3.6

Python 学习笔记

第三版 · 上卷

前言

写这本书的时候，我已摆脱萌新身份，勉强算得上是个有经验的作者。可即便如此，依然无法保证内容正确，且满足某某人的胃口。显然，这不可能做到。

在我看来，书大抵分两类：学习和研究。学习类书籍满足日常学习和提升需要，用简练语言把问题说清楚。最关键的是有条清晰线索，把散乱的知识串联起来。学习者可据此了解前因后果，为其在茫茫网海中导航，用以探寻神秘之地。至于研究类图书或论文，应摆脱基础，摆脱语法，全心关注算法、架构、性能，乃至内部实现等等。所有这些，以思想为支撑，超脱语言窠臼，构建并完善自有体系。

不同于写散文小说，技术类文字并不好组织。自然语言易阅读，但不便描述有复杂流程分支的逻辑，易导致歧义。更何况，这其中还有各种转译带来的麻烦。有句话大家都耳熟，所谓“一图胜千言”。故应以自然语言开宗明义，阐述理论和规则，随后用代码继续对这段文字进行详细解释，毕竟代码先天有描述逻辑的优势。

很多书，尤其是英文版，习惯用大量篇幅对代码示例作各种讲解。感觉有些啰嗦，怕是很少去读第二遍，最多用记号笔划出重点。既如此，为何不相信读者能阅读并理解这些代码呢？这本就是程序猿吃饭的本钱，最多在关键位置辅以注释便可。当然，前提怕是要设定为非入门读者。好在，我一再强调写的是第二本，或者闲书。

在这本书里，对于理论层面，我会尝试说得明白些。还会引入一些类比，或许不大合适，总归其原意是为加深理解，毕竟不是所有人都能立即明白那些云里雾里的抽象理念。一如上面所言，文字与代码相辅相承，应静下心来用代码去验证那些文字背后的东西。在我眼里，代码也是种自然语言，缩排跳转仿若图形，本就是最好的笔记注释。起码它离机器尚有些距离，为人类而发明。

无论我说得多悦耳动听，本书终归是本学习笔记。算不上多专业，适合闲暇翻阅一二。

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 2.7 升级到 3.6 环境。至于初学者，建议寻本从零开始，循序渐进介绍如何编码的图书为佳。

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
微博：weibo.com/qyuhen

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。
另，为阅读方便，对输出结果作了裁剪处理，请以实际运行结果为准。

Python 始于 1989 年，早于 1995 年发布的 Java 和 Ruby。发展至今，甚至比大部分 Pythoner 年纪还大。如此长的时间跨度，其身上不可避免有各个时期的思想风格。

单就语言而论，Python 复杂度远超 C、Go ，称得上是易学难精。
但众多规则也带来编码上的便利，算是冰火两重天。

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。

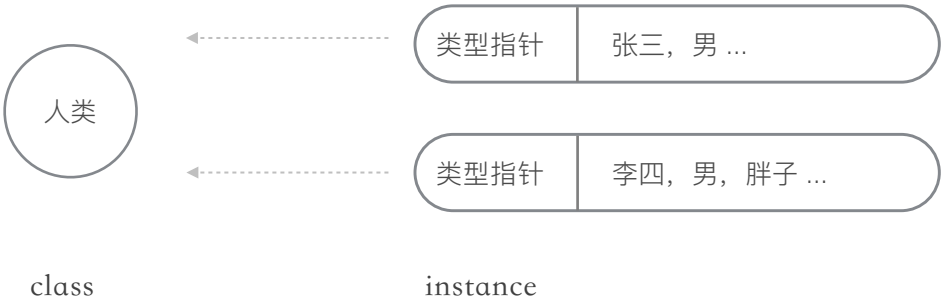
1.1 类型

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。

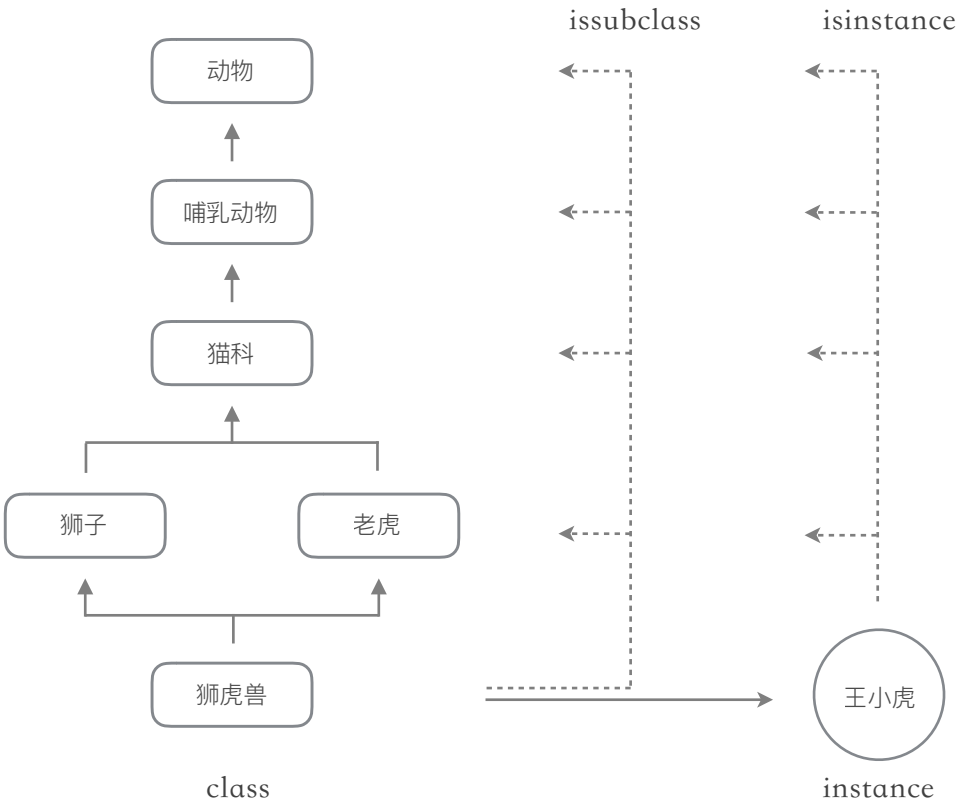
可用 type 返回实例所属类型。

```
>>> 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完全不同。静态编译型语言里的变量名，通常表达运行期某段内存，编译器会以直接或间接寻址指令替代。也就是说变量名不参与执行过程，可被剔除。但在多数解释型动态语言里，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成顾客直接按编号寻找银行服务柜台，那么 Python 名字就是一个接待员。任何时候，顾客都只能通过她间接与目标服务互动。从表面看，这似乎是 VIP 待遇，但实际增加了中间环节和额外成本开销，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：

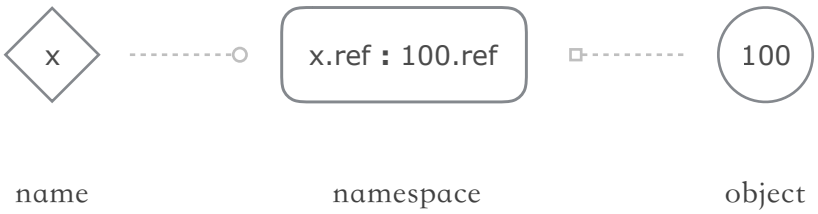
1. 准备好右值目标对象（比如上例中的整数对象 100）。
2. 准备好名字（通常是常量，保存在特定列表里。比如 x）。
3. 在名字空间（namespace）里为两者建立关联。

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。

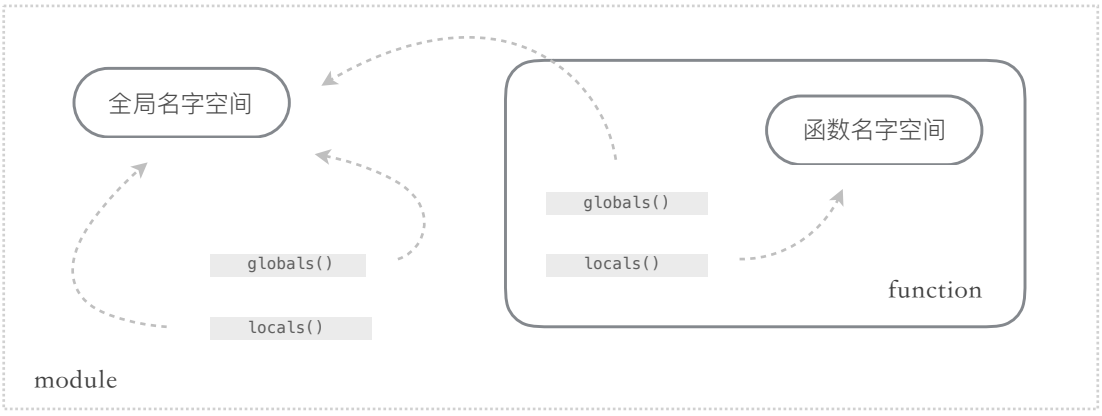
名字空间默认使用 dict 数据结构，由多个键值对（key/value）组成。每个 key 总是唯一。

```
>>> 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。

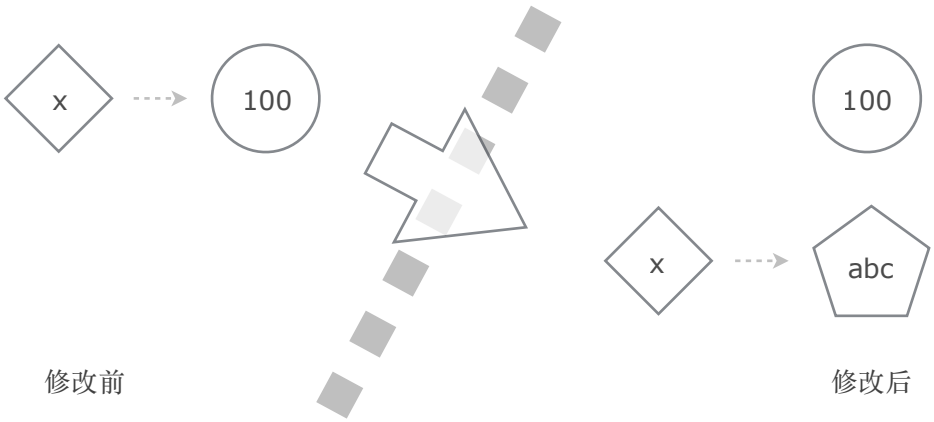
```
>>> globals()["hello"] = "hello, world!"
>>> globals()["number"] = 12345

>>> 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。

- 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
```

