1.2 编程要素

▶ INFO

编程语言不仅是一种指挥计算机执行任务的手段，它还应该成为一种框架，使我们能够在其中组织自己有关计算过程的思想。程序也用于在编程社区的成员之间传达想法，所以程序必须是人类可以阅读的，并且“恰巧”能被机器执行。

这样，当我们描述一种语言时，就需要特别注意该语言所提供的能够将简单思想组合成复杂思想的工具。每一种强大的语言都有这样三种机制：

- **原始表达式和语句**：语言所关心的最简单的个体
- **组合方法**：由简单元素组合构建复合元素
- **抽象方法**：命名复合元素，并将其作为单元进行操作

在编程中，我们只会处理两种元素：函数和数据（之后你会发现它们实际上并不是泾渭分明的），不那么正式的说法是：数据是我们想要操作的东西，而函数是操作这些数据的规则的描述。因此，任何强大的编程语言都必须能表达基本的数据和函数，并且提供对函数和数据进行组合和抽象的方法。

1.2.1 表达式

上一节中，我们完整尝试了 Python 解释器，而下面我们将重新开始，一步步地讲解 Python 语言。如果示例看起来过于简单，请耐心等待，更令人振奋的东西在后面呢。

我们从一种基本表达式“数字 number”开始，更准确地说，是你键入的，十进制数字组成的表达式。

py
```
>>> 42
42
```

表达式表示的数字可以与数学运算符组合形成一个复合表达式，解释器将对其进行求值：

py
```
>>> -1 - -1
0
>>> 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
0.9921875
```

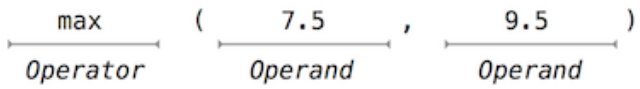
这些数学表达式使用中缀表示法（infix notation），运算符（例如 +、-、* 或 /）出现在操作数之间。Python 包含许多种形成复合表达式的方法，我们会在学习中慢慢引入新的表达式形式和它们所支持的语言特性，而不是立即把它们列举出来。

1.2.2 调用表达式

最重要的一种复合表达式是调用表达式，它将函数运用于一些参数上。回想一下代数中的函数的数学概念：函数就是从一些输入参数到输出值的映射。例如， `max` 函数会输出一个最大的输入值，也就是将多个输入映射到了单个输出上。Python 中函数应用的方式与传统数学相同。

py
```
>>> max(7.5, 9.5)
9.5
```

这个调用表达式包含子表达式（subexpressions）：在括号之前是一个运算符表达式，而括号里面是一个以逗号分隔的操作数表达式的列表。



运算符指定了一个函数，在对这个调用表达式进行求值时，我们会说：使用参数 7.5 和 9.5 来调用函数 `max` ，最后返回 9.5。

调用表达式中参数的顺序是很重要的。例如， `pow` 函数的第二个参数是第一个参数的幂。

py
```
>>> pow(100, 2)
10000
>>> pow(2, 100)
1267650600228229401496703205376
```

函数符号相比传统的中缀数学符号有三个主要优点。首先，因为函数名总是在参数前面，函数可以接收任意数量的参数而不会产生歧义。

py
```
>>> max(1, -2, 3, -4)
3
```

其次，函数可以直接扩展为嵌套（nested）表达式，其元素本身就是复合表达式。不同于中缀复合表达式，调用表达式的嵌套结构在括号中是完全明确的。

py
```
>>> max(min(1, -2), min(pow(3, 5), -4))
-2
```

这种嵌套的深度（理论上）没有任何限制，Python 解释器可以解释任何复杂的表达式。但人类很快就会被多层嵌套搞晕，所以作为一个程序员，你的一个重要目标就是：构造你自己、你的编程伙

伴和其他任何可能阅读你代码的人都可以解释的表达式。

第三点，数学符号在形式上多种多样：星号表示乘法，上标表示幂指数，水平横杠表示除法，带有倾斜壁板的屋顶表示平方根，而其中一些符号很难被输入！但是，所有这些复杂事物都可以通过调用表达式的符号来进行统一。Python 除了支持常见的中缀数学符号（如 + 和 -）之外，其他任何运算符都可以表示为一个带有名称的函数。

