3.6

Python 学习笔记

第三版

前言

读者定位

本书着重于剖析语言相关背景和实现方式，适合有一定 Python 编程基础的读者，比如准备从 2.x 升级到 3.x 环境。至于初学者，建议寻一本从零开始，循序渐进介绍如何编写代码的图书。

联系方式

鉴于能力有限，书中难免错漏。如您发现任何问题，请与我联系，以便更正。谢谢！

邮件：qyuhen@hotmail.com

雨痕

二〇一七，初夏

更新

2013-01-09　第二版，基于 Python 2.7。
2017-06-01　第三版，基于 Python 3.6。

上卷 语言详解

Python 3.6

一. 概述

示例运行环境: CPython 3.6，macOS 10.12

> 鉴于不同运行环境差异，示例输出结果会有所不同。尤其是 id，以及内存地址等信息。
> 请以实际运行结果为准。

二. 类型

1. 基本环境

作为一种完全面向对象，且兼顾函数式的编程语言，Python 复杂程度要远高出许多人的设想，诸多概念被隐藏在看似简单的代码背后。为了更好且更深入理解相关规则，在讲述语言特征以前，我们需对世界背景做个初步了解。

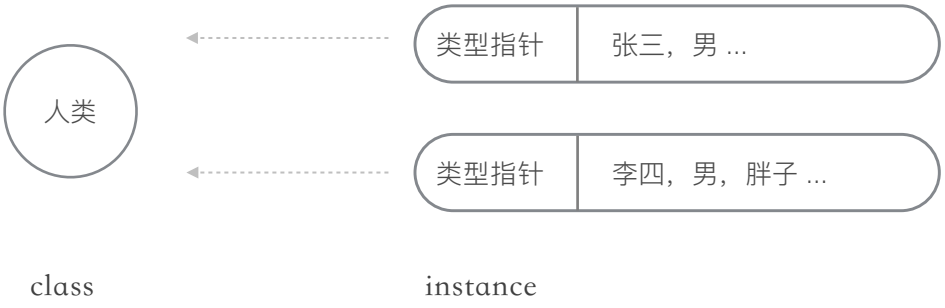
1.1 类型

从抽象角度看，每个运行中的程序（进程）都由数量众多的“鲜活”对象组成。每个对象都有其独特的状态和逻辑，通过行为触发，或与其他对象交互来体现设计意图。面对如此众多的个体，作为设计师自然需要以宏观视角来规划世界。那么首先要做的，就是将所有个体分门别类归置于某个族群。

我们习惯将生物分为动物、植物，进而又有猫科、犬科等等之类的细分。通过对多个个体的研究，归纳其共同特征，抽象成便于描述的种族模版。另一方面，有了模版后，可据此创建大量行为类似的个体。所以，分类是个基础工程。

在专业术语上，我们将类别称做类型（class），而个体叫做实例（instance）。类型持有所有个体的共同行为和共享状态，而实例仅保存私有特性即可。如此，在内存空间布局上才是最高效的。

以张三、李四为例，他们属于人类的特征，诸如吃饭、走路等等，统统放到“人类”这个类型里。个人只要保存姓名、性别、胖瘦、肤色这些即可。



每个实例都会存储所属类型指针。需要时，透过它间接访问目标即可。但从外在逻辑接口看，任何实例都是“完整”的。

存活的实例对象都有个“唯一”的 ID 值。

```
>>> id(123)
4487080432

>>> id("abc")
4488899864
```

> 不同 Python 实现使用不同算法，CPython 用内存地址作为 ID 值。这意味着它只能保证在某个时间，在所有存活对象里是唯一的。不保证整个进程生命周期内是唯一的，因为内存地址会被复用。如此，ID 也就不适合作为全局身份标识。

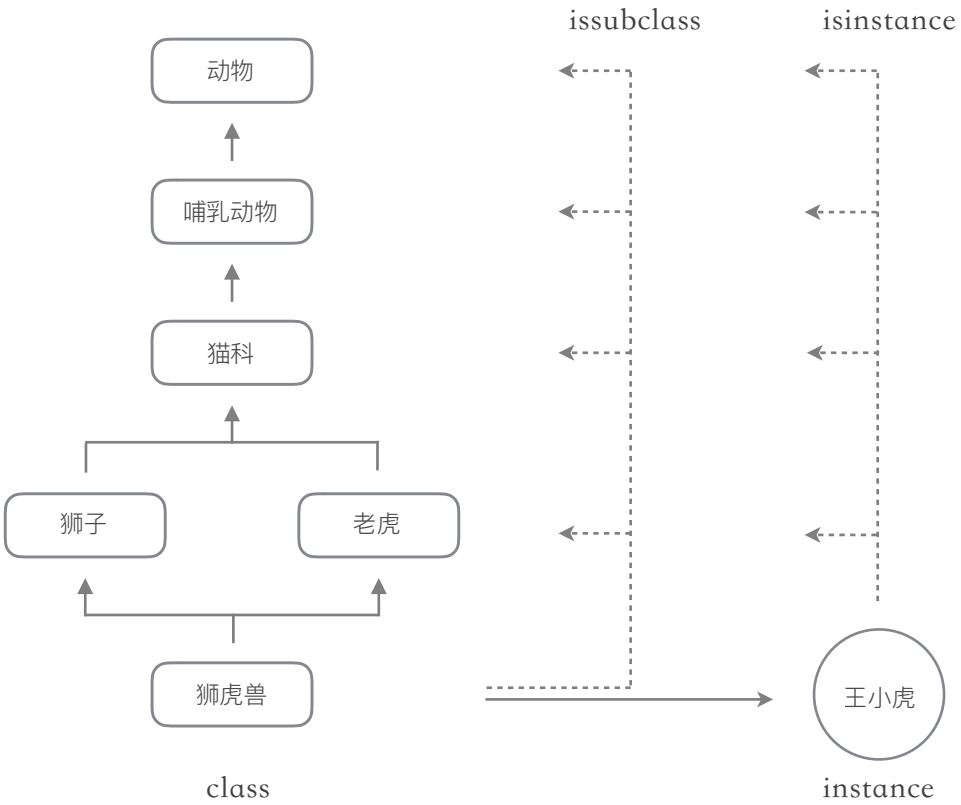
可用 type 返回实例所属类型。

```
>>> type(1)
int

>>> type(1.2)
float

>>> type("hello")
str
```

如要判断实例是否属于特定类型，须用 isinstance 函数。

```
>>> isinstance(1, int)
True

>>> isinstance(1, float)
False
```

类型之间可构成继承关系。就像老虎继承自猫科，而猫科又继承自哺乳动物，再往上还有更顶层类型。继承关系让一个类型拥有其所有祖先类型的特征。历史原因让 Python 允许多继承，也就是说可有多个父类型，好似人类同时拥有父族、母族的遗传特征。



任何类型都是其祖先类型的子类，同样对象也可以判定为其祖先类型的实例。这与面向对象三大特性中的多态有关，后文再做详述。

```
class Animal: pass                    # 动物
class Mammal(Animal): pass            # 哺乳动物（继承自动物）
class Felidae(Mammal): pass           # 猫科
class Lion(Felidae): pass             # 狮子
class Tiger(Felidae): pass            # 老虎
class Liger(Lion, Tiger): pass        # 狮虎兽（继承自两个直系父类型）
```

```
>>> issubclass(Liger, Lion)
True

>>> issubclass(Liger, Tiger)
True

>>> issubclass(Liger, Animal)          # 是任何层次祖先类型的子类。
True
```

```
>>> wxh = Liger()                     # 王小虎（狮虎兽实例）
```

```
>>> isinstance(wxh, Lion)
True

>>> isinstance(wxh, Tiger)
True

>>> isinstance(wxh, Animal)            # 是任何祖先类型的实例。
True
```

事实上，所有类型都有一个共同祖先类型 object，它为所有类型提供原始模版，以及系统所需的基本操作方式。

```
>>> issubclass(Liger, object)
True

>>> issubclass(int, object)
True
```

类型虽然是个抽象的族群概念，但在实现上也只是个普通的实例。区别在于，所有类型对象都是 type 的实例，这与继承关系无关。

```
>>> id(int)
4486772160

>>> type(int)
<class 'type'>

>>> isinstance(int, type)
True
```

怎么理解呢？想想看，单就类型对象自身而言，其本质就是个用来存储方法和字段成员的特殊容器，用同一份设计来实现这些容器才是正常思路。

就好比扑克牌，从玩法逻辑上看，J、Q、K、A 等等都有不同含义。但从材质上看，它们完全相同，没道理用不同材料去制作这些内容不同的卡片。同理，继承也只是说明两个类型在逻辑上存在关联关系。如此，所有类型对象都属于 type 实例就很好理解了。

无论是在编码，还是设计时，我们都要正确区分逻辑与实现的差异。

```
>>> type(int)
```

```
<class 'type'>

>>> type(str)
<class 'type'>

>>> type(type)
<class 'type'>
```

当然，类型对象属于特殊存在。默认情况下，它们由系统在载入时自动创建，生命周期通常与进程（虚拟机、解释器）相同，且仅有一个实例。

```
>>> type(100) is type(1234) is int    # 指向同一类型对象。
True
```

1.2 名字

在通常认知里，变量是一段有特定格式的可读写内存，变量名则是这段内存的别名。因为在编码阶段，无法使用直接或间接手段确定内存具体位置，所以使用名称符号来代替。

> 注意区分变量名和指针的不同。

接下来，静态编译和动态解释型语言对于变量名的处理方式就完全不同。类似 C、Go 这类静态编译型语言，其编译器会使用直接或间接寻址方式替代作为变量名符号。也就是说变量名不参与执行过程，是可被剔除的。但在 Python 这类动态语言里，名字和对象是两个实体。名字不但有自己的类型，要分配内存，还会介入实际执行过程。甚至可以说，名字才是其动态执行模型的基础。

如果将寻址方式比喻为顾客直接按编号寻找银行服务柜台，那么 Python 的名字就是一个接待员。任何时候，顾客都只能通过接待员间接与目标服务进行互动。从表面上看，这似乎是 VIP 待遇，但实际增加了中间环节和额外的成本开销，于性能不利。好处是，接待员与服务之间拥有更多的可调整空间。

鉴于此，名字必须与目标对象关联起来才有意义。

```
>>> x
NameError: name 'x' is not defined
```

最直接的关联操作就是赋值，而后对名字的任何引用都被解释为对目标对象的操作。

```
>>> x = 100
>>> x
100

>>> x += 2
>>> x
102
```

赋值操作步骤：

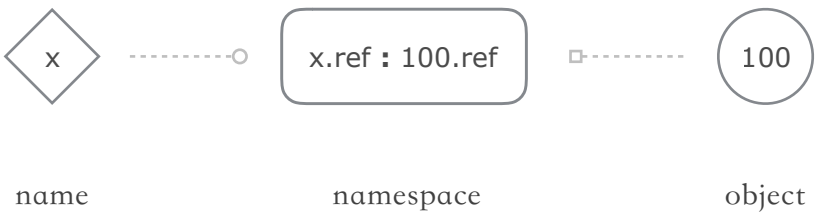
1. 准备好右值目标对象（比如上例中的整数对象 100）。
2. 准备好名字（通常是常量，保存在特定列表里。比如 x）。
3. 在名字空间（namespace）里为两者建立关联。

即便如此，名字和目标对象之间也仅只是引用关联。名字只负责找到正确的人，但对于该人的能力一无所知。鉴于在运行期才能知道名字引用的目标类型，所以 Python 是一种动态类型语言。

> Names have no type, but objects do.

名字空间

名字空间是上下文环境里专门用来存储名字和目标引用关联的容器。



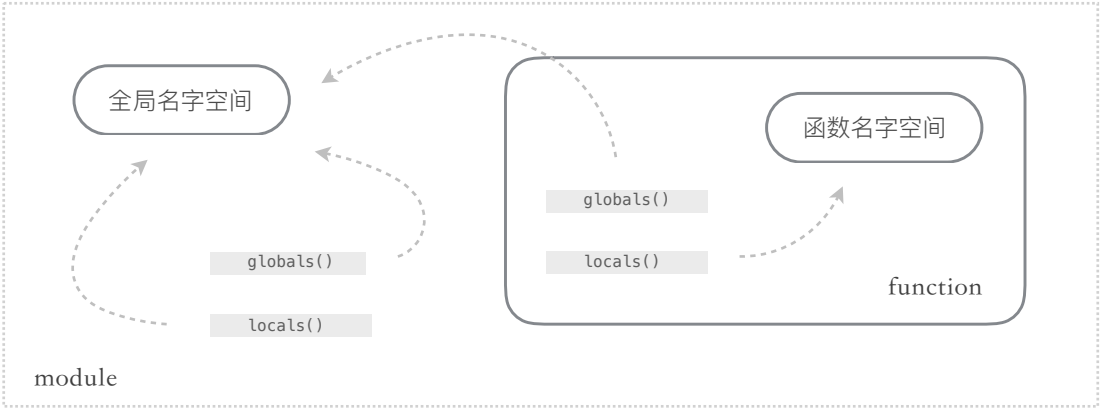
对 Python 而言，每个模块（源码文件）都有一个全局名字空间（globals）。而根据代码作用域，又有当前或本地名字空间（locals）一说。如果直接在模块级别执行，那么当前名字空间和全局名字空间相同。但在某个函数内，当前名字空间就专指函数执行栈帧（stack frame）作用域。

