# 入口

“速成”，对绝大多数人[1]来说，在绝大多数情况下，是不大可能的。

编程如此，自学编程更是如此。有时，遇到复杂度高一点的知识，连快速入门都不一定是很容易的事情。

所以，这一章的名称，特意从 “入门” 改成了 “入口” —— 它的作用是给你 “指一个入口”，至于你能否从那个入口进去，是你自己的事了……

不过，有一点不一样的地方，我给你指出的入口，跟别的编程入门书籍不一样 —— 它们几乎无一例外都是从一个 “Hello World!” 程序开始的…… 而我们呢？

让我们从认识一个人开始罢……

## 乔治·布尔

1833 年，一个 18 岁的英国小伙脑子里闪过一个念头：

> **逻辑关系**应该能用**符号**表示。

这个小伙子叫乔治·布尔（George Boole，其实之前就提到过我的这位偶像），于 1815 年出生于距离伦敦北部 120 英里之外的一个小镇，林肯。父亲是位对科学和数学有着浓厚兴趣的鞋匠。乔治·布尔在父亲的影响下，靠阅读自学成才。14 岁的时候就在林肯小镇名声大噪，因为他翻译了一首希腊语的诗歌并发表在本地的报纸上。

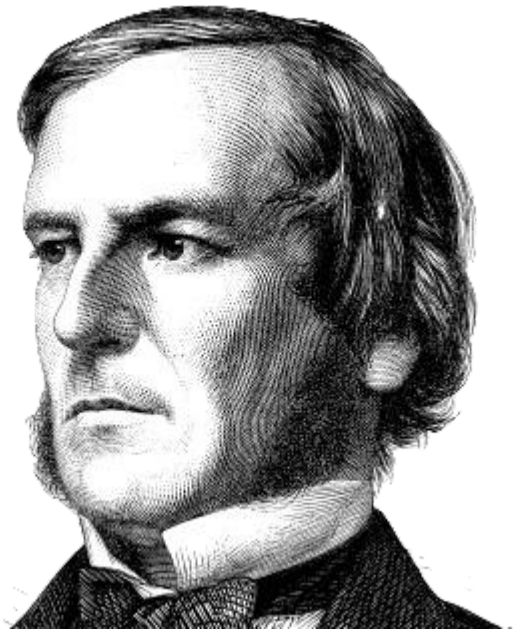
到了 16 岁的时候，他被本地一所学校聘为教师，那时候他已经在阅读微积分书籍。19 岁的时候布尔创业了 —— 他办了一所小学，自任校长兼教师。23 岁，他开始发表数学方面的论文。他发明了 “操作演算”，即，通过操作符号来研究微积分。他曾经考虑过去剑桥读大学，但后来放弃了，因为为了入学他必须放下自己的研究，还得去参加标准本科生课程。这对一个长期只靠自学成长的人来说，实在是太无法忍受了。

1847 年，乔治 32 岁，出版了他人生的第一本书籍，THE MATHEMATICAL ANAYSIS OF LOGIC —— 18 岁那年的闪念终于成型。这本书很短，只有 86 页，但最终它竟然成了人类的瑰宝。在书里，乔治·布尔很好地解释了如何使用代数形式表达逻辑思想。

1849 年，乔治·布尔 34 岁，被当年刚刚成立的女皇学院（Queen's College）聘请为第一位数学教授。随后他开始写那本最著名的书，AN INVESTIGATION OF THE LAWS OF THOUGHT。他在前言里写到：

> “The design of the following treatise is to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic and construct its method; …”
>
> “本书论述的是，探索心智推理的基本规律；用微积分的符号语言进行表达，并在此基础上建立逻辑和构建方法的科学……”

在大学任职期间，乔治·布尔写了两本教科书，一本讲微分方程，另外一本讲差分方程，而前者，A TREATISE ON DIFFERENTIAL EQUATIONS，直到今天，依然难以超越。