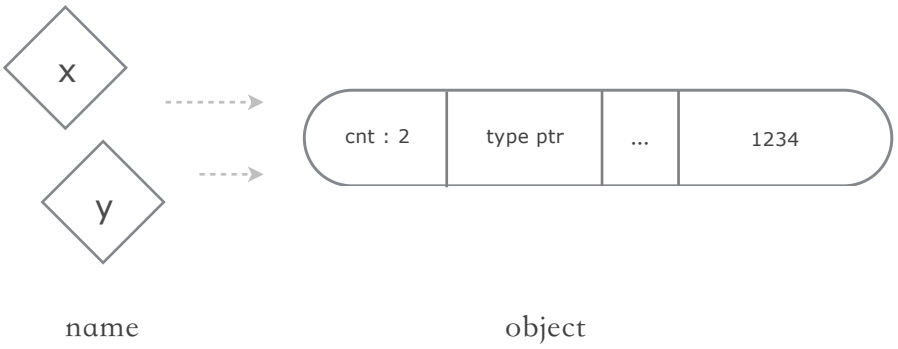
以下划线开头的名字，有特殊含义。

- 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。

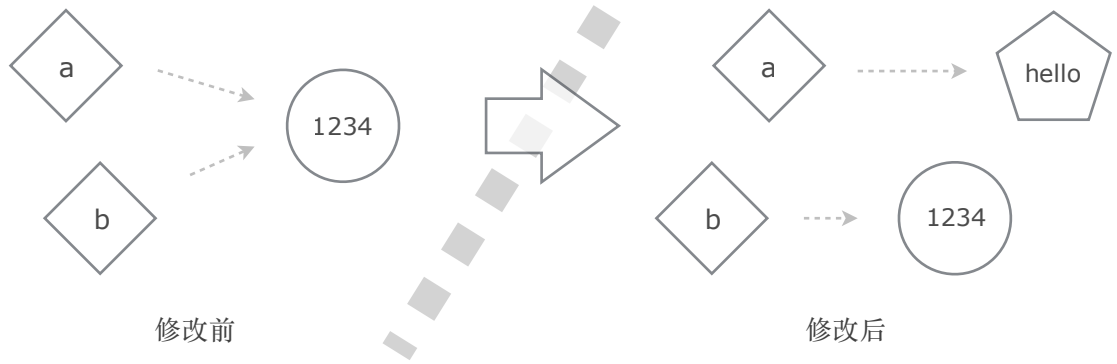
```
>>> a = 1234

>>> 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)              # 弱引用并未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。

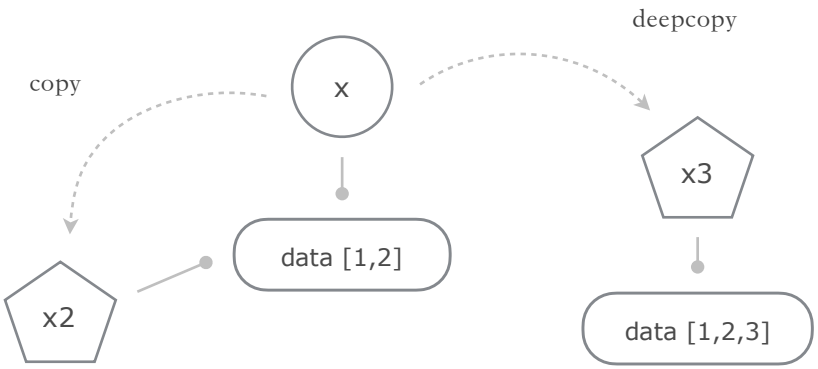
不可变类型包括 int、float 、str、bytes、tuple、frozenset 等。因不能改变其状态，所以被多方只读引用也没什么问题。

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

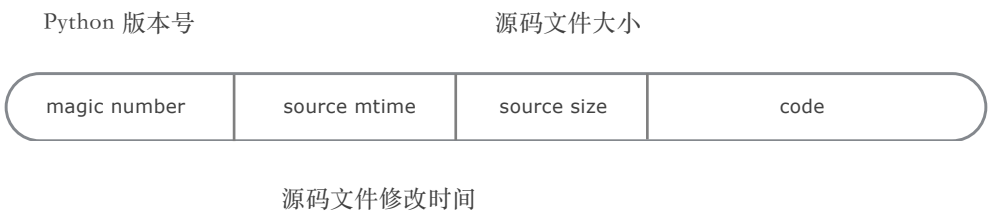
源码须编译成字节码（byte code）指令后，才能交由解释器（interpreter）解释执行。这也是 Python 性能为人诟病的一个重要原因。

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。从这些元数据里，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

可用内置函数将整数转换为指定进制的字符串，或反向用 int 还原。其默认识别为十进制，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，也就是说 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。示例中使用了扩展库 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
```

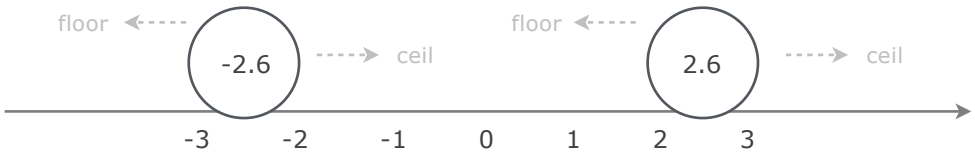
```
>>> float("0.12345678901234567890")
0.12345678901234568
```

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）或专业的扩展库，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。

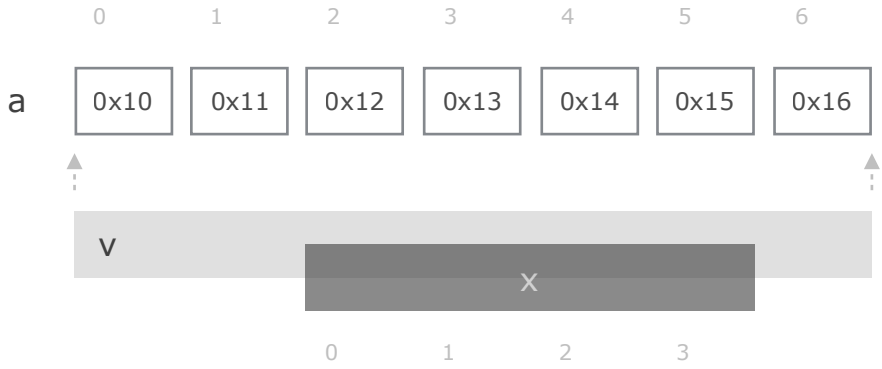
鉴于 Python 没有指针概念，外加内存安全模型限制，要做到这点似乎并不容易。为此，须借助一种名为内存视图（Memory Views）的方式来访问底层内存数据。

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，以及著名扩展库 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。

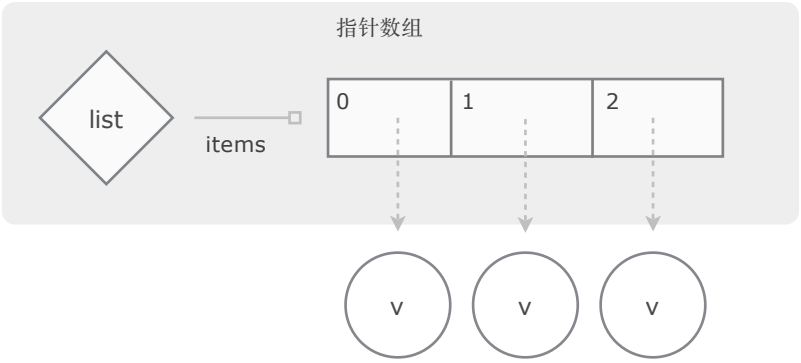
```
>>> x = [1, 2]
>>> x[1]
2

>>> 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。而当列表被回收时，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，同是加法（或乘法）运算，但结果却不同。

```
>>> 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
```

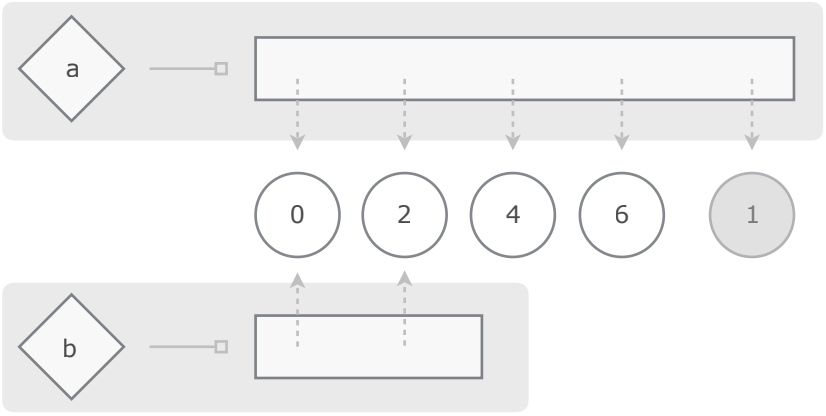
至于删除操作，可以索引序号指定单个元素，或用切片指定删除范围。

```
>>> 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。

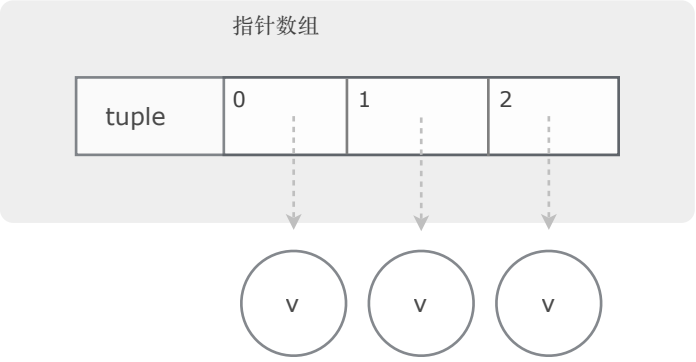
```
>>> a = [1, "abc"]
>>> b = tuple(a)

>>> 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
```

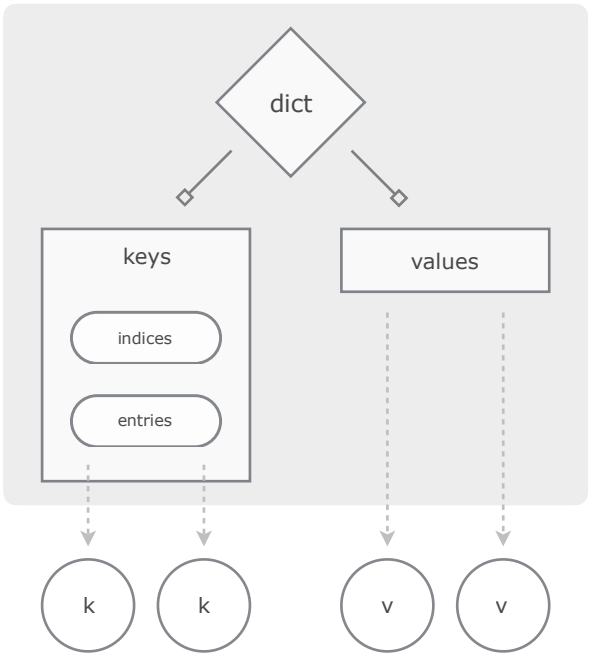
```
>>> hash((1, 2, 3))
2528502973977326415

>>> 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

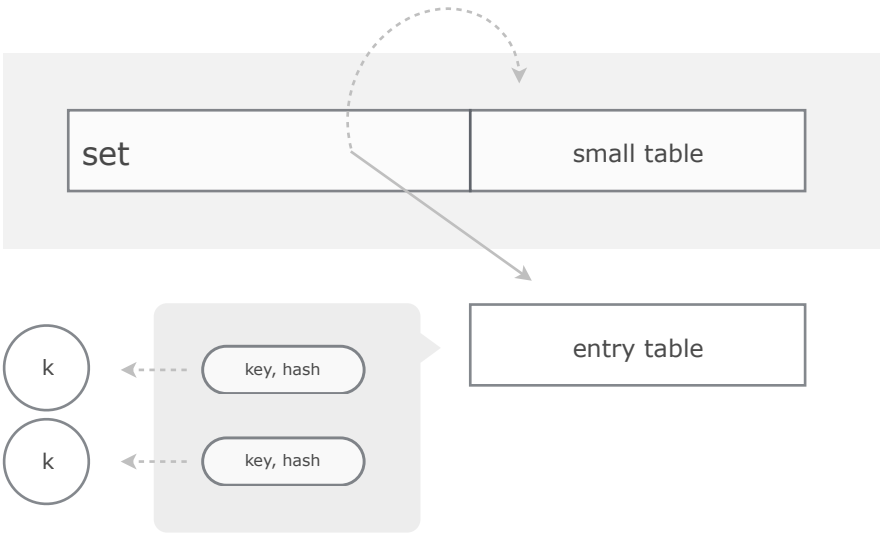
```
(a is b) or (hash(a) == hash(b) and a == b)
--------    ------------------------------
```

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
```

