1. 概述

本手册仅描述 Python 编程语言，不宜当作教程。

我希望尽可能地保证内容精确无误，但还是选择使用自然词句进行描述，正式的规格定义仅用于句法和词法解析。这样应该能使文档对于普通人来说更易理解，但也可能导致一些歧义。因此，如果你是来自火星并且想凭借这份文档把 Python 重新实现一遍，也许有时需要自行猜测，实际上最终大概会得到一个十分不同的语言。而在另一方面，如果你正在使用 Python 并且想了解有关该语言特定领域的精确规则，你应该能够在这里找到它们。如果你希望查看对该语言更正式的定义，也许你可以花些时间自己写上一份 --- 或者发明一台克隆机器 :-)

在语言参考文档里加入过多的实现细节是很危险的 --- 具体实现可能发生改变，对同一语言的其他实现可能使用不同的方式。而在另一方面，CPython 是得到广泛使用的 Python 实现 (然而其他一些实现的拥护者也在增加)，其中的特殊细节有时也值得一提，特别是当其实现方式导致额外的限制时。因此，你会发现在正文里不时会跳出来一些简短的 "实现注释"。

每种 Python 实现都带有一些内置和标准的模块。相关的文档可参见 Python 标准库 索引。少数内置模块也会在此提及，如果它们同语言描述存在明显的关联。

1.1. 其他实现

虽然官方 Python 实现差不多得到最广泛的欢迎，但也有一些其他实现对特定领域的用户来说更具吸引力。

知名的实现包括:

CPython
: 这是最早出现并持续维护的 Python 实现，以 C 语言编写。新的语言特性通常在此率先添加。

Jython
: 以 Java 语言编写的 Python 实现。 此实现可以作为 Java 应用的一个脚本语言，或者可以用来创建需要 Java 类库支持的应用。 想了解更多信息请访问 Jython 网站。

Python for .NET
: 此实现实际上使用了 CPython 实现，但是属于 .NET 托管应用并且可以引入 .NET 类库。它的创造者是 Brian Lloyd。想了解详情可访问 Python for .NET 主页。

IronPython
: 另一个 .NET 版 Python 实现，不同于 Python.NET，这是一个生成 IL 的完整 Python 实现，并会将 Python 代码直接编译为 .NET 程序集。 它的创造者就是当初创造 Jython 的 Jim Hugunin。想了解更多信息，请参看 IronPython 网站。

PyPy

一个完全使用 Python 语言编写的 Python 实现。 它支持多个其他实现所没有的高级特性，例如非栈式支持和实时编译器等。 此项目的目标之一是通过允许方便地修改解释器（因为它是用 Python 编写的）来鼓励对语言本身的试验。 更多信息可在 PyPy 项目主页 获取。

以上这些实现都可能在某些方面与此参考文档手册的描述有所差异，或是引入了超出标准 Python 文档范围的特定信息。请参考它们各自的专门文档，以确定你正在使用的这个实现有哪些你需要了解的东西。

1.2. 标注

词法分析和语法的描述采用混合了 EBNF 与 PEG 的语法标记形式。 例如：

```
name:   letter (letter | digit | "_")*
letter: "a"..."z" | "A"..."Z"
digit:  "0"..."9"
```

在这个示例中，第一行是说 `name` 是一个 `letter` 后面跟由零个或多个 `letter`, `digit` 和下划线组成的序列。 而 `letter` 则是从 `'a'` 到 `'z'` 以及从 `A` 到 `Z` 的任意单个字符；`digit` 是从 `0` 到 `9` 的任意单个字符。

每条规则以一个名称打头（它用来标识所定义的规则）后面跟一个冒号 `:`。 冒号右边的定义使用下列语法元素：

- `name`: 一个指向其他规则的名称。 如果可能，它将是一个指向规则定义的链接。
  - `TOKEN`: 一个指向特定 token 的大写形式的名称。 对于语法定义的场景，token 与规则是一回事。
- `"text"`, `'text'`: 单引号或双引号内的文本必须在字面上匹配（不带引号）。 引号的类型将根据 `text` 的含义来选择：
  - `'if'`: 单引号内的名称标记了一个 关键字。
  - `"case"`: 双引号内的名称表示一个 软关键字。
  - `'@'`: 单引号中的一个非字母符号表示一个 `OP` 记号，即 定界符 或 运算符。
- `e1 e2`: 仅用空格分隔的项表示一个序列。在这里，`e1``后面必须跟着``e2`。
- `e1 | e2`: 竖条用于分隔选项。它表示 PEG 的“有序选择”: 如果匹配 `e1`，则不考虑 `e2`。 在传统的 PEG 语法中，它被写成斜杠 `/`，而不是竖条。 有关更多背景和详细信息，请参阅 **PEP 617**。
- `e*`: 星号表示前一项重复零次或多次。
- `e+`: 同样，加号表示一次或多次重复。
- `[e]`: 用方括号括起来的短语表示出现零次或一次。换句话说，所包含的短语是可选的。
- `e?`: 问号与方括号的含义完全相同：前一项为可选项。
- `(e)`: 括号用于分组。

以下符号表示法仅在 词法定义 中使用。

- `"a"..."z"`: 由三个点分隔的两个字面值字符表示在给定（包括）ASCII 字符范围内选择任何单个字符。
- `<...>`: 尖括号之间的短语给出了匹配符号的非正式描述（例如，`<any ASCII character except "\">`），或者在附近文本中定义的缩写（例如，`<Lu>`）。

部分定义还使用了*前瞻断言*（lookaheads），这类断言用于指示某个元素必须在（或不能在）特定位置匹配，但不会消耗任何输入内容。

- `&e`：正向肯定前瞻断言（即要求必须匹配 `e`）
- `!e`：负向否定前瞻断言（即要求 `e` **必须不匹配**）

一元运算符（`*`、`+`、`?`）尽可能紧密地绑定；竖条（`|`）绑定最松散。

空格只对分隔记号有意义。

规则通常包含在一行中，但太长的规则可能会被换行：

```
literal: stringliteral | bytesliteral
         | integer | floatnumber | imagnumber
```

或者，规则可以格式化为第一行以冒号结束，其余每一行以竖线开始。例如：

```
literal:
   | stringliteral
   | bytesliteral
   | integer
   | floatnumber
   | imagnumber
```

这并不意味着第一个选项是空的。

1.2.1. 词法和语法定义

词法分析 和 *语法分析* 之间有一些区别: lexical analyzer 对输入源的单个字符进行操作，而 *解析器* (语法分析器) 对词法分析生成的 词元 流进行操作。 然而，在某些情况下，这两个阶段之间的确切界限是 CPython 的实现细节。

两者之间的实际区别在于，在 *词法* 定义中，所有空白符都是重要的。 词法分析器会 丢弃 所有未转换为 `token.INDENT` 或 `NEWLINE` 等记号的空白符。 *语法* 定义随后使用这些记号，而不是源字符。

本文档对两种定义样式使用相同的BNF语法。下一章（词法分析）中BNF的所有用法都是词法上的定义；后面章节中的用法是语法定义。