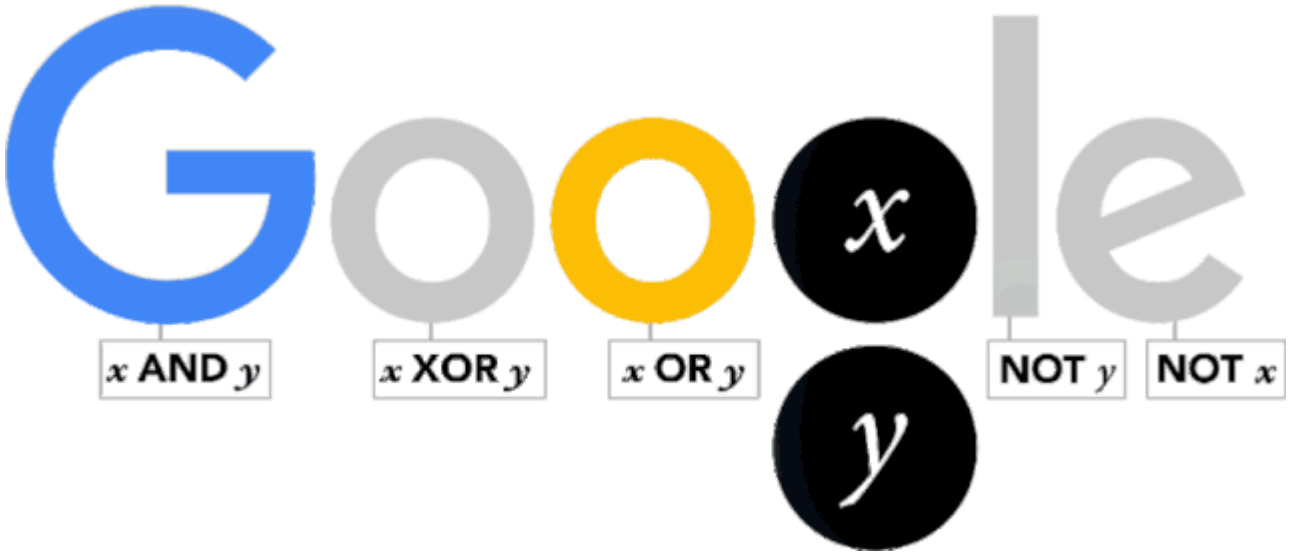


乔治·布尔于 1864 年因肺炎去世。

乔治·布尔在世的时候，人们并未对他的布尔代数产生什么兴趣。直到 70 年后，克劳德·香农（Claude Elwood Shannon）发表那篇著名论文，A SYMBOLIC ANALYSIS OF RELAY AND SWITCHING CIRCUITS 之后，布尔代数才算是开始被大规模应用到实处。

有本书可以闲暇时间翻翻，The Logician and the Engineer: How George Boole and Claude Shannon Created the Information Age。可以说，没有乔治·布尔的**布尔代数**，没有克劳德·香农的**逻辑电路**，就没有后来的计算机，就没有后来的互联网，就没有今天的信息时代 —— 世界将会怎样？

2015 年，乔治·布尔诞辰 200 周年，Google 设计了专门的 Logo 纪念这位为人类作出巨大贡献的自学奇才。




Google Doodle 的寄语是这样的：

> A very happy **11001000***th* birthday to genius George Boole!