三. 表达式

1. 词法

交互式环境通常用来学习、测试，甚至可替代系统命令行做些管理工作，但着实不适合做正式开发。因为程序复杂度和篇幅，很难保证一次性完成编码。事实上，任何优秀代码，都是在频繁修改和长期调整后才会出现。一个便捷，带有辅助插件，能提升开发效率的编辑器对于开发人员是很重要的事情。

个人倾向于用 iPython 作为学习和测试工具，因其非连贯性操作方式，允许我们实时从不同角度观察目标各个细节。而用编辑器时，更多是编写完整逻辑。或者说交互式操作方式属于调试器范畴，与 pdb 工作方式类似。

当然，对于系统性学习而言，Jupyter Notebook 要更方便些，既有交互式操作，还能编辑说明文字，最终以内嵌可执行代码的笔记形式留存下来。

正式可分发的程序，由一到多个源码文件组成，各自对应一个运行期模块。模块内有很多条语句，用于定义类型，创建实例，执行逻辑指令。其中包含表达式，完成数据计算和函数调用。

表达式（expression）由标识符、字面量和操作符组成。完成运算、属性访问，以及函数调用等。它像数学公式，总是返回一个结果。

语句（statement）则由一到多行代码组成，着重于逻辑过程，完成变量赋值、类型定义，以及控制流方向等。说起来，表达式算是语句的一种，但语句不一定是表达式。

简单归纳：表达式完成计算，语句执行逻辑。

1.1 源文件

如果以 Python 2 为起点，那么下面这个错误可能就是第一只拦路虎。事故源头要追溯到远古时代，解释器以 ASCII 为默认编码，如果源码里出现 Unicode 字符，就会导致其无法正确解析。

```
print "您好"
```

输出：

```
$ python2 test.py

SyntaxError: Non-ASCII character ..., but no encoding declared; ...
```

解决办法是在文件头部添加专门的编码声明，指示解释器按此格式读取。

```
# -*- coding: utf-8 -*-

print "您好"
```

好在 Python 3 将默认编码改成 UTF-8，免去我们为每个源码文件添加此类信息的麻烦。当然，如果您打算使用其他编码格式，依然需要额外声明。

执行

启动程序，只需将入口文件名作为命令行参数即可。

```
$ python3 main.py
```

或者，在入口文件头部添加 Shebang 信息，指示系统程序载入器用指定解释器执行。别忘了赋予该文件可执行权限。

main.py

```
#!/usr/bin/env python

print("您好")
```

输出：

```
$ chmod a+x main.py

$ ./main.py
```

考虑到不同系统安装方式上的差异，不建议使用类似“/usr/bin/python”这样的绝对路径，而是以 env 从当前环境设置里查找与之匹配的解释器。当然，还可指定解释器版本，以及相关执行参数。

> 系统命令 env 通过环境变量 PATH 查找目标，这对于使用 VirtualEnv 之类的虚拟环境有益。

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

命令行

命令行参数分解释器和程序两种，分别以 sys.flags、sys.argv 读取。

```
import sys

print(sys.flags.optimize)
print(sys.argv)
```

输出：

```
$ python -OO main.py 1 2 "hello, world"

2
['main.py', '1', '2', 'hello, world']
```

对于简单测试代码，无需创建文件或启用交互环境，直接在命令行以 -c 参数执行即可。

```
$ python -c "import sys; print(sys.platform)"
darwin

$ python -c "import sys; print(sys.version_info)"
major=3, minor=6, micro=1, releaselevel='final', serial=0
```

退出

终止进程的正式做法是调用 sys.exit 函数，它确保退出前完成相关清理操作。

> 常见清理操作包括 finally 和 atexit。前者是结构化异常子句，无论异常是否发生，它总被执行。而 atexit 用于注册在退出前才执行的清理函数。

终止进程应返回退出状态码（exit status），以便命令行管理工具据此做出判断。依惯例返回零表示正常结束，其他值为错误。标准库 os 模块里有部分平台的常见定义，但也可自行设定，以表达不同结果。

> 辅助函数 exit、quit 由 site 模块创建，适用于交互式环境。但不建议在源文件中使用，可能会导致该文件在 iPython 等环境下执行出错。至于 os._exit，会立即终止进程，不执行任何清理操作。

```
import atexit
import sys

atexit.register(print, "atexit")

try:
    sys.exit()
finally:
    print("finally")
```

输出：

```
finally
atexit
```

```
$ python -c "import os; exit(os.EX_DATAERR)"; echo $?        # 使用 os 定义。
65
```

```
$ python -c "exit('error')"; echo $?                        # 非整数退出码。
error
1
```

1.2 代码

在软件整个生命周期中，我们面对代码的时间远超过最终部署的可执行系统。若要评述，代码与运营各取其半，乃是极重要的资产。

对开发人员而言，编写代码不仅仅是为了实现逻辑需求，完成日常任务。作为一种另类记述文档，代码同样有所追求。如同写文章，不应使用恢宏架构和过度设计来体现优雅，而是追求最自然的逻辑抽象，最简单的直述文字。

综其所言，可阅读度性和可测试性是代码的基本要求。前者保证代码自身的可持续性发展，后者维护其最终价值。

缩进

对于强制缩进规则，批评和欣赏两方各持己见。可不管怎么说，强约束规则总好过流于形式的规范手册，这对新手培养严谨的编码风格尤其重要。

PEP8 推荐使用 4 个半角空格表达缩进，但也有人习惯使用制表符。空格可保证在大多数环境下风格一致，而制表符则可依个人喜好设定显示宽度，算是各有所长。好在，有很多工具能自动转换，算不上什么大问题。

PEP : Python Enhancement Proposals
PEP8 : Style Guide for Python Code

代码格式检查可使用 pycodestyle，或 autopep8、isort、yapf 等增强改进工具。

Python 2 允许在同一文件里混用两种风格，但 Python 3 只能选其一。建议选用能自动识别并转换格式的编辑器，避免出现这种无厘头的非语法性错误。

```
def add(a, b):
    z = a + b          # tab 缩进
    return z           # space 缩进，与 tab 保持等宽。

add(1, 2)
```

输出：

```
$ python2 -t test.py                                    # 参数 -t 检查缩进风格是否一致。
inconsistent use of tabs and spaces in indentation

$ python3 test.py
TabError: inconsistent use of tabs and spaces in indentation
```

缩进的另一个麻烦在于格式丢失导致逻辑性错误。

```
def sum(x):
    n = 0
    for i in x:
        n += i
        return n                # 缩进错误，导致逻辑错误。

print(sum(range(10)))
```

输出：

```
0
```

鉴于 Python 没有代码块结束标志，我们可添加注释作为排版和检查标记。当出现问题时，该标记能让我们手工重排格式，这对因网页粘贴而导致混乱的大段代码尤其有用。

```
def sum(x):
    n = 0
    for i in x:
        n += i
    # end_for                   # 注释表示块结束。虽对工具无效，但可肉眼发现问题所在。
    return n
```

如嫌注释不够优雅，还可创建一个伪关键字作为结束符号。

```
import builtins
builtins.end = None             # 在内置模块，为 None 添加一个别名。确保后续任意模块都能使用。
```

```
def sum(x):
    n = 0
    for i in x:
        n += i
    end                         # 块结束符号。
    return n
```

Python 编译器不够聪明，不会将这个无意义的 end / None 忽略掉。不过，与好处相比，它对性能的影响微乎其微。当然，如您有严格的编码习惯，可用空行区分不同代码块。

语句

无论是为了更好的阅读体验，还是便于在不同环境下编辑，代码行都不宜过长。通常，每条语句独占一行。仅在必要时，以分号分隔多条语句。

但时常也有单语句超过限宽的情况。此时相比编辑器自动换行，手工硬换行的可读性要更好，因为可调整缩进对齐。即便没有超限，将多条件分成多行，也易于修改，比如调整优先级次序，或临时注释掉某些条件。

大多数编码规范都将行宽限制在 80 字符。
考虑到该规则源自早期低分辨率显示环境，现在可适当放宽到 100 字符。

反斜线续行符后面不能有空格和注释。

```
if (0 < a < 10)     and \
   (b is not None)  and \
   (c in b):
```

当然，那些有成对括号的表达式不用续行符就可分成多行。

```
a = [
    1,                      # 这里没有续行符，可以有注释。
    2,                      # 多行可以方便调整顺序，或注释掉某些初始值。
    3,
]

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

```
def long_function_name(                             # 利用多行将参数分组。
        var_one, var_two, var_three,
        var_four):
    pass


long_function_name(var_one, var_two, var_three,
                   var_four)
```

注释

注释作为代码的有益补充存在，决不能与之相悖。以自然语言描述，力求简洁。针对当前代码，言之有物，无多余赘述。应控制注释数量，避免影响代码修改。更不应画蛇添足，对无歧义内容多做说明。

在修改代码时，请同步更新注释，避免造成误解。

注释以井号开头，作用到行尾，分块注释（block）和内联注释（inline）两类。块注释与代码块平级缩进，用于描述整块代码的逻辑意图和算法设计。内联注释在代码行尾部，补充说明其作用。

多数编码规范中要求使用英文注释。本书为阅读方便，依然采用中文。

```
def test():

    # block comment
    # line 2
    # line 3
    print()

    x = 1        # inline comment
```

帮助

与被编译器忽略的注释不同，帮助属基本元数据，可在运行期查询和输出。除在交互环境手工查看外，还用于编辑器智能提示，以改善编码体验，甚至直接导出生成开发手册。

从表面看，帮助就是简单的字符串字面量。考虑跨行需要，建议总是用三引号格式。

应为所有用户使用的导出成员（public）添加帮助（docstrings）信息。
文档生成可使用 pydoc，或更专业的 Sphinx。

---

test.py

```
"""
模块帮助信息
"""
```

```
def test():
    """
    函数帮助信息
    """
    pass
```

> 模块帮助信息不能放到 shebang 前面。

帮助信息保存在 __doc__ 属性里，可直接输出，或以 help 函数查看。

```
>>> import test

>>> test.__doc__
模块帮助信息

>>> test.test.__doc__
函数帮助信息
```

```
>>> help(test)

NAME
    test - 模块帮助信息

FUNCTIONS
    test()
        函数帮助信息
```

当解释器以“OO”优化方式运行时，帮助信息被移除。

```
$ python -c "import test; print(test.__doc__)"
模块帮助信息
```

```
$ python -OO -c "import test; print(test.__doc__)"
None
```

2. 赋值

前文提及，赋值操作为名字和目标对象建立关联。但这只是其结果，或者说基本用途。作为历史悠久的动态语言，Python 如同西方巨龙一般，收集了许多看着眼花缭乱，实际操作便捷的语法风格。样式多变的赋值操作，不过是其中繁花一朵。

> 对应用开发来说，丰富而便捷的语言功能可提升开发效率。但凡事有度，应限制花式代码和魔法实现。任何时候，代码可读性都须优先保障。

比如，为多个名字同时建立关联。

```
>>> a = b = c = 1234
>>> a is b is c                  # 引用同一目标。
True
```

仅以逗号分隔的多个右值被视作元组初始化元素。

```
>>> x = 1, "abc", [10, 20]
>>> x
(1, 'abc', [10, 20])
```

2.1 增量赋值

顾名思义，增量赋值（augmented assignment，+=、*=、...）试图直接修改原对象内容，实现累加效果。当然，前提是目标对象允许，否则退化为普通赋值。

既是增量，那么就须确保目标对象已存在。

