

master ▾

...

the-craft-of-selfteaching / markdown / Part.1.E.2.values-and-their-operators.md



xiaolai From wilton selfteaching#756



3 contributors



Raw

Blame




494 lines (337 sloc) 22.9 KB

值及其相应的运算

从结构上来看，一切的计算机程序，都由且只由两个最基本的成分构成：

- 运算 (Evaluation)
- 流程控制 (Control Flow)

没有流程控制的是计算器而已；有流程控制的才是可编程设备。

看看之前我们见过的计算质数的程序：（按一下 ，即 `ESC`，确保已经进入命令模式， 可以切换是否显示代码行号）

```
def is_prime(n):                # 定义 is_prime(), 接收一个参数
    if n < 2:                    # 开始使用接收到的那个参数 (值) 开始计算.....
        return False           # 不再是返回给人，而是返回给调用它的代码.....
    if n == 2:
        return True
    for m in range(2, int(n**0.5)+1):
        if (n % m) == 0:
            return False
    else:
        return True

for i in range(80, 110):
```

```
if is_prime(i):          # 调用 is_prime() 函数,
    print(i)             # 如果返回值为 True, 则向屏幕输出 i
```

```
83
89
97
101
103
107
109
```

`if...` , `for...` 在控制流程：在什么情况下运算什么，在什么情况下重复运算什么；

第 13 行 `is_prime()` 这个函数的调用，也是在控制流程 —— 所以我们可以**把函数看作是“子程序”**；

一旦这个函数被调用，流程就转向开始执行在第 1 行中定义的 `is_prime()` 函数内部的代码，而这段代码内部还是计算和流程控制，决定一个返回值 —— 返回值是布尔值；再回到第 13 行，将返回值交给 `if` 判断，决定是否执行第 14 行.....

而计算机这种可编程设备之所以可以做流程控制，是因为它可以做布尔运算，即，它可以对布尔值进行操作，而后将布尔值交给分支和循环语句，构成了程序中的流程控制。

值

从本质上看，程序里的绝大多数语句包含着运算 (Evaluation)，即，在对某个值进行评价。这里的“评价”，不是“判断某人某事的好坏”，而是“计算出某个值究竟是什么”—— 所以，我们用中文的“运算”翻译这个“Evaluation”可能表达得更准确一些。

在程序中，被运算的可分为常量 (Literals) 和变量 (Variables)。

```
a = 1 + 2 * 3
a += 1
print(a)
```

在以上代码中，

`1`、`2`、`3`，都是常量。Literal 的意思是“字面的”，顾名思义，常量的值就是它字面上的值。`1` 的值，就是 `1`。

`a` 是变量。顾名思义，它的值将来是可变的。比如，在第 2 句中，这个变量的值发生了改变，之前是 `7`，之后变成了 `8`。

第 1 句中的 `+`、`*`，是操作符 (Operators)，它用来对其左右的值进行相应的运算而后得到一个值。先是由操作符 `*` 对 `2` 和 `3` 进行运算，生成一个值 `6`；然后再由操作符 `+` 对 `1` 和 `6` 进行运算，生成一个值 `7`。先算乘除后算加减，这是操作符的优先级决定的。

`=` 是赋值符号，它的作用是将它右边的值保存到左边的变量中。

值是程序的基础成分 (Building blocks)，它就好像盖房子用的砖块一样，无论什么样的房子，到最后都主要是由砖块构成。

常量，当然有个值——就是它们字面所表达的值。

变量必须先赋值才能使用，也就是说，要先把一个值保存到变量中，它才能在其后被运算。

在 Python 中每个函数都有返回值，即便你在定义一个函数的时候没有设定返回值，它也会加上默认的返回值 `None` (请注意 `None` 的大小写！)

```
def f():  
    pass  
print(f())          # 输出 f() 这个函数被调用后的返回值, None  
print(print(f()))   # 这一行最外围的 print() 调用了一次 print(f()), 所以输出一个 N  
                    # 而后再输出这次调用的返回值, 所以又输出一次 None
```

```
None  
None  
None
```

