测试和调试

- 字符串和字符串操作
  - 字符串的格式化（生成）
- 程序测试（testing）
  - 试验性地运行程序，设法发现程序里的错误
- 调试（debugging）
  - 发现错误后找出原因（而后设法纠正）
- IDLE 的调试支持

字符串（复习）

- 字符串是一种基本数据类型
  - 类型名：**str**
  - 字面量形式：一对单引号或一对双引号括起的字符序列，其中可以包括空格和一些特殊字符

    还可以用连续三个单引号或三个双引号作为“括号”
  - 语义：得到相应的字符串对象
- 几个问题
  - 单个单引号或双引号括起的字符串中间不能换行
  - 连续写出几个字符串自动拼成一个长字符串
  - 三引号形式的字符串中间可以换行，换行符，换行后的空格等都看作字符串的内容

字符串

- 特殊字符（常用的，其他见参考材料）

  \n 表示换行字符　　**** 表示字符 ****

  \' 表示单引号　　**\"** 表示双引号

  其他见语言手册 **2.4.1, String and Bytes literals** 一节

- 两个主要的字符串构造操作（**s, t** 表示字符串，**n** 表示整数）

  s + t　拼接两个字符串，得到拼接串

  结果字符串的前一段是 **s** 的拷贝，后一段是 **t** 的拷贝

  "abc" + "123" 得到 **"abc123"**

  s * n 或 **n * s**　做出 **s** 的 **n** 个拷贝拼接而成的串

  两种写法等价。如，**"ab" * 2** 和 **2 * "ab"** 都得到 **"abab"**

字符串操作

- 字符串的“长度”就是其中的字符个数

  len(s) 得到字符串 **s** 的长度

  长度等于 **0** 的串称为空串

- 字符串里每个字符有一个位置，位置称为“下标”，从 **0** 开始计数，直至 **len(s) - 1**

- 下标表达式 **s[k]** 确定 **s** 中下标为 **k** 的字符

  s[0]　给出 **s** 的首字符

  s[-1]　给出 **s** 的末字符，类似地可写 **s[-2]** 等

  s[n]　给出 **s** 第 **n** 个字符

  取字符，实际得到的是指定位置的字符形成的单字符串

  如果下标超出这个字符串的范围，系统报错

在字符串上循环

- 可以通过下标在字符串上循环

```
i = 0
while i < len(s) :
    print(s[i])
    i += 1
```

```
for i in range(len(s)):
    print(s[i])
```

- Python 把字符串看作一种序列（sequence），序列对象的特点是包含有序的一系列元素，常需要逐一操作其中的元素。字符串的元素就是其中的字符
- 为了方便对序列中各元素的操作，Python 允许直接把序列对象作为迭代器，用在 for 语句里和其他需要迭代器的地方

  用 for 描述输出字符串里字符，很方便（其他操作类似）

```
for c in s:
    print(c)
```

字符串切片（slice）

- 对于序列对象，可以做切片。一个对象的切片就是用从该对象里选出的一些元素做成的另一个同类型对象
- 下面用字符串介绍 **Python** 的切片描述方法（其他序列对象的切片操作也都是这样描述）：

  s[m : n] 得到由 **s** 里下标为 **m, m+1, ..., n-1** 的字符构成的字符串（**[ ]** 里的表达式描述切片）

  s[: n] 得到由 **s** 里下标为 **0, 1, ..., n-1** 的字符构成的串

  s[m :] 是 **s** 下标为 **m, m+1, ..., len(s)-1** 的字符构成的串

  s[:] 得到 **s** 的一个拷贝

  s[m:n:d] 得到 **s** 中下标为 **m, m+1, ..., n-1** 位置的字符中间隔为 **d** 的字符构成的串。这里的 **m, n** 也可以省略，表示从头开始或到结束，但 **:** 不能省略。例如 **s[::2]**

字符串是“不变”对象

- 建立了一个字符串之后，不能修改其内容
  - 前面所有操作都是建立新字符串（不是修改已有字符串）
  - 对于基本类型的对象做运算，都是创建新对象
- 一些例子：

```
y = 2
x = y + 1
x += 3
ss = s = "Thank"
s += " you! "
s *= 2
print(ss)
print(s)
```

