1.5 控制

::: details INFO
译者：Mancuoj

来源：1.5 Control

对应：Lab 01
:::

我们现在可以定义的函数的能力十分有限，因为我们还没有引入一种方法来进行比较，并根据比较的结果执行不同的操作。控制语句将赋予我们这种能力，就是根据逻辑比较的结果来控制程序执行流程的语句。

语句与我们目前研究过的表达式有着根本的不同，它们没有值。执行一个控制语句决定了解释器接下来应该做什么，而不是计算某些东西。

1.5.1 语句

到目前为止，我们虽然主要思考的是如何计算求解表达式，但我们已经见过了三种语句：赋值（assignment）、`def` 和 `return` 语句。尽管这些 Python 代码都包含表达式作为它们的一部分，但它们本身并不是表达式。

语句不会被求解，而会被执行。每个语句都描述了对解释器状态的一些更改，并且执行语句就会应用该更改。正如我们在 `return` 和赋值语句中看到的那样，执行语句可能涉及求解其包含的子表达式。

表达式也可以作为语句执行，在这种情况下，它们会被求值，但它们的值会被丢弃。执行纯函数没有效果，但执行非纯函数会因为调用函数而产生效果。

思考一下，例如：

```
>>> def square(x):
        mul(x, x) # 小心！此调用不返回值。
```

这个例子是有效的 Python 代码，但可能不能达到预期。函数体由一个表达式组成。表达式本身是一个有效的语句，但语句的效果是调用 `mul` 函数，然后把结果丢弃。如果你想对表达式的结果做些什么，你需要用赋值语句存储它或用 `return` 语句返回它：

```
>>> def square(x):
        return mul(x, x)
```

有时，在调用 `print` 等非纯函数时，拥有一个主体为表达式的函数确实有意义。

```
>>> def print_square(x):
        print(square(x))
```

在最高层级上，Python 解释器的工作是执行由语句组成的程序。然而，很多有趣的计算工作都来自对表达式的求值。语句用来管理程序中不同表达式之间的关系，以及它们产生的结果。

1.5.2 复合语句

通常，Python 代码是一系列语句。简单语句是不以冒号结尾的单行，而由其他语句（简单语句和复合语句）组成被称为复合语句。复合语句通常跨越多行，以单行头部（header）开始，并以冒号结尾，其中冒号标识语句的类型。头部和缩进的句体（suite）一起称为子句。复合语句由一个或多个子句组成：

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

我们可以用这些术语来理解我们之前介绍过的语句。

- 表达式、返回语句和赋值语句都是简单语句。
- `def` 语句是复合语句，`def` 头后面的句体定义了函数体。

对每类 header 都有专门的求值规则来规定其何时执行以及是否执行其句体中的语句。我们说“the header controls its suite”，例如，在 `def` 语句中，`return` 表达式不会立即求值，而是存储起来供以后调用该函数时使用。

我们现在也可以理解多行程序了。

- 要执行一系列语句，会先执行第一个语句。如果该语句不重定向控制，则继续执行语句序列的其余部分（如果还有的话）。

这个定义揭示了递归定义序列（sequence）的基本结构：一个序列可以分解成它的第一个元素和其余元素。语句序列的“其余部分”本身也是语句序列！因此，我们可以递归地应用这个执行规则。这种将序列视为递归的数据结构的观点将在后面的章节中再次出现。

此规则的重要结论是语句会按顺序执行，但由于重定向控制（redirected control），后面的语句可能永远不会被执行到。

实践指南：缩进句体时，所有行必须以相同的方式缩进相同的量（使用空格，而不是制表符）。缩进的任何变化都会导致错误。

1.5.3 定义函数 II：局部赋值

最初，我们声明用户定义函数的主体仅由包含单个返回表达式的 `return` 语句组成。事实上，函数可以定义超出单个表达式的一系列操作。

每当用户定义的函数被调用时，其句体中的子句序列将会在局部环境中执行 --> 该环境通过调用函数创建的局部帧开始。`return` 语句会重定向控制：每当执行一个 `return` 语句时，函数应用程序就会终止，`return` 表达式的值会作为被调用函数的返回值。

赋值语句可以出现在函数体内。例如，以下函数使用了两步计算，首先计算两个数的差的绝对值，然后求出它与第一个数的百分比值并返回：

赋值语句的作用是将名称与当前环境中的第一帧的值绑定。因此，函数体内的赋值语句不会影响全局帧。“函数只能操纵其局部帧”是创建模块化程序的关键，而在模块化程序中，纯函数仅通过它们接收和返回的值与外界交互。