1.2.3 导入库函数

Python 定义了大量的函数，包括上一节中提到的运算符函数，但默认情况下我们不能直接使用名字来调用它们。Python 将已知函数和其他东西组织起来放入到了模块中，而这些模块共同组成了 Python 库。我们要使用的时候需要导入它们，例如， `math` 模块提供了各种熟悉的数学函数：

```py
>>> from math import sqrt
>>> sqrt(256)
16.0
```

`operator` 模块提供了中缀运算符对应的函数：

```py
>>> from operator import add, sub, mul
>>> add(14, 28)
42
>>> sub(100, mul(7, add(8, 4)))
16
```

`import` 语句需要指定模块名称（例如 `operator` 或 `math`），然后列出要导入该模块里的具名函数（例如 `sqrt`）。一个函数被导入后就可以被多次调用。

使用运算符函数（例如 `add`）和运算符号（例如 `+`）之间并没有任何区别。按照惯例来说，大多数程序员使用符号和中缀表示法来表达简单的算术。

Python 3 库文档 列出了每个模块中定义的函数，例如 math 模块。但是，该文档是为熟悉整个语言的开发人员编写的。现在来说，尝试使用函数可能会比阅读文档更能使你了解函数的行为。而当你熟悉了 Python 语言和词汇时，这个文档就将会成为你宝贵的参考资料。

1.2.4 名称与环境

编程语言的一个要素就是使用名称来引用计算对象，如果一个值被赋予了名称，我们说名称绑定到了值上面。

在 Python 中，我们可以使用赋值语句建立新的绑定， `=` 左边是名称，右边是值。

```py
>>> radius = 10
>>> radius
10
>>> 2 * radius
20
```

名称也可以通过 `import` 语句绑定。

```py
>>> from math import pi
>>> pi * 71 / 223
1.0002380197528042
```

`=` 在 Python 中称为 **赋值** 符号（即 assignment operator，许多其他语言也是如此），赋值是最简单的 **抽象** 方法，因为它允许我们使用简单名称来指代复合操作的结果，例如上面计算的 `area` 。这样，复杂的程序由复杂性递增的计算对象逐步构建。

将名称与值绑定，之后通过名称检索可能的值，就意味着解释器必须维护某种内存来记录名称、值和绑定，这种内存就是环境（environment）。

名称也可以与函数绑定。例如，名称 `max` 就和我们之前使用的 `max` 函数进行了绑定。与数字不同，函数很难以文本呈现，因此当询问一个函数时，Python 会打印一个标识来描述：

```py
>>> max
<built-in function max>
```

赋值语句可以为现有函数赋予新名称。

```py
>>> f = max
>>> f
<built-in function max>
>>> f(2, 3, 4)
4
```

之后再次赋值可以将已有名称与新值绑定。

```py
>>> f = 2
>>> f
2
```

在 Python 中，名称通常被称为“变量名 variable names”或“变量 variables”，因为它们可以在执行程序的过程中与不同的值绑定。当一个变量通过赋值语句与一个新值绑定，它就不再绑定以前的值。你甚至可以将内置名称与新值绑定。

```py
>>> max = 5
>>> max
5
```

将 `max` 赋值为 5 后，名称 `max` 不再绑定函数，因此调用 `max(2, 3, 4)` 将导致错误。

执行赋值语句时，Python 会先求解 `=` 右侧的表达式，再将结果与左侧的名称绑定，所以可以在右侧表达式中引用一个已绑定的变量。

```py
>>> x = 2
>>> x = x + 1
>>> x
3
```

还可以在单个语句中为多个变量分配值，左右都用逗号隔开。

```py
>>> area, circumference = pi * radius * radius, 2 * pi * radius
>>> area
314.1592653589793
>>> circumference
62.83185307179586
```