名字空间默认使用 dict 数据结构，由多个键值对（key/value）组成。每个 key 总是唯一。

```
>>> x = 100

>>> id(globals())                          # 在模块作用域调用。
4344493328

>>> id(locals())                           # 比对 ID 值，可以看到在模块级别，两者相同。
4344493328

>>> globals()                              # 包含模块作用域名字和对象的映射关系。
{'x': 100}
```

```
>>> def test():
        x = "hello, test!"
        print(locals())                         # 指向函数本地名字空间。
        print("locals:", id(locals()))
        print("globals:", id(globals()))        # 指向全局名字空间。

>>> test()                                      # 在函数作用域调用。
{'x': 'hello, test!'}                           # 此时，locals 输出函数栈帧名字空间。
locals : 4347899696                             # locals <> globals。
globals: 4344493328
```

globals 总是指向所在模块名字空间，而 locals 则指向当前作用域环境。



在了解名字空间后，我们甚至可以直接通过操控名字空间来建立名字和对象的引用。这与传统的变量定义方式有所不同。

> 并非所有时候都能直接操作名字空间。函数执行时会使用 FAST 缓存优化，直接修改其本地名字空间未必有效。正常编码时，应尽可能避免有修改操作。

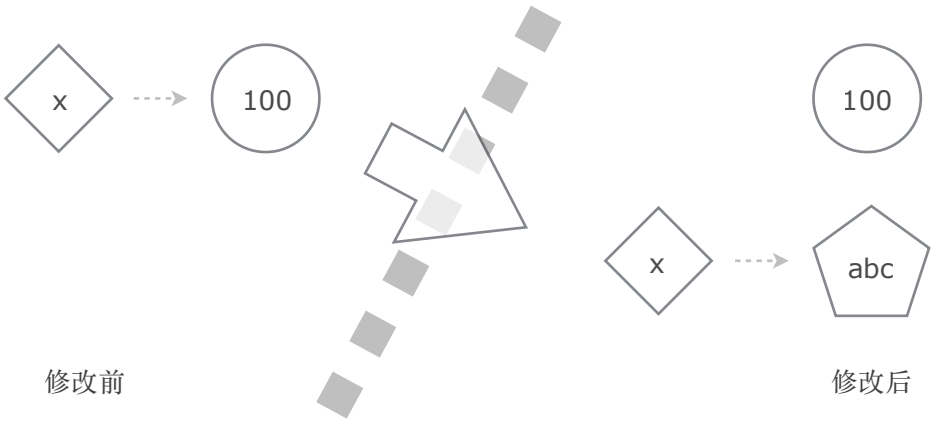
```
>>> globals()["hello"] = "hello, world!"
>>> globals()["number"] = 12345

>>> hello
'hello, world!'

>>> number
12345
```

在名字空间里，名字只是简单的字符串主键。也就是说名字自身数据结构里没有任何目标对象信息，因为引用关系是名字空间字典维护的。透过名字访问目标对象，无非是用名字作 key 去字典里读取 value，从而获得最终引用。也正因为如此，名字可随时重新关联另一个对象，而不在乎其类型是否相同。

```
>>> x = 100

>>> x
100

>>> id(x)
4319528720

>>> globals()
{'x': 100}
```

```
>>> x = "abc"               # 重新关联其他对象，没有类型限制。

>>> x
'abc'

>>> id(x)                   # 通过输出 id，可以看到关联新对象，而非修改原对象内容。
4321344792

>>> globals()               # 名字空间引用关系变更。
{'x': 'abc'}
```

赋值操作仅仅是让名字在名字空间里重新关联，而非修改原对象。



和一个名字只能引用一个对象不同，每个对象可以有多个名字，无论是在相同或不同的名字空间里。

一个人在办公室里（名字空间）可以叫老王、王大虎，或其他什么，这是单一空间里有多个名字。出办公室，在全公司（另一名字空间）范围内，还可有王处长等其他名字。

```
>>> x = 1234
>>> y = x                          # 总是按引用传递。

>>> x is y                         # 判断是否引用同一对象。
True

>>> id(x)                          # 输出 id 值来确认上述结论。
4350620272

>>> id(y)
4350620272

>>> globals()                      # 1234 有两个名字。
{'x': 1234, 'y': 1234}
```

应使用 is 语句判断两个名字是否引用同一对象。相等操作符并不能确定两个名字指向同一对象，这涉及到操作符重载，或许它被设定为比较值是否相等。

```
>>> a = 1234                # 请使用大数字。因为小数字常量会被缓存复用。
>>> b = 1234
```

```
>>> a is b               # 指向不同对象。
False

>>> a == b              # 值相等。
True
```

命名规则

名字应选用有实际含义，易于阅读和理解的字母或单词组合。

- 必须用以字母或下划线开头。
- 区分大小写。
- 不能使用保留关键字（reserved words）。

```
>>> x = 1
>>> X = "abc"

>>> x is X             # 区分大小写。
False
```

为统一命名风格，建议：

- 类型名称使用 CapWords 格式。
- 模块文件名，函数、方法成员等使用 lower_case_with_underscores 格式。
- 全局常量使用 UPPER_CASE_WITH_UNDERSCORES 格式。
- 避免与内置函数或标准库常用类型同名，易导致误解。

```
>>> OK = 200
>>> KEY_MIN = 10
>>> set_name = None
```

> 尽管 Python 3 支持用中文字符作为名字，但从习惯上讲，这并不是好选择。

保留关键字：

```
False      class        finally     is           return
```

```
True      continue    for         lambda      try
None      def         from        nonlocal    while
and       del         global      not         with
as        elif        if          or          yield
assert    else        import      pass
break     except      in          raise
```

提示：2.x 里的 exec、print，在 3.x 里已变成函数，不再是保留字。

```
>>> class = 100
SyntaxError: invalid syntax
```

如要检查动态生成代码是否违反保留字规则，可用 keyword 模块。

```
>>> import keyword

>>> keyword.kwlist
['False', 'None', ..., 'with', 'yield']

>>> keyword.iskeyword("is")
True

>>> keyword.iskeyword("print")
False
```

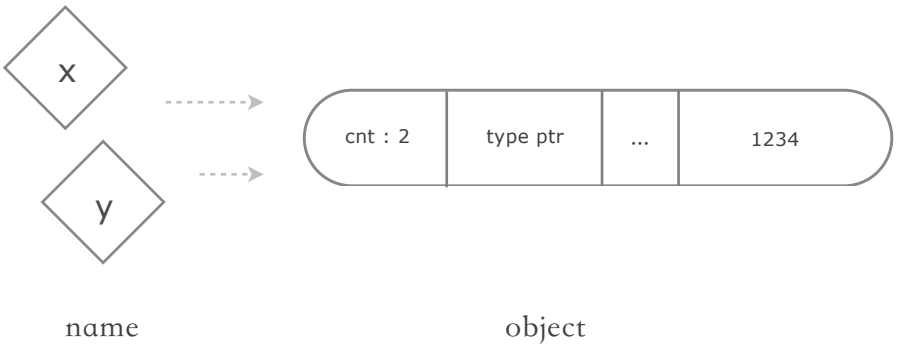
以下划线开头的名字，有特殊含义。

- 模块成员以单下划线开头（_x），属私有成员，不会被“import *”语句导入。
- 类型成员以双下划线开头，但无结尾（__x），属私有成员，会被自动重命名。
- 以双下划线开头和结尾（__x__），通常是系统成员，避免使用。
- 交互模式（shell）下，单下划线（_）返回最后一个表达式结果。

```
>>> 1 + 2 + 3
6

>>> _
6
```

1.3 内存

没有值类型、引用类型之分。事实上，每个对象都很“重”。即便是简单的整数类型，都有一个标准对象头，保存类型指针和引用计数等信息。如果是变长类型（比如 str、list 等），还会记录数据项长度，然后才是对象状态数据。




```
>>> x = 1234

>>> import sys

>>> sys.getsizeof(x)     # Python 3 里 int 也是变长结构。
28
```

总是通过名字来完成“引用传递”（pass-by-reference）。名字关联会增加计数，反之减少。如删除全部名字关联，那么该对象引用计数归零，会被系统自动回收。这就是默认的引用计数垃圾回收机制（Reference Counting Garbage Collector）。

关于 Python 是 pass-by-reference，还是 pass-by-value，会有一些不同的说法。归其原因，是因为名字不是内存位置符号造成的。如果变量（variable）不包括名字所关联的目标对象，那么就是值传递。因为传递是通过“复制”名字来实现的，这类似于复制指针，或许正确说法是 pass-by-object-reference。不过在编码时，我们通常关注的是目标对象，而非名字自身。从这点上来说，用“引用传递”更能清晰解释代码的实际意图。

基于立场不同，对某些问题会有不同的理论解释。有时候，反过来用实现来推导理论，或许更能加深理解，更好地掌握相关知识。

```
>>> a = 1234
>>> b = a

>>> sys.getrefcount(a)       # getrefcount 自身也会通过参数引用目标对象，导致计数 +1 。
3
```

```
>>> del a                       # 删除其中一个名字，减少计数。

>>> sys.getrefcount(b)
2
```

所有对象都由内存管理系统在特定区域统一分配。赋值、传参，亦或是返回局部变量都无需关心内存位置，并没有什么逃逸或者隐式复制（目标对象）行为发生。

基于性能考虑，像 Java、Go 这类语言，编译器会优先在栈（stack）上分配对象内存。但考虑到闭包、接口、外部引用等因素，原本在栈上分配的对象可能会“逃逸”到堆（heap）。这势必会延长对象生命周期，加大垃圾回收负担。所以，会有专门的逃逸分析（escape analysis），便于优化代码和算法。

Python 虚拟机虽然也有执行栈的概念，但并不会在栈上为对象分配内存。从这点上来说，可以认为所有原生对象（非 C、Cython 等扩展）都在“堆”上分配。

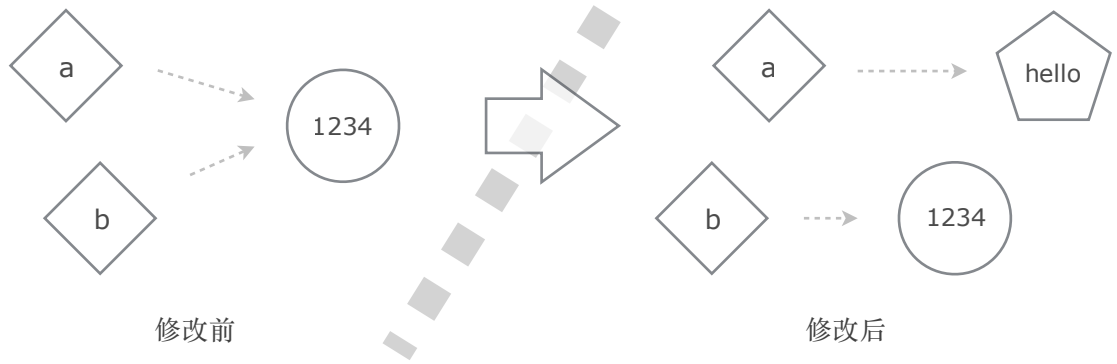
```
>>> a = 1234

>>> def test(b):
        print(a is b)        # 参数 b 和 a 都指向同一对象。

>>> test(a)
True
```

```
>>> def test():
        x = "hello, test"
        print(id(x))
        return x             # 返回局部变量。

>>> a = test()               # 对比 id 结果，确认局部变量被导出。
4371868400

>>> id(a)
4371868400
```

将对象多个名字中的某个重新赋值，并不会影响其他名字。

```
>>> a = 1234
>>> b = a
```

```
>>> a is b
True

>>> a = "hello"             # 将 a 重新关联到字符串对象。
>>> a is b                  # 此时，a、b 分别指向不同对象。
False

>>> a
'hello'

>>> b
1234
```




```
>>> a = 1234

>>> def test(b):
        print(b is a)        # True; 此时 b 和 a 指向同一对象。
        b = "hello"          # 在 locals 中，b 重新关联到字符串。
        print(b is a)        # False; 这时 b 和 a 分道扬镳。

>>> test(a)
True
False

>>> a                       # 显然对 a 没有影响。
1234
```

注意，只有对名字赋值才会变更引用关系。如下面示例，函数仅仅是透过名字引用修改列表对象自身，而并未重新关联到其他对象。

```
>>> a = [1, 2]

>>> def test(b):
        b.append(3)          # 通过名字 b 向列表追加数据。
        print(b)             # 查看修改结果。
```

```
>>> test(a)
[1, 2, 3]

>>> a                          # a 和 b 指向同一对象，自然也能获得“同步”结果。
[1, 2, 3]
```

弱引用

如果说，名字与目标对象关联构成强引用关系，会增加引用计数，会影响其生命周期。那么，弱引用（weak reference）就是减配版，在保留引用的前提下，不增加计数，也不阻止目标被回收。不过，并不是所有类型都支持弱引用。

```
class X:
    def __del__(self):               # 析构方法，观察实例被回收。
        print(id(self), "dead.")
```

```
>>> a = X()                         # 创建实例。

>>> sys.getrefcount(a)
2
```

```
>>> import weakref

>>> w = weakref.ref(a)              # 创建弱引用。

>>> w() is a                        # 通过弱引用访问目标对象。
True

>>> sys.getrefcount(a)              # 弱引用并为增加目标对象引用计数。
2
```

```
>>> del a                           # 解除目标对象名字引用，对象被回收。
4384434048 dead.

>>> w() is None                     # 弱引用失效。
True
```

标准库里另有一些相关实用函数，以及弱引用字典、集合等容器。

```
>>> a = X()
>>> w = weakref.ref(a)

>>> weakref.getweakrefcount(a)
1

>>> weakref.getweakrefs(a)
[<weakref at 0x10548f778; to 'X' at 0x10553b668>]

>>> hex(id(w))
'0x10548f778'
```

弱引用可用于一些类似缓存、监控等场合，这类“外挂”场景不应该影响到目标对象。另一个典型应用是实现 Finalizer，也就是在对象被回收时执行额外的“清理”操作。

为什么不使用析构方法（__del__）？

很简单，析构方法作为目标成员，其用途是完成该对象内部资源的清理。它无法感知，也不应该处理与之无关的外部场景。但在实际开发中，类似的外部关联会有很多，那么用 Finalizer 才是合理设计，因为这样只有一个不会侵入的观察员存在。

```
>>> a = X()

>>> def callback(w):
        print(w, w() is None)

>>> w = weakref.ref(a, callback)        # 创建弱引用时，附加回调函数。

>>> del a                               # 当回收目标对象时，回调函数被调用。
4384343488 dead.
<weakref at 0x1057f2818; dead> True
```

注意，回调函数参数为弱引用而非目标对象。另外，被调用时，目标已无法访问。

抛开对生命周期的影响不说，弱引用最大的区别在于其类函数的 callable 调用语法。不过可用 proxy 来改进，使其和普通名字引用语法保持一致。

```
>>> a = X()
>>> a.name = "Q.yuhen"
```

```
>>> w = weakref.ref(a)

>>> w.name
AttributeError: 'weakref' object has no attribute 'name'

>>> w().name
'Q.yuhen'
```

```
>>> p = weakref.proxy(a)
>>> p
<__main__.X at 0x1055ebe58>

>>> p.name                          # 直接访问目标成员。
'Q.yuhen'

>>> p.age = 60                      # 通过 proxy 直接向目标添加或设置成员。
>>> a.age
60

>>> del a                           # 同样不会影响目标生命周期。
4387316960 dead.
```

