第5章

调试与测试

调试是纠正错误的过程, 测试是设法检查程序中是否存在错误的过程。程序调试和测试是保证程序质量的重要方法。

5.1 调试

人们在编写程序过程中会犯各种错误，包括语法错误、语义错误和运行时错误。查找错误并改正错误的过程称为**调试** (debug)[①]。调试无论对毫无经验的初学者还是有一定经验的程序员都是非常费时的过程。

5.1.1 语法错误

语法错误 (syntax error)，顾名思义，就是不符合程序设计语言语法规则的错误，例如 Python 语言的缩进问题、漏掉冒号、打字错误、括号不匹配等。这些错误通常可由编译器或者解释器发现并报告，因此也比较容易发现、定位并修正。

对于解释器报告的错误位置，如果该位置似乎不存在错误，需要仔细检查前面的代码行是否有问题。例如，运行下面两行代码：

```
n = int(input('请输入一个整数 n:')
sum = 0
```

解释器报告错误“SyntaxError: invalid syntax”，并指向第二行赋值语句，但是第二行赋值语句显然没有问题。仔细查看发现，第一行代码中括号不匹配。这里错误位置指向第二行的原因是，解释器认为第一行命令未完，可能在第二行继续，但是在第二行没有发现预期的括号。

5.1.2 语义错误

语义错误（semantic error，也称逻辑错误）是指程序不能给出正确结果，或者不能按照预期工作。这类错误并不会让编译器或者解释器返回一个错误信息，只有在程序运行期间或者运行之后才会发现。

例如，从键盘输入两个数值，然后打印它们的平均值：

① 英语文献中通常用 bug（虫子）表示程序中的错误，debug 意为去除虫子。

```
a = eval(input())
b = eval(input())
print('Average of ', a, 'and', b, ':', a+b/2)
```

输入 1 和 2，结果显示 1 和 2 的平均值为 2.0。错误在于没有正确处理加法和除法的优先级问题。

具有语义错误的程序可能在运行时崩溃，返回一个错误信息；也可能在运行时不会崩溃，但是返回的结果不正确，或者其表现不是程序的预期行为。这类错误也是最难定位和修正的错误。

另外一类错误称为**运行时错误** (runtime error)，因为这类错误在程序运行时发生，例如列表索引越界（IndexError）、类型错误（TypeError）或者程序打开文件时出错（如 IOError）。

例如，下面的程序试图打印一个二元组的两个分量：

```
p = (1, 2)
print(p[1],p[2])
```

运行时显示“IndexError: tuple index out of range”，因为正确的存取方法是 p[0] 和 p[1]。

再如，以下程序要求用户输入一个整数 n，然后对后续输入的 n 个数求和，并打印结果：

```
n = int(input('请输入一个整数n:'))
sum = 0
for i in range(n):
    x = input()
    sum += x
print(sum)
```

在 IDLE 下执行 Check Module 命令没有发现语法错误；但是执行 Run Module 命令，程序在运行过程中发生错误：

```
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

以上信息报告第 5 行代码中的赋值运算不支持 int 类型和 str 类型相加。改正的方法是先将 x 转换为数值类型再相加。

5.1.3　调试基本技术


调试基本技术

调试程序的第一步是找到错误，然后分析错误原因，并改正错误。

定位错误的基本方法是跟踪程序变量的状态或值，基本技术是在程序中适当位置插入 print 语句，查看变量的值是否等于预期值。

打印变量值语句可以选择以下插入位置：

- 程序的中部，根据变量的值确定错误在插入位置之前或者之后。

- 循环开始前和结束后，确定错误位置在循环内还是循环外。
- 循环内部，进一步确定循环体内的错误位置。

例5.1 下面的代码中函数isPal检查字符列表是否构成回文，即从前往后和从后往前是同一个列表。例如，['a','b','a']和['a','a'] 都构成回文，但是['a','b','b']和['b','a']不是回文。[①]

```
def isPal(x):
    assert type(x) == list
    temp = x
    temp.reverse
    if temp == x:
        return True
    else:
        return False

def silly():
    n = int(input('enter length n:'))
    for i in range(n):
        result = []
        elem = input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