字符串处理函数

- 定义函数，判断某个字符是否在一个字符串里出现
  - 显然需要两个参数，检索用字符和被检索字符串
  - 需要一个个比较，用 **for** 循环最简单
  - 返回 **True** 或 **False** 表示出现或不出现
- 统计一个字符在一段正文里出现的次数
  - 正文也用字符串表示
  - 用 **for** 循环比较一个个字符
  - 用一个计数变量，初始值为 **0**，发现要找的字符就加一
  - 返回计数值
- 实际上，**str** 类型已经提供了这些操作

字符与编码

- 每个字符有一个编码
- 内置函数 **ord(c)** 给出字符 **c**的编码（**c** 必须是只包含一个字符的字符串）
  - 例如 **ord("a")** 得到 **97**
  - 不同字符的编码不同，字符也根据编码排大小（排序），称为字符序
- 从编码得到字符
  - 内置函数 **chr(n)** 给出编码为整数 **n** 的字符
  - 字符与编码一一对应

字符串包含和比较

- in 和 not in 运算符判断一个字符串是否为另一字符串的子串
  - 字符串 s 的子串是 s 中一段连续字符形成的字符串
  - t in s 和 t not in s 得到真假值，可用作逻辑条件
- 字符串可比较大小。字符串上定义了一种序，称为字典序，基于字符集合上的序定义（基于字符的 ord(c)）。s < t 的条件是，顺序比较两个串中各对应位置的字符
  - 如果发现第一对不同字符在 i 且 ord(s[i]) < ord(t[i])，或者
  - 两个串在能比较范围内的字符都相同但 s 较短
- 两个串长度相同，对应各字符分别相同，则这两个串相等
- 比较运算符（==, !=, <, <=, >, >=）均可用于字符串
  - 两个字符串之间比较，<, ==, > 三者之一成

字符串操作

- **Python 为字符串类型 str 的对象定义了很多常用函数**

  这些函数称为“方法”，要通过点号记法使用

  点号记法就是在字符串（或者值为字符串的变量）后写一个圆点符号，而后写函数名及实际参数

  字符串是不变对象，一些字符串方法生成新字符串

- 做出当前串的大写或小写拷贝

  s.lower() 做出 s 的小写拷贝

  s.upper() 做出 s 的大写拷贝

  s.capitalize() 做出 s 的首字符大写其余小写的拷贝

  s.swapcase() 做出 s 的大小写对换的拷贝

字符串操作

- 完成判断的方法（满足条件时返回 **True**）
  - **s.isupper()**　　**s** 不空其其中存在大小写的字符都是大写
  - **s.islower()**　　**s** 不空其其中存在大小写的字符都是小写
  - **s.isdigit()**　　**s** 不空且其中所有字符都是数字
  - **s.isalpha()**　　**s** 不空且其中所有字符都是字母
  - **...**
- **s.find(sub) sub** 是一个串

  查找 **sub** 在 **s** 里第一次出现的位置，没出现时返回 **-1**

  s.find(sub, start, end) 在 **s** 的指定范围里找

  例：在一个文本里找到一个串的所有出现位置

字符串操作

- **s.count(sub)** 统计 **sub** 在 **s** 里互不重叠的出现的次数

  s.count(sub, start, end) 在指定范围内统计出现次数

- **s.replace(old, new)**

  建立字符串 **s** 的一个拷贝，在其中把 **s** 中子串 **old** 的所有出现都替换成另一个串 **new**

  s.replace(old, new, count) 只做前 **count** 个替换

- **s.strip()**，**s.lstrip()**，**s.rstrip()** 得到 **s** 删去两端空白（或删去左边空白，或删去右边空白）后的串