对象复制

从编程初学者的角度看，基于名字的引用传递要比值传递自然得多。试想，日常生活中，谁会因为名字被别人呼来唤去就莫名其妙克隆出一个自己来？但在一个有经验的程序员眼里，事情恐怕得反过来。

当调用函数时，我们或许仅仅是想传递一个状态，而非整个实体。这好比把自家姑娘生辰八字，外加一幅美人图交给媒人，断没有直接把人领走的道理。另一方面，现在并发编程也算日常惯例，让多个执行单元共享实例引起数据竞争（data race），也是个大麻烦。

对象的控制权该由创建者负责，而不能寄希望于接收参数的被调用方。必要时，可使用不可变类型，或对象复制品作为参数传递。除自带复制方法（copy）的类型外，可选择方法包括标准库 copy 模块，或 pickle、json 等序列化（object serialization）方案。

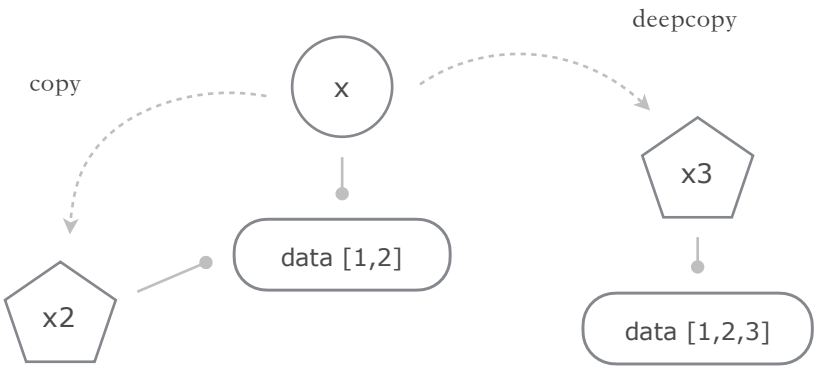
> 不可变类型包括 int、float 、str、bytes、tuple、frozenset 等。因不能改变其状态，所以被多方只读引用也没什么问题。

复制还分浅拷贝（shallow copy）和深度拷贝（deep copy）两类。对于对象内部成员，浅拷贝仅复制名字引用，而后者会递归复制所有引用成员。

```
>>> class X: pass
>>> x = X()                    # 创建实例。
>>> x.data = [1, 2]            # 成员 data 引用一个列表。
```

```
>>> import copy

>>> x2 = copy.copy(x)          # 浅拷贝。

>>> x2 is x                    # 复制成功。
False

>>> x2.data is x.data          # 但成员 data 依旧指向原列表，仅仅复制了引用。
True
```

```
>>> x3 = copy.deepcopy(x)      # 深拷贝。

>>> x3 is x                    # 复制成功。
False

>>> x3.data is x.data          # 成员 data 列表同样被复制。
False

>>> x3.data.append(3)          # 向复制的 data 列表追加数据。

>>> x3.data
[1, 2, 3]

>>> x.data                     # 没有影响原列表。
[1, 2]
```



相比 copy，序列化是将对象转换为可存储和传输的格式，反序列化则正好相反。至于格式，可以是 pickle 的二进制，也可以是 JSON、XML 等格式化文本。二进制拥有最好的性能，但从数据传输和兼容性看，JSON 更佳。

> 二进制序列化还可选择 MessagePack 等跨平台第三方解决方案。

```
>>> class X: pass

>>> x = X()
>>> x.str = "string"
>>> x.list = [1, 2, 3]
```

```
>>> import pickle

>>> d = pickle.dumps(x)          # 序列化，返回字节序列。
>>> x2 = pickle.loads(d)         # 反序列化，返回新 X 实例。

>>> x2 is x
False

>>> x2.list is x.list            # 基于字节序列构建，与原对象再无关系。
False
```

无论是哪种对象复制方案都存在一定限制，具体请参考相关文档为准。

循环引用垃圾回收

引用计数机制虽然实现简单，但可在计数归零时立即清理该对象所占内存，绝大多数时候都能高效运作。只是当两个或更多对象构成循环引用（reference cycle）时，该机制就会遭遇麻烦。因为彼此引用导致计数永不会归零，从而无法触发回收操作，形成内存泄漏。为此，另设了一套专门用来处理循环引用的垃圾回收器（以下简称 gc）作为补充。

> 单个对象也能构成循环引用，比如列表把自身作为元素存储。

相比在后台默默工作的引用计数，这个可选的附加回收器拥有更多的控制选项，包括将其临时或永久关闭。

```
class X:
    def __del__(self):
        print(self, "dead.")
```

```
>>> import gc
>>> gc.disable()                          # 关闭 gc。

>>> a = X()
>>> b = X()

>>> a.x = b                               # 构建循环引用。
>>> b.x = a

>>> del a                                 # 删除全部名字后，对象并未被回收，引用计数失效。
>>> del b
```

```
>>> gc.enable()                           # 重新启用 gc。

>>> gc.collect()                          # 主动启动一次回收操作，循环引用对象被正确回收。
<__main__.X object at 0x1009d9160> dead.
<__main__.X object at 0x1009d9128> dead.
```

虽然可用弱引用打破循环，但在实际开发时很难这么做。就本例而言，a.x 和 b.x 都需要保证目标存活，这是逻辑需求，弱引用无法确保这点。另外，循环引用可能由很多对象因复杂流程间接造成，很难被发现，自然也就无法提前使用弱引用方案。

在 Python 早期版本里， gc 不能回收包含 __del__ 的循环引用，但现在已不是问题。
另外，iPython 对于弱引用和垃圾回收存在干扰，建议用原生 Shell 或源码文件测试本节代码。

在进程（虚拟机）启动时，gc 默认被打开，并跟踪所有可能造成循环引用的对象。相比引用计数，gc 是一种延迟回收方式。只有当内部预设的阈值条件满足时，才会在后台启动。虽然可忽略该条件，强制执行回收，但不建议频繁使用。相关细节，后文再做阐述。

对某些性能优先的算法，在确保没有循环引用前提下，临时关闭 gc 可能会获得更好的性能。甚至某些极端优化策略里，会完全屏蔽垃圾回收，通过重启进程来回收资源。

例如，在做性能测试（比如 timeit）时，也会关闭 gc，避免回收操作对执行计时造成影响。

1.4 编译

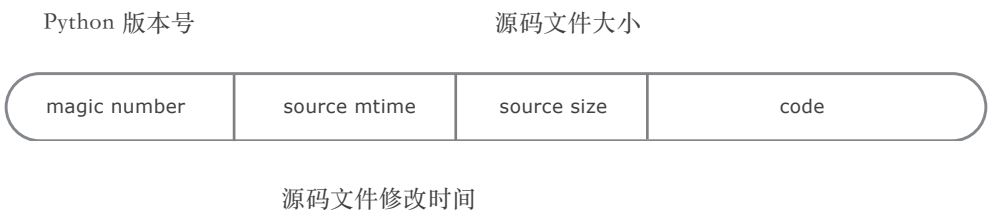
源码须编译成字节码（byte code）指令后，才能交由解释器（interpreter）解释执行。这也是 Python 性能为人诟病的一个重要原因。

字节码是中间代码，面向后端编译器或解释器。要么直接解释，要么二次编译成机器代码后执行。通常基于栈式虚拟机（stack-based VM）实现，没有寄存器等复杂概念，实现简单。且具备中立性，与硬件架构、操作系统等无关，便于将编译与平台实现分离，是跨平台语言的主流方案。

也可尝试内置即时编译（Just-In-Time compiler）的实现版本（比如 PyPy），它们提供更好的执行性能，更少的内存占用。只是相对 CPython 要滞后一些，另须注意兼容性。

除去交互模式（interactive）和手工编译。正常情况下，源码文件在被导入（import）时完成编译。编译后的字节码数据被缓存复用，并尝试保存到硬盘。

Python 3 使用专门的 __pycache__ 目录保存字节码缓存文件（.pyc）。这样在程序下次启动时，可避免再次编译，以提升导入速度。标准 pyc 文件大体上由两部分组成。头部存储有编译相关信息，在启动时，可判断源码文件是否被更新，是否需要重新编译。



下载 3.6 源码，阅读 Lib/importlib/text _bootstrap_external.py 文件里的 _code_to_bytecode 代码，可看到字节码头部信息构成。该文件还有“Finder/loader utility code”注释，对相关格式和内容做了更详细解释。

除作为执行指令的字节码外，还有很多元数据，共同组成执行单元。从这些源数据里，可以直接获得参数、闭包等诸多信息。

```
def add(x, y):
    return x + y
```

```
>>> add.__code__
```

```
<code object add at 0x1070ce9c0>

>>> dir(add.__code__)
['co_argcount', 'co_code', 'co_consts', ..., 'co_names', 'co_nlocals', 'co_varnames']

>>> add.__code__.co_varnames
('x', 'y')

>>> add.__code__.co_code
b'|\x00|\x01\x17\x00S\x00'
```

就像无法阅读机器代码一样，对于字节码指令，同样需要借助一点手段。所幸，标准库提供了一个 dis 模块，让我们可以直接“反汇编”（disassembly）。

```
>>> import dis

>>> dis.dis(add)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
```

相比 x86-64、ARM 等汇编指令，上面的输出结果非常易读。标准库帮助文档更是对所有指令做了详细说明。与隐藏大量细节的 Python 语言不同，字节码指令更能直接体现操作流程和细节。在后续章节，我们会大量使用该方式来验证语言特征。

> 本书第十章会对虚拟机、解释器、反汇编等作出详细解释，现在只需有个初步了解即可。

某些时候，需要手工完成编译操作。

```
source = """
    print("hello, world!")
    print(1 + 2)
"""
```

```
>>> code = compile(source, "demo", "exec")      # 编译，提供一个文件名便于输出提示。

>>> dis.show_code(code)                         # 输出相关信息。
Filename:          demo
Constants:
   0: 'hello, world!'
   1: 1
```

```
   2: 2
   3: None
   4: 3
Names:
   0: print

>>> dis.dis(code)
  2           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('hello, world!')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_NAME                0 (print)
             10 LOAD_CONST               4 (3)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               3 (None)
             18 RETURN_VALUE
```

```
>>> exec(code)
hello, world!
3
```

除内置 compile 函数外，标准库还有编译源码文件的相关模块。可用代码中调用，或直接在命令行执行。

> 可直接发布 pyc 文件用来执行，这算是对源码的一种简单保护。

```
>>> import py_compile, compileall

>>> py_compile.compile("main.py")
'__pycache__/main.cpython-36.pyc'

>>> compileall.compile_dir(".")
Listing '.'...
Compiling './test.py'...
True
```

```
$ python -m compileall .
```

1.5 执行

除预先准备好的代码外，还可在运行期动态执行一些“未知”代码。当然，这并不局限于从服务器或其他什么地方下载新的功能模块，常被用来实现动态生成设计。

以标准库 namedtuple 实现为例，它基于字符串模版在运行期构建新的类型。类似的还有使用频繁的 ORM 框架，完全可读取数据表结构动态生成数据模型和映射关系，极大减少了那堆看上去不太美观的架构（schema）定义代码。

不同于完全由专业开发人员编写的插件，动态执行逻辑可能来自于用户输入（比如计算公式），也可能是运维人员后台推送（新的加密方式或数据格式）。从这点上来说，它为架构设计和功能扩展带来更大的灵活性。

> 动态执行与是否是动态语言无关。就算是 Java、C# 这类静态语言，也可通过 CodeDOM、Emit 等方式动态生成、编译和执行代码。

```
>>> import collections

>>> User = collections.namedtuple("User", "name,age", verbose=True)   # 动态生成 User 类型。

class User(tuple):
    _fields = ('name', 'age')
    ...
```

```
>>> User
__main__.User

>>> u = User("Q.yuhen", 60)

>>> u
User(name='Q.yuhen', age=60)
```

且不管代码生成过程如何，其结果要么以模块导入方式执行，要么通过调用 eval、exec 函数执行。这两个内置函数使用很简单，eval 执行单个表达式，exec 应对复杂代码块。它们接受字符串或已编译好的代码对象（code）作为参数。如果直接传入字符串，会在编译和执行前检查是否符合 Python 语法规则。

```
>>> s = input()                          # 输入表达式，比如数学运算。
(1 + 2) * 3
```

```
>>> eval(s)                             # 执行表达式。
9
```

```
>>> s = """
    def test():
        print("hello, world!")

    test()
"""

>>> exec(s)                             # 执行多条语句。
hello, world!
```

无论选择哪种方式执行，都必须有相应的上下文环境。默认情况下，直接使用当前 globals、locals 名字空间。如同普通代码那样，从中读取目标对象，或写入新值。

```
>>> x = 100

>>> def test():
        y = 200
        print(eval("x + y"))     # 从上下文名字空间读取 x、y 值。

>>> test()
300
```

```
>>> def test():
        print("test:", id(globals()), id(locals()))
        exec("print('exec:', id(globals()), id(locals()))")    # 对比名字空间 id 值。

>>> test()
test: 4471161768 4468227456
exec: 4471161768 4468227456
```

有了操作上下文名字空间的能力，动态代码就可向外部环境“注入”新的成员。比如说构建新的类型，或导入新的算法逻辑等。最终达到将动态逻辑或其结果融入，成为当前体系组成部分的设计目标。

```
>>> s = """
    class X: pass

    def hello():
```

```
        print("hello, world!")
"""

>>> exec(s)                          # 执行后，会在名字空间内生成 X 类型和 hello 函数。
```

```
>>> X
__main__.X

>>> X()
<__main__.X at 0x10aafe8d0>

>>> hello()
hello, world!
```

某些时候，因动态代码来源的不确定性，基于安全考虑，必须对执行过程进行隔离，阻止其直接读写环境数据。如此，就需要显式传入容器对象作为动态代码的专用名字空间，以类似简易沙箱（sandbox）方式执行。