当然， `percent_difference` 函数可以写成单个表达式，如下所示，但返回表达式会更复杂。

```
>>> def percent_difference(x, y):
        return 100 * abs(x-y) / x
>>> percent_difference(40, 50)
25.0
```

到目前为止，局部赋值并没有增强函数定义的表达能力，而当它与其他控制语句结合时，就会增强。此外，局部赋值在“通过为中间量赋名来解释复杂表达式的含义”方面也起着至关重要的作用。

1.5.4 条件语句

Python 有一个用于计算绝对值的内置函数。

```
>>> abs(-2)
2
```

我们希望能够自己实现这样一个函数，但是没有清晰的方法来定义一个具有比较和选择的函数。我们想表达的是，如果 `x` 为正，则 `abs(x)` 返回 `x` ；此外，如果 `x` 为 0，则 `abs(x)` 返回 0；否则，`abs(x)` 返回 `-x`。在 Python 中，我们可以用条件语句来表达这种选择。

这个 `absolute_value` 函数的实现提出了几个重要的问题：

条件语句（Conditional statement）：Python 中的条件语句由一系列头部和句体组成：必需的 `if` 子句、可选的 `elif` 子句序列，最后是可选的 `else` 子句：

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

执行条件语句时，每个子句都会按顺序被考虑。执行条件子句的计算过程如下。

1. 求解头部的表达式
2. 如果它是真值，则执行该句体。然后，跳过条件语句中的所有后续子句。

如果到达 `else` 子句（仅当所有 `if` 和 `elif` 表达式的计算结果为假值时才会发生），则执行其句体。

布尔上下文（Boolean contexts）：上面，执行过程提到了“假值 a false value”和“真值 a true value”。条件块头部语句内的表达式被称为布尔上下文：它们值的真假对控制流很重要，另外，它们的值不会被赋值或返回。Python 包含多个假值，包括 0、 `None` 和布尔值 `False`，所有其他数字都是真值。在第二章中，我们将看到 Python 中的每种内置数据都具有真值和假值。

布尔值（Boolean values）：Python 有两个布尔值，分别叫做 `True` 和 `False` 。布尔值表示逻辑表达式中的真值。内置的比较运算符 >, <, > =, <=, ==, != 会返回这些值。

```
>>> 4 < 2
False
>>> 5 >= 5
True
```

第二个例子读作“5 大于或等于 5”，对应于 `operator` 模块中的函数 `ge`。

```
>>> 0 == -0
True
```

最后一个示例读作“0 等于 -0”，对应于 `operator` 模块中的 `eq`。请注意，Python 会区分赋值符号 `=` 与相等比较符号 `==`，这也是许多编程语言共享的约定。

布尔运算符（Boolean operators）：Python 中还内置了三个基本的逻辑运算符：

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

逻辑表达式具有相应的求值过程。而这些过程利用了这样一个理论 --> 有时，逻辑表达式的真值可以在不对其所有子表达式求值的情况下确定，这一特性称为短路（short-circuiting）。

求解表达式 `<left> and <right>` 的步骤如下：

1. 求解子表达式 `<left>`。
2. 如果左边的结果为假值 v，则表达式的计算结果就是 v。
3. 否则，表达式的计算结果为子表达式 `<right>` 的值。

求解表达式 `<left> or <right>` 的步骤如下：

1. 求解子表达式 `<left>`。
2. 如果左边的结果为真值 v，则表达式的计算结果就是 v。
3. 否则，表达式的计算结果为子表达式 `<right>` 的值。

求解表达式 `not <exp>` 的步骤如下：

1. 求解 `<exp>`，如果结果为假值，则值为 `True`，否则为 `False`。

这些值、规则和运算符为我们提供了一种组合比较结果的方法。执行比较并返回布尔值的函数通常以 `is` 开头，后面不跟下划线（例如 `isfinite, isdigit, isinstance` 等）。

1.5.5 迭代

除了选择要执行的语句外，控制语句还用于重复。如果我们编写的每一行代码只执行一次，那么编程将是一项非常低效的工作。只有通过重复执行语句，我们才能释放计算机的全部潜力。我们之前已经见过了一种重复形式：一个函数只用定义一次，就可以被多次调用。迭代控制（Iterative control）结构是另一种多次执行相同语句的机制。

思考斐波那契数列，其中每个数都是前两个数的和：

$0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$

每个值都是通过重复应用 `sum-previous-two` 的规则构建的，第一个和第二个值固定为 0 和 1。

我们可以使用 `while` 语句来枚举 n 项斐波那契数列。我们需要跟踪已经创建了多少个值（`k`），和第 k 个值（`curr`）及其前身（`pred`）。单步执行此函数并观察斐波那契数如何一个一个地演化，并绑定到 curr。