- 注意，**replace**、**strip**、**lstrip**、**rstrip** 都是基于给定字符串 **s** 建立新字符串

- 还有许多字符串方法，详情见标准库手册 **4.7** 节

输出格式控制

- 前面输出都采用 **print**（屏幕输出）产生的“自然形式”，如果需要也可以控制输出形式，这种工作称为输出格式控制
- 生成文本形式输出，第一步是把程序内部的对象字符串化
  - 用 **str(...)** 或 **repr(...)** 都可以由程序对象生成相应的字符串
  - **str** 的工作意图是生成对象的易读文本表示形式，**str** 对任何对象都能生成一个字符串表示
  - **repr**（出自 **representation**）的意图是生成的文本表示还可以重新输入，由 **Python** 解释器解释。如果 **x** 没有可以重新输入的字符串表示，**repr(x)** 将报 **SyntaxError** 异常
  - **print** 对参数自动调用 **str(...)**，再加上分隔符和结束符
- 所谓格式化输出，就是做字符串的形式处理。可以通过字符串提供的一些操作完成，**Python** 还提供了专门的处理功能

简单格式控制

- **str** 本身的格式功能包括在一个具体的宽度内对齐，生成一个指定宽度（长度）的字符串，把原串放在其中左边、中间或右边，其余部分用某个字符（默认为空格）填充。设 **s** 是字符串：

  s.center(n)　得到将 **s** 串居中的长度为 **n** 的字符串

  s.ljust(n)　得到将 **s** 串居左的长度为 **n** 的字符串

  s.rjust(n)　得到将 **s** 串居右的长度为 **n** 的字符串

  可指定填充字符（默认为空格），如 **s.rjust(6, '0')** 用 **'0'** 填充

- 例：

```
for i in range(0, 150, 13) :
    print(str(i).rjust(5), str(i**2).rjust(7), str(i**3).rjust(7))
```

- 注意，这些方法的参数是表达式，自然可以通过变量控制

格式转换

- **Python** 有两套专门用于格式化的机制。下面介绍新推荐的机制
- 新格式化功能通过 **str** 的 **.format** 函数（方法）完成

  使用形式是 **s.format(*args, **kwargs)**

  s 是描述格式化的字符串

  ***args** 表示 **format** 方法可以有任意多个实参（表达式）

  ***kwargs** 表示可以有任意多个关键字实参（后面有例子）

  s 中可有任意多个用 **{...}** 表示的位置（称为替换域），生成结果串时分别用各实参的值经格式化后产生的字符串替代
- 例：**"The {} of 2 + 5 is {}".format("sum", 1+2)**

  生成的串是（生成输出时不处理数学上的正确性）：

  The sum of 2 + 5 is 3

格式转换

- 调用 **s.format(...)** 生成一个字符串：
  - 格式化串 **s** 里除替换域外的字符顺序拷贝到结果串，**s** 里的替换域用 **.format** 的实参值生成的字符串替换

    格式串 **s** 描述结果的框架， **format** 的实参充填片段
  - 格式串就是字符串，但其中有些字符在作为格式串时有特殊作用。可以把格式串赋给变量，而后通过变量使用
  - 格式串里实际的花括号字符用双写 **{{ , }}** 表示
- 理解格式化，需要弄清两件事：
  1. 如何用替换域描述对实际参数的格式化要求
  2. 实参与替换描述的匹配和代入关系

  详情见标准库手册 **6.1.3** 节，有许多繁琐细节。下面只介绍其中的道理和常用实例

格式转换

- 最简单的默认情况：按位置匹配，默认生成形式。如：

  "A {} is {} but {}.".format(arg_0, arg_1, arg_2)

- 可以在替换域里用整数（按一般下标规则）指明实参。如：

  "A {2} is {0} but {1}.".format(arg_0, arg_1, arg_2)

  要求用指定位置的实参替换，可以重复用同一个实参

- 可以在替换域里用名字指明关键字实参，如：

  "The {noun} is {adj} but also {adj2}.".format(
  noun="pig", adj2="smart", adj="fat")

  产生： **The pig is fat but also smart.**

- 在整数或关键码表示的替换域名后可有一个“:”后随一个转换描述，描述实参的特殊转换方式（不采用非默认方式）

格式转换

- 几个常用描述项，均可省略，如出现时应按下面的顺序
- 描述对齐方式的字符 **<, >,** 或 **^**，分别表示该域内容采用居左, 居右或居中方式，可以在对齐字符前给一个填充字符