函数silly()提示用户输入列表长度和列表的各个元素，构造一个列表，然后调用函数isPal()判断该列表是不是回文，最后打印函数 isPal()的判断结果。

以下通过调用函数silly()检查程序中是否存在问题。分别输入长度为 1、2、3 等的字符列表，查看判断结果是否正确。

注意，运行本例代码需要将源程序文件用UTF-8编码存储，并在源文件顶部说明编码使用UTF-8，即添加#coding=utf-8。

第一次尝试

输入长度为 1 的列表，结果输出 Yes，正确。输入一个长度为 2 的列表，如['a','b']，打印结果也是 Yes，结果显然是错误的。为此，在 silly()中插入 print 语句，查看调用 isPal()的输入参数 result 是否正确。

```
def silly():
    n = int(input('enter length n:'))
    for i in range(n):
```

① 本例来自 edx.org 课程 MITx:6.00.1x“Introduction to Computer Science and Programming Using Python”。

```
        result = []
        elem = input('Enter element: ')
        result.append(elem)
    print('result = ',result)  # 查看构造列表result是否正确
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

第二次尝试

再次调用silly()，长度为 2，依次输入元素 a 和 b，结果发现输出为

```
result = ['b']
```

而不是期望的result = ['a', 'b']。说明 silly()中 for 循环结束后并没有正确地构造输入列表。为此，可以继续查看 for 循环代码，或者在 for 循环体中 append 之后查看 result 结果是否正确。

```
def silly():
    n = int(input('Enter length n:'))
    for i in range(n):
        result = []
        elem = input('Enter element: ')
        result.append(elem)
        print('第', i, '次append之后 result = ', result)

    print('result =', result)  # 查看构造列表result是否正确
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

第三次尝试

在循环体内添加打印语句后再次调用silly()，结果发现，每次 append 之后列表总是只有一个元素：

```
第 0 次append之后 result =  ['a']
Enter element: b
第 1 次append之后 result =  ['b']
```

仔细检查 for 循环发现，列表 result 每次循环都要初始化，但是，result 本应在 for 循环前只初始化一次。改正的方法是将初始化语句移到 for 循环之前。修改后的 silly()函数如下：

```
def silly():
    n = int(input('enter length n:'))
    result = []
    for i in range(n):
        elem = input('Enter element: ')
        result.append(elem)
        # print('第', i, '次append之后 result = ', result)

    # print('result =', result)  # 查看构造列表result是否正确
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

第四次尝试

再次调用函数silly()，发现构造的输入列表result正确。

现在可以确定函数silly()中调用isPal(result)的输入正确，因此断定错误在函数isPal()中。

因为函数isPal()中通过检查输入 x 和它的逆 temp 来判断 x 是否回文，所以，可以在判断前插入print 语句，检查 x 和 temp 的值。

```
def isPal(x):
    assert type(x) == list
    temp = x
    temp.reverse
    print('x = ', x)
    print('temp = ', temp)
    if temp == x:
        return True
    else:
        return False
```

第五次尝试

调用silly()，输入长度仍然为 2，输入元素 a 和 b，结果发现输出为

```
x =  ['a', 'b']
temp =  ['a', 'b']
```

可见 temp 并不是我们期望的 x 的逆。仔细查看代码发现,在 temp 上调用方法 reverse 的语句有错误，漏掉了括号，即应该使用temp.reverse()。

第六次尝试

纠正错误后重新运行silly()，输出以下结果：

```
x =  ['b', 'a']
temp =  ['b', 'a']
```

此时发现 temp 确实是原来输入的['a','b']的逆，但是 x 也变成了它原来的逆，这不是我们期望的。我们期望 x 不变，temp 是 x 的逆。仔细查看代码发现，赋值语句temp = x使得 x 和 temp 指向同一个对象['a', 'b']，结果在下一个语句temp.reverse()之后，x 和 temp 仍然指向将输入变成逆的同一个对象['b', 'a']。两个不同的变量指向同一个对象的**别名问题** (alias) 容易导致这类难以发现的错误。解决该问题的方法是，让 temp 取 x 的副本，即将赋值语句temp = x 改为temp = x[:]。修改后的函数如下：