请记住，单行赋值语句可以用逗号分隔多个名称和值同时赋值。该行：

`pred, curr = curr, pred + curr`

将名称 `pred` 重新绑定到 `curr` 的值，同时将 `curr` 重新绑定到 `pred + curr` 的值。所有 `=` 右侧的所有表达式都会在绑定之前计算出来。

在更新左侧的绑定之前求出所有 `=` 右侧的内容 --> 这种事件顺序对于此函数的正确性至关重要。

`while` 子句包含一个头部表达式，后跟一个句体：

```
while <expression>:
    <suite>
```

要执行 `while` 子句：

1. 求解头部的表达式。
2. 如果是真值，则执行后面的句体，然后返回第 1 步。

在第 2 步中，`while` 子句的整个句体在再次计算头部表达式之前执行。

为了防止 `while` 子句的句体无限期地执行，句体应该总是在每次循环中更改一些绑定。

不会终止的 `while` 语句被称为无限循环（infinite loop）。按 `<Control>-C` 可以强制 Python 停止循环。

1.5.6 测试

测试一个函数就是去验证函数的行为是否符合预期。现在我们的函数语句已经足够复杂，所以我们需要开始测试我们的实现的函数功能。

测试是一种系统地执行验证的机制。它通常采用另一个函数的形式，其中包含对一个或多个对被测试函数的调用样例，然后根据预期结果验证其返回值。与大多数旨在通用的函数不同，测试需要选择特定参数值，并使用它们验证函数调用。测试也可用作文档：去演示如何调用函数，以及如何选择合适的参数值。

断言（Assertions）：程序员使用 `assert` 语句来验证是否符合预期，例如验证被测试函数的输出。`assert` 语句在布尔上下文中有一个表达式，后面是一个带引号的文本行（单引号或双引号都可以，但要保持一致），如果表达式的计算结果为假值，则显示该行。

```
>>> assert fib(8) == 13, '第八个斐波那契数应该是 13'
```

当被断言的表达式的计算结果为真值时，执行断言语句无效。而当它是假值时，`assert` 会导致错误，使程序停止执行。

fib 的测试函数应该测试几个参数，包括 n 的极限值。

```
>>> def fib_test():
        assert fib(2) == 1, '第二个斐波那契数应该是 1'
        assert fib(3) == 1, '第三个斐波那契数应该是 1'
        assert fib(50) == 7778742049, '在第五十个斐波那契数发生 Error'
```

当在文件中而不是直接在解释器中编写 Python 时，测试通常是在同一个文件或带有后缀 `_test.py` 的相邻文件中编写的。

文档测试（Doctests）：Python 提供了一种方便的方法，可以将简单的测试直接放在函数的文档字符串中。文档字符串的第一行应该包含函数的单行描述，接着是一个空行，下面可能是参数和函数意图的详细描述。此外，文档字符串可能包含调用该函数的交互式会话示例：

```
>>> def sum_naturals(n):
        """返回前 n 个自然数的和。

        >>> sum_naturals(10)
        55
        >>> sum_naturals(100)
        5050
        """
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total
```

然后，可以通过 doctest 模块 来验证交互，如下。

```
>>> from doctest import testmod
>>> testmod()
TestResults(failed=0, attempted=2)
```

如果仅想验证单个函数的 doctest 交互，我们可以使用名为 `run_docstring_examples` 的 `doctest` 函数。不幸的是，这个函数调用起来有点复杂。第一个参数是要测试的函数；第二个参数应该始终是表达式 `globals()` 的结果，这是一个用于返回全局环境的内置函数；第三个参数 `True` 表示我们想要“详细”输出：所有测试运行的目录。

```
>>> from doctest import run_docstring_examples
>>> run_docstring_examples(sum_naturals, globals(), True)
Finding tests in NoName
Trying:
    sum_naturals(10)
Expecting:
    55
ok
Trying:
    sum_naturals(100)
Expecting:
    5050
ok
```

当函数的返回值与预期结果不匹配时，`run_docstring_examples` 函数会将此问题报告为测试失败。

当你在文件中编写 Python 时，可以通过使用 doctest 命令行选项启动 Python 来运行文件中的所有 doctest：

```
python3 -m doctest <python_source_file>
```

有效测试的关键是在实现新功能后立即编写（并运行）测试。在实现之前编写一些测试也是一种很好的做法，以便在你的脑海中有一些示例输入和输出。调用单个函数的测试称为单元测试（unit test）。详尽的单元测试是良好程序设计的标志。