当我们调用一个函数的时候，本质上来看，就相当于：

我们把一个值交给某个函数，请函数根据它内部的运算和流程控制对其进行操作而后返回另外一个值。

比如，`abs()` 函数，就会返回传递给它的值的绝对值；`int()` 函数，会将传递给它的值的小数部分砍掉；`float()` 接到整数参数之后，会返回这个整数的浮点数形式：

```
from IPython.core.interactiveshell import InteractiveShell  
InteractiveShell.ast_node_interactivity = "all"  
  
abs(-3.14159)  
int(abs(-3.14159))  
float(int(abs(-3.14159)))
```

```
3.14159  
3  
3.0
```

值的类型

在编程语言中，总是包含最基本的三种数据类型：

- 布尔值 (Boolean Value)
- 数字 (Numbers)： 整数 (Int)、浮点数 (Float)、复数 (Complex Numbers)
- 字符串 (Strings)

既然有不同类型的数据，它们就分别对应着不同类型的值。

运算的一个默认法则就是，通常情况下应该是相同类型的值才能相互运算。

显然，数字与数字之间的运算是合理的，但你让 `+` 这个操作符对一个字符串和一个数字进行运算就不行：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
11 + 10 - 9 * 8 / 7 // 6 % 5
'3.14' + 3                      # 这一句会报错
```

```
20.0
```

```
-----
----
```

```
TypeError                                Traceback (most recent call
last)
```

```
<ipython-input-18-e922b7565e53> in <module>
      3
      4 11 + 10 - 9 * 8 / 7 // 6 % 5
----> 5 '3.14' + 3                      # 这一句会报错
```

```
TypeError: can only concatenate str (not "int") to str
```

所以，在不得不对不同类型的值进行运算之前，总是要事先做 Type Casting (类型转换)。比如，

- 将字符串转换为数字用 `int()`、`float()`；
- 将数字转换成字符串用 `str()`；

另外，即便是在数字之间进行计算的时候，有时也需要将整数转换成浮点数字，或者反之：

- 将整数转换成浮点数字用 `float()`；
- 将浮点数字转换成整数用 `int()`；

有个函数，`type()`，可以用来查看某个值属于什么类型：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
type(3)
type(3.0)
type('3.14')
type(True)
type(range(10))
type([1,2,3])
type((1,2,3))
type({1,2,3})
type({'a':1, 'b':2, 'c':3})
```

```
int
float
str
bool
range
list
tuple
set
dict
```

操作符

针对不同类型的数据，有各自专用的操作符。

数值操作符

针对数字进行计算的操作符有加减乘除商余幂：`+`、`-`、`*`、`/`、`//`、`%`、`**`。

其中 `+` 和 `-` 可以对单个值进行操作，`-3`；其它的操作符需要有两个值才能操作。

从优先级来看，这些操作符中：

- 对两个值进行操作的 `+`、`-` 的优先级最低；
- 稍高的是 `*`、`/`、`//`、`%`；
- 更高的是对单个值进行操作的 `+`、`-`；
- 优先级最高的是 `**`。

完整的操作符优先级列表，参见官方文档：

布尔值操作符

针对布尔值，操作符有与、或、非：`and`、`or`、`not`。

它们之中，优先级最低的是或 `or`，然后是与 `and`，优先级最高的是非 `not`：

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

```
False
```

最先操作的是 `not`，因为它优先级最高。所以，上面的表达式相当于 `True and False or (not True)`，即相当于 `True and False or False`；

然后是 `and`，所以，`True and False or False` 相当于是 `(True and False) or False`，即相当于 `False or False`；

于是，最终的值是 `False`。

逻辑操作符

数值之间还可以使用逻辑操作符，`1 > 2` 返回布尔值 `False`。逻辑操作符有：`<`（小于）、`<=`（小于等于）、`>`（大于）、`>=`（大于等于）、`!=`（不等于）、`==`（等于）。

逻辑操作符的优先级，高于布尔值的操作符，低于数值计算的操作符。 即：数值计算的操作符优先级最高，其次是逻辑操作符，布尔值的操作符优先级最低。

```
n = -95
n < 0 and (n + 1) % 2 == 0
```

```
True
```

字符串操作符

针对字符串，有三种操作：

- 拼接：`+` 和 `' '`（后者是空格）
- 拷贝：`*`
- 逻辑运算：`in`、`not in`；以及，`<`、`<=`、`>`、`>=`、`!=`、`==`

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

'Awesome' + 'Python'
'Awesome' 'Python'
'Python, ' + 'Awesome! ' * 3
'o' in 'Awesome' and 'o' not in 'Python'
```