根据需要，分别提供 globals、locals 参数，也可共用同一名字空间。
为保证代码正确执行，会自动导入 __builtins__ 模块，以便调用内置函数等成员。

```
>>> g = {"x": 100}
>>> l = {"y": 200}

>>> eval("x + y", g, l)                 # 为 globals、locals 分别指定字典。
300
```

```
>>> ns = {}

>>> exec("class X: pass", ns)           # globals、locals 共用同一字典。

>>> ns
{'X': X,  '__builtins__': {...}}
```

当同时提供两个名字空间参数时，默认总是 locals 优先，除非在动态代码里明确指定使用 globals 作用域。这涉及到多层次名字搜索规则 LEGB。在本书后续内容里，你会看到搜索规则和名字空间一样无处不在，有着巨大的影响力。

```
>>> s = """
    print(x)             # locals.x
```

```
    global y             # globals.y
    y += 100

    z = x + y            # locals.z = l.x + g.y
"""
```

```
>>> g = {"x": 10, "y": 20}
>>> l = {"x": 1000}

>>> exec(s, g, l)
1000

>>> g
{'x': 10, 'y': 120}

>>> l
{'x': 1000, 'z': 1120}
```

前文曾提及，在函数作用域内，locals 函数总是返回执行栈帧（stack frame）名字空间。因此，即便我们显式提供 locals 名字空间，也无法将其“注入”到动态代码的函数内。

```
>>> s = """
    print(id(locals()))            # 我们提供的 locals 参数。

    def test():
        print(id(locals()))        # 函数调用栈帧名字空间。

    test()
"""
```

```
>>> ns = {}

>>> id(ns)
4473689720

>>> exec(s, ns, ns)               # 显然，test.locals 和我们提供的 ns locals 不同。
4473689720
4474406808
```

如果是“动态下载”的源码文件，可尝试用标准库 runpy 导入执行。

2. 内置类型

相比自定义类型（user-defined），内置类型（built-ins）算是特权阶层。除去它们是复合数据结构基本构成单元以外，最重要的是享受编译器和虚拟机（运行时）深度支持。比如，核心级别的指令和性能优化，专门设计的高效缓存，以及垃圾回收时特别对待等等。

作为“自带电池”（batteries included）的 Python，在这点上可谓丰富之极。很多时候，只用基本数据类型就可完成相对复杂的算法逻辑，更勿用说标准库里的那些后备之选。

对于内置的基本数据类型，可简单分为数字、序列、映射和集合等几类。另根据其实例内容是否可被更改，又有可变（mutable）和不可变（immutable）之别。

名称	分类	可变类型
int	number	N
float	number	N
str	sequence	N
bytes	sequence	N
bytearray	sequence	Y
list	sequence	Y
tuple	sequence	N
dict	mapping	Y
set	set	Y
frozenset	set	N

在标准库 collections.abc 里列出了相关类型的抽象基类，可据此判断类型基本行为方式。

```
>>> import collections.abc

>>> issubclass(str, collections.abc.Sequence)
True

>>> issubclass(str, collections.abc.MutableSequence)
False
```

日常开发时，应优先选择内置类型（含标准库）。除基本性能考虑外，还可得到跨平台兼容性，以及升级保障。轻易引入不成熟的第三方代码，会提升整体复杂度，增加潜在错误风险，更不便于后续升级和代码维护。

2.1 整数

Python 3 将 2.7 里的 int、long 两种整数类型合并为 int，默认采用变长结构。虽然这会导致更多的内存开销，但胜在简化了语言规则。

同样不再支持表示 long 类型的 L 常量后缀。

变长结构允许我们创建超大天文数字，理论上仅受可分配内存大小限制。

```
>>> x = 1

>>> type(x)
int

>>> sys.getsizeof(x)
28
```

```
>>> y = 1 << 10000

>>> y
199506311688075838488374216268358508382349683188619 24...04792596709376

>>> type(y)
int

>>> sys.getsizeof(y)
1360
```

从输出结果看，尽管都是 int 类型，但 x 和 y 所占用内存大小差别巨大。在底层实现上，通过不定长结构体（variable length structure）按需分配内存。

对于较长的数字，人们为了便于阅读，会以千分位等方式进行分隔。但因逗号在 Python 语法中有特殊含义，所以改用下划线表示，且不限分隔位数。

```
>>> 78,654,321          # 表示 tuple 语法，不能用做千分位表达。
(78, 654, 321)
```

```
>>> 78_654_321
78654321
```

```
>>> 786543_21
78654321
```

除十进制外，数字还可以二进制（bin）、八进制（oct），以及十六进制（hex）表示。下划线分隔符号同样适用于这些进制的数字常量。

二进制可用来设计类似位图（bitmap）这类开关标记类型，系统管理命令 chmod 使用八进制设置访问权限，至于十六进制常见于反汇编等逆向操作。

> 八进制不再支持 012 这样的格式，只能以 0o（或大写）前缀开头。

```
>>> 0b110011       # bin
51

>>> 0o12           # oct
10

>>> 0x64           # hex
100

>>> 0b_11001_1
51
```

转换

内置多个函数将整数转换为指定进制的字符串，反向操作用 int。它默认识别为十进制，会忽略空格、制表符等多余字符。如指定进制参数，还可省略字符串前缀。

```
>>> bin(100)
'0b1100100'

>>> oct(100)
'0o144'

>>> hex(100)
'0x64'
```

```
>>> int("0b1100100", 2)
100
```

```
>>> int("0o144", 8)
100

>>> int("0x64", 16)
100

>>> int("64", 16)           # 省略进制前缀。
100
```

```
>>> int(" 100 ")            # 忽略多余空白字符。
100

>>> int("\t100\t")
100
```

当然，用 eval 也能完成转换，无非是将字符串当作常量表达式而已。但相比直接以 C 实现的转换函数，性能要差很多，毕竟动态运行需要额外的编译和执行开销。

```
>>> x = eval("0o144")

>>> x
100
```

还有一种转换操作是将整数转换为字节数组，常见于二进制网络协议和文件读写。在这里需要指定字节序（byte order），也就是常说的大小端（big-endian, little-endian）。

无论什么类型的数据，在系统底层都是以字节方式存储。以整数 0x1234 为例，可分做两个字节，高位字节 0x12，低位 0x34。不同硬件架构会采取不同的存储顺序，高位在前（big-endian）或低位在前（little-endian），这与其设计有关。

日常使用较多的 Intel x86、AMD64 都采用小端。但 ARM 则两者都支持，由具体的设备制造商（高通、三星等）指定。另外，TCP/IP 网络字节序采用大端，这属协议定义，与硬件架构和操作系统无关。

转换操作须指定目标字节数组大小，考虑到整数类型是变长结构，故通过二进制位长度计算。另调用 sys.byteorder 获取当前系统字节序。

```
>>> x = 0x1234
>>> n = (x.bit_length() + 8 - 1) // 8        # 计算按 8 位对齐所需字节数。
```

```
>>> b = x.to_bytes(2, sys.byteorder)
>>> b.hex()
'3412'
```

```
>>> hex(int.from_bytes(b, sys.byteorder))
'0x1234'
```

运算符

支持常见数学运算，但要注意除法在 Python 2 和 3 里的差异。

Python 2.7

```
>>> 3 / 2
1

>>> type(3 / 2)
<type 'int'>
```

Python 3.6

```
>>> 3 / 2
1.5

>>> type(3 / 2)
<class 'float'>
```

除法运算分单斜线和双斜线两种。单斜线称作 True division，无论是否整除，总是返回浮点数。而双斜线 Floor division 会截掉小数部分，仅返回整数结果。

```
>>> 4 / 2           # true division
2.0

>>> 3 // 2          # floor division
1
```

如要获取余数，可用取模运算符（mod）或 divmod 函数。

```
>>> 5 % 2
```

```
1

>>> divmod(5, 2)
(2, 1)
```

另一个不同之处是，Python 3 不再支持数字和非数字类型的大小比较操作。

Python 2.7

```
>>> 1 > ""
False

>>> 1 < []
True
```

Python 3.6

```
>>> 1 > ""
TypeError: '>' not supported between instances of 'int' and 'str'

>>> 1 < []
TypeError: '<' not supported between instances of 'int' and 'list'
```

布尔

布尔是整数类型子类，也就算说 True、False 可直接当做数字使用。

```
>>> issubclass(bool, int)
True

>>> isinstance(True, int)
True
```

```
>>> True == 1
True

>>> False == 0
True

>>> True + 1
2
```

布尔转换时，数字零、空值（None）、空序列和空字典被视作 False，反之为 True。

```
>>> data = (0, 0.0, None, "", list(), tuple(), dict(), set(), frozenset())

>>> any(map(bool, data))
False
```

对于自定义类型，可通过重写 __bool__ 或 __len__ 方法影响 bool 转换结果。

枚举

Python 语言规范里并没有枚举（enum）定义，而是采用标准库实现。

在多数语言里，枚举是面向编译器，类似数字常量的存在。但到了 Python 这里，事情变得有些复杂。首先，需要定义枚举类型，随后由内部代码生成对应的枚举值。

```
>>> import enum

>>> Color = enum.Enum("Color", "BLACK YELLOW BLUE RED")

>>> isinstance(Color.BLACK, Color)
True
```

```
>>> list(Color)
[<Color.BLACK: 1>, <Color.YELLOW: 2>, <Color.BLUE: 3>, <Color.RED: 4>]
```

并没有规定枚举值就必须是整数。通过继承 Enum，可将其指定为任何类型。

```
>>> class X(enum.Enum):
        A = "a"
        B = 100
        C = [1, 2, 3]

>>> X.C
<X.C: [1, 2, 3]>
```

枚举类型内部以字典方式实现，每个枚举值都有 name 和 value 属性。可通过名字或值查找对应的枚举实例。

```
>>> X.B.name
'B'

>>> X.B.value
100
```

```
>>> X["B"]                      # by-name
<X.B: 100>

>>> X([1, 2, 3])                # by-value
<X.C: [1, 2, 3]>
```

```
>>> X([1, 2])
ValueError: [1, 2] is not a valid X
```

按照字典规则，值（value）可以相同，但名字（name）不许重复。

```
>>> class X(enum.Enum):
        A = 1
        B = 1
```

```
>>> class X(enum.Enum):
        A = 1
        A = 2

TypeError: Attempted to reuse key: 'A'
```

只是当值相同时，无论基于名字还是值查找，都返回第一个定义项。

```
>>> class X(enum.Enum):
        A = 100
        B = "b"
        C = 100

>>> X.A
<X.A: 100>
```

```
>>> X.C
<X.A: 100>

>>> X["C"]
<X.A: 100>

>>> X(100)
<X.A: 100>
```

如要避免值相同的枚举定义，可用 enum.unique 装饰器。

相比传统枚举常量，标准库 enum 提供了丰富的扩展功能，包括自增数字（auto）、标志位（Flag），以及整型枚举等。只是这些需额外的内存和性能开销，算是各有利弊。

内存

对于常用小数字，系统初始化时会预先缓存。稍后使用，直接将名字关联到这些预缓存对象即可。如此，可避免对象创建过程，提高性能，且节约内存开销。

以 Python 3.6 为例，其预缓存范围是 [-5, 256]。
具体过程请阅读 Objects/longobject.c : _PyLong_Init 源码。

```
>>> a = -5
>>> b = -5
>>> a is b
True

>>> a = 256
>>> b = 256
>>> a is b
True
```

如果超出范围，那么每次都要新建对象，这其中自然包括内存分配等操作。

```
>>> a = -6
>>> b = -6
>>> a is b
False
```

```
>>> a = 257
>>> b = 257
>>> a is b
False
```

如果用 Python 开发大数据等应用，免不了要创建并持有海量数字对象（比如超大 ID 列表）。Python 2.7 对回收后的整数复用缓存区不作收缩处理，这会导致大量闲置内存驻留。而 Python 3 则改进了此设计，极大减少了内存占用，这也算是迁移到新版的一个理由。

> 公平起见，下面测试代码在 Ubuntu 16.04 下分别以 Python 2.7 和 Python 3.6 运行，输出不同阶段 RSS 内存大小。示例中使用了第三方库 psutil。

```
from __future__ import print_function
import psutil


def mem_usage():                            # 输出进程 RSS 内存大小。
    m = psutil.Process().memory_info()
    print(m.rss >> 20, "MB")


mem_usage()                                 # 起始内存占用。

x = list(range(10000000))                   # 使用列表持有海量数字对象。
mem_usage()                                 # 输出海量数字造成的内存开销。

del x                                       # 删除列表，回收数字对象。
mem_usage()                                 # 输出回收后内存占用。
```

输出：

```
$ python2.7 ./main.py
11  MB
323 MB
246 MB
```

```
$ python3.6 ./main.py
11  MB
397 MB
11  MB
```

2.2 浮点数

默认 float 类型存储双精度（double）浮点数，可表达 16 到 17 个小数位。

```
>>> 1/3
0.3333333333333333

>>> 0.1234567890123456789
0.12345678901234568
```

从实现方式看，浮点数以二进制存储十进制数的近似值。这可能会导致执行结果与编码预期不符，造成算法结果不一致性缺陷。对精度有要求的场合，应选择固定精度类型。

可通过 float.hex 方法输出实际存储值的十六进制格式字符串，以检查执行结果为何不同。另外，还可用该方式实现浮点值的精确传递，避免精度丢失。

```
>>> 0.1 * 3 == 0.3
False

>>> (0.1 * 3).hex()           # 显然两个存储内容并不相同。
0x1.3333333333334p-2

>>> (0.3).hex()
0x1.3333333333333p-2
```

```
>>> s = (1/3).hex()

>>> float.fromhex(s)          # 反向转换回浮点数。
0.3333333333333333
```

对于简单比较操作，可尝试将浮点数限制在有效固定精度内。但考虑到 round 算法实现的一些问题，更精确做法是用 decimal.Decimal 类型。

```
>>> round(0.1 * 3, 2) == round(0.3, 2)    # 避免不确定性，左右都使用固定精度。
True

>>> round(0.1, 2) * 3 == round(0.3, 2)    # 将 round 返回值作为操作数，导致精度再次丢失。
False
```

不同类型的数字之间，可直接进行加减法和比较等运算。

```
>>> 1.1 + 2
3.1

>>> 1.1 < 2
True

>>> 1.1 == 1
False
```