```
>>> s += 10
NameError: name 's' is not defined
```

```
>>> s = 0
>>> s += 10
```

我们分别以可变列表、不可变元组对比增量赋值结果。

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

>>> a is b                    # 依然指向原对象。
True
```

```
>>> c = d = ()
>>> c += (1, 2)

>>> c is d                    # 新对象。
False
```

相同增量赋值运算符，结果全然不同。列表直接修改原内容，而元组新建对象。尽管编译器都处理成 INPLACE 指令，但最终执行还是依目标类型而定。

> 以 += 为例，对应 __iadd__ 方法，这也是能否执行增量操作的标志。
> 如该方法不存在，解释器则尝试执行 __add__，变成普通加法操作。

```
>>> dis.dis(compile("a += [1, 2]", "", "exec"))
  1           0 LOAD_NAME                0 (a)
              2 LOAD_CONST               0 (1)
              4 LOAD_CONST               1 (2)
              6 BUILD_LIST               2
              8 INPLACE_ADD
             10 STORE_NAME               0 (a)
             12 LOAD_CONST               2 (None)
             14 RETURN_VALUE
```

```
>>> dis.dis(compile("a += (1, 2)", "", "exec"))
  1           0 LOAD_NAME                0 (a)
              2 LOAD_CONST               3 ((1, 2))
              4 INPLACE_ADD
              6 STORE_NAME               0 (a)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

2.2 序列解包

不同于将多名字关联到单一对象，序列解包（sequence unpacking）展开所有元素对象，继而与多个名字分别建立关联。

```
>>> a, b, c = [1, 2, 3]
>>> a, b, c
1, 2, 3
```

```
>>> a, b, c = "xyz"
>>> a, b, c
'x', 'y', 'z'
```

```
>>> a, b, c = range(3)
>>> a, b, c
0, 1, 2
```

下面示例可能会造成误解。

```
>>> a, b = [1, 2], (3, 4)        # 右值表达式构建元组对象（[1, 2], (3, 4)）。

>>> a
[1, 2]

>>> b
(3, 4)
```

显然，解包操作还可用来交换变量，且无需借助第三方。事实上，对三个以内的变量交换，编译器直接优化成 ROT 指令，直接交换栈帧数据，而不是构建元组。

```
>>> a, b = b, a
```

```
>>> dis.dis(compile("a, b, c = c, b, a", "", "exec"))
  1           0 LOAD_NAME                0 (c)
              2 LOAD_NAME                1 (b)
              4 LOAD_NAME                2 (a)
              6 ROT_THREE                              # 使用 ROT 指令直接操作栈帧数据。
              8 ROT_TWO
```

```
             10 STORE_NAME               2 (a)
             12 STORE_NAME               1 (b)
             14 STORE_NAME               0 (c)
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

```
>>> dis.dis(compile("a, b, c, d = d, c, b, a", "", "exec"))
  1           0 LOAD_NAME                0 (d)
              2 LOAD_NAME                1 (c)
              4 LOAD_NAME                2 (b)
              6 LOAD_NAME                3 (a)
              8 BUILD_TUPLE              4             # 构建元组，解包后分别赋值。
             10 UNPACK_SEQUENCE          4
             12 STORE_NAME               3 (a)
             14 STORE_NAME               2 (b)
             16 STORE_NAME               1 (c)
             18 STORE_NAME               0 (d)
             20 LOAD_CONST               0 (None)
             22 RETURN_VALUE
```

解包操作支持深度嵌套。

左右值表达式以相同的方式嵌套。

```
>>> a, (b, c) = 1, [10, 20]                        # 简单嵌套方式。
>>> a, b, c
1, 10, 20
```

```
>>> a, ((b, c), (d, e)) = 1, [(10, 20), "ab"]       # 更深层次嵌套。
>>> a, b, c, d, e
1, 10, 20, 'a', 'b'
```

当然，也可忽略某些元素。

```
>>> a, _, b, _, c = "a0b0c"
>>> a, b, c
'a', 'b', 'c'
```

星号收集

当序列元素与名字数量不等时，解包出错。

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack (expected 2)

>>> a, b, c = 1, 2
ValueError: not enough values to unpack (expected 3, got 2)
```

如元素多过名字，可考虑将右值截取等长切片。但反过来，事情就比较麻烦了，要写更多代码进行判断。为此，Python 3 专门实现了扩展方式（extended sequence unpacking）。

> 在名字前添加星号，表示收纳所有剩余元素。

```
>>> a, *b, c = range(5)
>>> a, b, c
0, [1, 2, 3], 4
```

```
>>> a, b, *c = range(5)
>>> a, b, c
0, 1, [2, 3, 4]
```

```
>>> *a, b, c = range(5)
>>> a, b, c
[0, 1, 2], 3, 4
```

即便名字多过元素，也能处理。

```
>>> a, *b, c = 1, 2          # 收集不到数据，返回空列表。
>>> a, b, c
1, [], 2
```

注意，解包操作优先保障非收集名字赋值，所以右值元素不能少于此数量。另外，星号只能有一个，否则无法界定收集边界。

```
>>> a, *b, c, d = 1, 2
ValueError: not enough values to unpack (expected at least 3, got 2)
```

```
>>> a, *b, *c, d = range(10)
SyntaxError: two starred expressions in assignment
```

星号收集不能单独出现，要么与其他名字一起，要么放入列表或元组内。

```
>>> *a = 1, 2
SyntaxError: starred assignment target must be in a list or tuple
```

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

```
>>> (*a,) = 1, 2            # 注意是元组，别忘了逗号。
>>> a
[1, 2]
```

序列解包和星号收集还可用于控制流表达式等场合。

```
>>> for a, *b in ["abc", (1, 2, 3)]: print(a, b)

a ['b', 'c']
1 [2, 3]
```

星号展开

星号还可用来展开可迭代（iterable）对象。

简单点说，可迭代对象就是每次返回一个成员。
所有序列类型，以及字典、集合、文件等都是可迭代类型。

```
>>> a = (1, 2)
>>> b = "ab"
>>> c = range(10, 13)

>>> [*a, *b, *c]
[1, 2, 'a', 'b', 10, 11, 12]
```

对于字典，可用单星号展开主键，双星号展开键值。

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

>>> [*d]
['a', 'b']

>>> {"c": 3, **d}
{'c': 3, 'a': 1, 'b': 2}
```

展开操作同样可用于函数调用，将单个对象分解成多个实参。

```
def test(a, b, c):
    print(locals())
```

```
>>> test(*range(3))
{'c': 2, 'b': 1, 'a': 0}

>>> test(*[1, 2], 3)
{'c': 3, 'b': 2, 'a': 1}
```

```
>>> a = {"a": 1, "c": 3}
>>> b = {"b": 2}

>>> test(**b, **a)
{'c': 3, 'b': 2, 'a': 1}
```

2.3 作用域

作为隐式规则，赋值操作总是针对当前名字空间，可能是 locals，又或者 object.__dict__。在同一作用域内，即便存在先后顺序，名字也不会解析成不同名字空间引用。

```
>>> x = 10

>>> def test():
        print(x)
        x = x + 10

>>> test()
UnboundLocalError: local variable 'x' referenced before assignment
```

```
>>> dis.dis(test)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (x)           # 本地
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_FAST                0 (x)           # 本地
             10 LOAD_CONST               1 (10)
             12 BINARY_ADD
             14 STORE_FAST               0 (x)           # 本地
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

从反汇编结果看，函数内的 x 统统从本地名字空间引用。
并非如设想般，先从全局读取，最后才在本地建立关联。

如要对外部变量赋值，须显式声明变量位置。关键字 global 指向全局名字空间，nonlocal 为外层嵌套（enclosing）函数。

除非必要，否则应避免对外部变量赋值。可以返回值等方式，交由持有者处理。

```
>>> g = 1

>>> def outer():
        e = 2

        def inner():
            global g            # 声明全局变量。
            nonlocal e          # 声明外层嵌套函数变量。
            g = 10
            e = 20

        inner()
```

```
        return e

>>> outer()
20

>>> g
10
```

此显式声明指示编译器生成对外部空间的操作指令。

```
>>> def test():
        global x
        x = 10

>>> dis.dis(test)
  3           0 LOAD_CONST               1 (10)
              2 STORE_GLOBAL             0 (x)         # 全局
              4 LOAD_CONST               0 (None)
              6 RETURN_VALUE
```

我们可用 global 在函数内创建全局变量。

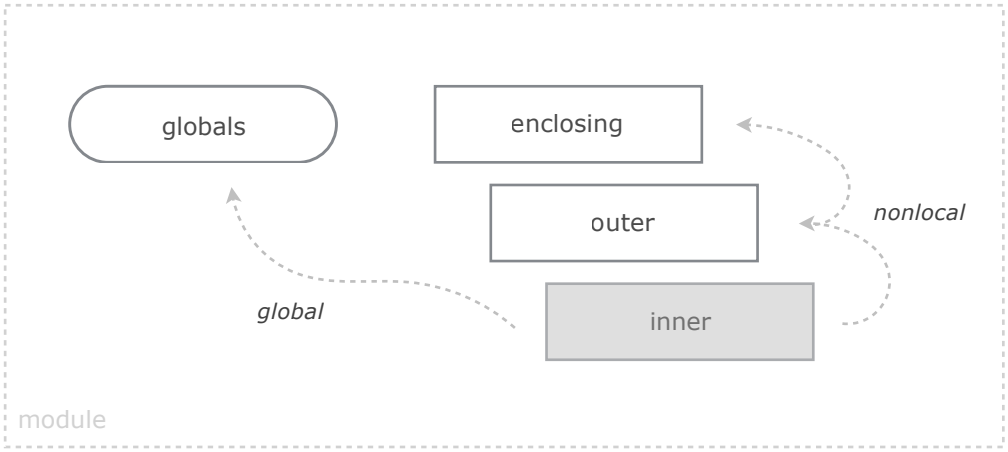
```
>>> x
NameError: name 'x' is not defined        # 检查 x 是否存在。

>>> def test():
        global x
        x = 100

>>> test()
>>> x
100
```

至于 nonlocal 则自内向外依次检索嵌套函数，但不包括全局名字空间。

如多层嵌套函数存在同名变量，依就近原则处理。
另，nonlocal 不能为外层嵌套函数新建变量。

```
>>> def enclosing():
        x = 1

        def outer():
            def inner():
```

```
                nonlocal x
                x = 999              # 向外查找，直到 enclosing.x 命中。

            inner()

        outer()
        print(x)