## 布尔运算

从定义上来看，所谓**程序**（Programs）其实一点都不神秘。

因为程序这个东西，不过是按照一定_顺序_完成任务的**流程**（Procedures）。根据定义，日常生活中你做盘蛋炒饭给自己吃，也是完成了一个“做蛋炒饭”的程序 —— 你按部就班完成了一系列的步骤，最终做好了一碗蛋炒饭给自己吃 —— 从这个角度望过去，所有的菜谱都是程序……

只不过，菜谱这种程序，编写者是人，执行者还是人；而我们即将要学会写的程序，编写者是人，执行者是计算机 —— 当然，菜谱用自然语言编写，计算机程序由程序员用编程语言编写。

然而，这些都不是最重要的差异 —— 最重要的差异在于计算机能做**布尔运算**（Boolean Operations）。

于是，一旦代码编写好之后，计算机在执行的过程中，除了可以“*按照顺序执行任务*”之外，还可以“*根据不同情况执行不同的任务*”，比如，“*如果条件尚未满足则重复执行某一任务*”。

计算器和计算机都是电子设备，但计算机更为强大的原因，用通俗的说法就是它“**可编程**”（Programable）—— 而所谓可编程的核心就是_布尔运算_及其相应的**流程控制**（Control Flow）；没有布尔运算能力就没有办法做_流程控制_；没有流程控制就只能“按顺序执行”，那就显得“很不智能”……

## 布尔值

在 Python 语言中，**布尔值**（Boolean Value）用 `True` 和 `False` 来表示。

**注意**：请小心区分大小写 —— 因为 Python 解释器是大小写敏感的，对它来说，`True` 和 `true` 不是一回事。

任何一个**逻辑表达式**都会返回一个_布尔值_。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
# 请暂时忽略以上两行……

1 == 2
1 != 2
```

```
True
```

`1 == 2`，用自然语言描述就是“*1 等于 2 吗？*” —— 它的布尔值当然是 `False`。

`1 != 2`，用自然语言描述就是“*1 不等于 2 吗？*” —— 它的布尔值当然是 `True`。

**注意**：自然语言中的“*等于*”，在 Python 编程语言中，使用的符号是 `==`，**不是一个等号！**

**请再次注意**：单个等号 `=`，有其他的用处。初学者最不适应的就是，在编程语言里所使用的操作符，与他们之前在其他地方已经习惯了的使用方法并不相同 —— 不过，适应一段时间就好了。

## 逻辑操作符

Python 语言中的**逻辑操作符**（Logical Operators）如下表所示 —— 为了理解方便，也可以将其称为“*比较操作符*”。

| 比较操作符 | 意义 | 示例 | 布尔值 |
| --- | --- | --- | --- |

| 比较操作符 | 意义 | 示例 | 布尔值 |
|---|---|---|---|
| `==` | 等于 | `1 == 2` | `False` |
| `!=` | 不等于 | `1 != 2` | `True` |
| `>` | 大于 | `1 > 2` | `False` |
| `>=` | 大于等于 | `1 >= 1` | `True` |
| `<` | 小于 | `1 < 2` | `True` |
| `<=` | 小于等于 | `1 <= 2` | `True` |
| `in` | 属于 | `'a' in 'basic'` | `True` |

除了等于、大于、小于之外，Python 还有一个逻辑操作符，`in`：

这个表达式 `'a' in 'basic'` 用自然语言描述就是：

> “`'a'` 存在于 `'basic'` 这个字符串之中吗？”（属于关系）

## 布尔运算操作符

以上的例子中，逻辑操作符的**运算对象**（Operands）是数字值和字符串值。

而针对布尔值进行运算的操作符很简单，只有三种：与、或、非：

> 分别用 `and`、`or`、`not` 表示

**注意**：它们全部是小写。因为布尔值只有两个，所以布尔运算结果只有几种而已，如下图所示：

| and | True | False |
|---|---|---|
| True | True | False |
| False | False | False |

| or | True | False |
|---|---|---|
| True | True | True |
| False | True | False |

| not | True | False |
|---|---|---|
| | False | True |

先别管以下代码中 `print()` 这个函数的工作原理，现在只需要关注其中布尔运算的结果：

