# 第3章. 数据模型

上一章讨论了如何将字符流转化为令牌序列，并识别其中的令牌。这就好像你听到一段话，或者看到一段文字，提取并识别出其中的词汇一样。从本章开始直到第15章，讨论的都是如何理解这些令牌形成的语句的含义，也就是所谓的“语义分析”。这比词法分析要高一个维度，就好像你领悟识别出的词汇构成的语句的含义一样。本章讨论Python通过其数据模型实现的抽象层次，以为后续各章打好基础。

## 3-1. 一切皆对象

（语言参考手册：3.1）
（标准库：内置函数）

发明了编程语言Algol W、Pascal和Modula，提出了“结构化程序设计（structured programming）”思想的图灵奖得主尼古拉斯·沃斯（Nicklaus Wirth）于1976年写了一本名为《算法 + 数据结构 = 程序》的书。该书名中的著名公式反映了中级语言设计者对于程序的理解：数据结构是通过数据体现的，算法是通过代码体现的，而用代码处理数据的过程就是将算法应用于数据结构的过程。然而高级语言设计者所追求的抽象层次是“一切皆对象”，程序的本质是对象间通过属性实现的相互联系，以及通过属性访问实现的信息传递，因此上述公式对高级语言已经不再适用。

解释器在解释Python脚本时，每遇到一条函数定义语句或类定义语句，就会在其内存空间中创建相应的“函数对象（function objects）”或“类对象（class objects）”。Python脚本的执行最终会演变为对函数对象的调用，而函数体对应的代码块经伪编译后会转换成内存空间中的“代码对象（code objects）”。综上所述，在Python解释器眼中，代码和数据都是储存在内存空间中的二进制序列，没有本质区别；而所谓的“对象”就是对这些二进制序列的抽象。

Python中的每个对象都可以通过如下三方面来描述：“编号（identity）”、“类型（type）”和“值（value）”。对象的编号是一个具有全局唯一性的非负整数（但生存期不重叠的对象的编号可能相同），一旦被创建就不会改变。在CPython中对象的编号就是与其对应的二进制序列的首字节在内存空间中的地址，亦即该对象的引用。其他Python解释器可以用别的方法来实现对象的编号，只需满足具有全局唯一性的非负整数这一条件。

可以通过内置函数id()取得对象的编号，其语法为：

```
id(object)
```

其中object参数代表一个对象，而该函数的返回值即代表该对象编号的非负整数。请多次以交互模式启动Python解释器，每次都执行如下语句：

```
>>> id(0)
4522590416
>>> id("abc")
4523876784
>>> id(0)
4522590416
>>> id("abc")
4523876784
>>>
```

该例子说明了在Python中数字和字符串都是对象。此外还可以观察到如下现象：
1. 每次启动Python解释器，id(0)和id("abc")的输出都不相同：这是因为每次解释器启动都在内存空间的随机位置储存0和“abc”。
2. Python解释器启动后，多次执行id(0)和id("abc")的输出是固定不变的：这是因为在Python脚本中多次出现的相同数字字面值或字符串字面值会指向同一对象，而对象的编号在其生存期内不会改变。

同样，对象一旦被创建其类型就不会改变。类型本身在内存空间中是用类对象表示的，而属于该类型的对象会储存指向相应类对象的引用。类对象本身默认储存指向内置函数type()的引用：type是Python解释器内置的唯一一个“元类（metaclasses）”。而type引用的类对象就是它自身，这就保证了每个对象都有自己的类型。在本章后面会说明，这种引用是通过特殊属性__class__实现的。

可以通过内置函数type()取得对象的类型，该内置函数用于该目的时语法为：

```
class type(object)
```

其中object参数代表一个对象，而该函数的返回值是该对象引用的类对象。注意类对象被输出时会自动转换为“<class 'xxx'>”格式的字符串，其中xxx为类名。请执行如下语句：

```
>>> type(0)
<class 'int'>
>>> type("abc")
<class 'str'>
>>> type(type)
<class 'type'>
>>>
```

Python官方手册没有说明什么是对象的值。我们可以模糊地将值理解成存储在对象中的数据，这些数据需要通过该对象引用的类对象的属性来访问。对象的类型决定了该对象可以取哪些值，以及可以将哪些操作施加于该值。有些类型支持多个取值（例如数字类型和字符串类型）；有些类型仅支持一个取值（例如None的类型）；有些类型的值是对其他对象的引用（例如元组类型和列表类型）；有些类型不支持值（例如用户自定义类）。事实上，只有内置类型和部分扩展模块中定义的类型支持值。

Python中一个较难理解的概念是对象的“可变性（mutability）”。为了便于说明，下面采用如下模型来想象对象在内存空间中的存储方式：每个对象对应的二进制序列开头一个字长（64位或32位）的部分储存着对相应类对象的引用，而后续部分用于储存该对象的值。显然，该二进制序列指向类对象的引用部分是不能改变的，如果它的其余部分（代表该对象的值）可以改变，则称该类型的对象是“可变的（mutable）”；否则，称该类型的对象是“不可变的（immutable）”。（注意，该模型假设对象除了__class__属性外没有其他属性，这是为了便于说明可变性而进行的简化。）

图3-1显示了内存空间中包含的4种类型的对象：数字0和1、字符串“abc”、元组（"abc", 0）和列表["abc", 1]（从["abc", 0]变化而来）。图中的蓝色小矩形代表的是对象对相应类对象的引用（类对象没有画出），而其右边的每个正方形都代表一个数据单元（可以是一个数字、一个字符或一个引用）。

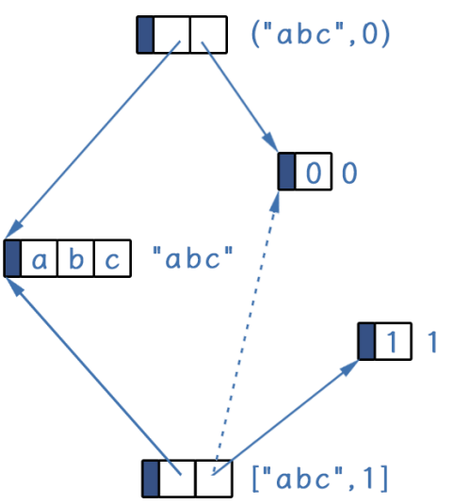


图3-1. 对象的可变性

从图3-1可以看出，数字对象或字符串对象的值就是数字或字符序列本身，它们一旦被创建所包含的数字或字符序列就不能改变，因此数字和字符串都是不可变的。元组对象的值储存着对其他对象的引用，它们一旦被创建所包含的引用也不能改变，因此元组也是不可变的。列表对象的值同样储存着对其他对象的引用，然而它们被创建后所包含的引用可以改变，因此列表是可变的。

需要强调的是，虽然元组的值包含的引用是不能改变的，但这些引用可以指向列表，后者的值改变后，虽然元组本身的值没有发生变化，但通过该元组访问到的列表的值依然与之前不同。这说明一个对象不可变并不意味着它在程序运行过程中每次被访问的结果都相同。

## 3-2. 名字的绑定

（教程：5.2、9.1）
（语言参考手册：3.1、3.2、4.2.1、6.2.1、6.2.2、6.12、6.16、7.5）

储存在内存空间中的对象必须通过绑定名字才能被Python脚本访问。所谓的“名字（names）”其实就是标识符。在第2章提到过，标识符是变量、常量、函数、类和模块的名字。但由于函数、类和模块本质上都是对象，而变量和常量则可以绑定任意对象，所以Python中的标识符可以统一被视为对象的名字：它被绑定到哪个对象，就成为该对象的一个名字。

在内存空间中，任意时刻类型和值都相同的对象只能存在一个，但属于同一类型但不具有值的对象可以有任意多个。一个对象可以被绑定到任意多个标识符，也可能不被任意标识符绑定但被其他对象引用。当一个对象的引用数为0时（既没有绑定到某个标识符也没有被任何其他对象引用），会被Python解释器通过“垃圾回收（garbage collection）”机制销毁以节省内存。但由于垃圾回收操作是周期性进行的，所以引用数变为0的对象不会立即被销毁，而会存在到下次垃圾回收操作被执行，这段时间内它虽然存在却不能被访问。（这是所有使用了垃圾回收机制自动管理内存的编程语言的共性。）

标识符本身是储存在“变量字典（variable dictionary）”中的，它本质上是一个字典：标识符是变量字典的键，而标识符和对象的绑定本质上是变量字典的键引用了对象。由于字典是可变的，所以标识符和对象的绑定永远是可以改变的，与对象本身的可变性无关。导致一个标识符被绑定到某个对象（新标识符将同时被创建）的操作包括如下7种：