>>> enclosing()
999
```



不同于 global 运行期行为，nonlocal 要求在编译期绑定，所以目标须提前存在。

```
>>> def inner():
        nonlocal x

SyntaxError: no binding for nonlocal 'x' found
```

```
>>> def outer():
        def inner():
            nonlocal x

SyntaxError: no binding for nonlocal 'x' found
```

作为写操作的赋值，其规则与读操作 LEGB 完全不同，注意区别对待。

---

3. 运算符

相比其他语言，Python 运算符更接近自然表达方式。也正因如此，优先级导致的错误更加隐蔽，不易察觉和排除。

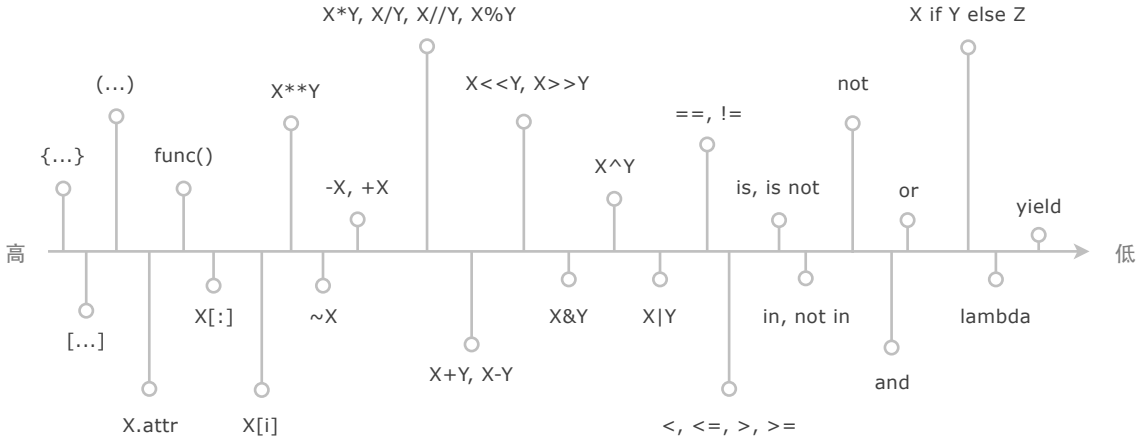
看下面示例，即便用括号调整优先级，也很难发现是否存在缺陷。

```
>>> not "a" in ["a", 1]              # 谁先谁后?
False

>>> (not "a") in ["a", 1]            # not 先?
False

>>> not ("a" in ["a", 1])            # in 先? 看不出来。
False
```

可一旦数据变化，其结果就可能不同。除非测试数据完整覆盖，否则导致带病上线。

```
>>> not "a" in [1]
True

>>> (not "a") in [1]
False

>>> not ("a" in [1])
True
```

该示例也提示我们，适当使用括号，不但可避免隐蔽错误，还能提高代码可读性。



每个运算符都有对应函数（方法）实现，可像普通函数那样作为逻辑传递。当然，用动态执行也未尝不可。

```
def calc(x, y, op):
    return op(x, y)
```

```
>>> import operator

>>> calc(1, 2, operator.add)
3

>>> calc(1, 2, operator.mul)
2
```

> 不仅仅是数学运算符，operator 还有 itemgetter、attrgetter 等用于索引和成员访问函数。

除此之外，还可用标准库提供的辅助函数，简化自定义类型运算符重载代码。

> 使用 functools.total_order 装饰器，基于 __eq__、__lt__ 方法，补全剩余比较方法。

```
@functools.total_ordering
class X:
    def __init__(self, n):
        self.n = n

    def __eq__(self, o):
        return self.n == o.n

    def __lt__(self, o):
        return self.n < o.n
```

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

>>> a <= b
True

>>> a >= b
False
```

Python 3 对运算符做了些调整。

- 移除“<>”，统一使用“!=”运算符。
- 移除 cmp 函数，自行重载相关运算符方法。
- 除法“/”表示 True Division，总是返回浮点数。
- 不再支持反引号 repr 操作，调用同名函数。
- 不再支持非数字类型混合比较，可自定义相关方法。
- 不再支持字典相等以外的比较操作。

3.1 链式比较

链式比较（chained comparison）将多个比较表达式组合到一起，更符合人类阅读习惯，而非面向机器。该方式可有效缩短代码，并稍稍提升性能。

```
>>> a, b = 2, 3

>>> a > 0 and b > a and b <= 5
True

>>> 0 < a < b <= 5                  # 可读性更好，更易维护。
True
```

反汇编查看两者差异。

```
>>> dis.dis(compile("1 < a and a < 2", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_NAME                0 (a)          # 载入。
              4 COMPARE_OP               0 (<)
              6 JUMP_IF_FALSE_OR_POP    14
              8 LOAD_NAME                0 (a)          # 载入。
             10 LOAD_CONST               1 (2)
             12 COMPARE_OP               0 (<)
        >>   14 RETURN_VALUE
```

```
>>> dis.dis(compile("1 < a < 2", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_NAME                0 (a)          # 载入。
              4 DUP_TOP                                 # 直接复制。
              6 ROT_THREE
              8 COMPARE_OP               0 (<)
```

```
              10 JUMP_IF_FALSE_OR_POP    18
              12 LOAD_CONST               1 (2)
              14 COMPARE_OP               0 (<)
              16 RETURN_VALUE
        >>    18 ROT_TWO
              20 POP_TOP
              22 RETURN_VALUE
```

显然，链式比较减少了载入指令，更多基于栈数据复制和交换。仅凭这点，其执行性能就有所提高。但整体上看，这点改善远不如代码可读性和可维护性吸引人。

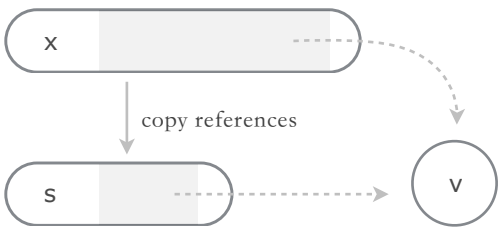
3.2 切片

切片（slice）用以表达序列对象的某个片段（或整体），其具体行为与其在出现在语句中的位置有关。当以右值出现时，复制序列数据；而左值则表达要操控的目标范围。

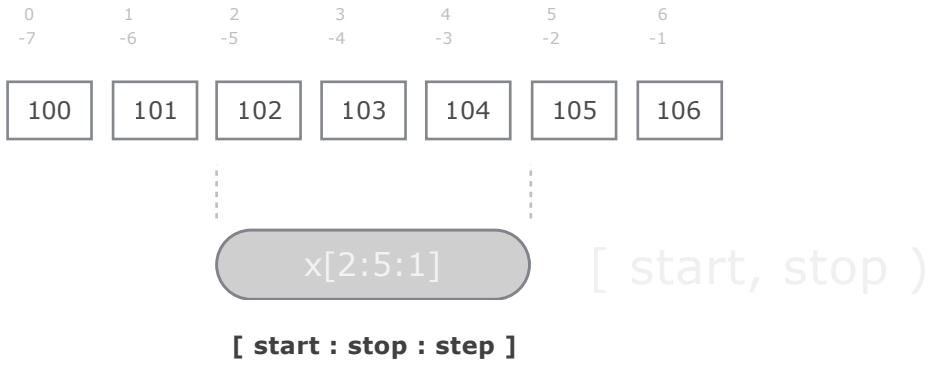
```
>>> x = [0, 1, 2, 3, 4, 5, 6]
>>> s = x[2:5]                      # 从列表中复制指定范围的引用。

>>> s
[2, 3, 4]
```

```
>>> x.insert(3, 100)                # 对原列表的修改，不影响切片。

>>> x
[0, 1, 2, 100, 3, 4, 5, 6]

>>> s
[2, 3, 4]
```

注意，列表存储的是元素对象引用（指针），那么复制的自然也是引用，而非元素对象。切片所返回新列表与原列表除共享部分元素对象外，其他毫无干系。



完整的切片操作由三个参数构成。



以起始和结束索引构成一个半开半闭区间（不含结束位置）。默认起始位置为 0，结束位置 len(x)，以容纳最后一个元素。

```
>>> x = [100, 101, 102, 103, 104, 105, 106]

>>> x[2:5:1]
[102, 103, 104]
```

```
>>> x[:5]                                  # 省略起始索引。
[100, 101, 102, 103, 104]

>>> x[2:]                                  # 省略结束索引。
[102, 103, 104, 105, 106]

>>> x[:]                                   # 完整复制。
[100, 101, 102, 103, 104, 105, 106]
```

可指定步进幅度，间隔选取元素。甚至可以是负值，从右至左反向行进。

索引 0 表示正向第一元素，所以反向索引从 -1 起始。

```
>>> x[2:6:2]
[102, 104]

>>> x[::2]
[100, 102, 104, 106]
```

```
>>> x[::-1]                                  # 反向步进，全部复制。
[106, 105, 104, 103, 102, 101, 100]

>>> x[5:2:-1]                                # 反向步进，使用正索引表示起始、结束位置。
[105, 104, 103]

>>> x[-2:-5:-1]                              # 使用负索引表达起始、结束位置。
[105, 104, 103]
```

除表达式外，也可使用 itertools.islice 函数执行切片操作。

事实上，负索引不仅用于切片，也可直接访问序列元素。

```
>>> (0, 1, 2)[-2]
1

>>> "abcd"[-3]
'b'
```

删除

用切片指定要删除的序列范围。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> del x[3:7]

>>> x
[0, 1, 2, 7, 8, 9]
```

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> del x[3:7:2]                             # 步进删除。

>>> x
[0, 1, 2, 4, 6, 7, 8, 9]
```

赋值

以切片方式进行序列局部赋值，相当于先删除，后插入。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[3:7] = [100, 200]

>>> x
[0, 1, 2, 100, 200, 7, 8, 9]
```

如设定步进，则删除和插入元素数量必须相等。

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[::2]                                    # 查看要删除的元素。
[0, 2, 4, 6, 8]

>>> x[::2] = [100, 200, 400, 600, 800]        # 步进插入。

>>> x
[100, 1, 200, 3, 400, 5, 600, 7, 800, 9]
```

```
>>> x[::2] = [0, 2, 4]
ValueError: attempt to assign sequence of size 3 to extended slice of size 5

>>> x[::2] = [0, 2, 4, 6, 8, 10]
ValueError: attempt to assign sequence of size 6 to extended slice of size 5
```

3.3 逻辑运算

逻辑运算用于判断多个条件的布尔结果，或返回有效操作数。

分别以 and、or、not 运算符表示逻辑与、或、非三种关系。
其中 and 返回最后，或导致短路的操作数；or 返回第一真值，或最后操作数。

```
>>> 1 and 2                   # 最后操作数。
2
```

```
>>> 1 and 0 and 2             # 导致短路的操作数。
0
```

```
>>> 1 or 0                    # 第一真值。
1

>>> 0 or 1 or 2               # 第一真值。
1

>>> 0 or []                   # 最后操作数。
[]
```

相同逻辑运算符一旦短路，后续计算被终止。

```
def x(o):                     # 输出执行信息。
    print("op:", o)
    return o
```

```
>>> x(0) and x(1)             # 0 导致短路。
op: 0
0

>>> x(1) or x(2)              # 返回 1 真值，短路。
op: 1
1
```

用反汇编可以看得更清楚些。

```
>>> dis.dis(compile("0 and 1 and 2 and 3", "", "eval"))
  1           0 LOAD_CONST               0 (0)
              2 JUMP_IF_FALSE_OR_POP    14                 # 如果为 False，跳转到 14。
              4 LOAD_CONST               1 (1)
              6 JUMP_IF_FALSE_OR_POP    14
              8 LOAD_CONST               2 (2)
             10 JUMP_IF_FALSE_OR_POP    14
             12 LOAD_CONST               3 (3)
        >>   14 RETURN_VALUE
```

```
>>> dis.dis(compile("1 or 2 or 3 or 4", "", "eval"))
  1           0 LOAD_CONST               0 (1)
              2 JUMP_IF_TRUE_OR_POP     14
```

```
              4 LOAD_CONST               1 (2)
              6 JUMP_IF_TRUE_OR_POP     14
              8 LOAD_CONST               2 (3)
             10 JUMP_IF_TRUE_OR_POP     14
             12 LOAD_CONST               3 (4)
        >>   14 RETURN_VALUE
```

