s)
    else:
        print("%s is not an integer." % s)
```

```
# with exception handling

n = None
while n is None:
    try:
        s = input("Please enter an integer: ")
        n = int(s)
    except ValueError:
        print("%s is not an integer." % s)
```



2. 异常处理

小结：

右侧这个案例展现了绝大多数异常处理中的用法。

```
In [5]: def devide(x, y):  
        try:  
            result = x / y  
        except ZeroDivisionError:  
            print("devision by zero!")  
        else:  
            print("result is", result)  
        finally:  
            print("executing finally clause")  
        devide(2, 1)  
        devide(2, 0)  
        devide("2", "1")
```

```
result is 2.0  
executing finally clause  
devision by zero!  
executing finally clause  
executing finally clause
```

TypeError Traceback (most recent call last)

```
<ipython-input-5-b5053ce19f8b> in <module>  
    10 devide(2, 1)  
    11 devide(2, 0)  
----> 12 devide("2", "1")  
  
<ipython-input-5-b5053ce19f8b> in devide(x, y)  
      1 def devide(x, y):  
      2     try:  
----> 3         result = x / y  
      4     except ZeroDivisionError:  
      5         print("devision by zero!")
```

TypeError: unsupported operand type(s) for /: 'str' and 'str'





2. 异常处理

练习：

- 这个函数在列表的字典中向列表添加一个元素。重写它，使用try-except语句处理可能的KeyError，如果提供的名称的列表还不存在于字典中，而不是事先检查是否存在。在try-except语句块中包含else和finally子句：

```
def add_to_list_in_dict(thedict, listname, element):  
    if listname in thedict:  
        l = thedict[listname]  
        print("%s already has %d elements." % (listname, len(l)))  
    else:  
        thedict[listname] = []  
        print("Created %s." % listname)  
  
    thedict[listname].append(element)  
    print("Added %s to %s." % (element, listname))
```




3. 文本输入和输出

❶ Q：为什么程序运行过程中需要文件的存在？

❷ A：因为内存是有限的，并且我们希望能够保存一些重要的数据。

❸ 当程序运行时，它的数据存储在随机存取存储器(RAM)中。RAM既快又便宜，但当程序结束或计算机关闭时，RAM中的数据就会丢失。

❹ 为了使数据在下一次启动计算机和启动程序时可用，必须将数据写入非易失性存储介质，如硬盘驱动器、usb驱动器或CD-RW。

❺ 非易失性存储介质上的数据存储在称为文件的介质上的指定位置。

❻ 使用文件很像使用笔记本。要使用笔记本，必须打开它。完成后，必须关闭。

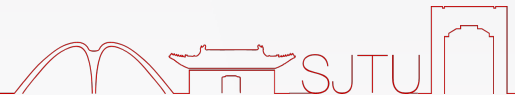


1：文本输入和输出



函数介绍：open()

- 在可以读写文件之前，必须使用Python的内置open()函数打开它。这个函数创建一个file对象，该对象将被用来调用与其关联的其他支持方法
- 格式：file object = open(file_name [, access_mode][, buffering])
- 参数介绍：
 - file_name-file_name为字符串值，包含要访问的文件名称。
 - access_mode-access_mode决定打开文件的方式，即读、写、追加等。下表给出了一个完整的可能值列表。这是一个可选参数，默认的文件访问模式是read (r)。
 - buffering-当buffer值设置为0时，表示不进行缓存。如果缓冲值为1，则在访问文件时执行行缓冲。如果您将缓冲值指定为大于1的整数，那么将按照指定的缓冲大小执行缓冲操作。如果为负数，则缓冲块大小为系统默认值(默认行为)。





3. 文本输入和输出

open () 的相关格式

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	<u>Opens a file for writing only.</u> <u>Overwrites</u> the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for <u>appending</u> . <u>The file pointer is at the end of the file if the file exists.</u> That is, <u>the file is in the append mode.</u> <u>If the file does not exist, it creates a new file for writing.</u>
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.





3. 文本输入和输出

⊗ File对象简介

⊗ 下面是与文件对象相关的所有属性的列表

⊗ File.closed:如果文件关闭返回true，否则返回false

⊗ File.mode:返回文件打开时的访问模式

⊗ File.name:返回文件名

⊗ 当你使用open（）打开了一个文件，你需要在完成操作后使用close（）去关闭它。