- 赋值语句。
- 赋值表达式。
- 函数定义。
- 类定义。
- 模块的导入。
- 函数被调用时的参数传递。
- try语句中的except子句、with语句、for语句和match语句。

而导致一个标识符到某个对象的绑定被解除的操作则包括如下2种：

- del语句。
- 函数被调用后的返回。

本节仅通过讨论赋值语句、赋值表达式和del语句来阐述标识符和对象的绑定，对其他操作的讨论将放在后续章节中。

最基本的赋值语句的语法可以概括为：

```
identifier = expression
```

其含义是对“=”右边的表达式求值，将得到的结果（肯定是一个对象）与“=”左边的标识符绑定。“=”两端的空白符不是必须的，但PEP 8推荐总是在赋值语句“=”两端各添加一个空格以提高可读性。最简单的表达式就是单独的一个字面值，它代表具有该值的数字对象或字符串对象。请通过如下例子验证最基本的赋值语句的用法：

```
>>> id(0)
4388962512
>>> a = 0
>>> id(a)
4388962512
>>> b = 0
>>> id(b)
4388962512
>>> c = a
>>> id(c)
4388962512
>>>
```

注意在上面例子中id(0)、id(a)、id(b)和id(c)的返回值是相同的（不一定是上面给出的非负整数），这证明标识符a、b和c都被绑定到了数字字面值0对应的对象上。此外，该例子还说明了两点：

1. 同样的字面值对应同一对象，因此赋值语句“a = 0”和“b = 0”将标识符a和b绑定到了同一对象，即数字0。
2. 一个已经绑定对象的标识符作为表达式被求值时将得到被绑定的对象，所以赋值语句“c = a”背后的逻辑是：先对表达式“a”求值，得到a被绑定的对象，即数字0，然后将标识符c也绑定到该对象。

赋值语句只能完成标识符和对象的绑定，得不到任何值，因此只能作为一条独立的语句使用。而赋值表达式则在完成标识符和对象绑定的同时还会得到被绑定的对象，因此可以嵌入其他语句来使用。（这与很多其他编程语言不同，例如在C中没有专门的赋值语句，赋值表达式单独使用时就被视为赋值语句。）赋值表达式的语法为：

```
identifier:=expression
```

其中“:=”又被称为“海象运算符”（因为它的形状像海象的脸）。注意“:=”两端也可以有空白符，但这会降低可读性。下面这个例子说明了赋值表达式的作用：

```
>>> t = (s:='abc', 0)
>>> l = [s, 0]
>>> l[1] = 1
>>> s
'abc'
>>> t
('abc', 0)
>>> l
['abc', 1]
>>>
```

该例子将赋值表达式嵌入赋值语句来使用，使得第一条赋值语句同时绑定了标识符t和s。图3-2显示了执行上述赋值语句之后标识符和对象之间的绑定关系。（该图中的灰色矩形代表的是储存有标识符t、s和l的变量字典。）

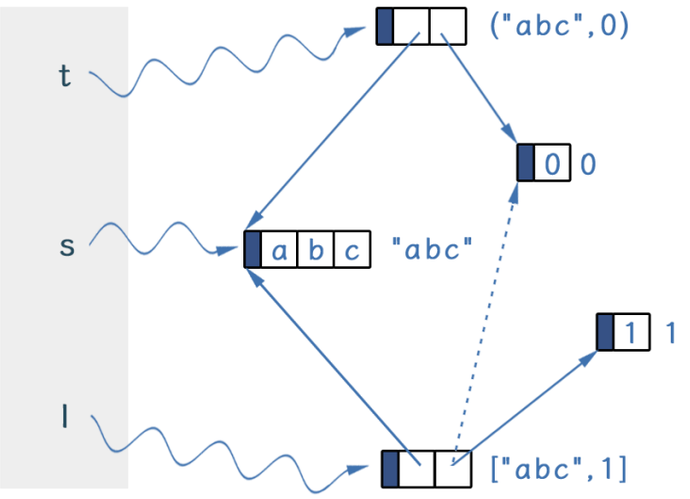


图3-2. 标识符和对象的绑定

表2-2列出的表达式中，只有赋值表达式对作为其操作数的子表达式的求值顺序是从右向左的，所有其他表达式都会按照从左向右的顺序依次对作为其操作数的子表达式求值。

最基本的del语句的语法可以概括为：

```
del identifier
```

注意这会删除指定的标识符，但并不一定会删除被绑定的对象（只会导致该对象的引用数减1）。下面的例子说明了这点：

```
>>> h = 'Hello World!'
>>> w = h
>>> del h
>>> h
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'h' is not defined
>>> w
'Hello World!'
>>>
```

## 3-3. 对象的属性

（语言参考手册：3.2、3.3.3、6.3.1）
（标准库：内置类型、内置函数）

本章开头已经说明，对于面向对象语言来说程序的本质是对象间的相互联系和信息传递。而对象间的相互联系和信息传递又是如何实现的呢？一言以蔽之，对象之间通过属性的创建产生了相互联系，通过属性的访问实现了信息传递。因此，理解面向对象语言内在逻辑的关键点就在于理解“属性（attributes）”这个概念。

**首先**，让我们讨论属性的访问。

定位到一个对象后，可以通过如下语法访问该对象的属性：

```
object.attribute
```

其中object是绑定到该对象的标识符，而attribute是绑定到该对象的属性的标识符（也称为“属性名”）。一个对象的属性必然引用某个对象，因此这种访问是可以递归的，即可以写成：

```
object.attribute.attribute...
```

属性和变量一样，可以引用任意类型的对象。事实上，属性其实就是它所属对象的名字空间中的变量（这会在第6章详细说明），因此属性和变量是可以互换的（属性名本质上就是变量名）。

**接下来**讨论属性的本质。我们已经知道Python中的每个对象都有自己的类型，而这是通过让对象的__class__属性引用相应的类对象实现的。事实上，__class__是Python对象最特殊的属性，每个对象都必然具有。__class__实现了一个对象和它所属类之间的这样一种关系：对象是它所属类的一个“实例（instances）”。

由于类本身也是对象，即类对象，所以每个类对象也都需要通过__class__引用它所属的类，后者就是所谓的“元类”。从逻辑上讲，元类本身也是某种对象，也需要属于某种类，而这一过程可以递归下去直至无穷。为了避免这种递归，Python规定元类本身也是一种特殊的类，因此所属的类也是元类。

前面提到过type是Python解释器内置的唯一元类，而我们还可以自定义其他元类（会在第15章详细讨论）。不考虑自定义元类，假设所有类都以type作为元类，那么Python对象之间的“实例”关系可以用图3-3表示，即箭头尾部的对象是箭头头部的对象的实例。

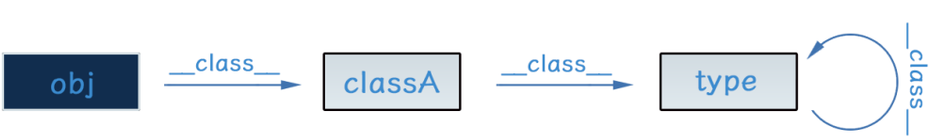


图3-3. 对象间的“实例”关系

目前我们已经接触到了下列类型的对象：数字、字符串、元组、列表以及内置函数id()和type()。下面的例子检查所有这些对象的__class__属性：

```
>>> (0).__class__
<class 'int'>
>>> "abc".__class__
<class 'str'>
>>> ("abc",0).__class__
<class 'tuple'>
>>> ["abc",1].__class__
<class 'list'>
>>> id.__class__
<class 'builtin_function_or_method'>
>>> type.__class__
<class 'type'>
>>>
```

可以看出__class__永远引用一个类对象，而type的__class__引用它自身。需要注意，整数字面值必须放在括号中才能访问其属性，否则会报错：

```
>>> 0.__class__
  File "<stdin>", line 1
    0.__class__
     ^
SyntaxError: invalid decimal literal
>>>
```

这是因为Python解释器无法分辨整数字面值末尾的“.”到底是对属性的引用还是小数点。对于浮点数字面值来说就不存在这一问题：

```
>>> 0.0.__class__
<class 'float'>
>>>
```

此外，直接访问一个对象的__class__属性，与将该对象作为参数传递给type()，得到的结果是相同的。换句话说，type()的功能其实就是访问指定对象的__class__属性。

下面围绕图3-3中的模型阐述对象属性的来源。首先考虑从类对象实例化出对象的场景。一个对象的属性有三种：

- 该对象引用的类对象的属性，被所有属于该类的对象所共有的，称为“类属性”。
- 该对象本身的属性，与其引用的类对象无关，被称为“实例属性”。
- 既不是类属性又不是实例属性的属性，被称为“特殊属性”。