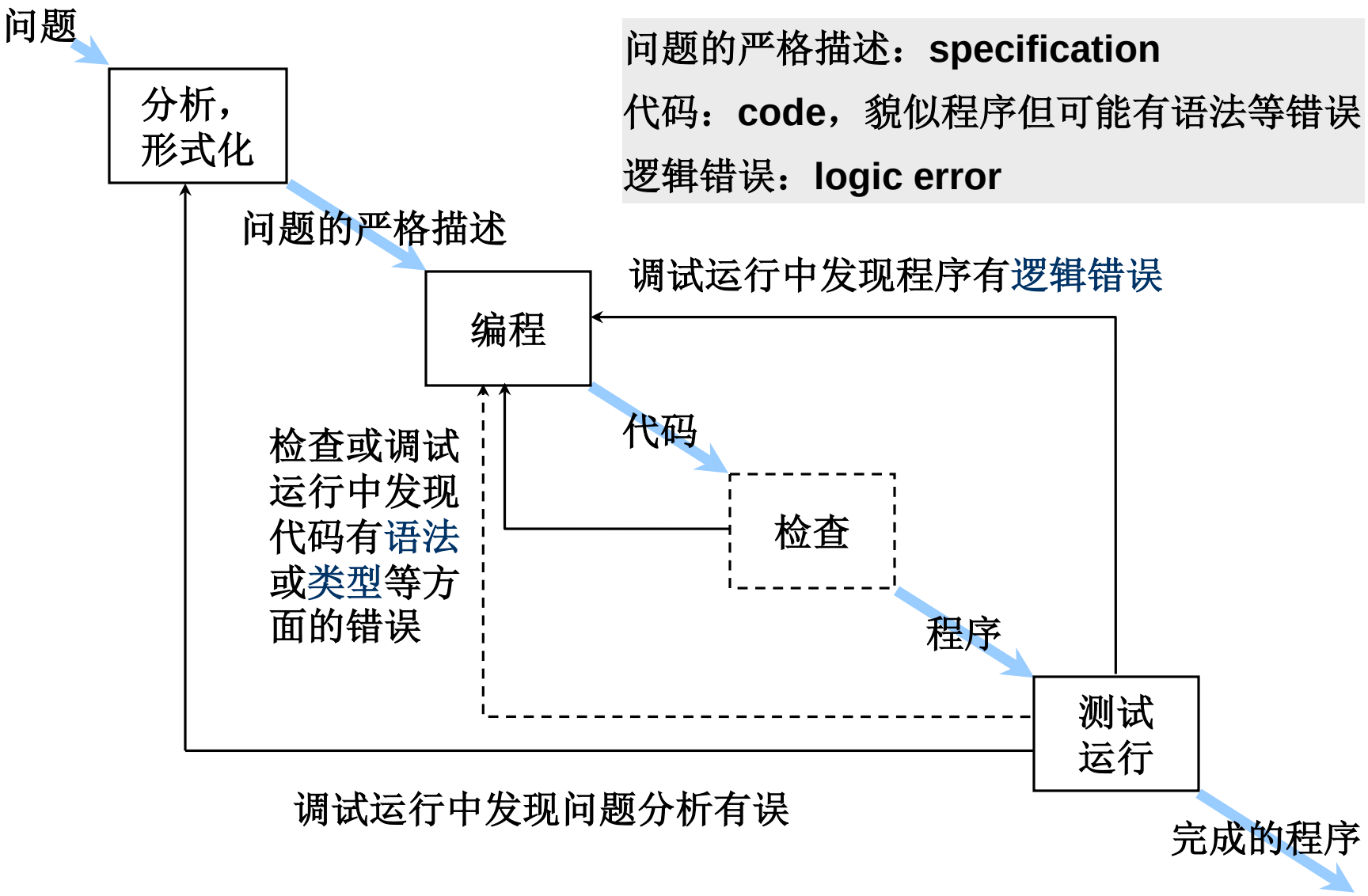
  无对齐描述时，字符串采用居左对齐，数值采用居右对齐
- 一个表示域的最小宽度的整数，实际数据内容需要输出更多字符时可以输出得更宽。默认的输出宽度由实际数据内容确定
- 对浮点数转换 **f** 和 **F**，可以有圆点和一个整数表示浮点数输出中小数部分的位数（精度）。默认输出精度为 **6** 位
- 转换类型用一个字符表示：**s** 表示字符串，**d** 表示整数用十进制方式输出，**f** 和 **F** 表示用浮点数形式输出，**e** 和 **E** 表示用科学记数法输出，**g** 和 **G** 根据情况采用浮点形式或科学形式

