1.了解错误与异常

在Python中，错误（Error）是指语法或逻辑不正确导致程序无法正常运行的问题。与异常不同的是，错误通常会导致程序崩溃或无法执行。

语法错误（Syntax Error）：由于代码中存在拼写错误、缺少冒号等情况而导致的编译错误。
名称错误（Name Error）：使用未定义的变量或函数名称导致的错误。
类型错误（Type Error）：尝试针对无效的数据类型进行操作导致的错误。
数组越界错误（Index Error）：尝试访问列表、元组或字典中不存在的元素导致的错误 。
文件不存在错误（File Not Found Error）：尝试打开不存在文件导致的错误。

及时识别和处理这些错误非常重要。可以通过调试工具、错误信息输出和捕获机制等方式来诊断和修复程序中存在的错误。

错误类型	描述
SyntaxError	语法错误，通常指代码有拼写错误、缺少括号、引号配对错误等。
NameError	名称错误，通常指变量或函数名称未定义或拼写错误。
TypeError	类型错误，通常因为尝试使用不支持的数据类型进行操作，例如对整数类型执行字符串方法。
IndexError	数组越界错误，通常因为尝试访问列表、元组或字典中不存在的索引导致。
KeyError	字典键错误，通常因为尝试访问不存在的字典键导致。
ValueError	值错误，通常指传递给函数的参数无效，例如使用负数调用平方根函数。
ZeroDivisionError	零除错误，通常因为尝试做除法时分母为零导致。

错误类型	描述
FileNotFound Error	文件不存在错误，通常因为尝试打开不存在的文件导致。
……	……

异常（Exception）是指程序运行时出现的问题。这些问题可能包括语法错误、逻辑错误或系统故障等情况，例如打开一个不存在的文件、除数为零、尝试访问无效的内存地址等。

当Python解释器遇到异常时，它会停止当前代码块的执行，并从调用堆栈中搜索异常处理程序来处理该异常。如果没有找到合适的处理程序，将会抛出未处理的异常并终止程序运行。

异常囊括错误 错误导致异常

```
# python的异常结构都是继承于BaseException
Exception 是最常见的异常类

BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
 ├── Exception
 |    ├── StopIteration
 |    ├── StopAsyncIteration
 |    ├── ArithmeticError
 |    |    ├── FloatingPointError
 |    |    ├── OverflowError
 |    |    └── ZeroDivisionError
 |    ├── AssertionError
 |    ├── AttributeError
 |    ├── BufferError
 |    ├── EOFError
 |    ├── ImportError
 |    |    └── ModuleNotFoundError
 |    ├── LookupError
 |    |    ├── IndexError
 |    |    ├── KeyError
 |    |    └── _NamespacePathFull
 |    ├── MemoryError
 |    ├── NameError
 |    |    └── UnboundLocalError
 |    ├── OSError
 |    |    ├── BlockingIOError
 |    |    ├── ChildProcessError
 |    |    ├── ConnectionError
 |    |    |    ├── BrokenPipeError
 |    |    |    ├── ConnectionAbortedError
 |    |    |    └── ConnectionRefusedError
 |    |    ├── FileExistsError
 |    |    ├── FileNotFoundError
 |    |    ├── InterruptedError
 |    |    ├── IsADirectoryError
 |    |    ├── NotADirectoryError
```

```
 |    |    ├── PermissionError
 |    |    ├── ProcessLookupError
 |    |    ├── TimeoutError
 |    |    └── UnsupportedOperation
 |    ├── ReferenceError
 |    ├── RuntimeError
 |    |    ├── NotImplementedError
 |    |    ├── RecursionError
 |    |    └── _TimeoutError
 |    ├── SyntaxError
 |    |    ├── IndentationError
 |    |    └── TabError
 |    ├── SystemError
 |    ├── TypeError
 |    ├── ValueError
 |    |    ├── UnicodeError
 |    |    └── _RemoteTraceback
 |    ├── Warning
 |    |    ├── BytesWarning
 |    |    ├── DeprecationWarning
 |    |    ├── FutureWarning
 |    |    ├── ImportWarning
 |    |    ├── PendingDeprecationWarning
 |    |    ├── ResourceWarning
 |    |    ├── RuntimeWarning
 |    |    ├── SyntaxWarning
 |    |    ├── UnicodeWarning
 |    |    └── UserWarning
 |    └── _OptionError
 └── SystemError

# 这个继承结构表明，所有的异常类都是从BaseException中派生而来的。其中，Exception是最常见的异常类，包括了很多其他的具体异常，例如ArithmeticError、ImportError、TypeError等。
```

Python中异常类的继承关系非常复杂，其中的每个异常类都有其特定的用途和含义。熟悉这些异常类的继承关系可以帮助我们在处理程序中出现的异常情况时更加得心应手。

```
Traceback (most recent call last):
  File "D:\pycharm_文件\pythonProject\demo\day_22\异常处理.py", line 11, in
<module>
    print(data[4])
IndexError: list index out of range

# 错误回溯信息
```

2.异常处理-捕捉

在计算机编程中，异常是指出现了不可预测、不符合正常规则和预期的情况。这些异常可能会导致程序崩溃、数据丢失等问题。为了解决这些问题，通常采取异常处理机制来对这些异常进行捕获和处理，以保证程序的正确性和稳定性。