更改一个变量的值不会影响其他变量。即使下列代码中 `area` 的值由最初定义的 `radius` 绑定，但改变 `radius` 的值并不能更新 `area` 的值，我们需要另一个赋值语句来更新它。

```py
>>> radius = 11
>>> area
314.1592653589793
>>> area = pi * radius * radius
380.132711084365
```

对于多重赋值，所有 `=` 右边的表达式都会先求值，然后再与左边的名称绑定。在这个规则下，我们可以在单个语句内交换两个变量的值。

```py
>>> x, y = 3, 4.5
>>> y, x = x, y
>>> x
4.5
>>> y
3
```

1.2.5 求解嵌套表达式

本章的目标之一是在“以程序的角度思考”中隔离其他的问题，举一个恰当的例子，就是思考一下在求解嵌套表达式时，解释器自身的操作过程。

为了求值一个表达式，Python 将执行以下操作：

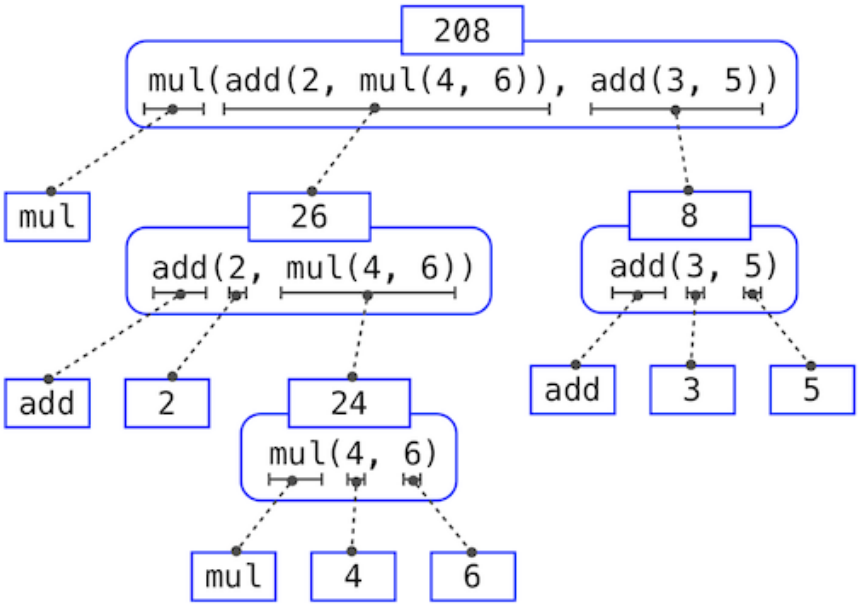
- 求解运算符子表达式和操作数子表达式
- 然后将操作数子表达式的值作为运算符子表达式的函数的参数

这个简单的过程也说明了有关流程的一些要点。第一步规定：为了求出调用表达式，必须首先求出其他表达式。因此，求值程序本质上是 **递归** 的，也就是说它会自己调用自己作为步骤之一。

例如，此式需要应用四次求值过程。

```
>>> sub(pow(2, add(1, 10)), pow(2, 5))
2016
```

如果把每个需要求解的表达式都抽出来，我们可以看到这个求值过程的层次结构。



这个图叫做表达式树，在计算机科学中，树通常从上到下增长。树中每个点的对象都叫做节点。这里节点分别是表达式和表达式的值。

求解根节点（即顶部的完整表达式），需要首先求解子表达式，也就是分支节点。叶子节点（也就是没有分支的节点）表示函数或数值。内部节点有两部分：我们想要应用的求值规则的调用表达式，以及该表达式的结果。观察这棵树的求解过程，我们可以想象操作数的值会向上流动，从叶子节点开始一步步向上组合。

接下来，观察第一步的重复应用，将我们带到我们需要求解的原始表达式，而不是调用表达式，例如数字（例如 2）和名称（例如 add）。我们规定基本逻辑为：

- 数字的值就是它们所表示的数值
- 名称的值是环境中关联此名称的对象

注意环境在决定表达式中的符号意义上有重要作用。在 Python 中，不指定任何环境信息去谈论一个值是没有意义的，例如名称 `x` 和 `add` 。环境为求解提供了上下文信息，对理解程序执行过程

