3.3 异常

::: details INFO
译者：Bryan Zhang

来源：3.3 Exceptions

对应：无
:::

程序员必须时刻注意程序中可能出现的错误。举例来说：一个函数没有收到它所预期的参数，必要的信息可能缺失，或者网络连接可能中断。在设计程序时，必须预见可能发生的异常，并采取措施。

处理程序中的错误没有唯一正确的方法。比如，对于用于提供持续服务的程序（如网络服务器），他们需要具备鲁棒性，将错误日志记录下来以供作考虑，同时尽可能继续为新的请求提供服务。另一方面，Python 解释器处理错误时会立即终止程序并打印错误消息，这样程序员可以及时解决问题。无论哪种情况，程序员都必须在如何处理异常时做出明智的选择。

异常（Exceptions）是本节的主题。在程序当中，异常通过添加错误处理的逻辑提供了一种通用的机制。抛出异常（raising an exception）是一种中断程序正常执行流程的技术，它表示发生一些异常情况，并直接返回到程序中预定处理该情况的部分。Python 解释器在检测到表达式或语句中出现错误时会抛出异常。用户也可以使用 `raise` 和 `assert` 语句来抛出异常。

抛出异常：异常是一个对象实例，其类（class）直接或间接继承自 `BaseException` 类。在第一章中引入的 `assert` 语句会抛出一个类为 `AssertionError` 的异常。通常情况下，可以使用 `raise` 语句来抛出任何异常实例。Python 文档 中描述了 `raise` 语句的一般形式。最常见的用法是构造一个异常实例并将其抛出。

```
>>> raise Exception(' An error occurred')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
Exception: an error occurred
```

当抛出异常时，当前代码块中的后续语句将不会执行。除非异常被处理（如下所述），否则解释器将直接返回到交互式的读取 - 求值 - 打印（read-eval-print-loop）循环，或者在 Python 是通过文件参数启动时会完全终止。此外，解释器将打印一个堆栈回溯（stack backtrace），它是一个结构化的文本块，描述了在异常被抛出的执行分支中活动的嵌套函数调用集合。在上述示例中，文件名 `<stdin>` 表示该异常是由用户在交互会话中引发的，而不是来自文件中的代码。

处理异常（handling exceptions）。异常可以由封闭的 `try` 语句来处理。`try` 语句由多个子句组成；第一个以 `try` 开头，其余的以 `except` 开头：

```
try
    <try suite>
except <exception class> as <name>:
    <except suite>
```

在执行 try 语句时，`<try suite>` 总是立即执行。只有在执行 `<try suite>` 过程中发生异常时，except 子句的内容才会执行。每个 except 子句指定了要处理的特定异常类。例如，如果 `<exception class>` 是 `AssertionError`，那么在执行 `<try suite>` 过程中引发的任何继承自 `AssertionError` 类的实例都将由随后的 `<except suite>` 处理。在 `<except suite>` 内部，标识符 `<name>` 绑定到被引发的异常对象，但此绑定不会在 `<except suite>` 之外存在。

例如，我们可以使用 `try` 语句处理 `ZeroDivisionError` 异常，当异常被引发时，将名称 `x` 绑定到 0。

```
>>> try:
        x = 1 / 0:
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
handling a <class 'ZeroDivisionError'>
>>> x
0
```

`try` 语句将处理 `<try suite>` 中发生的异常（包括 应用在 `<try suit>` 的函数，无论是直接还是间接应用）。当引发异常时，控制权会直接跳转到处理该类型异常的最近一次 `try` 语句的 `<except suite>` 中。

```
>>> def invert(x):
        result = 1/x # 抛出一个异常（ZeroDivisionError) 如果 x 为 0
        print('Never printed if x is 0')
        return result
>>> def invert_safe(x):
        try:
            return invert(x)
        except ZeroDivisionError as e:
            return str(e)

>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
'division by zero'
```

这个例子说明了在 `invert` 中的 `print` 表达式永远不会被评估，而是转移到 `invert_safe` 的 `except` 子句的内容中。将 `ZeroDivisionError` `e` 强制转换为字符串会得到可解释字符串：“division by zero”。

3.3.1 异常对象（Exception Object）

异常对象本身可以具有属性，例如在 `assert` 语句中的错误消息，以及关于异常在执行过程中被引发的位置的信息。用户自定义的异常类可以具有额外的属性（attributes）。

在第一章中，我们实现了牛顿法来寻找任意函数的零点。下面的示例定义了一个异常类，每当出现数值错误（ValueError）时，它返回在迭代改进过程中发现的最佳猜测。当将 `sqrt` 函数应用于负数时，会引发数学域错误（ValueError 的一种类型）。通过引发一个 `IterImproveError` 来处理此异常，并将牛顿法中最近的猜测存储为属性。

首先，我们定义一个新的类，它继承自 `Exception`。

```
>>> class IterImproveError(Exception):
        def __init__(self, last_guess):
            self.last_guess = last_guess
```

接下来，我们定义 `improve` 方法，这是我们通用的迭代改进算法。这个版本通过引发一个 `IterImproveError` 来处理任何数值异常（ValueError），并将最近的猜测存储起来。与之前一样，`improve` 接受两个函数作为参数，每个函数都接受一个数值参数。`update` 函数返回新的猜测，而 `done` 函数返回一个布尔值，表示更新的值是否正确。

```
>>> def improve(update, done, guess=1, max_updates=1000):
        k = 0
        try:
            while not done(guess) and k < max_updates:
                guess = update(guess)
                k = k + 1
            return guess
        except ValueError:
            raise IterImproveError(guess)
```

最后，我们定义 `find_zero` 函数，它用于返回 `improve` 函数的结果（`update` 函数为 `newton_update`）。牛顿更新函数（newton_update）在第一章中定义，并且对于这个例子不需要进行任何更改。这个版本的 `find_zero1` 通过返回最后的猜测来处理 `IterImproveError` 异常。

```
>>> def find_zero(f, guess=1):
        def done(x):
            return f(x) == 0
        try:
            return improve(newton_update(f), done, guess)
        except IterImproveError as e:
            return e.last_guess
```

考虑通过 `find_zero` 函数来寻找 $2x^{2}+\sqrt{ x }$ 的零点。该函数在 0 处有一个零点，但在任何负数上评估它都会引发 `ValueError` 异常。我们第一章中实现的牛顿法将引发该错误，并且无法返回任何零点的猜测。而我们修订后的实现会在错误发生前返回最后的猜测。

```
>>> from math import sqrt
>>> find_zero(lambda x: 2*x*x + sqrt(x))
-0.030211203830201594
```

尽管这个近似值仍然远离正确的答案 0，但某些应用程序更倾向于粗略的近似值，而不是数值异常（ValueError）。

异常是另一种帮助我们将程序的关注点模块化的技术。在这个例子中，Python 的异常机制允许我们将迭代改进的逻辑（在 try 子句中内容中保持不变）与处理错误的逻辑（在 except 子句中出现）分开。我们还会发现，在 Python 中实现解释器时，异常也是一个有用的功能。