当然，不同运算符需多次计算。

```
>>> x(0) and x(1) or x(2)
op: 0
op: 2
2
```

```
>>> dis.dis(compile("0 and 1 and 2 or 9", "", "eval"))
  1           0 LOAD_CONST               0 (0)
              2 POP_JUMP_IF_FALSE       12                 # 如果 False，跳转到 12。
              4 LOAD_CONST               1 (1)
              6 POP_JUMP_IF_FALSE       12
              8 LOAD_CONST               2 (2)
             10 JUMP_IF_TRUE_OR_POP     14
        >>   12 LOAD_CONST               3 (9)
        >>   14 RETURN_VALUE
```

条件表达式

另一常见逻辑运算是条件表达式（conditional expression），类似功能在其他语言被称作三元运算符（ternary operator）。

> T if X else F：当条件 X 为真时，返回 T，否则返回 F。等同 X ? T : F。

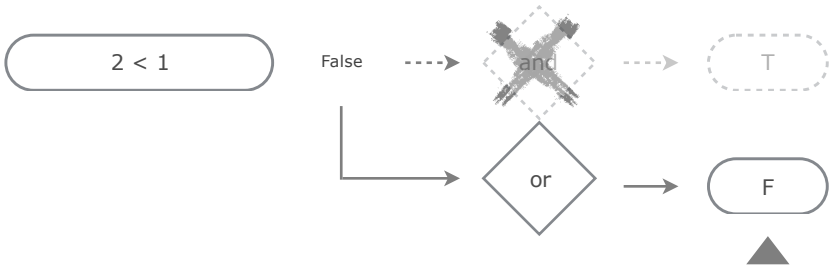
```
>>> "T" if 2 > 1 else "F"      # 等同 2 > 1 ? T : F
'T'

>>> "T" if 2 < 1 else "F"
'F'
```

也可用逻辑运算符实现同等效果，且方式更接近传统习惯。

```
>>> 2 > 1 and "T" or "F"        # 2 > 1 ? T : F
'T'

>>> 2 < 1 and "T" or "F"
'F'
```

分解执行步骤，这很容易理解。

```
>>> 2 < 1 and "T" or "F"
'F'

>>> 2 < 1 and "T"            # ① 2 < 1 导致短路。
False

>>> False or "F"             # ② 返回真值 F。
'F'
```



可惜此方法存在缺陷：当 T 为假时，那么 or 必然返回最后操作数，与预期不符。

```
>>> 2 > 1 and "" or "F"
'F'

>>> 2 > 1 and ""             # ① 返回最后操作数“”。
''

>>> "" or "F"                # ② 返回真值 F，与期望值“” 不符。
'F'
```

显然，当 T 和 F 是动态数据时，条件表达式更安全一些。

```
>>> "" if 2 > 1 else "F"
''
```

逻辑运算符还常被用来简化默认值设置。

```
>>> x = None
>>> y = x or 100
>>> y
100
```

```
>>> x = None
>>> y = x and x * 2 or 100
>>> y
100
```

4. 控制流

当我们用面向对象技术构建好了舞台和演员，剩下的就是用故事线串联起整个世界，让所有成员都活过来。故事的每个场景都由一系列矛盾冲突推动前行，不总是那么平淡悠闲。其中有选择，有轮回，还需大魄力抽身事外。

4.1 选择

语法除 elif 缩写外，其他未见有何不同。多选择分支依次执行条件表达式，最终全部匹配失败，或仅一条得以执行。

单 if 多分支，与多 if 语句意义和执行方式完全不同，注意区别。

```
def test(x):
    if x > 0:
        print("+")
    elif x < 0:
        print("-")
    else:
        print("=")
```

```
>>> dis.dis(test)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (0)
              4 COMPARE_OP               4 (>)
              6 POP_JUMP_IF_FALSE       18                # 当前条件失败，转入下一分支。

  3           8 LOAD_GLOBAL              0 (print)
             10 LOAD_CONST               2 ('+')
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 JUMP_FORWARD            26 (to 44)        # 分支执行后，立即跳出整个 if 语句块。

  4     >>   18 LOAD_FAST                0 (x)
             20 LOAD_CONST               1 (0)
             22 COMPARE_OP               0 (<)

             ...

        >>   44 LOAD_CONST               0 (None)
             46 RETURN_VALUE
```

对于 else 分支，通常建议代码前置作为默认实现。应尝试以重构或设计模式（比如多态）来减少选择分支，减少缩进层次，避免将执行细节与框架流程混到一起。

> 在描述汽车换挡流程时，会说：“挂 R 档倒车”，并不会加入变速器和发动机如何工作等细节。因流程和细节所处层次不同，混到一起，未免有主次不分之嫌。在阅读和研究代码时，更期望有简单清晰的逻辑主线，让我们能快速了解整个过程，然后才是选择性深入细节。

```
def lenx(s):
    n = -1
    if s: n = len(s)
    return n
```

```
def lenx(s):
    if s: return len(s)
    return -1
```

常见问题和建议：

- 将过长的分支代码重构为函数。相比细节，有意义的函数名更友好。
- 将复杂或过长的条件表达式重构为函数，更易阅读和维护。
- 代码块跨度太长（比如需要翻屏），极易造成缩进错误。
- 嵌套易造成缩进错误，且太多层次可读性不好，应当避免。
- 简单的选择语句，可用条件表达式或逻辑运算替代。

> 在逻辑流程中，用友好的名字遮蔽细节是极重要的。

对新手而言，过多的选择分支容易造成死代码（dead code）。可惜编译器对此漠不关心，未做任何提示和消除（DCE, dead code elimination）处理，只能借助第三方检查工具。

```
def test(x):
    if x > 0:
        print("a")
    elif x > 5:              # 死代码，永远不会被执行。
        print("b")
```

```
>>> dis.dis(test)
```

```
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (0)
              4 COMPARE_OP               4 (>)
              6 POP_JUMP_IF_FALSE       18

                ...

  4     >>   18 LOAD_FAST                0 (x)            # 死代码分支代码依然存在。
             20 LOAD_CONST               3 (5)
             22 COMPARE_OP               4 (>)
             24 POP_JUMP_IF_FALSE       34

  5          26 LOAD_GLOBAL              0 (print)
             28 LOAD_CONST               4 ('b')
             30 CALL_FUNCTION            1
             32 POP_TOP
        >>   34 LOAD_CONST               0 (None)
             36 RETURN_VALUE
```

4.2 循环

循环语句分 while、for 两种，两者间并不存在替代关系。前者用于执行逻辑循环，而后者则偏向对象内容迭代，或许改名 foreach 更易区分和理解。

```
>>> while n > 0:                    # 基于特定条件，循环执行指令。
        print(n)
        n -= 1
```

```
>>> for i in range(3):              # 依次从可迭代对象中提取元素。
        print(i)
```

迭代

遍历可迭代对象（iterate）时，可进一步执行序列解包。次序是先取迭代元素，随后对该元素做解包操作。

```
>>> for a, b in [(1, 2), (3, 4)]:
        print(a, b)

1 2
```

```
3 4
```

类似下面的错误，纯属误解。

```
>>> for a, b in [1, 2]:
        print(a, b)

TypeError: 'int' object is not iterable
```

如要实现传统 for 循环，可借助 enumerate 类型。它为迭代元素添加自增序号，如此解包操作就可获取索引值。

> 迭代操作用 __next__ 方法返回元素，无需修改原对象，在此添加一个计数器即可。
> 默认起始序号为 0，可用参数自行设定。

```
>>> for i, x in enumerate([100, 200, 300]):
        print(i, x)

0 100
1 200
2 300
```

如目标对象以函数返回，那么该函数仅调用一次。

```
>>> def data():
        print("data")
        return range(3)

>>> for i in data():
        print(i)

data                            # 仅执行一次。
0
1
2
```

```
>>> dis.dis(compile("for i in data(): print(i)", "", "exec"))
  1           0 SETUP_LOOP              22 (to 24)
              2 LOAD_NAME                0 (data)
              4 CALL_FUNCTION            0                 # 调用 data 函数，将结果存储在堆栈。
```

```
              6 GET_ITER
        >>    8 FOR_ITER                12 (to 22)         # 开始迭代。
             10 STORE_NAME               1 (i)
             12 LOAD_NAME                2 (print)
             14 LOAD_NAME                1 (i)
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 JUMP_ABSOLUTE            8                 # 跳转到 8 继续迭代，而非调用 data 函数。
        >>   22 POP_BLOCK
        >>   24 LOAD_CONST               0 (None)
             26 RETURN_VALUE
```

可选分支

Python 循环语句与其他语言最大差别在于，它们可自选 else 分支，用于在循环正常结束后执行额外操作。

> 正常结束是指循环没有被 break、return 中断。当然，循环体没被执行也属正常。
> 另外，执行 continue 是允许的，它不是中断。

```
>>> n = 3
>>> while n > 0:
        print(n)
        n -= 1
    else:
        print("over")

3
2
1
over
```

```
>>> n = 3
>>> while n > 0:
        print("break:", n)
        break                       # break 导致 else 不会执行。
    else:
        print("over")

break: 3
```

```
>>> n = 0
```

```
>>> while n > 0:
        print(n)
        break                       # 尽管有 break，但并未执行，也算正常结束。
    else:
        print("over")

over
```

该特性可用来处理一些默认行为，比如日志记录，重置缓冲区等。

```
def match(data, x):
    tmp = data[:]

    while tmp:
        if x in tmp: return True        # 找到后，以 return 中断循环。
        tmp = tmp[1:]
    else:                               # 没有中断，自然是没找到。
        print("log: not found")

    return False
```

```
>>> match(range(3), 4)
log: not found
False
```

临时变量

循环语句并没有单独名字空间，其内部临时变量直接影响语句所在上下文。

```
def test():
    while True:
        x = 100
        break

    for i in range(10, 20):
        pass

    print(locals())                 # x、i 实际使用 test.locals 名字空间。
    print(x, i)
```

```
>>> test()
```

```
{'i': 19, 'x': 100}
100 19
```

跳转

Python 不支持 goto 和 label，想要在多层嵌套循环中跳转，稍稍有些麻烦。最简单的做法是设定跳转标志，并在相应位置检查。

```
def test():
    stop = False

    while True:
        while True:
            stop = True              # 中断前设定跳出标志，供外循环判断。
            break

        if stop: break
```

或者，将内层循环重构为函数，基于返回值作出判断。

```
def test():

    def inner():                     # 内循环重构函数，以返回值作为跳转标志。
        while True:
            return False
        return True

    while True:
        if not inner(): break
```

更激进的做法是抛出异常，可绕开循环语句和函数限制，在调用堆栈层面拦截捕获。当然，这涉及函数调用和异常处理内容，后文另作详述。

网上有修改字节码插入跳转指令实现 goto 的方法，可供研究，不建议使用。
另定义多个异常，分别 raise 和 except，也算是变向实现 goto label 机制。

---

5. 推导式

相比之下，推导式（comprehensions）文法更像出自某位数学家之手。用自然而简便的方式整合 for、if 语句，用于构造列表、字典和集合对象。


```
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]

>>> [x + 10 for x in range(10) if x % 2 == 0]
[10, 12, 14, 16, 18]
```

将推导式拆解成普通语句，更易理解其执行过程，但稍显不够优雅。

```
l = []

for x in range(10):
    if x % 2 == 0: l.append(x + 10)
```

除列表外，还可以大括号创建字典和集合，区别在于输出表达式是否为键值对。

```
>>> {k:v for k, v in zip("abc", range(10, 13))}    # 字典
{'a': 10, 'b': 11, 'c': 12}

>>> {k:0 for k in "abc"}
{'a': 0, 'b': 0, 'c': 0}
```

```
>>> {x for x in "abc"}                              # 集合
{'b', 'c', 'a'}
```