```
print('(True and False) yields:',  True and False)
print('(True and True) yields:',  True and True)
print('(False and True) yields:',  False and True)
print('(True or False) yields:',  True or False)
print('(False or True) yields:',  False or True)
print('(False or False) yields:',  False or False)
print('(not True) yields:',  not True)
print('(not False) yields:',  not False)
```

```
(True and False) yields: False
(True and True) yields: True
(False and True) yields: False
(True or False) yields: True
```

```
(False or True) yields: True
(False or False) yields: False
(not True) yields: False
(not False) yields: True
```

千万不要误以为布尔运算是_理科生_才必须会、才能用得上的东西..... 文理艺分科是中国的特殊分类方式，真挺害人的。比如，设计师们在计算机上创作图像的时候，也要频繁使用_或与非_的布尔运算操作才能完成各种图案的拼接..... 抽空看看这个网页： Boolean Operations used by Sketch App —— 这类设计软件，到最后是每个人都用得上的东西呢。另，难道艺术生不需要学习文科或者理科？—— 事实上，他们也有文化课.....


## 流程控制

有了布尔运算能力之后，才有_根据情况决定流程_的所谓**流程控制**（Flow Control）的能力。

```
import random
r = random.randrange(1, 1000)
# 请暂时忽略以上两句的原理，只需要了解其结果：
# 引入随机数，而后，每次执行的时候，r 的值不同

if r % 2 == 0:
    print(r, 'is even.')
else:
    print(r, 'is odd.')
```

```
693 is odd.
```

你可以多执行几次以上程序，看看每次不同的执行结果。执行方法是，选中上面的 Cell 之后按快捷键 `shift + enter`。

现在看代码，先忽略其它的部分，只看关键部分：

```
    ...
    if r % 2 == 0:
        ...
    else:
        ...
```

这个 `if/else` 语句，完成了流程的**分支**功能。 `%` 是计算余数的符号，如果 `r` 除以 `2` 的余数等于 `0`，那么它就是偶数，否则，它就是奇数 —— 写成布尔表达式，就是 `r % 2 == 0`。

这一次，你看到了单个等号 `=`：`r = random.randrange(1, 1000)`。

这个符号在绝大多数编程语言中都是“**赋值**”（Assignment）的含义。

在 `r = 2` 之中，`r` 是一个名称为 `r` 的**变量**（Variable）—— 现在只需要将变量理解为程序_保存数值的地方_；而 `=` 是赋值符号，`2` 是一个整数**常量**（Literal）。

语句 `r = 2` 用自然语言描述就是：

> “把 `2` 这个值保存到名称为 `r` 的变量之中”。

现在先别在意头两行代码的工作原理，只关注它的工作结果：`random.randrange(1, 1000)` 这部分代码的作用是返回一个 *1 到 1000 之间*（含左侧 `1` 但不含右侧 `1000`）的随机整数。每次执行以上的程序，它就生成一个新的随机整数，然后因为 `=` 的存在，这个数就被保存到 `r` 这个变量之中。

计算机程序的所谓“智能”（起码相对于计算器），首先是因为它能做_布尔运算_。计算机的另外一个好处是“不知疲倦”（反正它也不用自己交电费），所以，它最擅长处理的就是“重复”，这个词在程序语言中，术语是**循环**（Loop）。以下程序会打印出 `10` 以内的所有奇数：

```
for i in range(10):
    if i % 2 != 0:
        print(i)
```

```
1
3
5
7
9
```

其中 `range(10)` 的返回值，是 `0`～`9` 的整数序列（默认起始值是 `0`；含左侧 `0`，不含右侧 `10`）。

用自然语言描述以上的程序，大概是这样的 —— 自然语言写在 `#` 之后：