  默认是根据实际数据类型输出，整数可用浮点形式输出，但必须写出具体的输出形式 **f/e/g** 等

格式转换

- 下面是一些转换描述的实例：
  - **{1:->10s}**　字符串形式，第1个实参宽10，右对齐，填充 –
  - **{price:10.2f}**　浮点形式，名字 price，宽10，小数点后2位
  - **{:<<10d}**　　十进制整数形式，宽10，居左，填充 <
- 注意：
  - 转换类型为 **s** 时，参数必须是字符串
  - 转换类型为 **d** 时，参数必须是整数
  - 转换类型是 **f/F/e/E/g/G** 时，参数可以是整数或浮点数
  - **d** 等整数转换类型不允许出现精度描述（圆点加精度）
  - 非数值类型的转换中精度描述规定输出域的最大宽度

从问题到程序



问题的严格描述：**specification**

代码：**code**，貌似程序但可能有语法等错误

逻辑错误：**logic error**

程序开发

- 通过了语法和类型检查的程序，可以实际运行，下一步工作是设法确认它是否确实满足需求（requirements）
  - 功能正确？
  - （函数）对正确的参数都能得到正确的结果？
  - 对合适的输入都能产生正确的输出？
- 如何确认程序满足需求（“正确”）？
  - 常规做法是通过一系列试验运行，检查程序的行为或输出

    如果发现问题，就设法纠正或改进
  - 这是工程中通行的做法

    例如：飞机，机床，电视机等

    设计、制造、试验运行、修改设计并迭代（如 J-20）

程序测试（Testing）

- 测试：通过试验运行确定程序能“正确”工作的过程
  - 用实际数据检查程序的行为，设法找出程序中的问题
  - 是实际软件开发中确认程序满足需要的基本手段
  - 在实际中工作量很大（整个开发工作量的一半以上），需要耗费大量的人工、物力和其他资源
- 测试的局限性
  - 一次测试运行只能针对一组特定数据进行
  - 程序处理的数据可能有大量不同情况，不可能穷尽

    例如，一个整型参数就有非常多可能取值（2^{32} 或 2^{64}）
- “测试只能发现程序有错，不能确认程序没有错”

  —— E. W. Dijkstra（1930-2002）

程序测试

- 虽然测试具有本质局限性，但
  - 仍然是发现程序错误的最重要手段，需要认真做好
  - 已经发展为计算机科学技术的一个重要研究领域
    - 怎样高效率地做，如果选择测试用例（test cases）
    - 如果评价测试的效果，怎样确定测试可以结束？
- 基本想法是选择有用的（？）数据，尝试运行被测程序，通过检查输出（或观察运行过程）确定程序功能是否正确，工作中
  - 可能发现程序错误，下一步是设法修改程序去消除错误
  - 经过“足够多”的测试，认为程序已经可以使用了
- 另一种检验技术是程序验证，通过逻辑推理等数学方法（严格）证明程序正确性。为此需要建立程序的语义理论。该领域和相关技术正在发展，已有了许多成果，也存在很多重大困难

单元测试和集成测试

- 复杂程序通常由一些部分组成
  - Python 程序的基本功能单元是函数，函数是具有独立性的代码体，有清晰的边界和功能
  - 复杂的 Python 程序可能由一组模块（代码文件）组成，一个模块也是一个代码单元（还有其他单元）
- 复杂程序（系统）的测试需要分步进行
  - 首先，尽可能彻底地检查一个个独立的代码单元，确认它们能的功能满足需求，这种工作称为单元测试
  - 各代码单元测试完成后，逐步将其集成起来测试（集成测试）
    - 检查越来越大的一组单元的整体功能满足需求
    - 最终完成整个系统的测试
- 良好的程序结构、功能分解（如函数分解），有利于有效的测试