推导式还可直接用作函数调用实参。

```
def test(data):
    print(type(data), data)
```

```
>>> test({x for x in range(3)})
<class 'set'> {0, 1, 2}
```

嵌套

让我们用推导式改造如下面这样一个层次过多的嵌套循环。

```
>>> l = []

>>> for x in "abc":
        if x != "c":
            for y in range(3):
                if y != 0:
                    l.append(f"{x}{y}")

>>> l
['a1', 'a2', 'b1', 'b2']
```

推导式允许有多个 for 子句，每个子句都可选一个 if 条件表达式。

```
>>> [f"{x}{y}" for x in "abc" if x != "c"
               for y in range(3) if y != 0]

['a1', 'a2', 'b1', 'b2']
```

两相对比，可以说推导式绝对是您日后时常提及，并积极使用的功能。

性能

除去文法因素外，推导式还有性能上的优势。

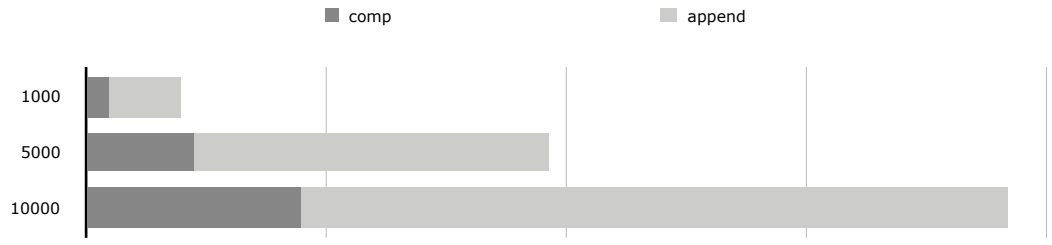
```
def test_comp(data):
    return [x for x in data]


def test_append(data):
    l = []
    for x in data:
        l.append(x)

    return l
```

为更准确测试两者性能差异，避免额外干扰，特预置静态数据源（列表）。但从结果看，数倍差异已是不小距离。

> 比较对象是 for + append 语句，而非内置以 C 实现的 list 构造。
> 毕竟构造方法不支持过滤表达式。

```
>>> data = list(range(10000))                    # 准备测试数据。

>>> test_comp(data) == test_append(data)         # 确保结果一致。
True
```

```
>>> %timeit test_comp(data)
269 µs ± 4.72 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %timeit test_append(data)
885 µs ± 11.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```



查看反汇编输出，可看到推导式被优化处理过。

```
>>> dis.dis(test_comp)
```

```
  2           0 LOAD_CONST               1 (<code object <listcomp>)
              2 LOAD_CONST               2 ('test_comp.<locals>.<listcomp>')
              4 MAKE_FUNCTION            0
              6 LOAD_FAST                0 (data)
              8 GET_ITER
             10 CALL_FUNCTION            1
             12 RETURN_VALUE
```

临时变量

不同于普通循环语句，推导式临时变量不会影响上下文名字空间。

```
def test():
    a = "abc"
    data = {a:b for a, b in zip("xyz", range(10, 13))}
    print(locals())
```

```
>>> test()
{'data': {'x': 10, 'y': 11, 'z': 12}, 'a': 'abc'}
```

从输出结果看，推导式临时变量既没有修改外部同名变量，也没有在名字空间新建关联。

异步、生成器

自 Python 3.5 引入新协程模型（async/await coroutines） 后，3.6 又增加了异步推导式支持。不过要理解这些内容，须对 Python 异步编程有全面了解。本书下卷，会详细解析包含 asyncio 在内的异步架构和使用方法。

如在推导式语法中使用小括号，其结果并非创建元组，而是生成器（generator）对象。所以，我们区别于推导式，将其称作生成器表达式。

```
>>> (x for x in range(3))
<generator object <genexpr> at 0x10812f468>
```

四. 函数

1. 定义

函数是基本代码复用和模块化单位，用于执行单一逻辑。

相比语句，函数以整体形式面向调用者，其签名（名称、参数列表、返回值）构成接口原型，天生带有设计味道。函数还是代码重构（code refactoring）源头，涉及重命名、参数整理等诸多技巧。莫小看函数，应以此为起点，追求精致整洁、优雅简约风格，写出没有坏味道的代码。

> 整洁的代码简单直接，如同优美的散文。它从不隐藏设计者意图，充满干净利落的抽象和直截了当的控制语句。—— Grady Booch

函数应减少依赖关系，具备良好的可测试性和可维护性，这是性能调优关键所在。另外，还应遵循一个基本原则，就是专注于做一件事，不受外在干扰和污染。

比如，当某个函数同时具备修改和查询功能时，正确做法是拆分成查询和修改两个不同函数。从逻辑上讲，查询可能返回零，但修改必然是预设存在条件的。再则，查询性能优化面向缓存，这于修改不利，两者无法共处。

函数要短而精，使用最小作用域。如有可能，应确保其行为一致性。如果函数逻辑受参数影响而有所不同，那不如将多个逻辑分支分别重构成独立函数，使其从变转为不变。结果是更易测试，更易扩展，算是遵循开闭原则（OCP），可逐步冻结代码。

最后，请及时清理掉不再使用的参数、代码，以及注释。任何用不到的元素，无论出发点如何，终究也只是拖累。与其覆满霜尘，莫若“茅檐长扫，花木成畦”，以待未来客。

创建

任何可被调用执行的函数都由两部分组成：代码对象持有字节码和指令元数据，负责指令执行；函数对象则为上下文提供调用实例，并管理执行所需的状态数据。两者相加，算是典型的“指令 + 数据”格局。

假定函数是生产线，那么代码对象就是已定义工序的数控机床。即便如此，机床本身也只是一个生产工具，并不能直接对外服务。需要专门的工厂为其提供场地、电力、人员（状态），然后为其承接生产任务（调用），这就是函数对象的角色。当然，同一种机床可卖给不同工厂。

```
def test(x, y = 10):
    x += 100
    print(x, y)
```

```
>>> test
<function __main__.test>

>>> test.__code__
<code object>
```

代码对象（__code__）相关属性由编译器静态生成，为只读模式。存储指令运行所需相关信息，诸如源码行、指令常量，以及参数和变量名等。

```
>>> test.__code__.co_varnames              # 参数及变量名列表。
('x', 'y')

>>> test.__code__.co_consts                # 指令常量。
(None, 100)
```

```
>>> dis.dis(test.__code__)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (100)
              4 INPLACE_ADD
              6 STORE_FAST               0 (x)

  3           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                0 (x)
             12 LOAD_FAST                1 (y)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

如果 dis(test.__code__.co_code)，会看到没有元数据符号的反汇编结果。

与代码对象不同，函数对象作为外部实例存在，负责管理运行期状态。比如上例中的参数默认值，以及动态添加的属性等。

```
>>> test.__defaults__                   # 参数默认值。
(10,)

>>> test(1)
101 10
```

```
>>> test.__defaults__ = (1234,)         # 修改默认值。

>>> test(1)
101 1234
```

```
>>> test.abc = "hello, world"           # 为函数实例添加属性。

>>> test.__dict__
{'abc': 'hello, world'}
```

事实上，def 是运行期指令。以代码对象为参数，创建函数实例，并在当前上下文与指定名字相关联。

```
>>> dis.dis(compile("def test(): pass", "", "exec"))
  1           0 LOAD_CONST               0 (<code object>)
              2 LOAD_CONST               1 ('test')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)
```

伪码

```
test = make_function("test", code)
```

> 通常反汇编操作，是在函数实例创建后执行。目标针对 __code__，而非创建过程。

正因如此，可用 def 指令以单个代码对象为模版创建多个函数实例。

```
def make(n):
    ret = []
```

```
    for i in range(n):
        def test(): print("hello")               # test = make_function(code)

        print(id(test), id(test.__code__))
        ret.append(test)

    return ret
```

```
>>> make(3)
4428346232 4425999248                            # 不同实例，相同代码。
4430389456 4425999248
4430390136 4425999248

[<function __main__.make.<locals>.test>,
 <function __main__.make.<locals>.test>,
 <function __main__.make.<locals>.test>]
```

多个实例，和多个名字引用同一实例并非一回事。
用列表持有多实例。如循环内 test 立即释放，可能因内存复用而出现相同 id 值。

同一名字空间，名字只能与单个目标关联。如此，就不能实现函数重载（overload）。另外，作为第一类对象（first-class object），函数可作为参数和返回值传递。

```
>>> def test(op, x, y):
        return op(x, y)

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

>>> test(add, 1, 2)                      # 将函数作为参数。
3
```

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

        return hello                      # 将函数作为返回值。

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

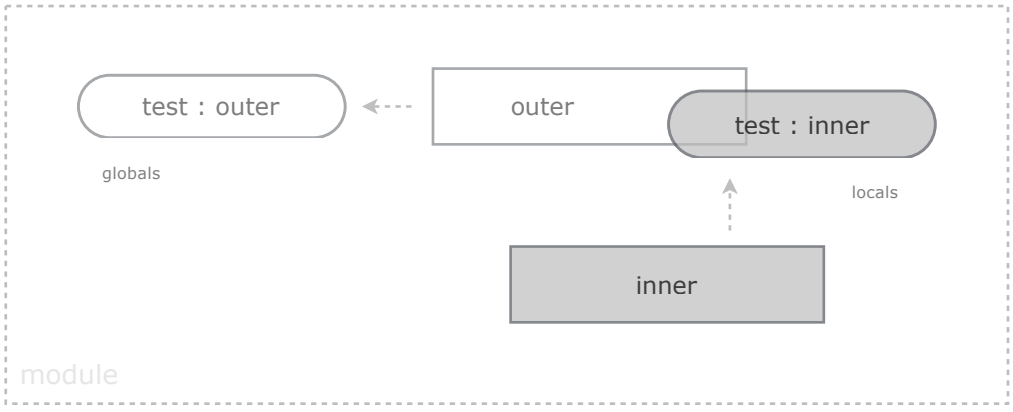
嵌套

支持函数嵌套，甚至与外层函数同名。

```
def test():
    print("outer test")

    def test():
        print("inner test")

    return test
```

```
>>> x = test()
outer test

>>> x
<function __main__.test.<locals>.test>

>>> x()
inner test
```

内外函数名字虽相同，但分属不同层次名字空间。依优先级而定，并不冲突。



outer

```
>>> dis.dis(test)
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('outer test')
              4 CALL_FUNCTION            1
              6 POP_TOP

  3           8 LOAD_CONST               2 (<code object test>)
             10 LOAD_CONST               3 ('test.<locals>.test')
             12 MAKE_FUNCTION            0
```

```
              14 STORE_FAST               0 (test)              # 保存到当前名字空间。

  5           16 LOAD_FAST                0 (test)              # 从当前名字空间返回。
              18 RETURN_VALUE
```

匿名函数

匿名函数正式名称为 lambda 表达式。

相比普通函数，有所限制。其内容只能是单个表达式（函数调用也属表达式），不能使用语句。当然，也不能提供函数名，否则就不能称作匿名函数。除此之外，使用方法与普通函数并无差异。

```
>>> add = lambda x, y: x + y
>>> add
<function __main__.<lambda>>

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

普通函数总有个默认名字（__name__），用以标识真实身份。它是编译期静态绑定，与运行期变量名引用无关。

PEP3155 : 还可用 __qualname__ 获取目标在模块中的完整定义路径。

```
>>> def test(): pass

>>> a = test

>>> a.__name__
'test'