转换

将整数或字符串转换为浮点数很简单，且能自动处理字符串内正负符号和空白符。只是超出有效精度时，结果与字符串内容存在差异。

```
>>> float(100)
100.0

>>> float("-100.123")                # 符号
-100.123

>>> float("\t 100.123\n")            # 空白符
100.123

>>> float("1.234E2")                 # 科学记数法
123.4
```

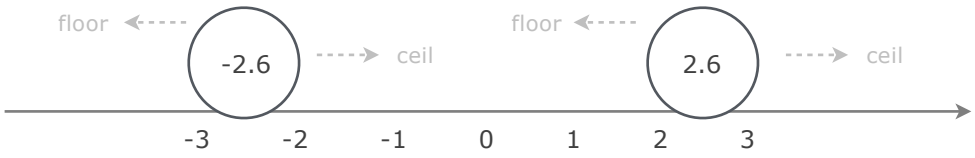
```
>>> float("0.12345678901234567890")
0.12345678901234568
```

反过来，将浮点数转换为整数时，有多种不同方案可供选择。可截掉（int, trunc）小数部分，或分别往大（ceil）、小（floor）方向取临近整数。

```
>>> int(2.6), int(-2.6)
(2, -2)
```

```
>>> from math import trunc, floor, ceil
```

```
>>> trunc(2.6), trunc(-2.6)              # 截断小数部分。
(2, -2)

>>> floor(2.6), floor(-2.6)              # 往小数字方向取最近整数。
(2, -3)

>>> ceil(2.6), ceil(-2.6)                # 往大数字方向取最近整数。
(3, -2)
```



十进制浮点数

相比 float 基于硬件的二进制浮点类型，decimal.Decimal 是十进制实现，最高可提供 28 位有效精度。能准确表达十进制数和运算，不存在二进制近似值问题。

```
>>> 1.1 + 2.2                            # 结果与 3.3 近似。
3.3000000000000003

>>> (0.1 + 0.1 + 0.1 - 0.3) == 0         # 同样二进制近似值计算结果与十进制预期不符。
False
```

```
>>> from decimal import Decimal

>>> Decimal("1.1") + Decimal("2.2")
Decimal('3.3')

>>> (Decimal("0.1") + Decimal("0.1") + Decimal("0.1") - Decimal("0.3")) == 0
True
```

在创建 Decimal 实例时，应该传入一个准确的数值，比如整数或字符串等。如果是 float 类型，那么要知道在构建之前，其精度就已丢失。

```
>>> Decimal(0.1)
```

```
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> Decimal("0.1")
Decimal('0.1')
```

在需要时，可通过上下文（context）修改 Decimal 默认的 28 位精度。

```
>>> from decimal import Decimal, getcontext

>>> getcontext()
Context(prec=28, ...)

>>> getcontext().prec = 2

>>> Decimal(1) / Decimal(3)
Decimal('0.33')
```

或者用 localcontext 限制某个区域的精度。

```
>>> from decimal import localcontext

>>> getcontext().prec = 28

>>> with localcontext() as ctx:            # 在该范围内将精度修改为 2。
        ctx.prec = 2
        print(getcontext().prec)
        print(Decimal(1) / Decimal(3))

2
0.33

>>> getcontext().prec                      # 不会影响外部精度。
28
```

除非有明确需求，否则不要用 Decimal 替代 float，要知道前者运算速度要慢许多。

四舍五入

同样因为近似值和精度问题，造成对 float 进行“四舍五入”（round）操作存在不确定性，其结果会导致一些不易察觉的陷阱。

```
>>> round(0.5)          # 5 舍
0

>>> round(1.5)          # 5 入
2
```

首先，按照 round 算法规则，按临近数字（舍入后）的距离远近来考虑是否进位。如此，四舍六入就是确定的，相关问题都集中在两边距离相等的 5 是否进位。

> 以 0.4 为例，其舍入后的相邻数字是 0 和 1，从距离上看自然是 0 更近一些。

对于 5，还要考虑后面是否还有小数位。如果有，那么左右距离就不可能是相等的，这自然是要进位的。

```
>>> round(0.5)            # 与 0、1 距离相等，不确定。
0

>>> round(0.500001)       # 哪怕 5 后面的小数部分再小，那也表示它更接近 1。
1

>>> round(1.25, 1)
1.2

>>> round(1.25001, 1)
1.3
```

剩下的，要看是返回整数还是浮点数。如果是整数，取临近的偶数。

```
>>> round(0.5)           # 0 ---> 0.5 ---> 1
0

>>> round(1.5)           # 1 ---> 1.5 ---> 2
2

>>> round(2.5)           # 2 ---> 2.5 ---> 3
2
```

> 在不同版本下，规则存在差异。比如 Python 2.7，round 2.5 返回 3.0。
> 从这点来看，我们应谨慎对待此类行为差异，并严格测试其造成的影响。

如果依旧返回浮点数，事情就变的有点莫名其妙。有些文章宣称“奇舍偶入”或“五成双”等，也就是看数字 5 前一位小数奇偶来决定是否进位，但事实未必如此。

```
>>> round(1.25, 1)        # 偶舍
1.2

>>> round(1.245, 2)       # 偶进
1.25

>>> round(2.675, 2)       # 和下面的都是奇数 7，但有舍有入。
2.67

>>> round(2.375, 2)
2.38
```

对此，官方文档 《Floating Point Arithmetic: Issues and Limitations》宣称并非错误，而属事出有因。对此，我们可改用 Decimal，按需求选取可控制的进位方案。

```
>>> from decimal import Decimal, ROUND_HALF_UP

>>> def roundx(x, n):
        return Decimal(x).quantize(Decimal(n), ROUND_HALF_UP)    # 严格按照四舍五入进行。
```

```
>>> roundx("1.24", ".1")
Decimal('1.2')

>>> roundx("1.25", ".1")
Decimal('1.3')

>>> roundx("1.26", ".1")
Decimal('1.3')
```

```
>>> roundx("1.245", ".01")
Decimal('1.25')

>>> roundx("2.675", ".01")
Decimal('2.68')

>>> roundx("2.375", ".01")
Decimal('2.38')
```

2.3 字符串

字符串（str）存储 Unicode 文本，是不可变序列类型。相比 Python 2 里的混乱，Python 3 总算顺应时代发展，将文本和二进制彻底分离。

Unicode 设计意图是为了解决跨语言和跨平台转换和处理需求，用统一编码方案容纳不同国家地区的文字，以解决传统编码方案的不兼容问题，故又称作统一码、万国码等等。

Unicode 为每个字符分配一个称作码点（code point）的整数序号，此对应编码方案叫做通用字符集（Universal Character Set, UCS）。依据编码整数长度，可分做 UCS-2 和 UCS-4 两种，后者可容纳更多字符。UCS 只规定了字符和码点的对应关系，并不涉及如何显示和存储。

UTF（Unicode Transformation Format）的作用是将码点整数转换为计算机可存储的字节格式。发展至今，有 UTF-8、UTF-16、UTF-32 等多种方案。其中 UTF-8 采用变长格式，因与 ASCII 兼容，是当下使用最广泛的一种。对于英文为主的内容，UTF-8 可获得最好的存储效率。而使用两字节等长方案的 UTF-16，有更快的处理效率，常被用作执行编码。

UTF 还可在文本头部插入称作 BOM（byte order mark）的标志来标明字节序信息，以区分大小端（BE、LE）。如此，又可细分为 UTF-16LE、UTF-32BE 等。

```
>>> s = "汉字"

>>> len(s)
2
```

```
>>> hex(ord("汉"))             # code point
0x6c49

>>> chr(0x6c49)
汉
```

```
>>> ascii("汉字")               # 对 non-ASCII 进行转义。
\u6c49\u5b57
```

字符串字面量（literal）以成对单引号、双引号，或跨行三引号语法构成，自动合并相邻字面量。支持转义、八进制、十六进制，或 Unicode 格式字符。

用单引号还是双引号，并没有什么特殊限制。如果文本内引用文字使用双引号，那么外面用单引号可避免转义，更易阅读。通常情况下，建议遵循多数编程语言惯例，使用双引号标示。除去单引号在英文句法里的特殊用途外，它还常用来表示单个字符。

```
>>> "h\x69, \u6C49\U00005B57"
hi, 汉字
```

注意：Unicode 格式大小写分别表示 16 位和 32 位整数，不能混用。

```
>>> "It's my life"                                      # 英文缩写。
>>> 'The report contained the "facts" of the case.'     # 包含引文，避免使用 \" 转义。
```

```
>>> "hello" ", " "world"                                # 合并多个相邻字面量。
hello, world
```

```
>>> """                                                 # 换行符、前导空格、空行都是组成内容。
    The Zen of Python, by Tim Peters

    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
"""
```

可在字面量前添加标志，指示构建特定格式字符串。

最常用的原始字符串（r, raw string），它将反斜线视作字符内容，而非转义标志。这在构建类似 Windows 路径、正则表达式匹配模式（pattern）之类的文法字符串时很有用。

```
>>> open(r"c:\windows\readme.txt")           # Windows 路径。

>>> re.findall(r"\b\d+\b", "a10 100")        # 正则表达式。
['100']
```

```
>>> type(u"abc")                             # 默认 str 就是 unicode，无需添加 u 前缀。
str
```

```
>>> type(b"abc")                              # 构建字节数组。
bytes
```

操作

支持用加法和乘法运算符拼接字符串。

```
>>> s = "hello"
>>> s += ", world"

>>> "-" * 10
----------
```

编译器会尝试在编译期直接计算出字面量拼接结果，避免运行时开销。不过此类优化程度有限，并不总是有效。

```
>>> def test():
        a = "x" + "y" + "z"
        b = "a" * 10
        return a, b


>>> dis.dis(test)
  2           0 LOAD_CONST               7 ('xyz')             # 直接给出结果，省略加法运算。
  3           4 LOAD_CONST               8 ('aaaaaaaaaa')      # 省略乘法运算。
```

至于多个动态字符串拼接，应优先选择 join 或 format 方式。

相比多次加法运算和多次内存分配（字符串是不可变对象），join 这类函数（方法）可预先计算出总长度，一次性分配内存，随后直接拷贝内存数据填充。另一方面，将固定内容与变量分离的模版化 format，更易阅读和维护。

```
>>> username = "qyuhen"
>>> datetime = "2017010"

>>> "/data/" + username + "/message/" + datetime + ".txt"
/data/qyuhen/message/20170101.txt
```

```
>>> "/data/{user}/message/{time}.txt".format(user = username, time = datetime)
/data/qyuhen/message/20170101.txt
```

我们用 line_profiler 对比用加法和 join 拼接 26 个大写字母的性能差异。虽然该测试不具备代表性，但可以提供一个粗略的验证方法。

```
#!/usr/bin/env python3

import string
x = list(string.ascii_uppercase)


@profile
def test_add():
    s = ""
    for c in x:
        s += c
    return s


@profile
def test_join():
    return "".join(x)


test_add()
test_join()
```

输出：

```
$ kernprof -l ./test.py && python -m line_profiler test.py.lprof

行号  #   执行次数          耗时    每次耗时   耗时百分比    源码
==============================================================================
     7                                          @profile
     8                                          def test_add():
     9         1            8        8.0       20.0        s = ""
    10        27           13        0.5       32.5        for c in x:
    11        26           18        0.7       45.0            s += c
    12         1            1        1.0        2.5        return s


行号  #   执行次数          耗时    每次耗时   耗时百分比    源码
==============================================================================
    15                                          @profile
    16                                          def test_join():
    17         1            3        3.0      100.0        return "".join(x)
```

> 有关 line_profiler 使用方法请参阅第十二章。

编写代码除保持简单外，还应具备良好的可阅读性。比如判断是否包含子串，in、not in 操作符就比 find 方法自然，更贴近日常阅读习惯。

```
>>> "py" in "python"
True

>>> "Py" not in "python"
True
```

作为序列类型，可以使用索引序号访问字符串内容，单个字符或者某一个片段。支持负索引，也就是从尾部以 -1 开始（索引 0 表示正向第一个字符）。

```
>>> s = "0123456789"

>>> s[2]
2

>>> s[-1]
9

>>> s[2:6]
2345

>>> s[2:-2]
234567
```

使用两个索引号表示一个序列片段的语法称作切片（slice），可以此返回字符串子串。但无论以哪种方式返回与原字符串内容不同的子串时，都会重新分配内存，并复制数据。不像某些语言那样，仍旧以指针引用原字符串内容缓冲区。

先看相同或不同内容时，字符串对象构建情形。

```
>>> s = "-" * 1024
>>> s1 = s[10:100]               # 片段，内容不同。
>>> s2 = s[:]                    # 内容相同。
>>> s3 = s.split(",")[0]         # 内容相同。

>>> s1 is s                      # 内容不同，构建新对象。
False
```

```
>>> s2 is s                        # 内容相同时，直接引用原字符串对象。
True

>>> s3 is s
True
```

再进一步用 memory_profiler 观察内存分配情况。

```
@profile
def test():
    a = x[10:-10]
    b = x.split(",")
    return a, b


x = "0," * (1 << 20)
test()
```

输出：

```
$ python -m memory_profiler ./test.py

 行号 #         内存使用             增量      源码
==============================================================
     4     39.082 MiB      0.000 MiB     @profile
     5                                   def test():
     6     41.086 MiB      2.004 MiB         a = x[10:-10]
     7     43.090 MiB      2.004 MiB         b = x.split(",", 1)
     8     43.090 MiB      0.000 MiB         return a, b
```

此类行为，与具体的 Python 实现版本有关，不能一概而论。

字符串类型内置丰富的处理方法，可满足大多数操作需要。对于更复杂的文本处理，还可使用正则表达式（re）或专业的第三方库，比如 NLTK、TextBlob 等。

转换

除去与数字、Unicode 码点的转换外，最常见的是在不同编码间进行转换。

Python 3 使用 bytes、bytearray 存储字节数组，不再和 str 混用。

```
>>> s = "汉字"

>>> b = s.encode("utf-16")     # to bytes
>>> b.decode("utf-16")         # to unicode string
汉字
```