```
def isPal(x):
    assert type(x) == list
    temp = x[:]
    temp.reverse()
    print('x = ', x)
    print('temp = ', temp)
    if temp == x:
        return True
    else:
        return False
```

第七次尝试

调用silly()，输入不同的列表，发现结果正确。

5.2　测试


测试的概念和方法

如果一个程序是正确的，那么它对于任意合法输入都能给出正确的输出。例如，一个对于整数列表从小到大排序的程序，其正确性含义是：对于任意整数序列 xs，程序都应该给出 xs 从小到大的重新排列。编写程序是非常容易犯错误的过程。为了降低排序程序包含错误的可能性，应该设计大量的整数列表输入作为测试用例，并观察每个输出的结果是否正确。这种设法发现软件中错误的过程称为**软件测试** (software testing)，用于执行测试的输入称为**测试用例** (test case)。

软件测试按照是否运行程序可以分为静态测试和动态测试。静态测试主要通过阅读代码发现错误，动态测试则需要在计算机上执行代码。动态测试按照与程序结构的相关性分为**白盒测试** (white box testing) 和**黑盒测试** (black box testing)。白盒测试要根据程序结构设计测试用例，并让测试达到一定覆盖率，例如要求测试的运行能够覆盖程序中的所有语句，即所有语句或者分支都得到至少一次运行，或者所有可能的执行路径都得到执行。黑盒测试则根据软件功能或者规格说明设计测试用例，并要求测试用例覆盖输入的各种情况。软件测试按照是否需要人工干预分为人工测试和自动测试。

下面以 5.1.3节中的函数 isPal() 为例简单介绍黑盒测试。

5.2.1 程序的规格说明

一个程序的**规格说明** (specification) 是程序功能的具体描述，包括程序的合法输入以及对应的正确输出。例如，每个函数的文档串可被看作一个函数的简化规格说明。程序的规格说明是测试程序的重要依据。以函数 isPal() 为例，它的合法输入包括哪些列表？合法输入只包括字符的列表，如 ['a','b','a']，还是包括字符串的列表，如 ['ab','ba']？如果是字符列表，那么字符范围是什么？是否同时包括大小写？是否包括数字字符，如'1'、'2'？如果同时包括大小写，那么输入列表 ['a','A'] 的输出是'Yes'还是'No'？所有这些都应该在规格说明中给出明确说明，这样函数的实现、使用以及测试才有依据。

这里假定 isPal() 的合法输入是 26 个英文小写字母构成的列表，例如 ['a', 'b','a']、['a'] 是合法输入，而 ['A','2'] 则不是合法输入。如果一个列表和它的逆完全相同，则称为回文；否则不是回文。依据这样的规格说明便可以给出下面的实现：

```
def isPal(x):
    """x是26个英文小写字母构成的列表，如[]、['a']、['a','b','a']
    和['a','z']。如果x和它的逆完全相同，则返回True；否则返回False。
    例如，对于以上前3个输入返回True，对第4个输入返回False
    """
    temp = x[:]
    temp.reverse()
    if temp == x:
        return True
    else:
        return False
```

5.2.2 人工测试

这里只考虑对于合法输入，函数是否给出正确的期望结果。因为 isPal() 的合法输入是无穷的，但是只能测试有限个输入作为测试用例。如何选择测试用例是测试的关键问题。一般来说，选择的测试用例应该尽可能多，尽可能覆盖各种可能类型的输入。例如，如果把输入分成奇数长度和偶数长度，那么输入既要有奇数长度的输入列表，也要有偶数长度的输入列表。对于同一长度的输入，既要有回文的列表，也要有不是回文的列表。测试输入还要特别覆盖边界或者极端的情况。例如，长度为 0 是极端的情况；长度为 1 也比较极端，因为长度为 1 的列表都是回文。再如，同一个字符构成的列表，如 ['a']、['a','a']、['a','a','a'] 等，也是极端的情况。

如果在测试中发现错误，那么改正错误后不仅要测试原先发生错误的测试用例现在是否给出正确输出，而且需要对所有的测试用例重新进行测试，称为**回归测试** (regression test)，这是因为在改正错误的过程中可能引入新的错误。

另一种选择测试输入的方法是随机法，即让计算机生成随机输入，然后人工检查输

出是否正确。例如，函数 gen_str(n) 生成长度为 n 的随机字符列表，函数 auto_test(n) 生成 n 个随机输入并打印 isPal() 的判断结果：