这三种属性在被访问时的语法没有区别，但实现的方式却截然不同。

类属性是这样实现的：当一个对象被创建时，相关类对象的__new__属性会被调用，使得该对象的__class__属性被设置为引用该类对象，因此该对象也自动具有了该类对象的所有实例属性。（换句话说，一个类对象的实例属性，就是所有属于该类的对象的类属性。）

实例属性的实现方式则要复杂得多。最一般的情况是，在相关类对象的__new__属性被调用时会自动创建一个空字典作为该对象的变量字典，并添加特殊属性__dict__以引用该变量字典。在这之后，相关类对象的__init__属性会被调用，而在其被执行的过程中可以给变量字典添加键值对，这样就初始化了该对象的实例属性。而由于变量字典是可变的，所以在该对象被创建之后依然可以动态地添加实例属性。下面通过一个例子说明这种最一般的情况。首先编写脚本Point2D.py（内置函数print()的作用是向标准输出写入字符串，将在第7章详细讨论）：

```
class Point2D:
    #类属性dimension。
    dimension = 2

    #在初始化时设置了实例属性x和y。
    def __init__(self, x, y):
        self.x = x
        self.y = y

    #类属性move。
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    #类属性location。
    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")
```

然后通过如下命令行执行该脚本，并在脚本执行完成后进入交互式状态：

```
$ python3 -i Point2D.py
```

接下来先创建一个点p1，访问p1的属性，并检查其特殊属性__class__和__dict__引用的对象：

```
>>> p1 = Point2D(3.2, 0.0)
>>> p1.location()
(3.2, 0.0)
>>> p1.move(3.4, -6.9)
>>> p1.location()
(6.6, -6.9)
>>> p1.dimension
2
>>> p1.x
6.6
>>> p1.y
-6.9
>>> p1.__class__
<class '__main__.Point2D'>
>>> p1.__dict__
```

```
{'x': 6.6, 'y': -6.9}
>>>
```

再创建一个点p2，访问p2的属性，并检查其特殊属性__class__和__dict__引用的对象：

```
>>> p2 = Point2D(10, 10)
>>> p2.location()
(10, 10)
>>> p2.move(-1, 0.1)
>>> p2.location()
(9, 10.1)
>>> p2.dimension
2
>>> p2.x
9
>>> p2.y
10.1
>>> p2.__class__
<class '__main__.Point2D'>
>>> p2.__dict__
{'x': 9, 'y': 10.1}
>>>
```

下面在p1和p2被创建后分别给它们添加一个radius属性：

```
>>> import math
>>> p1.radius = math.sqrt(p1.x**2 + p1.y**2)
>>> p1.radius
9.54829827770373
>>> p1.__dict__
{'x': 6.6, 'y': -6.9, 'radius': 9.54829827770373}
>>> p2.radius = math.sqrt(p2.x**2 + p2.y**2)
>>> p2.radius
13.528118864054973
>>> p2.__dict__
{'x': 9, 'y': 10.1, 'radius': 13.528118864054973}
>>>
```

最后验证类属性和实例属性的区别：

```
>>> id(p1.dimension) == id(p2.dimension)
True
>>> id(p1.move) == id(p2.move)
True
>>> id(p1.location) == id(p2.location)
True
>>> id(p1.x) == id(p2.x)
False
>>> id(p1.y) == id(p2.y)
False
>>> id(p1.radius) == id(p2.radius)
False
>>>
```

**上述**实现实例属性的方式存在两个问题：
1. 在很多情况下我们希望对象被创建后其实例属性就不再增加。
2. 变量字典占用的内存空间并不小，为每个对象都创建一个变量字典会增大物理内存的压力。所以若对象的实例属性固定不变，最好用更节省内存空间的方式实现它们。

解决这两个问题的办法是给类对象添加实例属性__slots__，该属性引用的是以字符串为成员的容器，表示该对象的实例属性只能使用这些字符串作为属性名。具体来说：

➤ 如果__slots__引用的是一个字符串、字节串和字节数组之外的序列，或者一个集合，则以其内的元素作为属性名。
➤ 如果__slots__引用的是一个映射，则以该映射的键作为属性名，而该映射的值会被视为文档字符串（将在第6章讨论）的一部分。
➤ 如果__slots__引用的是一个字符串，则以该字符串为属性名。这意味着该对象只有一个实例属性。

（这里提到的类型在本章的后面会解释。）

如果一个类对象具有__slots__属性，则该类的实例将不具有__dict__特殊属性（除非“__dict__”出现在了__slots__属性列出的标识符中）。该类对象会自动为__slots__指定的每个标识符创建一个数据描述器（将在第5章讨论），而该类的实例将具有一个用C实现的数组，该数组为__slots__指定的每个标识符分配了一个元素，以储存被对应该标识符的数据描述器访问的对象的引用。因此该类的实例可以具有哪些实例属性已经固定下来了，只可能改变这些实例属性引用的对象，不能添加新的实例属性，但可以删除被允许的实例属性。下面用一个例子说明__slots__属性的作用。首先编写脚本Point2DFixed.py（类Point2DFixed只是给Point2D增加了__slots__属性）：

```
class Point2DFixed:
    dimension = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")

    #通过__slots__指定实例属性。
    __slots__ = ("x", "y", "area")
```

然后通过如下命令行执行该脚本，并在脚本执行完成后进入交互式状态：

```
$ python3 -i Point2DFixed.py
```

尝试如下语句：

```
>>> p = Point2DFixed(100,100)
>>> p.x
100
>>> p.y
100
>>> p.radius
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point2DFixed' object has no attribute 'radius'
>>> p.area
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point2DFixed' object has no attribute 'area'
>>> p.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point2DFixed' object has no attribute '__dict__'. Did you
mean: '__dir__'?
>>> import math
>>> p.radius = math.sqrt(p.x**2 + p.y**2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point2DFixed' object has no attribute 'radius'
>>> p.area = p.x * p.y
>>> p.area
10000
>>>
```

至此我们讨论完了从类对象实例化出对象的场景。上面的例子都涉及一些简单的类定义，而关于类定义的更多语法将在第5章详细讨论。

接下来考虑从元类实例化出类的场景。虽然从概念上讲类是元类的实例，但元类的实例化过程与类的实例化过程存在一个重要的区别：如果元类具有实例属性__prepare__，该属性会在调用元类的__new__之前被调用，并返回一个映射来设置变量字典；否则使用一个空字典来设置变量字典。这意味着元类可以对类的属性进行更多控制。

由于类依然通过__class__属性引用其元类，所以依然会以元类的实例属性作为其类属性。类自身的实例属性依然存放在变量字典中，但其__dict__属性不会直接引用变量字典，而是引用变量字典的一个只读代理，所以类被创建后其实例属性不能被修改。

关于从元类实例化出类的更多细节将在第15章讨论。

**让我们**总结一下上述属性模型：每个对象都通过特殊属性__class__获得了它所属类的实例属性作为其类属性。如果该对象具有特殊属性__dict__，则它所引用的变量字典包含了该对象的实例属性；否则该对象的实例属性是通过其所属类的__slots__属性指定的标识符实现的，而__slots__是相应类的实例属性，存在于类的变量字典中。

__class__和__dict__之所以是特殊属性，是因为一方面它们与特定对象绑定，不与其他对象共享，因此并非类属性；另一方面它们也不会出现在该对象的变量字典中，因此并非实例属性。特殊属性是直接通过在对象的相应二进制序列上预留槽位实现的，每个槽位都是一个字长，能储存一个对其他对象的引用。除了这两个特殊属性之外，有些特定类型的对象还具有其他特殊属性。表3-1～表3-4总结了函数、类、模块和方法的额外特殊属性。

表3-1. 函数的额外特殊属性

| 属性名 | 说明 | 访问限制 |
| --- | --- | --- |
| __name__ | 该函数的名字。默认为定义该函数时使用的标识符。 | 读写 |
| __doc__ | 该函数的文档。用于描述该函数的功能。默认引用None。 | 读写 |
| __annotations__ | 该函数的函数标注。 | 读写 |
| __module__ | 该函数所属模块的名字。 | 读写 |
| __qualname__ | 该函数的限定名后缀。 | 读写 |
| __code__ | 该函数的函数体经伪编译产生的代码对象。 | 读写 |
| __defaults__ | 包含该函数的位置对应形式参数的默认实参值的元组。如无则引用None。 | 读写 |
| __kwdefaults__ | 包含该函数的仅关键字参数的默认实参值的字典。如无则引用None。 | 读写 |
| __globals__ | 该函数所属模块的全局名字空间。 | 只读 |
| __closure__ | 包含该函数的单元对象的元组。 | 只读 |