```
for i in range(10): # 对于 0～9 中的所有数字都带入 i 这个变量，执行一遍以下任务：
    if i % 2 != 0:  #     如果 i 除以 2 的余数不等于零的话，执行下面的语句：
        print(i)    #         向屏幕输出 i 这个变量中所保存的值
```

就算你让它打印出一百亿以内的奇数，它也毫不含糊 —— 你只需要在 `range()` 这个函数的括号里写上一个那么大的整数就行……

让它干一点稍微复杂的事吧，比如，我们想要打印出 `100` 以内所有的_质数_（Primes）。

根据质数的定义，它大于等于 2，且只有在被它自身或者 1 做为除数时余数为 0。判断一个数字是否是质数的**算法**是这样的：

> - 设 n 为整数，n >= 2；
> - 若 n == 2，n 是质数；
> - 若 n > 2，就把 n 作为被除数，从 2 开始一直到 n - 1 都作为除数，_逐一_计算看看余数是否等于 0？
>
>> - 如果是，那就不用接着算了，它不是质数；
> - 如果全部都试过了，余数都不是 0，那么它是质数。

于是，你需要两个**嵌套**的循环，第一个负责作为被除数 n 从 2 到 99（题目是 100 以内，所以不包含 100）的循环；而这内部，需要另外一个作为除数负责 i 从 2 到 n 的循环：

```
for n in range(2, 100): #range(2,100)表示含左侧 2，不含右侧 100，是不是第三次看到这个说法了？
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
            break
    else:                   # 这里目前你可能看不懂…… 但先关注结果吧。
        print(n)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
```

```
89
97
```

## 所谓算法

以上的**算法**可以改进（程序员们经常用的词汇是“*优化*”）：

> 从 2 作为除数开始试，试到 $\sqrt{n}$ 之后的一个整数就可以了……

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, int(n ** 0.5)+1): #为什么要 +1 以后再说…… n 的 1/2 次方，相当于根号 n。
        if (n % i) == 0:
            break
    else:
        print(n)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

你看，寻找更有效的算法，或者说，不断优化程序，提高效率，最终是程序员的工作，不是编程语言本身的工作。关于判断质数最快的算法，可以看 Stackoverflow 上的讨论，有更多时间也可以翻翻 Wikipedia。

到最后，**所有的工具都一样，效用取决于使用它的人**。所以，学会使用工具固然重要，更为重要的是与此同时自己的能力必须不断提高。

虽然写代码这事刚开始学起来好像门槛很高，那只不过是幻觉，其实门槛比它更高的多的去了。到最后，它就是个最基础的工具，还是得靠思考能力 —— 这就好像识字其实挺难的 —— 小学初中高中加起来十来年，我们才掌握了基本的阅读能力；可最终，即便是本科毕业、研究生毕业，真的能写出一手好文章的人还是少之又少一样 —— 因为用文字值得写出来的是思想，用代码值得写出来的是创造，或者起码是有意义的问题的有效解决方案。有思想，能解决问题，是另外一门手艺 —— 需要终生精进的手艺。

## 所谓函数

我们已经反复见过 `print()` 这个**函数**（Functions）了。它的作用很简单，就是把传递给它的值输出到屏幕上 —— 当然，事实上它的使用细节也很多，以后慢慢讲。

现在，最重要的是初步理解一个函数的基本构成。关于**函数**，相关的概念有：*函数名*（Function Name）、*参数*（Parameters）、*返回值*（Return Value）、*调用*（Call）。

拿一个更为简单的函数作为例子，`abs()`。它的作用很简单：接收一个数字作为参数，经过运算，返回该数字的绝对值。