黑箱测试

- 测试的一种想法是设法确定程序单元的功能，如正确参数/结果关系，输入/输出关系，这相当于把程序单元看作黑箱
  - 只考虑程序单元功能的测试称为黑箱测试
  - 黑箱测试的关键是两方面：
    - 能选择一组“合适的”测试用例
    - 对每个用例，必须有办法判断运行结果（效果）是否满足需求
- 常见的重要测试用例：
  - 基本情况，一些易于判断正误的情况
  - 边界情况，例如可能取到的最大最小值，特殊参数值等
  - 典型错误情况（考查程序单元能否正确处理错误）
  - 一些常见情况，一组一般性情况

白箱测试

- 对程序功能的另一种考虑：一个程序单元功能正确，那么其所有代码/所有可能的执行路径都“功能正确”
  - 程序代码和可能执行路径由代码结构确定，顺序/分支/循环等
  - 基于这一观点测试需考察程序的内部结构，称为白箱测试
- 白箱测试的做法：
  - 考察程序单元的代码，确定语句或设法确定可能执行路径
  - 设计测试用例，设法
    - “覆盖”程序代码中的每个语句，即要求代码中的每个语句至少被某一个测试用例的执行经过一次
    - “覆盖”所有执行路径。如果有循环，一般而言不可能穷尽所有路径（例如，**while** 循环通常意味着无穷多条执行路经），只能选择测试其中的一些典型路径

白箱测试

- 语句和路径覆盖的例子：

```
def isPrime(n) :
    if n < 2:
        return False
    k = 2
    while k * k <= n :
        if n % k == 0 :
            return False
        k += 1
    return True
```

考虑语句的覆盖

考虑路径“覆盖”

在测试中利用计算机和程序

- 测试通常有很多细节和机械性工作，应尽可能利用计算机进行，为此可能需要积累测试数据，写测试脚本
  - 所谓测试脚本，就是专为测试写的代码（函数，代码片段）
  - 测试 **Python** 程序时，直接用 **Python** 写测试脚本很方便
  - 还可能需要另外编写一些专门服务于测试的程序（下面介绍）
- 对复杂程序（单元、模块）要建立自动测试脚本和数据集
  - 设法使程序测试和结果检查都能自动进行。设法做到用一个命令就能完成整个测试，可以节省大量时间。修改程序后应该重新运行脚本，做一遍完整的测试
  - 不但要准备测试用例的数据，还要准备运行结果。用某组数据测试程序发现错误，就应把这组数据加入测试数据集
  - 一些软件公司在开发中每天都对整个系统做一遍完整测试

程序桩和驱动代码

- 测试程序时，经常需要写一些专门为测试服务的代码
- 需要测试的单元可能依赖一些尚未开发的单元。例如，被测函数可能需要调用几个尚未实现（尚未定义）的函数
  - 这时需要写出模仿被调函数的程序桩，使测试能够进行
  - 桩函数应与被模仿函数的参数相同，可简单返回结果
- 被测函数（等）要在一定环境中用，调用它的部分尚未开发
  - 需要写出临时调用函数的代码，自动或交互地生成实际调用
  - 这种代码称为驱动代码（driver）。调试脚本也是驱动代码，前面课堂上已经展示过很多这种代码，可供参考
- 人们开发了一些专门用于帮助做测试的系统，如单元测试工具
  - 基本功能就是帮助组织测试用例和单元测试过程
  - 复杂的系统功能都是特殊的，需要专门去做

程序调试（Debugging）

- 测试中发现程序的错误，就需要修改程序去纠正错误
  - 发现程序错误，绝大多数都是我们编程中写在程序里的错误，偶尔会遇到语言系统或基础系统的错误等
  - 纠正错误的工作最好由编程序的人自己做
- 找出程序中的实际错误并予以改正的工作称为 **debugging**（排除错误，排错，调试，原意是“捉虫子”）。有历史渊源
- 排除程序中的错误，主要靠人观察和分析，基本方法：
  - 人工检查代码，根据已知的错误情况设法确定出错的原因
  - 设计有针对性的数据，设法使错误重现或不出现，排除疑问
  - 确定错误根源后设法修改代码，消除错误（**fix a bug**）
  - 良好的程序结构有助于确定错误原因和修改程序、消除错误