表3-2. 类的额外特殊属性

| 属性名 | 说明 | 访问限制 |
| --- | --- | --- |
| __name__ | 该类的名字。默认为定义该类时使用的标识符。 | 读写 |
| __doc__ | 该类的文档。用于说明该类的功能。默认引用None。 | 读写 |
| __annotations__ | 该类的类体包含的变量标注。 | 读写 |
| __module__ | 该类所属模块的名字。 | 读写 |
| __qualname__ | 该类的限定名后缀。 | 读写 |
| __bases__ | 代表该类的直接基类列表的元组。 | 读写 |
| __mro__ | 储存该类的MRO的元组。 | 只读 |

表3-3. 模块的额外特殊属性

| 属性名 | 说明 | 访问限制 |
| --- | --- | --- |
| **__name__** | 该模块的名字。默认为实现该模块的文件的文件名。 | 读写 |
| **__doc__** | 该模块的文档。用于说明该模块的功能。默认引用None。 | 读写 |
| **__annotations__** | 该模块包含的函数标注和变量标注。 | 读写 |
| **__file__** | 该模块对应的文件的路径。 | 读写 |
| **__path__** | 搜索该模块的子模块时使用的位置。 | 读写 |
| **__cached__** | 该模块的编译版本的路径。 | 读写 |

表3-4. 方法的额外特殊属性

| 属性名 | 说明 | 访问限制 |
| --- | --- | --- |
| **__func__** | 该方法包装的函数。 | 只读 |
| **__self__** | 调用该方法的对象。 | 只读 |
| **__name__** | 该方法的名字。与__func__.__name__引用同一对象。 | 只读 |
| **__doc__** | 该方法的文档。与__func__.__doc__引用同一对象。 | 只读 |
| **__annotations__** | 该方法的函数标注。与__func__.__annotations__引用同一对象。 | 只读 |
| **__module__** | 该方法所属模块的名字。与__func__.__module__引用同一对象。 | 只读 |
| **__qualname__** | 该方法的限定名后缀。与__func__.__qualname__引用同一对象。 | 只读 |

表3-1～表3-4中列出的特殊属性有一部分是可读可写的，而__class__和__dict__都是可读可写的。然而即便是可读可写的特殊属性，在大部分情况下都应由Python解释器自行维护，而不要手工修改它们。（在第4章和第5章会遇到需要手工修改特殊属性的情况。）

此外需要特别强调，内置函数与函数虽然在使用接口上具有相似性，但本质上是不同的：内置函数用C定义，集成到Python解释器中；函数用Python定义，储存在Python脚本中。内置函数在Python解释器内部是以“内置函数对象（built-in function objects）”的形式存在，这些对象的类型是“内置函数或方法”。（而包装内置函数的内置方法在Python解释器内部是以“内置方法对象（built-in method objects）”的形式存在，这些对象的类型也是“内置函数或方法”。）内置函数对象没有下列特殊属性：__code__、__defaults__、__kwdefaults__、__globals__和__closure__。下面的例子说明了内置函数与函数的不同（将自定义函数f()与内置函数id()做对比）：

```
>>> def f():
...     pass
...
>>> type(f)
<class 'function'>
>>> type(id)
<class 'builtin_function_or_method'>
```

```
>>> f.__code__
<code object f at 0x1079f9a50, file "<stdin", line 1>
>>> id.__code__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttribtueError: 'builtin_function_or_method' object has no attribute
'__code__'. Did you mean: '__call__'?
>>>
```

类似的，内置类型与类在使用接口上具有相似性，但本质上是不同的：内置类型用C定义，集成到Python解释器中；类用Python定义，储存在Python脚本中的。内置类型在Python解释器内部其实是以“类型对象（type objects）”的形式存在的。在Python 2中类型对象与类对象是截然不同的，而Python 3有意模糊了类型对象与类对象的区别。但这两者的区别依然存在：

1. 类型对象没有__annotations__特殊属性。
2. 类型对象没有__slots__属性，但属于内置类型的对象默认就不具有__dict__特殊属性，因此也就无法具有实例属性。

请通过下列语句验证上述结论（输出的结果被省略）：

```
>>> type(0).__dict__
>>> (0).__dict__
>>> (0).a = 1
>>> type('abc').__dict__
>>> 'abc'.__dict__
>>> 'abc'.a = 1
>>> type([0, "abc"]).__dict__
>>> [0, "abc"].__dict__
>>> [0, "abc"].a = 1
```

访问一个对象的__dict__特殊属性本质上是在查看该对象的变量字典，而我们也可以用内置函数vars()来达到相同目的，该函数的语法为：

```
vars(object)
```

其实vars()的作用就是读取指定对象的__dict__并返回，而如果该对象没有__dict__就抛出TypeError异常（将在第8章解释）。下面的例子说明了vars()的用法：

```
>>> class trivial:
...     pass
...
>>> obj = trivial()
>>> vars(obj)
{}
>>> vars(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: vars() argument must have __dict__ attribute
>>>
```

**最后**，有必要阐明一个对象的属性和值之间的关系。

图3-1和图3-2中给出的模型假设一个对象对应的二进制序列只有一个实现特殊属性__class__的槽位，其余部分都用于实现该对象的值，这与真实情况差别很大。事实上，一个对象对应的二进制序列有多个用于实现特殊属性的槽位。如果该对象所属类具有__slots__，则其对应的二进制序列还要有一个槽位引用一个C数组，以储存对__slots__指定的标识符所需访问的对象的引用。二进制序列除这些槽位之外的部分才是该对象的值，其内既可以存放数字、字符串之类的数据，也可以存放对其他对象的引用。

由于Python类定义的语法中没有设置值的方法，所以用户自定义类的实例都是没有值的，所有具有值的对象都属于某种内置类型（或某种扩展模块定义的类型）。但属于内置类型的对象不一定有值，例如函数对象的所有数据都是通过表3-1列出的特殊属性间接储存的，因而没有值。没有值的对象显然是不可变的。

## 3-4. 类之间的关系

（语言参考手册：3.2、3.3）
（标准库：内置函数）

图3-3给出的模型理论上已经足以实现任何对象，但这还不够经济高效。考虑这样的场景：可以对属于类B1的对象和属于类B2的对象施加同一种操作。如果按照图3-3的模型，就需要在类B1和类B2中分别实现这一操作，因此内存中需要存在两个功能相同的函数对象。

**一个**更好的办法是定义类A，让类A包含实现该操作的属性，然后让类B1和类B2都“继承（inherit）”类A，进而获得该属性，这样在内存中就只存在一个函数对象。

表3-2列出的类对象的特殊属性中，__bases__和__mro__就是用于描述这种继承关系的。这意味着Python支持类的继承，因此Python类型不是完全独立的。如果类C继承了类B，而类B又继承了类A，则：

➤ 从类C的角度，类B和类A都是它的“基类（base classes）”，但类B是它的直接基类，而类A是它的间接基类。
➤ 从类A的角度，类C和类B都是它的“子类（subclasses）”，但类B是它的直接子类，类C是它的间接子类。
➤ 从类B的角度，类C是它的（直接）子类，类A是它的（直接）基类。

规定一个类至多有一个直接基类的编程语言仅支持“单继承（single inheritance）”。单继承语言保证类的“继承链”是一个单链表，如图3-4左边所示。而允许一个类有多个直接基

类的编程语言支持“多继承（multiple inheritance）”。多继承语言中类的继承链可以是一个具有菱形结构的有向图，如图3-4右边所示。Python是多继承语言。

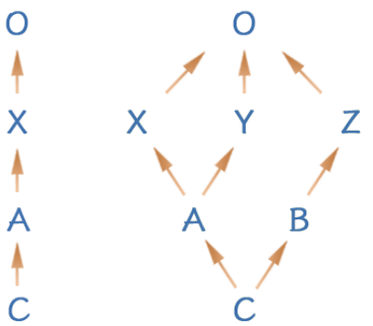


图3-4. 单继承和多继承

Python解释器内置了一个特殊类型object，它是所有类型（包括用户自定义类和内置类型）的基类，而其自身没有基类。也就是说，object是Python中所有类型相互间依靠继承关系形成的有向图的顶点。另外，bool类型继承自int类型。下面的例子说明了特殊属性__bases__和__mro__的区别：

```
>>> bool.__bases__
(<class 'int'>,)
>>> bool.__mro__
(<class 'bool'>, <class 'int'>, <class 'object'>)
>>>
```