```
'AwesomePython'
'AwesomePython'
'Python, Awesome! Awesome! Awesome! '
False
```

字符之间，字符串之间，除了 `==` 和 `!=` 之外，也都可以被逻辑操作符 `<`、`<=`、`>`、`>=` 运算：

```
'a' < 'b'
```

```
True
```

这是因为字符对应着 Unicode 码，字符在被比较的时候，被比较的是对应的 Unicode 码。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

'A' > 'a'
ord('A')
ord('a')
```

```
False
65
97
```

当字符串被比较的时候，将从两个字符串各自的第一个字符开始逐个比较，“一旦决出胜负马上停止”：

```
'PYTHON' > 'Python 3'
```

```
False
```

列表的操作符

数字和字符串（由字符构成的序列）是最基本的数据类型，而我们往往需要批量处理数字和字符串，这样的時候，我们需要数组（Array）。不过，在 Python 语言中，它提供了一个容器（Container）的概念，用来容纳批量的数据。

Python 的容器有很多种——字符串，其实也是容器的一种，它的里面容纳着批量的字符。

我们先简单接触一下另外一种容器：列表（List）。

列表的标示，用方括号 `[]`；举例来说，`[1, 2, 3, 4, 5]` 和 `['ann', 'bob', 'cindy', 'dude', 'eric']`，或者 `['a', 2, 'b', 32, 22, 12]` 都是一个列表。

因为列表和字符串一样，都是有序容器（容器还有另外一种是无序容器），所以，它们可用的操作符其实相同：

- 拼接：`+` 和 `' '`（后者是空格）
- 拷贝：`*`
- 逻辑运算：`in`、`not in`；以及，`<`、`<=`、`>`、`>=`、`!=`、`==`

两个列表在比较时（前提是两个列表中的数据元素类型相同），遵循的还是跟字符串比较相同的规则：“一旦决出胜负马上停止”。但实际上，由于列表中可以包含不同类型的元素，所以，通常情况下没有实际需求对他们进行“大于、小于”的比较。（比较时，类型不同会引发 `TypeError`）

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
a_list = [1, 2, 3, 4, 5]
b_list = [1, 2, 3, 5]
c_list = ['ann', 'bob', 'cindy', 'dude', 'eric']
a_list > b_list
10 not in a_list
'ann' in c_list
```

```
False
True
True
```

更复杂的运算

对于数字进行加、减、乘、除、商、余、幂的操作，对于字符串进行拼接、拷贝、属于的操作，对布尔值进行或、与、非的操作，这些都是相对简单的运算。

更为复杂一点的，我们要通过调用函数来完成 —— 因为在函数内部，我们可以用比“单个表达式”更为复杂的程序针对传递进来的参数进行运算。换言之，函数就相当于各种事先写好的子程序，给它传递一个值，它会对其进行运算，而后返回一个值（最起码返回一个 `None`）。

以下是 Python 语言所有的内建函数（[Built-in Functions](#)）：

(Python	Built-in	Functions)
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	import()
complex()	hasattr()	max()	round()	

现在倒不用着急一下子全部了解它们 —— 反正早晚都会的。

这其中，针对数字，有计算绝对值的函数 `abs()`，有计算商余的函数 `divmod()` 等等。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

abs(-3.1415926)
divmod(11, 3)
```

```
3.1415926
(3, 2)
```

这些内建函数也依然只能完成“基本操作”，比如，对于数字，我们想计算三角函数的话，内建函数就帮不上忙了，于是，我们需要调用标准库（Standard Library）中的 `math` 模块（Module）：

```
import math
math.sin(5)
```

```
-0.9589242746631385
```

代码 `math.sin(5)` 这里的 `.`，也可以被理解为“操作符”，它的作用是：

从其它模块中调用函数。

代码 `math.sin(5)` 的作用是：

把 `5` 这个值，传递给 `math` 这个模块里的 `sin()` 函数，让 `sin()` 根据它内部的代码对这个值进行运算，而后返回一个值（即，计算结果）。

类（Class）中定义的函数，也可以这样被调用——虽然你还不明白类（Class）究竟是什么，但从结构上很容易理解，它实际上也是保存在其他文件中的一段代码，于是，那段代码内部定义的函数，也可以这样调用。

比如，数字，其实属于一个类，所以，我们可以调用那个类里所定义的函数，比如，`float.as_integer_ratio()`，它将返回两个值，第一个值除以第二个值，恰好等于传递给它的那个浮点数字参数：

```
3.1415926.as_integer_ratio()
```

```
(3537118815677477, 1125899906842624)
```

关于布尔值的补充

当你看到以下这样的表达式，而后再看看它的结果，你可能会多少有点迷惑：

```
True or 'Python'
```

```
True
```

这是因为 Python 将 `True` 定义为：

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object.