```
f = open("file.txt", "wb")
print ("Name of the file: ", f.name)
print ("Closed or not : ", f.closed)
print ("Opening mode : ", f.mode)
f.close()
```

```
('Name of the file: ', 'file.txt')
('Closed or not : ', False)
('Opening mode : ', 'wb')
```



3. 文本输入和输出

④ 函数介绍write()

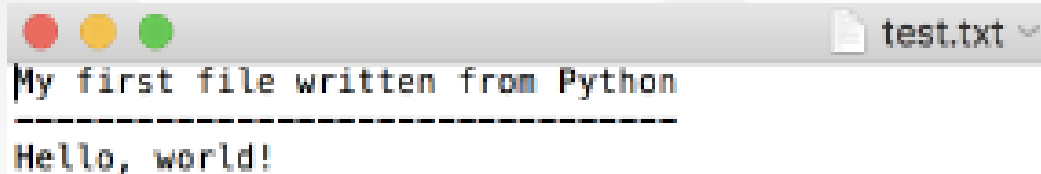
④ File对象可以调用write()将数据写入文件

④ f.write (string)将string的内容写入文件，返回写入的字符数。write()方法不会在字符串的末尾添加换行符('\n')

④ 在“w”模式下(意味着写)，如果磁盘上没有名为test.txt的文件，它将被创建。如果已经有一个，它将被替换为我们正在写的文件

④ 总是成对地open()和close()

```
In [6]: myfile = open("test.txt", "w")
        myfile.write("My first file written from Python\n")
        myfile.write("-----")
        myfile.write("Hello, world\n")
        myfile.close()
```



```
test.txt
My first file written from Python
-----
Hello, world!
```



3. 文本输入和输出

④ 函数介绍readline() & read()

④ 有两种主要的方法从文件中读取数据:

④ readline()方法返回换行符之前的所有内容(包括换行符)。“n”会被读出。

④ 如果读取到:" \n "或" ", 则文件结束。

④ read([n])方法从文件中读取n个字节。默认情况下, 它将把文件的完整内容读入一个字符串

```
In [7]: f = open("somefile.txt")
        content = f.read()
        f.close()
        words = content.split()
        print("There are {0} words in the file.".format(len(words)))
```





3. 文本输入和输出

❶ 二进制文件 (Binary files)

❶ 存放照片、视频、zip文件、可执行程序等的文件称为二进制文件:它们没有被组织成行，不能用普通的文本编辑器打开。

❶ python对二进制文件的处理同样简单，但是当我们从文件中读取时，我们将获得字节而不是字符串。

```
f = open("test.zip", "rb")      # r: read; b: binary file
g = open("thecopy.zip", "wb")    # w: write; b: binary file while True:

buf = f.read(1024)  # read 1024 bytes
while True:
    if len(buf) == 0:  # if buf is empty, then it is the end of the file.
        break
    g.write(buf)
    buf = f.read(1024)
f.close()
g.close()
```





3. 文本输入和输出

- 由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：忘记close()是编程中常见的错误。解决办法是回避它。

```
try:
    f = open('/path/to/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

- python的with语句提供了一种非常方便的方式来处理这种情况，在这种情况下，即使在某个时刻引发了异常，文件在套件结束后也会被正确地关闭。
- 这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。。

```
In [ ]: with open("test_with.txt", "w") as f:
        f.write("aa\n")
        f.write("--\n")
        f.write("bb\n")
```





为什么需要数据持久化

- 在程序运行的过程中，所有的变量都是在内存中
- 把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫 pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。



Python提供了pickle，dbm等数据持久化模块。

- pickle模块将具有层次的数据结构转换为有序的二进制比特流
- dbm模块全称为database manager，是访问UNIX数据库的一类Python接口





数据持久化：pickle

🔴 pickle.dump()可以将Python程序里的变量直接序列化到磁盘上，不用考虑变量的类型

```
1 import pickle
2 a = 10
3 f = open('/home/ycsongacemap/Lab/AI-Practice/test_1.pkl', 'wb')
4 pickle.dump(a, f)
```

[13] ✓ 0.4s

🔴 pickle.load()可以将磁盘里的序列化文件读取为Python变量

```
1 import pickle
2 f = open('/home/ycsongacemap/Lab/AI-Practice/test_1.pkl', 'rb')
3 a = pickle.load(f)
4 f.close()
5 print(a)
```

[21] ✓ 0.1s

... 10

+ 代码

+ 标记





感谢聆听

饮水思源 爱国荣校