```
a = abs(-3.1415926)
a
```

```
3.1415926
```

在以上的代码的第 1 行中，

- 我们_调用_了一个_函数名_为 `abs` 的函数；写法是 `abs(-3.1415926)`；
- 这么写，就相当于向它_传递_了一个_参数_，其值为： `-3.1415926`；
- 该函数接收到这个参数之后，根据这个参数的_值_在函数内部进行了*运算*；
- 而后该函数返回了一个值，_返回值_为之前接收到的参数的值的绝对值 `3.1415926`；
- 而后这个_值_被保存到*变量* `a` 之中。

从结构上来看，每个函数都是一个完整的程序，因为一个程序，核心构成部分就是_输入_、*处理*、*输出*：

- 它有输入 —— 即，它能接收外部通过参数传递的值；
- 它有处理 —— 即，内部有能够完成某一特定任务的代码；尤其是，它可以根据 “输入” 得到 “输出”；
- 它有输出 —— 即，它能向外部输送返回值……

被调用的函数，也可以被理解为**子程序**（Sub-Program） —— 主程序执行到函数调用时，就开始执行实现函数的那些代码，而后再返回主程序……

我们可以把判断一个数字是否是质数的过程，写成函数，以便将来在多处用得着的时候，随时可以调用它：

```
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
```

# 细节补充

## 语句

一个完整的程序，由一个或者多个**语句**（Statements）构成。通常情况下，建议每一行只写一条语句。

```
for i in range(10):
    if i % 2 != 0:
        print(i)
```

```
1
3
5
7
9
```

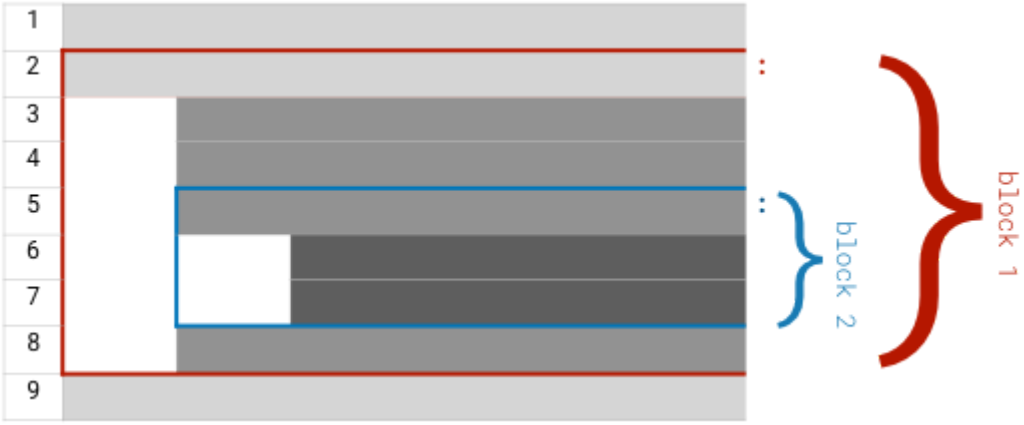
## 语句块

在 Python 语言中，**行首空白**（Leading whitespace，由空格 ' ' 或者 Tab ⇥ 构成）有着特殊的含义。

如果有行首空白存在，那么，Python 将认为这一行与其他邻近有着相同行首空白的语句同属于一个**语句块** —— 而一个语句块必然由一个行末带有冒号 `:` 的语句起始。同属于一个语句块中的语句，行首空白数量应该相等。这看起来很麻烦，可实际上，程序员一般都使用专门的文本编辑器，比如 Visual Studio Code，其中有很多的辅助工具，可以让你很方便地输入具备一致性的行首空白。

以上程序，一共三个语句，两个语句块，一个 `for` 循环_语句块_中包含着一个 `if` 条件_语句块_。注意第一行和第二行末尾的冒号 `:`。

在很多其他的语言中，比如，JavaScript，用大括号 `{}` 作为语句块标示 —— 这是 Python 比较特殊的地方，它组织语句块的方式如下图所示：




> **注意**：在同一个文件里，不建议混合使用 Tab 和 Space；要么全用空格，要么全用制表符。

## 注释

在 Python 程序中可以用 `#` 符号标示**注释**语句。

所谓的注释语句，就是程序文件里写给人看而不是写给计算机看的部分。本节中的代码里就带着很多的注释。

人写的 Python 语言代码，要被 Python **解释器**翻译成机器语言，而后才能让计算机 “读懂”，随后计算机才可以按照指令执行。解释器在编译程序的过程中，遇到 `#` 符号，就会忽略其后的部分（包括这个注释符号）。

## 操作符

在本节，我们见到的比较操作符可以比较它左右的值，而后返回一个布尔值。

我们也见过两个整数被**操作符** `%` 连接，左侧作为被除数，右侧作为除数，`11 % 3` 这个表达式的值是 `2`。对于数字，我们可用的操作符有 `+`、`-`、`*`、`/`、`//`、`%`、`**` —— 它们分别代表加、减、乘、除、商、余、幂。

## 赋值符号与操作符的连用

你已经知道变量是什么了，也已经知道赋值是什么了。于是，你看到 `x = 1` 就明白了，这是为 `x` 赋值，把 `1` 这个值保存到变量 `x` 之中去。

但是，若是你看到 `x += 1`，就迷惑了，这是什么意思呢？

这只是编程语言中的一种惯用法。它相当于 `x = x + 1`。

看到 `x = x + 1` 依然会困惑..... 之所以困惑，是因为你还没有习惯把单等号 `=` 当作赋值符号，把双等号 `==` 当作逻辑判断的 “等于”。

`x = x + 1` 的意思是说，把表达式 `x + 1` 的值保存到变量 `x` 中去 —— 如此这般之后，`x` 这个变量中所保存的就不再是原来的值了……