有着重要作用。

```py
>>> add(x, 1)
```

这个求解步骤并不能对所有 Python 代码求值，它仅能求解调用表达式、数字和名称。例如，它并不能处理赋值语句。

```py
>>> x = 3
```

因为赋值语句的目的是将名称与值绑定，它并不返回值，也不应用参数去求解函数。也就是说，赋值语句不被求解但“被执行”，它们只是做出一些改变但不产生值。每种类型的表达式或语句都有自己的求解或执行过程。

注意：当我们说“一个数字求解为一个数值”时，实际上是 Python 解释器将数字求解为数值，是解释器赋予了编程语言这个意义。鉴于解释器是一个始终表现一致的固定程序，我们就可以说数字（以及表达式）会在 Python 程序的上下文中被求解为值。

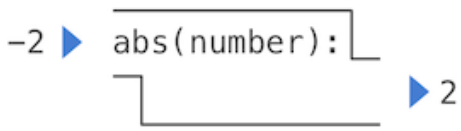
1.2.6 非纯函数 print

在本节中，我们将区分两种类型的函数。

纯函数（Pure functions）：函数有一些输入（参数）并返回一些输出（调用返回结果）。

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

可以将内置函数 `abs` 描述为接受输入并产生输出的小型机器。

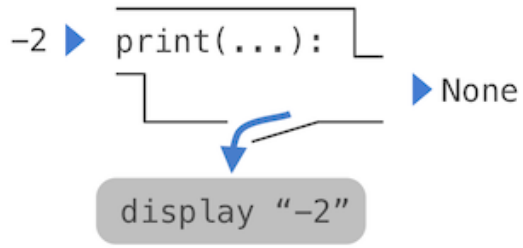


`abs` 就是纯函数，纯函数在调用时除了返回值外不会造成其他任何影响，而且在使用相同的参数调用纯函数时总是会返回相同的值。

非纯函数（Non-pure functions）：除了返回值外，调用一个非纯函数还会产生其他改变解释器和计算机的状态的副作用（side effect）。一个常见的副作用就是使用 `print` 函数产生（非返回值的）额外输出。

```py
>>> print(1, 2, 3)
1 2 3
```

虽然 `print` 和 `abs` 在这些例子中看起来很相似，但它们的工作方式基本不同。 `print` 返回的值始终为 `None` ，这是一个不代表任何内容的特殊 Python 值。而交互式 Python 解释器并不会自动打印 `None` 值，所以 `print` 函数的额外输出就是它的副作用。



下面这个调用 `print` 的嵌套表达式就展示了非纯函数的特征。

py

```
>>> print(print(1), print(2))
1
2
None None
```

如果你发现这个输出结果出乎你的意料，可以画一个表达式树来解释求解该表达式会产生特殊输出的原因。

小心使用 `print` 函数！它返回 `None` 意味着它不应该用于赋值语句。

py

```
>>> two = print(2)
2
>>> print(two)
None
```

纯函数不能有副作用，或是随着时间推移的改变的限制，但是对其施加这些限制会产生巨大的好处。首先，纯函数可以更可靠地组成复合调用表达式。在上面的示例中可以看到在操作数表达式中使用非纯函数 `print` 并不能返回有用的结果，但另一方面，我们已经看到 `max, pow, sqrt` 等函数可以在嵌套表达式中有效使用。

第二，纯函数往往更易于测试。相同的参数列表会返回相同的值，我们可以将其与预期的返回值进行比较。本章后面将更详细地讨论测试。

第三，第四章将说明纯函数对于编写可以同时计算多个调用表达式的并发程序来说是必不可少的。

此外，第二章会研究一系列非纯函数并描述它们的用途。

所以我们将在本章的剩余部分重点介绍纯函数的创建和使用， `print` 函数仅用于查看计算的中间结果。

上一页
1.1 开始

下一页
1.3 定义新的函数

正在加载评论.....