调试

- 如果通过人工检查代码的方式无法确认和消除错误，可以通过追踪程序运行过程的方式进一步仔细检查
  - 常用方法：在程序一些关键点加入打印语句（如用 **print**），输入一些变量（表达式）的值，帮助判断
  - 利用 **debug** 工具，如 **Python IDLE** 有关功能（下面介绍）
- 对执行过程的所有检查工作都只能是提供线索，最终还是需要人来判断和思考，确定出错的原因和修改代码消除错误的方法
  - 良好的设计和程序结构有利于错误定位
  - 总结一些常见错误也有参考意义， **Python** 的典型错误
    - 忘记范围描述不包括上界
    - 无参函数调用忘记写括号，等
  - 最重要的错误都是具体的，如编程逻辑或者算法的错误

IDLE 调试功能

- 首先回顾一下程序执行的基本情况：
  - 从当前模块的执行代码（非函数定义）开始
  - 调用一个函数时执行进入该函数，执行其函数体
  - 一个函数调用完成时退出该函数，回到调用点之后继续
- 执行中任何时刻，执行过程通常位于某个函数里，而且
  - 可能有一串已开始执行但尚未完成的函数调用
  - 最后一个调用就是当前正在执行的函数
  - 每个函数都有局部变量，还可以访问在该函数外围作用域里定义的（**nonlocal**）变量和全局变量
  - 函数调用的状态记在一个运行栈里，每个函数有一个状态记录，调用函数时增加一个新记录，退出时消去一个记录

IDLE 调试功能

- IDLE 里默认是 run 方式是执行当前模块的所有代码，直至
  - 代码执行完毕，回到 shell 状态等待交互式输入
  - 或程序执行中出错，输出错误信息并回到 shell 状态
- IDLE 的调试支持功能：
  - 提供了一组控制程序的执行过程的操作
    - 单步执行，一次一个语句
    - 进入/完成函数，或者一步完成一个函数调用的执行
  - 在代码中设置断点，要求执行暂停在断点等待命令
  - 检查当前函数和其他尚未结束的函数的状态（局部变量、非局部变量和全局变量的取值情况）
  - 其他相关操作

IDLE 调试功能

- Python Shell 窗口的 Debug 菜单，提供主要功能
  - Debugger 项启动调试器窗口（进入调试模式，使程序以调试方式执行），关闭调试器窗口导致系统退出调试模式
  - Stack Viewer 菜单项打开一个运行栈追踪窗口
  - 如果勾了 Auto-open Stack Viewer 选项，一旦程序运行出错就会自动打开运行栈追踪窗口
- 下面简单介绍 Stack Viewer 和 Debugger
- Stack Viewer 窗口追踪程序里的函数调用关系，显示出错时的函数调用链（从上到下）
  - 点击各项，可以查看相应的局部变量值等
  - 显示出错的语句，可以追踪回到程序文件里的语句

程序调试

- 启动 **Debugger**（调试器）
  - 系统显示 **[DEBUG ON]**，执行将在调试器控制下进行
  - 可选显示 **Stack**（运行栈），**Source**（源代码），**Locals**（局部定义），**Globals**（全局定义）
- 调试器的运行控制按钮：
  - **Go**：运行到结束
  - **Step**：运行一步（一个基本语句，遇到函数调用时进入函数）
  - **Over**：执行到完成一个函数调用（不像单步执行那样进入调用函数内部，而是一直运行到被调函数的执行完成）
  - **Out**：运行到当前函数退出，执行到函数调用语句后的位置
  - **Quit**：结束本次调试运行

程序调试

- 在 **IDLE** 编辑器里可以设置（程序执行）断点（**breakpoint**）
  - 在希望程序运行中断的那行按右键，选 **Set Breakpoint**
    - 允许同时设置任意多个断点
    - 可随时用 **Clear Breakpoint** 清除断点，包括调试运行中
  - 在 **Python Shell** 窗口启动调试器
  - 在用 **Go** 按钮启动运行时，程序运行到一个断点就会中断执行，停在尚未执行那一行的状态
  - 可以通过调试器的菜单按钮（以任何方式）继续中断的执行
- 执行中断时，可以在调试器窗口里检查当前的状态情况
- 总之，**IDLE** 的调试使人可能比较方便地查看程序执行中的状态和状态变化情况，寻找程序错误的线索