可以看出，__bases__和__mro__都引用一个以类对象为成员的元组，但前者仅包含直接基类（其顺序与这些基类在类定义语句中出现的顺序一致），而后者则包含所有基类（其顺序代表着这些基类被继承的顺序）。本例子只涉及单继承，第5章会给出多继承的例子。

有了继承，Python对象之间关系的模型就应该由图3-3升级为图3-5。注意图3-5只显示了单继承，但多继承也只是给继承链中添加了一些嵌套的菱形结构，不会改变该模型的本质。

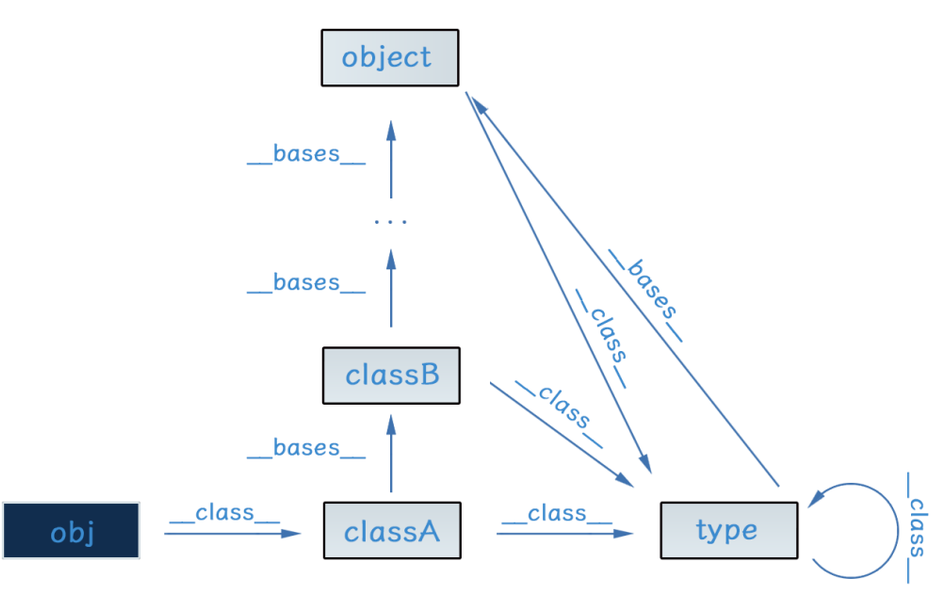


图3-5. 对象间的“实例”和“继承”关系

object和type是图3-5所示模型中最特殊的两个端点：所有类都是type的实例，而所有类又都是object的子类。因此，object也是type的实例，反过来type也是object的子类。这是Python面向对象设计中最难理顺的关系。

因为有了继承，一个对象的类属性就不一定来自它的__class__引用的类，还可能来自该类通过__bases__引用的其他类，以及这些类通过__bases__引用的类，……。换句话说，一个对象的类属性可能来自于它所属的类，也可能来自于它所属类的某个基类。当然，在继承链中的多个类可能具有同名的属性，那么就需要按照这些类的继承顺序以最接近对象的那个类的属性为准。该规则被称为属性的“重写（overwrite）”，会在第5章被进一步说明。

具体以图3-5为例，obj对象的类属性可能是classA的实例属性、classB的实例属性、……、object的实例属性，但不可能是type的实例属性。而classA、classB、……、object的类属性只可能是type的实例属性。一言以蔽之，一个对象的类属性只能是它所属类的继承链中某个类对象的实例属性。

---

**Python**中有一些系统定义的标识符具有特殊含义。如果一个对象具有以这些标识符为属性名的类属性，就将获得特定的语法特征，因此这些标识符所对应的属性被称为“魔术属性（magic attributes）”。

附录 II总结了所有魔术属性，注意它们都符合第2章介绍过的系统定义的标识符的要求。前面已经讨论过的__slots__其实就是一个魔术属性。但并非所有系统定义的标识符都是魔术属性。例如前面介绍的__dict__就不是魔术属性，因为它是一个特殊属性。此后本书讨论到某魔术属性时，请查阅附录 II以了解该魔术属性的语法，正文中不会再重复。

由于object是所有类的基类，所以它的实例属性成为了所有对象的类属性。事实上，这些实例属性都是魔术属性，也代表所有对象通用的语法特征。表3-5总结了object实现了的魔术属性。我们可以在继承链的合适位置重写这些魔术属性，以改变相关对象的相应语法特征。

表3-5. object的实例属性

| 属性名 | 说明 |
| --- | --- |
| `__new__()` | 一个静态方法，用于创建该类的实例。 |
| `__init__()` | 初始化该类的一个实例。 |
| `__str__()` | 返回对象的非正式字符串表示。 |
| `__repr__()` | 返回对象的正式字符串表示。 |
| `__format__()` | 返回对象的格式化字符串表示。 |
| `__lt__()` | 对<表达式求值时自动调用。 |
| `__le__()` | 对<=表达式求值时自动调用。 |
| `__eq__()` | 对==表达式求值时自动调用。 |
| `__ne__()` | 对!=表达式求值时自动调用。 |

| 属性名 | 说明 |
| --- | --- |
| **__gt__()** | 对>表达式求值时自动调用。 |
| **__ge__()** | 对>=表达式求值时自动调用。 |
| **__hash__()** | 返回对象的哈希值。 |
| **__dir__()** | 返回对象的有效属性列表。 |
| **__init_subclass__()** | 一个类方法，当该类被其他类继承时自动调用。 |
| **__subclasshook__()** | 一个类方法，检测指定类是否是该类的子类。 |
| **__getattribute__()** | 属性被读取时自动调用，如果属性不存在则抛出AttributeError异常。 |
| **__setattr__()** | 属性被设置时自动调用。 |
| **__delattr__()** | 属性被删除时自动调用。 |

我们可以通过内置函数object()来创建属于object的对象。需要强调，虽然object没有__slots__属性，但它的实例依然没有__dict__属性，这意味着此类对象只能具有表3-5列出的属性作为类属性，无法具有实例属性。一般而言，创建这样的对象是没有意义的。下面的例子说明了这点（注意__sizeof__、__reduce__和__reduce_ex__是CPython特有的实现细节，所以表3-5没有将它们列出；__getstate__本来是通过标准库中的pickle模块实现的，Python 3.11给object增加了该属性，本书不详细讨论）：

```
      >>> o1 = object()
      >>> o2 = object()
      >>> id(o1) == id(o2)
      False
      >>> o1.__dict__
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      AttributeError: 'object' object has no attribute '__dict__'. Did you mean:
'__dir__'?
      >>> o1.a = 0
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      AttributeError: 'object' object has no attribute 'a'
      >>> o1.__dir__()
      ['__new__', '__repr__', '__hash__', '__str__', '__getattribute__',
'__setattr__', '__delattr__', '__lt__', '__le__', '__eq__', '__ne__', '__gt__',
'__ge__', '__init__', '__reduce_ex__', '__reduce__', '__getstate__',
'__subclasshook__', '__init_subclass__', '__format__', '__sizeof__', '__dir__',
'__class__', '__doc__']
      >>>
```

另一方面，由于type是所有类的元类，所以它的实例属性成为了所有类的类属性。表3-6总结了type的实例属性，其中前三个也是魔术属性，而后三个则是进行元类编程才需要用到的实例属性（将在第15章讨论）。

表3-6. type的实例属性

| 属性名 | 说明 |
| --- | --- |
| `__call__()` | 保证所有类对象都可被调用。 |
| `__or__()` | 类对象的按位或，以支持泛型。 |
| `__ror__()` | 类对象反射操作数的按位或，以支持泛型。 |
| `__prepare__()` | 为创建类对象准备变量字典。 |
| `__instancecheck__()` | 检查某对象是否是指定类的实例。 |
| `__subclasscheck__()` | 检查某类是否是指定类的子类。 |

type实现了魔术属性__call__，这意味着所有类都能访问__call__。这使得所有类对象都能被调用，而调用的结果就是实例化出一个属于该类的对象。此外，所有函数属于的函数类型，以及所有方法属于的方法类型，也都实现了__call__，以使得函数和方法能被调用。事实上，任何对象只要能够访问类属性__call__就能够被调用。

如果一个对象能被调用，则称其是“可调用的（callable）”。我们可以通过内置函数callable()判断一个对象是否可调用，该函数的语法为：

```
callable(object)
```

事实上，callable()靠检查是否可以通过指定对象访问__call__来判断该对象是否可调用，是的时候返回True，否的时候返回False。

在这里介绍内置函数dir()的基本用法，其语法为：

```
dir(object)
```

该函数的作用是调用指定对象的__dir__魔术属性，返回一个属性名列表。__dir__的默认行为是：

➤ 对于模块对象，仅返回其实例属性。
➤ 对于类对象，返回其实例属性，且递归查找并返回该类的所有基类的实例属性。
➤ 对于其他对象，返回它本身和其所属类的实例属性，且递归查找并返回其所属类的所有基类的实例属性。