如要处理 BOM 信息，可导入 codecs 模块。

```
>>> s = "汉字"

>>> s.encode("utf-16").hex()
fffe496c575b

>>> codecs.BOM_UTF16_LE.hex()              # BOM 标志。
fffe
```

```
>>> codecs.encode(s, "utf-16be").hex()     # 按指定 BOM 转换。
6c495b57

>>> codecs.encode(s, "utf-16le").hex()
496c575b
```

还有，Python 3 默认编码不再是 ASCII，所以无需额外设置。

Python 3.6

```
>>> sys.getdefaultencoding()
utf-8
```

Python 2.7

```
>>> import sys
>>> reload(sys)
>>> sys.setdefaultencoding("utf-8")

>>> b = s.encode("utf-16")
>>> b.decode("utf-16")
u'\u6c49\u5b57'

>>> type(b)
<type 'str'>
```

格式化

长期发展下来，Python 累积了多种字符串格式化方式。相比古老的面孔，人们更喜欢或倾向于使用新的特征。

Python 3.6 新增了 f-strings 支持，这在很多脚本语言里属于标配。

使用 f 前缀标志，解释器解析大括号内的字段或表达式，从上下文名字空间查找同名对象进行值替换。格式化控制依旧遵循 format 规范，但阅读体验上更加完整和简洁。

```
>>> x = 10
>>> y = 20

>>> f"{x} + {y} = {x + y}"                  # f-strings
10 + 20 = 30
```

```
>>> "{} + {} = {}".format(x, y , x + y)
10 + 20 = 30
```

表达式除运算符外，还可以是函数调用。

```
>>> f"{type(x)}"
<class 'int'>
```

完整 format 格式化以位置序号、字段名匹配替换值参数，允许对其施加包括对齐、填充、精度等控制。从某种角度看，f-strings 有点像是 format 的增强语法糖。

将两者进行对比，f-strings 类模版方式更加灵活，一定程度上将输出样式与数据来源分离。但其缺点是与上下文名字耦合，导致模版内容与代码必须保持同步修改。而 format 的序号与主键匹配方式可避开这点，只可惜它不支持表达式。

另外，对于简短的格式化处理，format 拥有更好的性能。

手工序号和自动序号

```
>>> "{0} {1} {0}".format("a", 10)
a 10 a

>>> "{} {}".format(1, 2)                                # 自动序号，不能与手工序号混用。
1 2
```

主键

```
>>> "{x} {y}".format(x = 100, y = [1,2,3])
100 [1, 2, 3]
```

属性和索引

```
>>> x.name = "jack"

>>> "{0.name}".format(x)                                # 对象属性。
jack

>>> "{0[2]}".format([1,2,3,4])                          # 索引。
3
```

宽度、补位

```
>>> "{0:#08b}".format(5)
0b000101
```

数字

```
>>> "{:06.2f}".format(1.234)                            # 保留 2 位小数。
001.23

>>> "{:,}".format(123456789)                            # 千分位。
123,456,789
```

对齐

```
>>> "[{:^10}]".format("abc")                            # 居中
[   abc    ]
```

```
>>> "[{:.<10}]".format("abc")                          # 左对齐，以点填充。
[abc.......]
```

古老的 printf 百分号格式化方式已被官方标记为“obsolete”，加上其自身固有的一些问题，可能会被后续版本抛弃，不建议使用。另外，标准库里 string.Template 功能弱，且性能也差，同样不建议使用。

池化

字符串算是进程里实例数量较多的类型之一，因为无处不在的名字就是字符串实例。

鉴于相同名字会重复出现在各种名字空间里，那么有必要让它们共享对象。内容相同，且不可变，共享不会导致任何问题。关键是可节约内存，且省去创建新实例的调用开销。

对此，Python 的做法是实现一个字符串池（intern）。池负责管理实例，使用者只需引用即可。另一潜在好处是，从池返回的字符串，只需比较指针就可知道内容是否相同，无需额外计算。可用来提升哈希表等类似结构的查找性能。

```
>>> "__name__" is sys.intern("__name__")
True
```

除了以常量方式出现的名字和字面量外，动态生成字符串一样可加入池中。如此可保证每次都引用同一对象，不会有额外的创建和分配操作。

```
>>> a = "hello, world!"
>>> b = "hello, world!"

>>> a is b                                             # 不同实例。
False

>>> sys.intern(a) is sys.intern("hello, world!")       # 相同实例。
True
```

当然，一旦失去所有外部引用，池内字符串对象会被回收。

```
>>> a = sys.intern("hello, world!")
```

```
>>> id(a)
4401879024

>>> id(sys.intern("hello, world!"))        # 有外部引用。
4401879024
```

```
>>> del a                                  # 删除外部引用后被回收。

>>> id(sys.intern("hello, world!"))        # 从 id 值不同可以看到新建，入池。
4405219056
```

字符串池实现算法很简单，就是简单的字典结构。
详情参考 Objects/unicodeobject.c : PyUnicode_InternInPlace。

做大数据处理时，可能需创建海量主键，使用 intern 有助于减少对象数量，节约大量内存。

2.4 字节数组

虽然生物都由细胞构成，但在普通人眼里，并不会将人、狮子、花草这些当做细胞看待。因为对待事物的角度决定了，我们更关心生物外在形状和行为，而不是其构成组织。

从底层实现来说，所有数据都是二进制字节序列。但为了更好地表达某个逻辑，我们会将其抽象成不同类型，一如细胞和狮子的关系。当谈及字节序列时，更多关心的是存储和传输方式；而面向类型时，则着重于其抽象属性。尽管一体两面，但从不混为一谈。

如此，当 str 瘦身只为字符串而存在，专门用于二进制原始数据处理的类型也必然会出现。早在 Python 2.6 时就引入 bytearray 字节数组，后 Python 3 又新增了只读版本 bytes。

同作为不可变序列类型，bytes 与 str 有着非常类似的操作方式。

```
>>> b"abc"
>>> bytes("汉字", "utf-8")
```

```
>>> a = b"abc"
>>> b = a + b"def"
```

```
>>> b
b'abcdef'

>>> b.startswith(b"abc")
True

>>> b.upper()
b'ABCDEF'
```

相比 bytes 的一次性内存分配，bytearray 可按需扩张，更适合作为可读写缓冲区使用。如有必要，还可为其提前分配足够内存，避免中途扩张造成额外消耗。

```
>>> b = bytearray(b"ab")

>>> len(b)
2

>>> b.append(ord("c"))
>>> b.extend(b"de")

>>> b
bytearray(b'abcde')
```

同样支持加法、乘法等运算符。

```
>>> b"abc" + b"123"              # bytes
b'abc123'

>>> b"abc" * 2
b'abcabc'
```

```
>>> a = bytearray(b"abc")

>>> a * 2
bytearray(b'abcabc')

>>> a += b"123"

>>> a
bytearray(b'abc123')
```

内存视图

如果要引用字节数据的某个片段，该怎么做？需要考虑的问题包括：是否会有数据复制行为？是否能同步修改？

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])

>>> x = a[2:5]                                    # 引用片段?
>>> x
bytearray(b'\x12\x13\x14')
```

```
>>> a[3] = 0xEE                                   # 修改原数据。
>>> a
bytearray(b'\x10\x11\x12\xee\x14\x15\x16')

>>> x                                             # 并未同步发生变更，显然是数据复制。
bytearray(b'\x12\x13\x14')
```

为什么需要引用某个片段，而不是整个对象？

以自定义网络协议为例，通常由标准头和数据体两部分组成。如要验证数据是否被修改，总不能将整个包作为参数交给验证函数。这势必要求该函数了解协议包结构，显然是不合理设计。而拷贝数据体又可能导致重大性能开销，同样得不偿失。

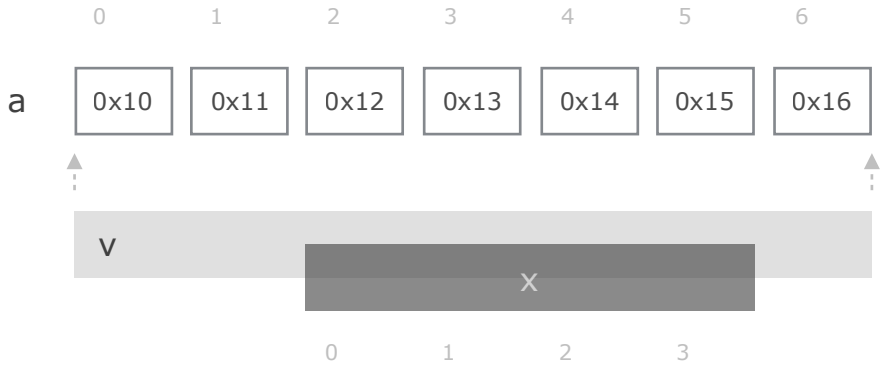
鉴于 Python 没有指针概念，外加内存安全模型限制，要做到这点似乎并不容易。为此，须借助一种名为内存视图（Memory Views）的方式来访问底层内存数据。

内存视图要求目标对象支持缓冲协议（Buffer Protocol）。它直接引用目标内存，没有额外复制行为。故此，可读取最新数据。在允许情况下，还可执行写操作。常见支持视图操作的有 bytes、bytearray、array.array，以及著名第三方库 NumPy 的某些类型。

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])

>>> v = memoryview(a)                         # 完整视图。

>>> x = v[2:5]                                # 视图片段。
>>> x.hex()
'121314'
```

```
>>> a[3] = 0xee                              # 对原数据修改，可通过视图观察到。

>>> x.hex()
'12ee14'
```

```
>>> x[1] = 0x13                              # 因为引用相同内存区域，也可通过视图修改原始数据。

>>> a
bytearray(b'\x10\x11\x12\x13\x14\x15\x16')
```



> 视图片段有自己的索引范围。读写操作以视图索引为准，但不得超出限制。

当然，能否通过视图修改数据，得看原对象是否允许。

```
>>> a = b"\x10\x11"
>>> v = memoryview(a)

>>> v[1] = 0xEE
TypeError: cannot modify read-only memory
```

如要复制视图数据，可调用 tobytes、tolist 方法。复制后的数据与原对象无关，同样不会影响视图自身。

```
>>> a = bytearray([0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16])
>>> v = memoryview(a)
>>> x = v[2:5]
```

```
>>> b = x.tobytes()            # 复制并返回视图数据。

>>> b
b'\x12\x13\x14'
```

```
>>> a[3] = 0xEE                # 对原数据修改。

>>> b                          # 不影响复制数据。
b'\x12\x13\x14'

>>> x.hex()                    # 但影响视图。
'12ee14'
```

除去上述所说，内存视图还为我们提供了一种内存管理手段。

比如说通过 bytearray 预申请很大一块内存，随后以视图方式将不同片段交由不同逻辑使用。因逻辑不能越界访问，故此可实现简易内存分配器模式。对于 Python 这种限制较多的语言，合理使用视图可在不使用 ctypes 等复杂扩展的前提下，改善算法性能。

可使用 memoryview.cast、struct.unpack 将字节数组转换为目标类型。

2.5 列表

仅从操作方式上看，列表（list）像是数组和链表的综合体。除按索引访问外，还支持插入、追加、删除等操作，完全可视作队列（queue）或栈（stack）结构使用。如不考虑性能问题，似乎是一种易用且功能完善的理想数据结构。

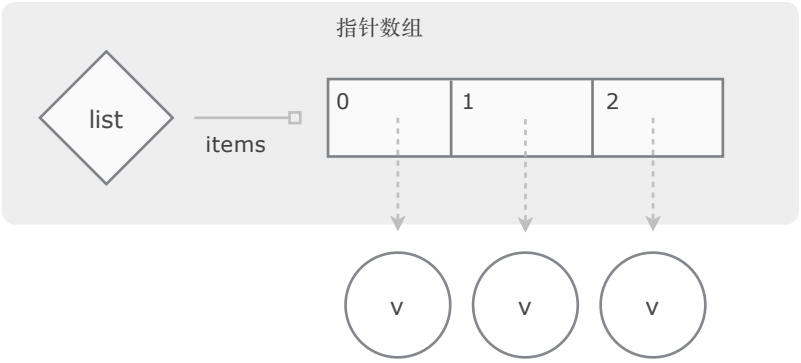
```
>>> x = [1, 2]
>>> x[1]
2

>>> x.insert(0, 0)
>>> x
[0, 1, 2]

>>> x.reverse()
>>> x
[2, 1, 0]
```

queue

```
>>> q = []

>>> q.append(1)              # 向队列追加数据。
>>> q.append(2)

>>> q.pop(0)                 # 按追加顺序弹出数据。
1

>>> q.pop(0)
2
```

对于有大量写操作的专职队列或栈，建议使用 collections.deque、queue 等类型。

列表内部结构由两部分组成，头部保存元素和内存分配计数，另引用独立指针数组。所有列表项（item）通过该数组保存指针引用，并不会嵌入元素实际内容。



作为使用频率最高的数据结构之一，列表的性能优化很重要。那么，固定长度的头部结构，很容易实现内存复用。创建时，优先从复用区获取。而当列表被回收，除非超出最大复用数量限制（默认 80），否则会被放回复用区，而不是交还内存。每次真正需要分配和释放内存的是指针数组。

用数组而非链表存储元素项引用，也有实际考虑。从读操作看，无论遍历还是基于序号访问，数组性能总是最高。尽管插入、删除等变更操作需移动内存，但也仅仅是指针复制，无关元素大小，不会有太高消耗。如果列表太大，或写操作远多于读，那么应当使用针对性的数据结构，而非通用设计的内置列表类型。

另外，指针数组内存分配算法基于元素数量和剩余空间大小，按相应比率进行扩容或收缩，而非逐项进行。如此，可避免太频繁的内存分配操作。

构建

用方括号指定显式元素的构建语法最为常用。当然，也可基于类型创建实例，接收一个可迭代对象作为初始内容。不同于数组，列表仅存储指针，对元素类型并不关心，故可以是不同类型混合。

```
>>> [1, "abc", 3.14]
[1, 'abc', 3.14]

>>> list("abc")                # iterable
['a', 'b', 'c']
```

另有一种称做推导式（comprehension）的语法。同样用方括号，但以 for 循环初始化元素，并可选 if 表达式作为过滤条件。

```
>>> [x + 1 for x in range(3)]
[1, 2, 3]

>>> [x for x in range(6) if x % 2 == 0]
[0, 2, 4]
```