```
x = 0
x += 1
print(x)
```

```
1
```

其实不难理解，只要习惯了就好。理论上，加减乘除商余幂这些操作符，都可以与赋值符号并用。

```
x = 11
x %= 3   # x = x % 3
print(x)
```

```
2
```

## 总结

以下是这一章中所提到的重要概念。了解它们以及它们之间的关系，是进行下一步的基础。

> - 数据：整数、布尔值；操作符；变量、赋值；表达式
> - 函数、子程序、参数、返回值、调用
> - 流程控制、分支、循环
> - 算法、优化
> - 程序：语句、注释、语句块
> - 输入、处理、输出
> - 解释器

你可能已经注意到了，这一章的小节名称罗列出来的话，看起来像是一本编程书籍的目录 —— 只不过是概念讲解顺序不同而已。事实上还真的就是那么回事。

这些概念，基本上都是**独立于**某一种编程语言的（Language Independent），无论将来你学习哪一种编程语言，不管是 C++，还是 JavaScript，抑或是 Golang，这些概念都在那里。

学会一门编程语言之后，再学其它的就会容易很多 —— 而且，当你学会了其中一个之后，早晚你会顺手学其它的，为了更高效使用微软办公套件，你可能会花上一两天时间研究一下 VBA；为了给自己做个网页什么的，你会顺手学会 JavaScript；为了修改某个编辑器插件，你发现人家是用 Ruby 写的，大致读读官方文档，你就可以下手用 Ruby 语言了；为了搞搞数据可视化，你会发现不学会 R 语言有点不方便……

你把这些概念装在脑子里，而后就会发现几乎所有的编程入门教学书籍结构都差不多是由这些概念构成的。因为，所有的编程语言基础都一样，所有的编程语言都是我们指挥计算机的工具。无论怎样，反正都需要输入输

出，无论什么语言，不可能没有布尔运算，不可能没有流程控制，不可能没有函数，只要是高级语言，就都需要编译器..... 所以，掌握这些基本概念，是将来持续学习的基础。

---

脚注

[1]：对于自学能力强、有很多自学经验的人来说，速成往往真的是可能、可行的。因为他们已经积累的知识与经验会在习得新技能时发挥巨大的作用，乃至于他们看起来相对别人花极少的时间就能完成整个自学任务。也就是说，将来的那个已经习得自学能力、且自学能力已经磨练得很强的你，常常真的可以做到在别人眼里 “速成”。

↑Back to Content↑