但__dir__是可以在定义类时被重写的，所以返回的结果不一定如上所述，通常仅包含（类的定义者所认为的）较重要的属性。

最后值得一提的是，在讨论类之间的关系时还经常会用到“派生（derive）”这个术语，它的含义正好与“继承”相反：如果A是B的基类，则B继承了A，反过来A派生了B。

## 3-5. 内置类型和内置常量

（语言参考手册：3.2）
（标准库：内置类型、内置常量、types）

至此我们讨论完了Python的数据模型的逻辑框架，下面介绍具体的类型。由于Python一切皆对象，类与类型等同，因此Python有无数多个类型。好在Python的内置类型是有限的，而在前面提到的函数、类、方法、数字、字符串、元组、列表和字典都属于内置类型。本节将系统地介绍这些内置类型，以及与它们相关的内置常量。

---

**Python**的内置类型只有一部分向用户提供了对应内置函数，而那些没向用户提供对应内置函数的内置类型被统称为“标准解释器类型”。属于这些类型的对象是Python解释器在运行过程中自动创建和维护的。通过标准库中的types模块可以得到标准解释器类型的名字，而附录 III则总结了这些类型。下面分类讨论这些标准解释器类型。

首先，如下三种类型都只支持一个值，分别对应一个具有特殊含义的对象：

➤ None的类型：仅支持特殊值None，代表空对象。标识符可以绑定到None，但这其实表明该标识符是悬空的，没有指向有效对象。
➤ NotImplemented的类型：仅支持特殊值NotImplemented，代表未实现对象。若访问一个属性（通常是魔术属性）的结果是NotImplemented，则表明该对象没有实现该属性。
➤ Ellipsis的类型：仅支持特殊值Ellipsis，代表省略对象。Python脚本中出现的“…”在语义分析时被解释为该对象，进而实现相关语法特性（例如自定义容器类型的扩展切片）。

None、NotImplemented和Ellipsis都属于Python解释器预定义的内置常量。下面的例子验证了这三种类型：

```
>>> type(None)
<class 'NoneType'>
>>> type(NotImplemented)
<class 'NotImplementedType'>
>>> type(Ellipsis)
<class 'ellipsis'>
>>> Ellipsis == ...
True
>>> import types
>>> type(None) == types.NoneType
True
>>> type(NotImplemented) == types.NotImplementedType
True
>>> type(Ellipsis) == types.EllipsisType
```

```
True
>>>
```

前面已经提到了函数对象，而下面是所有与函数相关的类型。注意它们实例化出的对象都是可调用的：

- 函数的类型：通过函数定义语句定义的函数。
- 匿名函数的类型：通过lambda表达式定义的匿名函数。
- 生成器的类型：由生成器函数或生成器表达式创建的对象。
- 协程的类型：由协程函数创建的对象。
- 异步生成器的类型：由异步生成器函数创建的对象。

由于属于这些类型的对象是Python脚本的基本构建模块，所以对这些类型的讨论将贯穿全书。下面的例子依次给出这些类型的可调用对象（暂时看不懂这些代码没有关系，因为它们会在后面被讨论）：

```
>>> import types
>>> def f():
...     return 0
...
>>> type(f)
<class 'function'>
>>> type(f) == types.FunctionType
True
>>>
>>> l = lambda x: x*2
>>> type(l)
<class 'function'>
>>> type(l) == types.FunctionType
True
>>> type(l) == types.LambdaType
True
>>> types.LambdaType == types.FunctionType
True
>>>
>>> g = (x for x in [1, 2, 3])
>>> type(g)
<class 'generator'>
>>> type(g) == types.GeneratorType
True
>>>
>>> async def corof():
...     pass
...
>>> coro = corof()
>>> type(coro)
<class 'coroutine'>
>>> type(coro) == types.CoroutineType
True
>>>
>>> async def agf():
...     for i in range(10):
...         yield i
```

```
...         await co
...
>>> ag = agf()
>>> type(ag)
<class 'async_generator'>
>>> type(ag) == types.AsyncGeneratorType
True
>>>
```

前面也提到了方法对象，事实上它们是对函数对象的包装。下面的例子定义了一个仅包含一个函数属性的类，创建了一个该类的实例，并验证了通过实例访问该属性时函数会被包装为方法，而通过类访问该属性时则不会（这会在第5章被详细讨论）：

```
>>> class OneMethodClass:
...     def method_example(self):
...         return 0
...
>>> obj = OneMethodClass()
>>> type(OneMethodClass.method_example)
<class 'function'>
>>> type(obj.method_example)
<class 'method'>
>>> import types
>>> type(obj.method_example) == types.MethodType
True
>>> obj.method_example.__func__ == OneMethodClass.method_example
True
>>>
```

前面也已经提到了Python中的模块，第6章会说明模块是维护和管理Python程序的核心机制。下面的例子显示了模块的类型：

```
>>> type(types)
<class 'module'>
>>> type(types) == types.ModuleType
True
>>>
```

前面同样已经提到了，内置函数与函数本质上是不同的：内置函数对应内置函数对象，而包装内置函数的内置方法则对应内置方法对象，它们的类型都是“内置函数或方法”。这里需要指出，上述说法是不准确的。附录 IV列出了所有的内置函数，表IV-1和表IV-2中的内置函数，以及type()、property()、super()和slice()，其实属于类型对象。下面的例子验证了这些事实：

```
>>> type(int)
<class 'type'>
>>> type(dir)
<class 'builtin_function_or_method'>
>>> import types
>>> type(dir) == types.BuiltinFunctionType
```

```
True
>>> type(dir) == types.BuiltinMethodType
True
>>> types.BuiltinFunctionType == types.BuiltinMethodType
True
>>>
```

最后，如下几种类型对应一些特殊对象，它们的存在几乎不会被意识到：

- 代码对象的类型：函数体经伪编译得到的可重复执行的中间代码。
- 帧对象的类型：函数被调用时生成的栈帧。
- 单元对象的类型：函数闭包需要访问的对象。
- 回溯对象的类型：抛出异常时自动创建的对象，用于支持回溯。
- 泛型别名的类型：将类型当成参数通过泛型生成的类型。
- Union结构的类型：将多个类型通过按位或运算符号连接起来形成的表达式。

---

**接下来**讨论向用户提供了对应内置函数的内置类型，它们可以被视为其他编程语言中的“基本类型”。

首先是数字相关类型，包括：

- 整数（integers）：可通过内置函数int()创建。
    - 布尔值（booleans）：整数的子类，可能取值是内置常量True和False，可通过内置函数bool()创建。
- 浮点数（floating point numbers）：可通过内置函数float()创建。
- 复数（complex numbers）：可通过内置函数complex()创建。

然后是字符串和二进制序列（虽然任何对象本质上都是个二进制序列，但“二进制序列”同时也是Python中的若干数据类型的统称），包括：

- 字符串（strings）：可通过内置函数str()创建。
- 字节串（bytes objects）：可通过内置函数bytes()创建。
- 字节数组（bytearrays）：可通过内置函数bytearray()创建。
- 内存视图（memoryviews）：可通过内置函数memoryview()创建。

这些类型其实都属于“容器（collections）”中的序列，但又具有一定的特殊性。

最后是容器，它们又被分为三类：“序列（sequences）”、“映射（mappings）”和“集合类型（set types）”。除了上面提到的字符串和二进制序列之外，序列还包括：

- 元组（tuples）：可通过内置函数tuple()创建。
- 列表（lists）：可通过内置函数list()创建。
- 范围（ranges）：可通过内置函数range()创建。

映射包括：

- 字典（dictionaries）：可通过内置函数dict()创建。

而集合类型则包括：

- 集合（sets）：可通过内置函数set()创建。
- 凝固集合（frozen sets）：可通过内置函数frozenset()创建。

---

以上就是所有的内置类型。Python标准库中的一些模块（例如datetime、enum、collections和array）提供了额外的数据类型，但它们不属于内置类型，而等同于自定义类。第5章会详细讨论自定义类的用法，而完整的类型理论将在第15章被讨论。

## 3-6. 逻辑值检测、逻辑运算和比较运算

（语言参考手册：3.3.1、3.3.7、6.10、6.11、6.13）
（标准库：内置函数、内置类型）

分支和循环都依赖于判断对一个表达式求值得到的结果等同于True还是False，而Python中一切都是对象，对表达式求值得到的结果也不例外。这就意味着必须有一套规则将对象映射到布尔值，该规则就是所谓的“逻辑值检测（truth value testing）”：