<https://docs.python.org/3/library/stdtypes.html#truth-value-testing>

这一段文字，初学者是看不懂的。但下一段就好理解了：

Here are most of the built-in objects considered `False` :

- constants defined to be false: `None` and `False` .
- zero of any numeric type: `0` , `0.0` , `0j` , `Decimal(0)` , `Fraction(0, 1)`
- empty sequences and collections: `''` , `()` , `[]` , `{}` , `set()` , `range(0)`

所以，`'Python'` 是个非空的字符串，即，不属于是 `empty sequences` ，所以它不被认为是 `False` ，即，它的布尔值是 `True`

于是，这么理解就轻松了：

每个变量或者常量，除了它们的值之外，同时还相当于有一个对应的布尔值。

关于值的类型的补充

除了数字、布尔值、字符串，以及上一小节介绍的列表之外，还有若干数据类型，比如 `range()` (等差数列)、`tuple` (元组)、`set` (集合)、`dictionary` (字典)，再比如 `Date Type` (日期) 等等。

它们都是基础数据类型的各种组合——现实生活中，更多需要的是把基础类型组合起来构成的数据。比如，一个通讯簿，里面是一系列字符串分别对应着若干字符串和数字。

```
entry[3662] = {
    'first_name': 'Michael',
    'last_name': 'Willington',
    'birth_day': '12/07/1992',
    'mobile': {
        '+714612234',
        '+716253923'
    }
    'id': 3662,
    ...
}
```

针对不同的类型，都有相对应的操作符，可以对其进行运算。

这些类型之间有时也有不得不相互运算的需求，于是，在相互运算之前同样要 Type Casting，比如将 List 转换为 Set，或者反之：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
a = [1, 2, 3, 4, 5, 6, 7]
b = set(a)
c = list(b)
a
b
c
```

```
[1, 2, 3, 4, 5, 6, 7]
{1, 2, 3, 4, 5, 6, 7}
[1, 2, 3, 4, 5, 6, 7]
```

总结

回到最开始：从结构上来看，一切的计算机程序，都由且只由两个最基本的成分构成：

- 运算 (Evaluation)
- 流程控制 (Control Flow)

这一章主要介绍了基础数据类型的运算细节。而除了基础数据类型，我们需要由它们组合起来的更多复杂数据类型。但无论数据的类型是什么，被操作符操作的总是该数据的值。所以，虽然绝大多数编程书籍按照惯例会讲解“数据类型”，但为了究其本质，我们在这里关注的是“值的类型”。虽然只是关注焦点上的一点点转换，但实践证明，这一点点的不同，对初学者更清楚地把握知识点有巨大的帮助。

针对每一种值的类型，无论简单复杂，都有相应的操作方式：

- 操作符
 - 值运算
 - 逻辑运算
- 函数
- 内建函数
- 其他模块里的函数
- 其本身所属类之中所定义的函数

所以，接下来要学习的，无非就是熟悉各种数据类型，及其相应的操作，包括能对它们的值进行操作的操作符和函数；无论是操作符还是函数，最终都会返回一个相应的值，及其相应的布尔值——这么看来，编程知识结构没多复杂。因为换句话讲，

接下来你要学习的无非是各种数据类型的运算而已。

另外，虽然现在尚未来得及对函数进行深入讲解，但最终你会发现它跟操作符一样，在程序里无所不在。

备注

另外，以下几个链接先放在这里，未来你会返回来参考它们，还是不断地参考它们：

- 关于表达式：<https://docs.python.org/3/reference/expressions.html>
- 关于所有操作的优先级：
<https://docs.python.org/3/reference/expressions.html#operator-precedence>
- 上一条链接不懂 BNF 的话根本读不懂：
https://en.wikipedia.org/wiki/Backus-Naur_form
- Python 的内建函数：<https://docs.python.org/3/library/functions.html>
- Python 的标准数据类型：<https://docs.python.org/3/library/stdtypes.html>

另外，其实所有的操作符，在 Python 内部也是调用函数完成的.....

<https://docs.python.org/3.7/library/operator.html>