其行为类似以下代码。

```
>>> d = []

>>> for x in range(6):
        if x % 2 == 0:
            d.append(x)

>>> d
[0, 2, 4]
```

> 有种称做 Pythonic 的习惯，核心是写出简洁的代码，推导式算其中一种。
> 有关推导式更多信息，可阅读后章。

无论是历史原因，还是实现方式，内置类型关注性能要多过设计。如要实现自定义列表，建议基于 collections.UserList 包装类型完成。除统一 collections.abc 体系外，最重要的是该类型重载并完善了相关操作符方法。

```
>>> list.__bases__
(object,)

>>> collections.UserList.__bases__
(collections.abc.MutableSequence,)
```

以加法操作符为例，对比不同继承的结果。

```
>>> class A(list): pass

>>> type(A("abc") + list("de"))               # 返回的是 list，而不是 A。
list
```

```
>>> class B(collections.UserList): pass

>>> type(B("abc") + list("de"))               # 返回 B 类型。
__main__.B
```

最小接口设计是个基本原则。应慎重考虑列表这种功能丰富的类型，是否适合作为基类。

操作

用加法运算符连接多个列表，乘法复制内容。

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]

>>> [1, 2] * 2
[1, 2, 1, 2]
```

注意，同时加法（或乘法）运算，但结果却有所不同。

```
>>> a = [1, 2]
>>> b = a

>>> a = a + [3, 4]                  # 新建列表对象，然后与 a 关联。
```

```
>>> a                                   # a、b 结果不同，确定 a 指向新对象。
[1, 2, 3, 4]

>>> b
[1, 2]
```

```
>>> a = [1, 2]
>>> b = a

>>> a += [3, 4]                         # 直接修改 a 内容。

>>> a                                   # a、b 结果相同，确认修改原对象。
[1, 2, 3, 4]

>>> b
[1, 2, 3, 4]

>>> a is b
True
```

> 编译器将“+=”运算符处理成 INPLACE_ADD 操作，也就是修改原数据，而非新建对象。其效果类似于执行 list.extend 方法。

判断元素是否存在时，同样习惯使用 in，而非 index 方法。

```
>>> 2 in [1, 2]
True
```

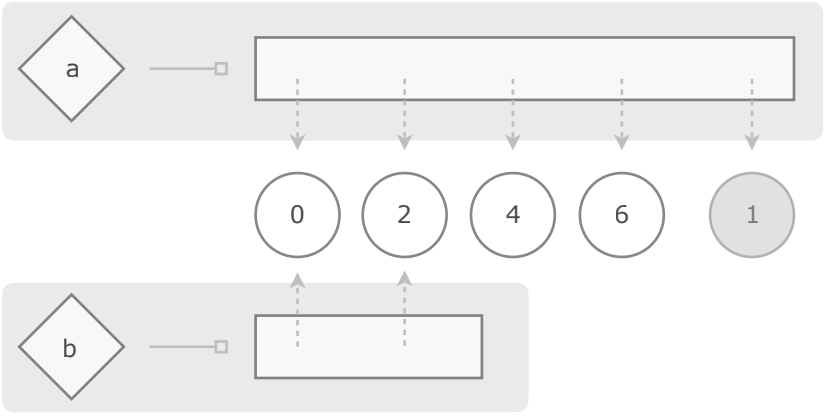
至于删除操作，可以索引序号指定单个元素，或用切片指定删除范围。

```
>>> a = [0, 1, 2, 3, 4, 5]

>>> del a[5]                    # 删除单个元素。

>>> a
[0, 1, 2, 3, 4]

>>> del a[1:3]                  # 删除范围。

>>> a
[0, 3, 4]
```

返回切片时会创建新列表对象，并复制相关指针数据到新的数组。除部分引用目标相同外，对列表自身的修改（插入、删除等）互不影响。

```
>>> a = [0, 2, 4, 6]
>>> b = a[:2]

>>> a[0] is b[0]            # 复制引用，指向同一对象。
True

>>> a.insert(1, 1)          # 对 a 列表的操作，不会影响 b。

>>> a
[0, 1, 2, 4, 6]

>>> b
[0, 2]
```

复制的是指针（引用），而不是目标元素对象。
对列表自身的修改互不影响，但对目标元素的修改是共享的。



对列表排序可设定自定义条件，比如按字段或长度等。

```
class User:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"{self.name} {self.age}"
```

```
>>> users = [User(f"user{i}", i) for i in (3, 1, 0, 2)]

>>> users
[user3 3, user1 1, user0 0, user2 2]
```

```
>>> users.sort(key = lambda u: u.age)       # 使用 lambda 匿名函数返回排序条件。

>>> users
[user0 0, user1 1, user2 2, user3 3]
```

如要返回排序复制品，可使用 sorted 函数。

```
>>> d = [3, 0, 2, 1]

>>> sorted(d)                               # 同样可指定排序条件，或倒序。
[0, 1, 2, 3]

>>> d                                       # 并未影响原列表。
[3, 0, 2, 1]
```

向有序列表插入元素，可借助 bisect 模块。它使用二分法查找适合位置，可用来实现优先级队列或一致性哈希算法等。

```
>>> d = [0, 2, 4]

>>> import bisect

>>> bisect.insort_left(d, 1)                 # 插入新元素后，依然保持有序状态。
>>> d
[0, 1, 2, 4]

>>> bisect.insort_left(d, 2)
>>> d
[0, 1, 2, 2, 4]

>>> bisect.insort_left(d, 3)
>>> d
[0, 1, 2, 2, 3, 4]
```

自定义复合类型，可通过重载比较运算符（__eq__、__lt__ 等）实现自定义排序条件。

元组

尽管两者并没有直接关系，但在操作方式上，元组（tuple）可当做列表的只读版本使用。

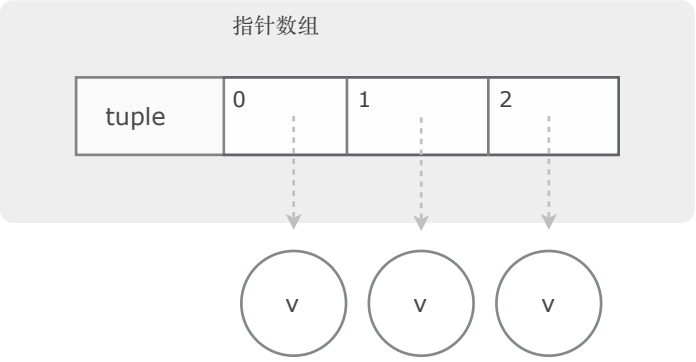
```
>>> a = [1, "abc"]
>>> b = tuple(a)

>>> b
(1, 'abc')
```

元组使用小括号语法，但要与普通括号区别开来。

```
>>> a = (1, )              # 仅一个元素的元组。
>>> type(a)
tuple

>>> b = (1)                # 普通括号。
>>> type(b)
int
```

因元组是不可变类型，它的指针数组无需变动，故内存分配乃一次性完成。另外，系统会缓存复用一定长度的元组内存。创建时，按长度提取复用，无需额外内存分配（包括指针数组）。从这点上看，元组的性能要好于列表。

> Python 3.6 缓存复用长度在 20 以内的 tuple 内存，每种 2000 上限。




```
>>> %%timeit
    [1, 2, 3]
```

```
64.8 ns ± 0.375 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

>>> %%timeit
    (1, 2, 3)

16.4 ns ± 0.156 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
```

支持与列表类似的运算符操作，但没有 INPLACE，总是返回新对象。

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)

>>> (1, 2) * 2
(1, 2, 1, 2)
```

```
>>> a = (1, 2, 3)
>>> b = a

>>> a += (4, 5)                 # 创建新 tuple，而不是修改原内容。

>>> a
(1, 2, 3, 4, 5)

>>> b
(1, 2, 3)
```

因列表支持插入、删除等操作，所以无法将某个位置（索引序号）的元素与固定内容等同起来。但元组不同，相同序号总是返回同一对象，故可为序号取个别名。

```
>>> User = collections.namedtuple("User", "name,age")      # 创建 User 类型，指定字段。

>>> issubclass(User, tuple)                                # tuple 子类。
True
```

```
>>> u = User("qyuhen", 60)

>>> u.name, u.age
qyuhen 60

>>> u[0] is u.name
True
```

对于定义纯数据类型，显然 namedtuple 要比 class 简洁。关键在于，名字要比序号更易阅读和维护，类似于数字常量定义。

数组

数组（array）与列表、元组的区别在于：元素单一类型和内容嵌入。

```
>>> import array

>>> a = array.array("b", [0x11, 0x22, 0x33, 0x44])

>>> memoryview(a).hex()                                    # 使用内存视图查看，内容嵌入而非指针。
11223344
```

```
>>> a = array.array("i")
>>> a.append(100)

>>> a.append(1.23)
TypeError: integer argument expected, got float
```

可直接存储包括 Unicode 字符在内的各种数字内容。至于复合类型，须用 struct、marshal、pickle 等转换为二进制字节后再行存储。

与列表类似，数组长度不固定，按需扩张或收缩内存。

```
>>> a = array.array("i", [1, 2, 3])

>>> a.buffer_info()                         # 返回缓冲区内存地址和长度。
(4481545888, 3)

>>> a.extend(range(100000))                 # 追加大量内容后，内存地址和长度发生变化。

>>> a.buffer_info()
(4460855296, 100003)
```

由于可指定更紧凑的数字类型，故数组可节约更多内存。再则，内容嵌入也避免了标准对象的额外开销，减少活跃对象数量和内存分配次数。

```
@profile
def test_list():
    x = []
    x.extend(range(1000000))
    return x


@profile
def test_array():
    x = array.array("l")
    x.extend(range(1000000))
    return x


test_array()
test_list()
```

输出：

```
$ python -m memory_profiler test.py

Line #      Mem usage      Increment     Line Contents
================================================================
     6     40.547 MiB      0.000 MiB     @profile
     7                                    def test_list():
     8     40.547 MiB      0.000 MiB         x = []
     9     79.129 MiB     38.582 MiB         x.extend(range(1000000))
    10     79.129 MiB      0.000 MiB         return x


Line #      Mem usage      Increment     Line Contents
================================================================
    13     32.605 MiB      0.000 MiB     @profile
    14                                    def test_array():
    15     32.609 MiB      0.004 MiB         x = array.array("l")
    16     40.547 MiB      7.938 MiB         x.extend(range(1000000))
    17     40.547 MiB      0.000 MiB         return x
```

2.6 字典

字典（dict）是内置类型中唯一的映射（mapping）结构，基于哈希表存储键值对数据。

值（value）可以是任意数据，但主键（key）必须是可哈希类型。常见的可变类型，如列表、集合等都不能作为主键使用。而元组等不可变类型，也不能引用可变类型元素。

```
>>> issubclass(list, collections.Hashable)
False
```

```
>>> issubclass(int, collections.Hashable)
True
```

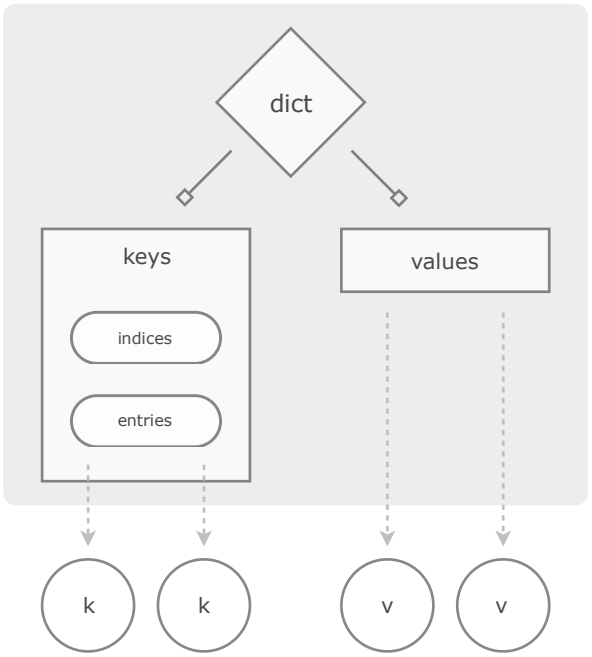
```
>>> hash((1, 2, 3))
2528502973977326415

>>> hash((1, 2, [3, 4]))                    # 包含可变列表元素。
TypeError: unhashable type: 'list'
```

哈希计算通过调用 __hash__ 方法返回整数值，用来快速比较内容是否相同。某些类型虽然有该方法，但实际无法执行，故不能作为主键使用。另外，主键对象哈希值须恒定不变，否则无法查找键值，甚至引发错误。

```
>>> callable(list().__hash__)
False
```

自定义类型默认实现了 __hash__ 和 __eq__ 方法，用于哈希和相等比较操作。前者为每个实例返回随机值，后者除非与自己比较，否则总是返回 False。可根据需要重载。

作为一种常用数据结构，以及名字空间的缘故，字典使用频率非常高。开发团队也一直致力于改进其数据结构和算法，这其中自然也包括惯用的缓存复用。



Python 3.6

> Python 3.6 借鉴 PyPy 字典设计，采用更紧凑的存储结构。keys.entries 和 values 用数组按添加顺序存储主键和值引用。实际哈希表由 keys.indices 数组承担，通过计算主键哈希值找到合适位置，然后在此存储主键在 keys.entries 的实际索引。如此，只要通过 indices 获取实际索引后，就可读取主键和值信息。
>
> 虽然该版本按添加顺序存储，但内部实现不能作为依赖条件。在后续版本中，可能有其他变化。如有明确顺序需求，建议使用 collections.OrderedDict。
>
> 系统分别缓存复用 80 个 dict 和 keys，其中包括长度为 8 的 entries 内存。对于大量小字典对象而言，直接使用，无需任何内存分配操作。回收时，有过内存扩张的都被放弃。

从字典开放地址法（open-address）的实现方式看，它并不适合用来处理大数据。轻量级方案可选用 shelve、dbm 等标准库模块，或直接采用 SQLite、LevelDB 专业数据库。

构建

以大括号键值对方式创建，或调用类型构造。

```
>>> {"a": 1, "b": 2}
{'a': 1, 'b': 2}

>>> dict(a = 1, b = 2)
{'a': 1, 'b': 2}
```