- True和False没有必要检测，可直接使用。
- 若该对象具有魔术属性__bool__，则将该对象映射到调用__bool__得到的返回值。（__bool__必须被实现为返回一个布尔值。）
- 否则，若该对象具有魔术属性__len__，则当调用该属性的返回值为0时映射到False，其他情况映射到True。（__len__必须被实现为返回一个非负整数。）
- 否则，将该对象映射到True。

按照逻辑值检测的规则，以及内置类型对__bool__和__len__的实现，默认情况下仅下列对象被映射到False：None、0、0.0、0j以及任何0个成员的容器对象（包括空字符串和空字节串）。此外，NotImplemented不被映射到任何布尔值，因此它出现在需要进行逻辑值检测的场合属于语法错误。

值得一提的是，内置函数bool()创建布尔值的原理就是返回对指定对象进行逻辑值检测的结果，其语法为：

```
class bool([object])
```

特别的，如果省略了object参数，则bool()返回False。但由于在Python语句中需要布尔值的地方都会自动对相应对象进行逻辑值检测，所以bool()几乎只被用在赋值语句中。

---

**Python**通过运算符not、and和or分别实现了逻辑非、逻辑与和逻辑或，且它们的优先级也如上述顺序。表2-2说明这三个运算符的优先级是相当低的。

需要强调的是，由于有逻辑值检测这种机制存在，所以这三个运算符与数学上的逻辑运算存在微妙的差别：

➤ not表达式格式为“not expr”，当对expr求值得到的对象的逻辑值检测结果为True时得到False，而当该对象的逻辑值检测结果为False时得到True。（这完全符合数学中的非运算。）
➤ and表达式格式为“expr1 and expr2”。先对expr1求值，若得到的对象的逻辑值检测结果为False，则该对象就是对整个and表达式求值的结果，不再对expr2求值。否则，对expr2求值，并将得到的对象作为对整个and表达式求值的结果。
➤ or表达式格式为“expr1 or expr2”。先对expr1求值，若得到的对象的逻辑值检测结果为True，则该对象就是对整个or表达式求值的结果，不再对expr2求值。否则，对expr2求值，并将得到的对象作为对整个or表达式求值的结果。

换句话说，and表达式和or表达式总是得到两个操作数之一，将其用在需要布尔值的地方时才再次进行逻辑值检测。这使得它们不仅兼容数学中的逻辑与和逻辑或，还有其他更巧妙的用法。例如“o = x and y”会在x的逻辑值检测结果为False时将其赋值给o，仅在该结果为True时才将y赋值给a。而“o = x or y”则正好反过来。

推而广之，“o = x1 and x2 and … and default”会从对象序列x1，x2，……中选出第一个逻辑值检测结果为False者赋值给o，如果它们的逻辑值检测结果都为True则赋值default。而“o = x1 or x2 or … or default”则会从对象序列x1，x2，……中选出第一个逻辑值检测结果为True者赋值给o，如果它们的逻辑值检测结果都为False则赋值default。

请看下面的例子：

```
>>> a = 1 and '' and 0 and None
>>> a
''
>>> b = () or {} or object()
>>> b
<object object at 0x102370860>
>>>
```

Python还参考C中的“... : ... ? ...”运算符引入了“... if ... else ...”运算符，相应表达式格式为“expr1 if condition else expr2”，含义是：如果对condition求值得到的对象的逻辑值检测结果为True，则得到对expr1求值得到的对象；否则，得到对expr2求值得到的对象。该运算符的优先级非常低，仅高于lambda运算符和:=运算符。

---

**接下来**讨论与逻辑运算密切相关的比较运算。由于比较运算的结果直接是True或False，所以比较表达式用在需要布尔值的场合不会引发逻辑值检测。

Python中有多种比较运算符，其中如下6种源自C，含义分别是“小于”、“小于等于”、“等于”、“不等于”、“大于”和“大于等于”：

```
<     <=    =     !=    >     >=
```

这些运算符的工作原理分别是调用魔术属性：__lt__、__le__、__eq__、__ne__、__gt__和__ge__。举例来说，在对如下表达式求值时：

```
x < y
```

其实执行的是：

```
x.__lt__(y)
```

在上一节已经说明，由于object实现了__lt__、__le__、__eq__、__ne__、__gt__和__ge__，所以任意两个对象之间都可以用上述6种运算符进行比较。但object在实现这6个魔术属性时，除了__eq__和__ne__检查两个操作数引用的是否是同一个对象之外，其余4个魔术属性都只是简单地抛出TypeError异常，因此默认情况下对象之间仅能进行==和!=比较。下面的例子说明这点：

```
>>> a = object()
>>> b = object()
>>> a == b
False
>>> a != b
True
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'object' and ‘object'
>>>
```

而数字和字符串都在各自的类中重写了__lt__、__le__、__gt__和__ge__：数字按照数学上的大小顺序进行比较，而字符串则按照字典顺序进行比较。例如：

```
>>> -1 < 1
True
>>> 13.9 > 14.8
False
>>> 5+9j <= 5-9j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<=' not supported between instances of 'complex' and 'complex'
>>> 'a' < 'b'
True
>>> 'begin' > 'budget'
False
>>>
```

数字相关类甚至还重写了__eq__和__ne__以使得整数、浮点数、复数和布尔值之间可以按照符合数学规则的方式进行比较，例如：

```
>>> 0 == 0.0
True
>>> 0.0 == 0j
True
>>> False == 0
True
>>> True == 1.0
True
>>> True == 2
False
>>>
```

除了上面6个比较运算符之外，Python还提供了如下四个较独特的比较运算符：

```
in    not in            is    is not
```

其中in和not in用于检查一个对象是否是一个容器的成员，因此被称为“成员检测（membership testing）”。“obj1 in obj2”仅当obj1是obj2的成员时才得到True，否则得到False；而“obj1 not in obj2”的行为则正好相反。需要强调的是，这两种表达式仅当obj2具有魔术属性__contains__时才是合法的，而所有内置容器类型都实现了该魔术属性。下面的例子说明了成员检测的行为：

```
>>> 1 in (1, 2, 3)
True
>>> 4 in (1, 2, 3)
False
>>> 1 not in (1, 2, 3)
False
>>> 4 not in (1, 2, 3)
True
>>> 'a' in 'abc'
True
>>> 'd' in 'abc'
False
>>> 'a' not in 'abc'
False
>>> 'd' not in 'abc'
True
>>>
```

is和is not用于检查两个标识符是否指向同一对象，又被称为“相同测试（identity test）”。“a is b”等价于“id(a) == id(b)”，而“a is not b”等价于“id(a) != id(b)”。易知，任意两个标识符都可以进行相同测试，不会产生语法错误。下面的例子说明了其用法：

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a is not b
False
>>> b = 1.0
>>> a is b
False
>>> a is not b
True
>>>
```

与C不同的是，Python中的比较运算符可以任意串连。举例来说，表达式“x < y >= z”会被解读为“x < y and y >=z”。下面是一些例子：

```
>>> 5 < 8.9 > 6
True
>>> 10 >= 7 < 8 in (8, 9)
True
>>>
```

该语法特性的前提是所有比较运算符具有相同的优先级，这在表2-2中已经反映出来了。

比较运算符的优先级高于所有逻辑运算符，包括not。所以“not a == b”不会被解读为“(not a) == b”，而是被解读为“not (a == b)”；而“a == not b”是一个语法错误。这与很多其他编程语言（包括C）是不一样的。

本节作为例子的比较表达式都很简单，在第9章～第11章会见到更复杂更难理解比较表达式。

---

最后，PEP 8推荐了如下用于提高代码可读性的规范：
1. 总是在逻辑运算符和比较运算符的两端添加一个空格。
2. 不要与True和False直接做比较（例如“if obj == True:”），而应利用逻辑值检测机制（例如“if obj:”）。
3. 利用逻辑值检测机制来判断一个容器是否非空（例如“if seq:”），而不要判断其成员个数是否大于零（例如“if len(seq) > 0:”）。
4. 如果要重写用于比较的魔术属性，则重写所有六个（__lt__、__le__、__eq__、__ne__、__gt__和__ge__），而不要只重写其中的一部分。
5. 与None、NotImplemented和Ellipsis（以及类似的具有特殊含义的对象）进行比较时，使用is或is not，而不要使用==或!=。
6. 在可以使用is not的场合，不要使用not ... is ...。

## 3-7. Python中最基本的语句

（教程：3.2、4.1、4.4、4.5、5.7）
（语言参考手册：7.1、7.2、7.9、7.10、8.1、8.2）

在本章的最后我们讨论Python中最基本的语句，它们保证了Python的图灵完备性。“图灵完备性（Turing completeness）”是一个听起来让人生畏的术语，因为它来自作为计算机科学基石的可计算性理论。用通俗的说法来解释，图灵完备性其实就是能够模拟一台图灵机的能力，具有这一能力的系统理论上可以完成计算机能够完成的一切工作（但效率可以大相径庭）。

图灵完备的编程语言从可计算性理论的角度来讲具有相同的能力，而几乎所有现代编程语言都是图灵完备的。所以本节的重点在于，如何以尽可能少的Python语句保证图灵完备性。这些语句就是最基本的语句，仅仅使用它们理论上就可以编写出能够解决任何可计算的问题的程序了。前面已经介绍过的赋值语句是最基本的语句之一。而任何表达式被独立使用时就形成一条“表达式语句（expression statements）”，这也是最基本的语句之一。除它们之外，我们只额外需要两条最基本的语句就可以保证图灵完备性——用于实现“分支（branches）”的if语句，和用于实现“循环（loops）”的while语句。

if语句是一条复合语句，其语法为：