在Python中，当程序执行过程中发生异常时，解释器会寻找相应的异常处理逻辑来处理这些异常。如果存在合适的异常处理逻辑，则程序可以继续正常执行，否则将会抛出一个未被处理的异常。

Python中提供了try...except语句和finally子句来实现异常处理机制。

```
try:
    # 可能会出现异常的代码
except [异常类型1]:
    # 当出现异常类型1时执行的代码
except [异常类型2]:
    # 当出现异常类型2时执行的代码
except:
    # 当出现其他异常时执行的代码
else:
    # 当没有出现任何异常时执行的代码
finally:
    # 无论是否出现异常都会执行的代码
```

try语句块用于包含可能出现异常的代码，然后except语句块用于定义当出现特定异常时需要采取的操作。
如果没有指定具体的异常类型，则except语句块将捕获所有类型的异常。

在try...except语句中，可以使用多个except子句来处理各种不同类型的异常。
同时，还可以使用else子句来在没有任何异常发生时执行一些特定的操作。
最后，我们可以使用finally子句来定义无论是否出现异常都需要执行的代码，例如释放资源等。

```
try:
    num1 = int(input("请输入第一个数："))
    num2 = int(input("请输入第二个数："))
    result = num1 / num2
    print(f"{num1}除以{num2}的结果为：{result}")
except ZeroDivisionError:
    print("除数不能为零！")
except ValueError:
    print("输入的不是有效的数字！")
except:
    print("发生了其他错误！")
else:
    print("没有出现任何错误！")
finally:
    print("程序结束！")
```

3.异常处理-主动抛出

raise语句用于手动抛出异常

```
# 异常类型表示需要抛出的异常类型，错误信息表示异常相关的错误信息。
```

```
def divide(num1, num2):
    if num2 == 0:
        raise ZeroDivisionError("除数不能为零！")
    return num1 / num2

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(f"发生了错误：{e}")
else:
    print(f"结果为：{result}")
finally:
    print("程序结束！")

# 定义了一个divide函数，在其中使用了raise语句手动抛出了一个除数为零的异常。
# 在主程序中，我们可以采用try...except语句来捕获这个异常并进行处理。
```

4.异常处理--自定义异常类

有时候，在程序中可能需要自定义一些异常类来表达特定的错误或者状态信息。在Python中，可以通过继承Exception类来定义自己的异常类。

```
class MyException(Exception):
    def __init__(self, message):
        super().__init__(message)
        self.message = message

    def __str__(self):
        return f"MyException: {self.message}"

try:
    raise MyException("这是一个自定义异常！")
except MyException as e:
    print(e)
finally:
    print("程序结束！")

# 首先定义了一个MyException异常类，它继承自Python内置的Exception类，并实现了__init__()和__str__()方法来初始化异常对象并定义异常信息输出格式。然后，在主程序中，使用raise语句手动抛出了一个MyException异常，并使用try...except语句捕获这个异常并进行处理
```

5.异常处理-断言

断言（Assert）是一种常用的调试工具，它用于对程序运行时的某些条件进行检查，并捕获由于不符合预期条件而导致程序崩溃或出现错误的情况。

assert语句用于在程序运行期间进行条件检查。
如果指定的条件表达式的结果为False，则会抛出一个AssertionError异常。

```
assert condition, [message]

# condition表示需要进行检查的条件表达式，message是可选参数，用于指定出现异常时输出的错误信息。
```

```
def divide(num1, num2):
    assert num2 != 0, "除数不能为零！"
    return num1 / num2

try:
    result = divide(10, 0)
except AssertionError as e:
    print(f"发生了错误：{e}")
else:
    print(f"结果为：{result}")
finally:
    print("程序结束！")

# 定义了一个divide函数，在其中使用了assert语句来检查除数是否为零。如果除数为零，则会抛出一个AssertionError异常。在主程序中，我们可以采用try...except语句来捕获这个异常并进行处理。
```

在程序开发和测试阶段使用assert语句可以帮助我们快速发现程序中的问题，但在生产环境中使用时需要慎重考虑，因为assert语句可能会影响程序的性能和稳定性。

当然，在一些特殊的情况下，例如对于安全性要求较高的系统或者对程序正确性要求较高的场景，使用assert语句是非常有用的

6.上下文管理器

```
with open()  # 可以不用去写close
```

with语句之所以这么强大，就是因为背后有上下文管理器作为支撑

6.1 什么是上下文管理器

一个类中只要实现了__enter__()和__exit__()这两个魔法方法，通过该类创建的对象就是一个上下文管理器对象。
__enter__()：表示的就是上文方法，需要返回一个操作对象

```
__exit__():表示的就是下文方法，with语句会自动执行下文方法，及时代码出现异常，也会执行下文方法


"""
需求: 定义一个上下文管理类，模拟文件操作
定义一个File类
实现__enter__()和__exit__()这两个魔法方法
然后用with语句调用这个类来完成文件操作，观察现象
"""

class File(object):
    def __init__(self, file_name, file_model):
        self.file_name = file_name
        self.file_model = file_model

    def __enter__(self):
        """上文方法"""
        print('这是上文方法')
        self.file = open(self.file_name,self.file_model)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        """下文方法"""
        print('这是下文方法')
        self.file.close()

with File('1.txt', 'r') as f:
    data = f.read()
    1/0
    print(data)
```

python基础

前端 打基础

python爬虫

框架

爬虫进阶

数据可视化