初始化键值参数也可以元组、列表等可迭代对象方式提供。

```
>>> kvs = (("a", 1), ["b", 2])

>>> dict(kvs)
{'a': 1, 'b': 2}
```

基于动态数据创建时，更多以 zip、map 函数或推导式方式完成。

```
>>> dict(zip("abc", range(3)))
{'a': 0, 'b': 1, 'c': 2}

>>> dict(map(lambda k, v: (k, v + 10), "abc", range(3)))     # 使用 lambda 匿名函数过滤数据。
{'a': 10, 'b': 11, 'c': 12}
```

```
>>> {k: v + 10 for k, v in zip("abc", range(3))}              # 使用推导式处理数据。
{'a': 10, 'b': 11, 'c': 12}
```

除直接提供内容外，某些时候，还需根据一定条件初始化字典对象。比如说，基于已有字典内容扩展，或者初始化零值等。

```
>>> a = {"a":1}
```

```
>>> b = dict(a, b = 2)                                      # 在复制 a 内容基础上，新增键值对。
>>> b
{'a': 1, 'b': 2}
```

```
>>> c = dict.fromkeys(b, 0)                                 # 仅用 b 主键，内容另设。

>>> c
{'a': 0, 'b': 0}
```

```
>>> d = dict.fromkeys(("counter1", "counter2"), 0)          # 显式提供主键。

>>> d
{'counter1': 0, 'counter2': 0}
```

相比 fromkeys 方法，推导式可完成更复杂的操作，比如额外的 if 过滤条件。

操作

字典不是序列类型，不支持序号访问，以主键为条件读取、新增或删除内容。

```
>>> x = dict(a = 1)
>>> x["a"]                      # 读取。
1

>>> x["b"] = 2                  # 修改或新增。
>>> x
{'a': 1, 'b': 2}
```

```
>>> del x["a"]                # 删除。
>>> x
{'b': 2}
```

当访问不存在主键时，会引发异常。可先以 in、not in 语句判断，或用 get 方法返回预设的默认值参数。

```
>>> x = dict(a = 1)

>>> x["b"]
KeyError: 'b'

>>> "b" in x
False
```

```
>>> x.get("b", 100)              # 主键 b 存在，返回默认值 100。
100

>>> x.get("a", 100)              # 主键 a 存在，返回字典内容。
1
```

方法 get 的默认值仅返回，不影响字典内容。但某些时候，我们还需向字典插入默认值。比如用字典存储多个计数器，那么在第一次取值时延迟初始化是很有必要的。在字典内有零值内容代表了该计数器被使用过，没有则无法记录该行为。

```
>>> x = {}
>>> x.setdefault("a", 0)          # 如果有 a，那么返回。否则新增 a = 0 键值。
0

>>> x
{'a': 0}
```

```
>>> x["a"] = 100

>>> x.setdefault("a", 0)
100
```

字典不支持加法、乘法、大小等运算，但可比较内容是否相同。

```
>>> {"b":2, "a":1} == {"a":1, "b":2}
True
```

视图

与早期版本某些方法复制并返回内容列表不同，Python 3 默认以视图（view object）方式关联字典内容。如此，即避免了复制开销，还能同步观察字典变化。

```
>>> x = dict(a = 1, b = 2)
>>> ks = x.keys()                               # 主键视图。

>>> "b" in ks                                   # 判断主键是否存在。
True

>>> for k in ks: print(k, x[k])                 # 利用视图迭代字典。
a 1
b 2
```

```
>>> x["b"] = 200                                # 修改字典内容。
>>> x["c"] = 3

>>> for k in ks: print(k, x[k])                 # 视图能同步变化。
a 1
b 200
c 3
```

字典没有类似元组那样的只读版本，无论直接传递引用还是复制品，都存在弊端。直接引用导致接收方存在修改内容的风险，而复制品又仅是一次性快照，无法获知字典本身变化。视图则不同，它能同步读取字典内容，却无法修改。且可选择不同粒度的内容传递，如此可将接收方限定为指定模式下的观察员。

```
def test(d):                    # 传递键值视图（items），只能读取，无法修改。
    for k, v in d:
        print(k, v)
```

```
>>> x = dict(a = 1)
>>> d = x.items()
```

```
>>> test(d)
a 1
```

另一方面，视图支持集合运算，弥补了字典功能上的不足。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(c = 3, b = 2)

>>> ka = a.keys()
>>> kb = b.keys()
```

```
>>> ka & kb                 # 交集：在 a、b 中同时存在。
{'b'}

>>> ka | kb                 # 并集：在 a 或 b 中存在。
{'a', 'b', 'c'}

>>> ka - kb                 # 差集：仅在 a 中存在。
{'a'}

>>> ka ^ kb                 # 对称差集：仅在 a 或仅在 b 中出现，相当于“并集 - 交集”。
{'a', 'c'}
```

利用视图集合运算，可简化一些操作。例如 update 只更新，不新增。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(b = 20, c = 3)

>>> ks = a.keys() & b.keys()               # 并集，也就是 a 中必须存在的主键。

>>> a.update({k: b[k] for k in ks})        # 利用并集提取待更新内容。

>>> a
{'a': 1, 'b': 20}
```

扩展

在标准库中，有几个字典的扩展类型可供使用。

默认字典（defaultdict）类似 setdefault 包装。当主键不存在时，调用构造参数提供的工厂函数返回默认值。

将字典直接作为对外接口时，无法保证用户是否会调用 setdefault 或 get 方法。那么，默认字典的内置初始化行为就好于对用户做额外要求。

```
>>> d = collections.defaultdict(lambda: 100)

>>> d["a"]
100

>>> d["b"] += 1

>>> d
{'a': 100, 'b': 101}
```

与 dict 内部实现无关，有序字典（OrderedDict）明确记录主键首次插入次序。

任何时候都要避免依赖内部实现，或者说遵循“显式优于隐式”规则。

```
>>> d = collections.OrderedDict()

>>> d["z"] = 1
>>> d["a"] = 2
>>> d["x"] = 3

>>> for k, v in d.items(): print(k, v)
z 1
a 2
x 3
```

与前面所说不同，计数器（Counter）对于不存在的主键返回零，但不会新增。

可通过继承并重载 __missing__ 方法新增键值。

```
>>> d = collections.Counter()

>>> d["a"]
0

>>> d["b"] += 1
```

```
>>> d
Counter({'b': 1})
```

链式字典（ChainMap）以单一接口访问多个字典内容，其自身并不存储数据。读操作按参数顺序依次查找各字典，但修改操作（新增、更新、删除）仅针对第一字典。

可利用链式字典设计多层次上下文（context）结构。

一个合理的上下文类型，需具备两个基本特征。首先是继承，所有设置可被调用链后续函数读取。其次是修改仅针对当前和后续逻辑，不应向无关的父级传递。如此，链式字典查找次序本身就是继承体现。而修改操作被限制在当前第一字典，自然也不会影响父级字典的同名主键设置。当调用链回溯时，利用 parents 属性抛弃 child 字典，也就等于丢弃了被修改的无关数据。当然，还可进一步封装 cancel、timeout 操作，通过 __getattribute__ 拦截访问并引发异常。

```
>>> a = dict(a = 1, b = 2)
>>> b = dict(b = 20, c = 30)

>>> x = collections.ChainMap(a, b)
```

```
>>> x["b"], x["c"]                          # 按顺序命中。
2  30

>>> for k, v in x.items(): print(k, v)      # 遍历所有字典。
b 2
c 30
a 1
```

```
>>> x["b"] = 999                            # 更新，命中第一字典。
>>> x["z"] = 888                            # 新增，命中第一字典。

>>> x
{'a': 1, 'b': 999, 'z': 888}, {'b': 20, 'c': 30}
```

2.7 集合

集合用于存储非重复对象。所谓非重复，除不是同一对象外，还包括值不能相等。

判重公式

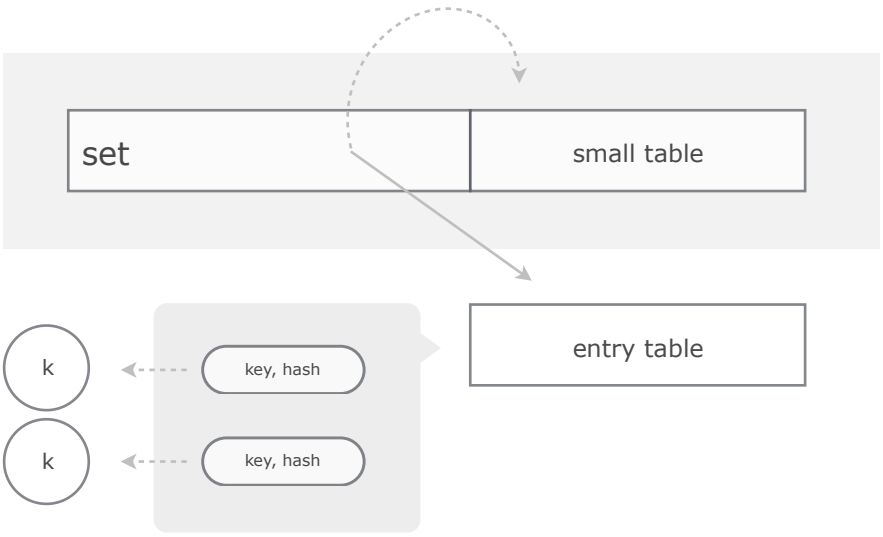
```
(a is b) or (hash(a) == hash(b) and a == b)
--------    ------------------------------
```

如果不是同一对象，那么先判断哈希值，然后比较内容。受限于哈希算法，不同内容可能返回相同哈希值（哈希碰撞），那么就有必要继续比较内容是否相同。

为什么先比较哈希值，而不直接比较内容？首先，相比大多数内容（例如字符串），整数类型哈希值比较运算，性能高得多。其次，哈希值不同，内容肯定不同，没有继续比较内容的必要。

```
>>> a = 1234
>>> b = 1234

>>> a is b                # a、b 内容相同，但非同一对象。
False

>>> s = {a}               # 创建集合，初始化元素 a。

>>> b in s                # 使用内容相同的 b 进行判重。
True
```

按操作方式，集合分可变（set）和不可变（frozenset）两个版本，其内部实现完全相同。用数组哈希表存储对象引用，这也就要求元素必须为可哈希类型。



查找元素对象时，先通过算法定位数组索引，继而比较哈希值和内容。

集合对象自带一个长度为 8 的小数组（small table），这对多数简单集合运算有益，避免额外内存分配。只有超出容量限制时，才分配更大数组内存（entry table）。集合使用频率不及列表和字典，内部没采用缓存复用策略。其实现方式决定了无序存储，标准库也没有提供有序实现。

创建

和字典一样使用大括号语法，但初始化数据非键值对。

```
>>> type({})               # 没有初始化值，表示创建一个空字典。
dict

>>> type({"a":1})          # 字典：键值对
dict

>>> type({1})              # 集合
set
```

可调用类型构造方法创建，或推导式。

```
>>> set((1, "a", 1.0))
{1, 'a'}

>>> frozenset(range(3))
frozenset({0, 1, 2})

>>> {x + 1 for x in range(6) if x % 2 == 0}
{1, 3, 5}
```

直接在不同版本间自由转换。

```
>>> s = {1}
>>> f = frozenset(s)

>>> set(f)
{1}
```

操作

支持大小、相等运算符。

```
>>> {1, 2} > {2, 1}
False

>>> {1, 2} == {2, 1}
True
```

子集判断不能使用 in、not in 语句，它仅用来检查是否包含某个特定元素。

```
>>> {1, 2} <= {1, 2, 3}          # 子集：issubset
True

>>> {1, 2, 3} >= {1, 2}          # 超集：issuperset
True
```

```
>>> {1, 2} in {1, 2, 3}          # 判断是否包含 {1, 2} 单一元素。
False
```

集合作为一个初等数学概念，其重点自然是并差集运算。合理使用这些操作，可简化算法筛选逻辑，使其具备更好的可阅读性。

交集：同时属于 A、B 两个集合。
并集：A、B 所有元素。
差集：仅属于 A，不属于 B。
对称差集：仅属于 A，加上仅属于 B，相当于“并集 - 交集”。

```
>>> {1, 2, 3} & {2, 3, 4}        # 交集：intersection
{2, 3}

>>> {1, 2, 3} | {2, 3, 4}        # 并集：union
{1, 2, 3, 4}

>>> {1, 2, 3} - {2, 3, 4}        # 差集：difference
{1}
```

```
>>> {1, 2, 3} ^ {2, 3, 4}        # 对称差集: symmetric_difference
{1, 4}
```

集合运算还可与 update 联合使用。

```
>>> x = {1, 2}
>>> x |= {2, 3}                 # update

>>> x
{1, 2, 3}
```

```
>>> x = {1, 2}
>>> x &= {2, 3}                 # intersection_update

>>> x
{2}
```

删除操作 remove 可能引发异常，可改用 discard。

```
>>> x = {2, 1}

>>> x.remove(2)
>>> x.remove(2)
KeyError: 2

>>> x.discard(2)
```

自定义类型

自定义类型虽是可哈希类型，但默认实现并不足以完成集合去重操作。

```
class User:
    def __init__(self, uid, name):
        self.uid = uid
        self.name = name

>>> issubclass(User, collections.Hashable)
True
```

```
>>> u1 = User(1, "user1")
>>> u2 = User(1, "user1")

>>> s = set()

>>> s.add(u1)
>>> s.add(u2)

>>> s
{<__main__.User at 0x10b69c780>, <__main__.User at 0x10b2eb438>}
```

根本原因是默认实现的 __hash__ 方法返回随机值，而 __eq__ 仅比较自身。为符合逻辑需要，须重载这两个方法。

```
class User:

    def __init__(self, uid, name):
        self.uid = uid
        self.name = name

    def __hash__(self):                          # 针对 uid 去重，忽略其他字段。
        return hash(self.uid)

    def __eq__(self, other):
        return self.uid == other.uid
```

```
>>> u1 = User(1, "user1")
>>> u2 = User(1, "user2")

>>> s = set()

>>> s.add(u1)
>>> s.add(u2)

>>> s
{<__main__.User at 0x10afd8b00>}

>>> u1 in s
True

>>> u2 in s                                    # 仅检查 uid 字段。
True
```