```
def gen_str(n):
    """生成长度为n的小写字母列表"""
    cs = string.ascii_lowercase
    result = []
    for i in range(n):
        result.append(random.choice(cs))
    return result

def test(n):
    """随机生成n个字符列表，并查看判断结果"""
    for i in range(n):
        k = random.randint(1,10)
        s = gen_str(k)
        b = isPal(s)
        print(s, b)

n = int(input())
test(n)
```

5.2.3　自动测试

因为人工输入测试用例并检查输出是否正确效率很低，所以，如果能让计算机生成测试用例，并判断结果是否正确，则可大大提高测试效率。对于本例，一种可能是编写生成给定长度的回文列表，然后调用函数 isPal()，并检查结果是否为 True，如果不是，则报告错误，并停止测试。当然，这个生成的回文要确保是正确的回文，而且具有代表性，例如是随机生成的回文列表。例如，下面的函数 gen_panlindrome(n) 生成长度为 n 的随机回文列表：

```
def gen_panlindrome(n):
    """生成长度为n的回文"""
    k = n//2
    result1 = gen_str(k)
    result = result1[:]
    cs = string.ascii_lowercase
    if n%2 == 1:
        result.append(random.choice(cs))
    for i in range(k):
        result.append(result1[k-i-1])
    return result
```

这里使用了模块 string 提供的常量 ascii_lowercase：

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

另外使用了 random 模块的 choice() 函数，choice(cs) 返回参数 cs 中随机选择的元素：

```
>>> random.choice(cs)
'q'
>>> random.choice(cs)
's'
```

在此基础上，可以编写自动测试函数 auto_test()：

```
def auto_test(n):
    """随机生成n个回文列表，并判断结果是否正确。
       如果isPal()返回False,则报告错误，终止测试
    """
    for i in range(n):
        k = random.randint(1,10)
        s = gen_panlindrome(k)
        b = isPal(s)
        if not b:
            print(s, b)
            break
    else:
        print('Done')

auto_test(n)
```

当然，另一方面还应该测试输入不是回文列表时 isPal() 判断结果是否正确。

习题

5.1 设计一个函数，检查一个字符串是否回文：

```
def ispanlindrome(s):
    """判断字符串s是否回文。
       如果s是回文，则返回True，否则返回False
    """
```

注意，函数用 return 返回 True 或者 False，不需要打印信息。另外，本章例子中 isPal() 函数输入实际上是字符列表，不是字符串。

5.2 设计一个能够随机生成给定长度字符串的函数：

```
def genstring(n):
    """返回长度为n的随机生成的字符串"""
```

注意，可以只考虑小写的字符串，只含 26 个英文字母。要用 return 返回一个指定长度的随机字符串。另外，可以考虑使用常量 string.ascii_lowercase，每次随机取其中一个字符构造一个给定长度字符串。

5.3　实现一个总是随机生成指定长度字符串回文的函数：

```
def genpanlindrome(n):
    """随机生成长度为n的字符串回文"""
```

5.4　尝试使用下列框架测试习题 5.1 中的函数 ispanlindrome()：

```
def test():
    n = int(input('type some int:'))
    #根据n的值生成随机的回文或者随机字符串
    if random.randint(0,1) ==0:
        s = genpanlindrome(n)
    else:
        s = genstring(n)
    b = ispanlindrome(s)
    if b:
        print(s, 'is a panlindrome')
    else:
        print(s, 'is not a panlindrome')
```

5.5　设计一个函数，可以进行任意给定多次的测试。例如：

```
请求用户输入一个测试次数N
for i<- to N:
    生成一个随机长度n
    生成一个长度为n的随机回文s
    如果s是回文，打印s和ispanlindrome(s)的结果
    如果s不是回文，则打印s和ispanlindrome(s)的结果，并停止测试
```

5.6　设计一个类似于习题 5.4 的函数，生成任意多个随机字符串，检查 ispanlindrome() 的判断结果。