```
if expr1:
    suite
elif expr2:
    suite
...
else:
    suite
```

也就是说，if语句被关键字if、elif和else分为若干“子句（clauses）”，每个子句包含一个用suite代表的代码块（可以包含其他复合语句）。if语句的执行逻辑是：依次对表达式expr1、expr2、……求值，直到得到的某个对象的逻辑值检测结果是True为止，然后执行该表达式所在子句包含的代码块。如果所有表达式求值得到的对象的逻辑值检测结果都是False，则执行else子句包含的代码块。易知，if语句根据表达式expr1、expr2、……来决定执行哪个子句包含的语句块，而每个子句就是一个分支。这些表达式则被视为if语句的“条件表达式（conditional expressions）”。

下面的脚本number_sign_1.py是一个使用标准if语句的例子（内置函数input()的作用是从标准输入读取一个字符串，将在第7章详细讨论）：

```
s = input("请输入一个整数: ")

n = int(s)
if n < 0:
    print("这是一个负数。")
elif n > 0:
    print("这是一个正数。")
else:
    print("这是零。")
```

请多次执行该脚本，并尝试输入不同的字符串以观察得到的反馈。

if语句还有两种变体。第一种变体是去掉所有elif子句，仅保留else子句，即：

```
if expr:
    suite1
else:
    suite2
```

这样if语句就从任意多个分支变成了两个分支：如果对表达式expr求值得到的对象的逻辑值检测结果是True，则执行suite1代表的代码块；否则执行suite2代表的代码块。

下面通过对number_sign_1.py的修改来说明if语句的这种变体的用法。该脚本有个缺点：如果输入的字符串不是整数的字符串表示就会抛出异常。下面的代码将原if语句嵌入了另一

个if语句：外层if语句的作用是通过Perl兼容正则表达式来判断输入的字符串是否是一个整数，是则执行内层if语句；否则提示用户应输入一个整数（本例子使用了标准库中的re模块，该模块提供了对正则表达式的支持，但本书不详细讨论该模块）：

```
import re

s = input("请输入一个整数: ")

if re.match(r"^[+\-]?[0-9]+$", s):
    n = int(s)
    if n < 0:
        print("这是一个负数。")
    elif n > 0:
        print("这是一个正数。")
    else:
        print("这是零。")
else:
    print("您输入的并非整数。")
```

请将这段代码保存为number_sign_2.py，然后多次运行该脚本。

if语句的第二种变体是仅保留if子句，即：

```
if expr:
    suite
```

这意味着仅当对表达式expr求值得到的对象的逻辑值检测结果是True时，suite代表的代码块才会被执行。下面将number_sign_2.py中的代码改造为：

```
import re

s = input("请输入一个整数: ")

if s != "quit":
    if re.match(r"^[+\-]?[0-9]+$", s):
        n = int(s)
        if n < 0:
            print("这是一个负数。")
        elif n > 0:
            print("这是一个正数。")
        else:
            print("这是零。")
    else:
        print("您输入的并非整数。")
```

这使得当输入字符串“quit”时程序终止。请将这段代码保存为number_sign_3.py，然后多次运行该脚本。

while语句也是一条复合语句，其语法为：

```
while expr:
    suite1
else:
    suite2
```

其含义是：当对表达式expr求值得到的对象的逻辑值检测结果是True时，重复执行suite1代表的代码块；直到对表达式expr求值得到的对象的逻辑值检测结果变为False，然后执行suite2代表的代码块，最后结束循环。易知，while语句是以表达式expr来决定是否执行suite1代表的代码块的，该代码块被称为“循环体（loop body）”，而该表达式则被视为while语句的条件表达式。

使用while语句最重要的一点是，循环体必须能够使条件表达式的值发生变化，并最终使得某个值的逻辑值检测结果为False。否则该while语句就成为了一个“死循环（dead loop）”，除了采取强制手段（例如按下Ctrl+C）之外无法使Python脚本的执行终止。

下面的例子是对number_sign_3.py的进一步改造，它通过一个while语句使得“从标准输入读取一个整数，然后判断它的正负号”这一行为循环进行，直到读取到了“quit”字符串：

```
import re

print("这是一个判断整数符号的小游戏。\n")

#对变量s赋初始值并保证其逻辑值检测的结果为True。    这样才能避免循环在第一次计算条件
# 表达式时终止。
s = "begin"

#只要读取到的字符串不是“quit”，就会继续读取。    而当读取到“quit”后，则输出游戏结束
# 的提示。
while s != "quit":
    #由于每次循环都读取了不同的字符串，而“quit”会使条件表达式的值的逻辑值检测结果
    # 为False，所以该循环不会是死循环。
    s = input("请输入一个整数：")

    if re.match(r"^[+\-]?[0-9]+$", s):
        n = int(s)
        if n < 0:
            print("这是一个负数。")
        elif n > 0:
            print("这是一个正数。")
        else:
            print("这是零。")
    else:
        print("您输入的并非整数。")
else:
    print("你已经退出了游戏。\n")
```

请将上面的代码保存为number_sign_game1.py，然后只需要执行一次就可以验证该脚本的所有行为。

number_sign_game_1.py有个缺点，即当读取到“quit”时，会先输出“您输入的并非整数。”，然后再输出“你已经退出了游戏。\n”。这显然不是我们期望的。为了弥补这一缺点，就需要在循环体中加判断，然后通过break语句跳出循环，或者通过continue语句继续下一次循环。这两条只能在循环体内部使用的语句都是简单语句，只需要输入相应的关键字break或continue即可。下面的例子用continue语句弥补了上述缺点：

```
import re

print("这是一个判断整数符号的小游戏。\n")

s = "begin"

while s != "quit":
    s = input("请输入一个整数: ")

    #当读取的字符串是“quit”时，继续下一次循环。
    if s == "quit":
        continue

    if re.match(r"^[+\-]?[0-9]+$", s):
        n = int(s)
        if n < 0:
            print("这是一个负数。")
        elif n > 0:
            print("这是一个正数。")
        else:
            print("这是零。")
    else:
        print("您输入的并非整数。")
else:
    print("你已经退出了游戏。\n")
```

请将上面的代码保存为number_sign_game_2.py，然后验证。

注意上面的代码中选择通过continue语句继续下一次循环，其背后逻辑是循环体中continue语句之后的部分被跳过，而下一次循环会继续进行。此时s引用了字符串“quit”，使else子句被执行，然后循环结束。但如果将continue语句替换成break语句的话，你会发现输入“quit”后脚本直接结束运行，不会输出“你已经退出了游戏。\n”。这是因为break语句会直接跳出整个循环，而else子句被视为循环的一部分，因此同样被跳过。要用break语句解决上述问题，就必须使用while语句的变体，也就是省略else子句：

```
while expr:
    suite
```

这样当对表达式expr求值得到的对象的逻辑值检测结果是False时，直接执行while语句后面的其他语句。下面是采用此策略修改number_sign_game_1.py得到的代码：

```
import re

print("这是一个判断整数符号的小游戏。\n")

while True:
    s = input("请输入一个整数: ")

    #当读取的字符串是“quit”时，跳出循环。
    if s == "quit":
        break

    if re.match(r"^[+\-]?[0-9]+$", s):
        n = int(s)
        if n < 0:
            print("这是一个负数。")
        elif n > 0:
            print("这是一个正数。")
        else:
            print("这是零。")
    else:
        print("您输入的并非整数。")

#本提示仅当循环语句执行完成后才会被输出。
print("你已经退出了游戏。\n")
```

请将上面的代码保存为number_sign_game_3.py，然后验证。