>>> a
<function test at 0x1099b29d8>
```

而 lambda 只有变量引用，没有自己的名字。

```
>>> test = lambda: None
```

```
>>> a = test

>>> a.__name__
'<lambda>'
```

就连创建过程都是路人甲待遇。

```
>>> dis.dis(compile("def test(): pass", "", "exec"))
  1           0 LOAD_CONST               0 (<code object test>)       # 有名有姓。
              2 LOAD_CONST               1 ('test')                   # 静态名字。
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)                     # 在上下文中使用自己的名字。
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

```
>>> dis.dis(compile("lambda: None", "", "exec"))
  1           0 LOAD_CONST               0 (<code object <lambda>>)   # 没有名字，统统路人甲。
              2 LOAD_CONST               1 ('<lambda>')               # 路人甲。
              4 MAKE_FUNCTION            0                            # 创建完了，直接返回。
              6 POP_TOP                                               # 如没有变量赋值，那就走丢了。
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

但是，在适用场合，lambda 远比普通函数更加灵活和自由。

```
>>> map(lambda x: x ** 2, range(3))                # 直接作为参数。
```

```
>>> ops = {
        "add": lambda x, y: x + y,                 # 构建方法表。
        "sub": lambda x, y: x - y,
   }

>>> ops["add"](2, 3)
5
```

```
>>> def make(n):
        return [lambda: print("hello") for i in range(n)]    # 作为推导式输出结果。
```

同样支持嵌套，还可直接调用。

```
>>> test = lambda x: (lambda y: x + y)       # 将另一个 lambda 作为返回值，支持闭包。

>>> add = test(2)
>>> add(3)
5
```

```
>>> (lambda x: print(x))("hello")            # 使用括号避免语法错误。
hello
```

2. 参数

如果动态语言是妖精，那么 Python 对参数的处理方式就是鼓动她去魅惑世人。我相信，再难找到如这般功能丰富得让人“头疼”的语言。

按定义和传参方式，参数可分作位置（positional）和键值（keyword）两类。允许设置默认值和余参收集，但不支持参数嵌套。

形参出现在函数定义的参数列表中，可视作函数局部变量，仅能在函数内部使用。而实参由调用方提供，通常以复制方式将值传递给形参。形参在函数调用结束后销毁，而实参则受调用方作用域影响。不同于形参以变量形式存在，实参可以是变量、常量、表达式等，总之须有确定值可供复制传递。

不管实参是名字、引用，还是指针，都以值复制方式传递，随后形参变化不会影响实参。当然，对该指针或引用目标的修改，与此无关。

形参如普通局部变量出现在函数名字空间内。实参按顺序传递，也可以星号展开。

```
def test(a, b, c = 3):
    print(locals())
```

```
>>> test(1, 2)                      # 忽略有默认值参数。
{'c': 3, 'b': 2, 'a': 1}

>>> test(1, 2, 30)                  # 为默认值参数显式提供实参。
{'c': 30, 'b': 2, 'a': 1}

>>> test(*(1, 2, 30))               # 星号展开。
{'c': 30, 'b': 2, 'a': 1}
```

使用命名方式传递时，无需理会参数顺序。这对于字典展开非常方便。

```
>>> test(b = 2, a = 1)
{'c': 3, 'b': 2, 'a': 1}

>>> test(**{"b": 2, "a":1})          # 键值展开后，等同命名传递。
{'c': 3, 'b': 2, 'a': 1}
```

如混用两种方式，须确保顺序在命名之前。

```
>>> test(1, c = 30, b = 2)
{'c': 30, 'b': 2, 'a': 1}

>>> test(c = 30, 1, 2)
SyntaxError: positional argument follows keyword argument
```

位置参数

位置参数按排列顺序，又可细分为：

1. 普通位置参数，零到多个。
2. 有默认值的位置参数，零到多个。
3. 单星号收集参数，仅一个。

收集参数将多余的参数值收纳到一个元组对象里。所谓多余，是指对普通参数和有默认值参数全部赋值以后的结余。

```
def test(a, b, c = 3, d = 4, *args):
    print(locals())
```

```
>>> test(1, 2, 33)                                    # 不足以填充普通参数和默认值参数。
{'args': (), 'd': 4, 'c': 33, 'b': 2, 'a': 1}
```

```
>>> test(1, 2, 33, 44, 5, 6, 7)                          # 填充完普通和默认值参数后，收集剩余参数值。
{'args': (5, 6, 7), 'd': 44, 'c': 33, 'b': 2, 'a': 1}
```

不能对收集参数命名传参。

```
def test(a, *args):
    pass
```

```
>>> test(a = 1, args = (2, 3))
TypeError: test() got an unexpected keyword argument 'args'
```

键值参数

最简单做法，是在位置参数列表后放置一个键值收集参数。

```
def test(a, b, *args, **kwargs):
    print(kwargs)
```

```
>>> test(1, 2, 3, x = 1, y = 2)           # 多余位置参数被 args 收集。
{'x': 1, 'y': 2}

>>> test(b = 2, a = 1, x = 3, y = 4)      # kwargs 收集多余的键值参数。
{'x': 3, 'y': 4}
```

键值收集仅针对命名传参，对多余的位置参数没兴趣。

```
def test(a, b, **kwargs): pass
```

```
>>> test(1, 2, 3, x = 1)
TypeError: test() takes 2 positional arguments but 3 were given
```

在此基础上，Python 3 新增了一种名为 keyword-only 的键值参数类型。

1. 与位置参数列表分隔边界，星号。
2. 普通 keyword-only 参数，零到多个。
3. 有默认值的 keyword-only 参数，零到多个。
4. 双星号键值收集参数，仅一个。

无默认值的 keyword-only 必须显式命名传参，否则视为普通位置参数。

```
def test(a, b, *, c):
    print(locals())
```

```
>>> test(1, 2, 3)
TypeError: test() takes 2 positional arguments but 3 were given

>>> test(1, 2)
TypeError: test() missing 1 required keyword-only argument: 'c'
```

```
>>> test(1, 2, c = 3)
{'c': 3, 'b': 2, 'a': 1}
```

即便没有位置参数，keyword-only 也须按规则传递。

```
def test(*, c): pass
```

```
>>> test(1)
TypeError: test() takes 0 positional arguments but 1 was given

>>> test(c = 1)
```

除单星号外，位置收集参数（*args）也可作为边界。但只取其一，不能同时出现。

```
def test(a, *args, c, d = 99, **kwargs):
    print(locals())
```

```
>>> test(1, 2, 3, c = 88, x = 10, y = 20)
{
         a : 1,
      args : (2, 3),
         c : 88,
         d : 99,
    kwargs : {'x': 10, 'y': 20},
}
```

同样，不应对键值收集参数命名传参。其结果是弄巧成拙，被当作普通参数收集。

```
def test(**kwargs):
    print(kwargs)
```

```
>>> test(kwargs = {"a":1, "b":2})      # 被当作普通键值参数收集，kwargs["kwargs"]。
{'kwargs': {'a': 1, 'b': 2}}

>>> test(**{"a":1, "b":2})             # 这才是正确的收集姿势。
{'a': 1, 'b': 2}
```

默认值

参数默认值允许省略实参传值，让函数调用更加灵活。尤其是那些参数众多，或具有缺省设定的函数。

但需注意，默认值在函数创建时生成，保存到特定属性（__defaults__），为每次调用所共享。如此，其行为类似静态局部变量，会“记住”以往调用状态。

```
def test(a, x = [1, 2]):
    x.append(a)
    print(x)
```

```
>>> test.__defaults__
```

```
([1, 2],)
```

默认值对象作为函数构建参数存在。

```
>>> dis.dis(compile("def test(a, x = [1, 2]): pass", "", "exec"))
  1           0 LOAD_CONST               0 (1)
              2 LOAD_CONST               1 (2)
              4 BUILD_LIST               2                         # 构建默认值对象。
              6 BUILD_TUPLE              1                         # 作为构建参数。
              8 LOAD_CONST               2 (<code object test>)
             10 LOAD_CONST               3 ('test')
             12 MAKE_FUNCTION            1                         # 参数 1 表示包含缺省参数。
```

如默认值为可变类型，且在函数内做了修改（比如本节示例）。那么后续调用会观察到本次改动，导致默认值失去原本含义。

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

>>> test(4)                    # 在上次调用基础上，添加。
[1, 2, 3, 4]
```

故建议默认值选用不可变类型，或以 None 表示可忽略。

```
def test(a, x = None):
    x = x or []                # 忽略时，主动新建。
    x.append(a)
    return x
```

```
>>> test(1)
[1]

>>> test(2)
[2]

>>> test(3, [1, 2])           # 提供非默认值实参。
[1, 2, 3]
```

要说静态局部变量还是有实际用处的，可在不用外部变量的情况下维持函数状态。比如，用来设计调用计数等。但相比参数默认值，正确做法是为函数创建一个状态属性。毕竟变量为函数内部使用，而参数属对外接口。所创建属性等同函数对象生命周期，不会随函数调用结束而终结。

```
def test():
    test.__x__ = hasattr(test, "__x__") and test.__x__ + 1 or 1
    print(test.__x__)
```

```
>>> test()
1

>>> test()
2

>>> test()
3
```

形参赋值

总结解释器对形参赋值过程。

1. 按顺序对位置参数赋值。
2. 按命名方式对指定参数赋值。
3. 收集多余的位置参数。
4. 收集多余键值参数。
5. 为没有赋值的参数设置默认值。
6. 检查参数列表，确保非收集参数都已赋值。

收集参数 args、kwargs 属习惯性命名，非强制。

对应形参顺序，实参也有些基本规则。

- 无默认值参数，必须有实参传入。
- 键值参数总是以命名方式传入。
- 不能对同一参数重复传值。

无论是以位置和命名两种不同方式，还是多个星号展开里有重复主键，都不能导致对同一参数重复传值。

```
def test(a, b): pass
```

```
>>> test(1, 2, a = 1)
TypeError: test() got multiple values for argument 'a'

>>> test(**{"a":1, "b":2}, **{"a":1})
TypeError: test() got multiple values for keyword argument 'a'

>>> test(*(1, 2), **{"a":1})
TypeError: test() got multiple values for argument 'a'
```

键值收集参数会维持传入顺序（PEP468）。如键值参数存在次序依赖，那么此功能就有实际意义。还有，收集参数并不计入 __code__.co_argcount 中。

签名设计

设计一个 print 函数，那么其理想签名应该是：

```
def printx(*objects, sep = ",", end = "\n"):
```

参数分作两部分：待显示对象为主，可忽略的显示设置（options）为次。从设计角度讲，待显示对象是外部资源，而设置项用于控制函数自身行为，分属不同范畴。如同卡车所装载货物，与车自身控制系统的差别。

据此接口，对象数量未定，设置除默认值外，还可显式调整。如使用 Python 2，作为收集参数的 objects 就只能放在参数列表尾部。

```
def printx(sep = ",", end = "\n", *objects):
    print(locals())
```

除主次不分导致不美观外，最大麻烦是不能绕开默认设置，单独为 objects 传值。加上收集参数无法命名传参，直接导致默认配置项毫无意义。

```
>>> printx(1, 2, 3)
{'objects': (3,), 'end': 2, 'sep': 1}

>>> printx(objects = (1, 2))
TypeError: printx() got an unexpected keyword argument 'objects'
```

幸好，Python 3 提供 keyword-only 参数方式可完美解决此问题。

```
def printx(*objects, sep = ",", end = "\n"):
    print(locals())
```

```
>>> printx(1, 2, 3)
{'objects': (1, 2, 3), 'end': '\n', 'sep': ','}

>>> printx(1, 2, 3, sep = "|")
{'objects': (1, 2, 3), 'end': '\n', 'sep': '|'}
```

参数列表还不宜过长，可尝试将部分参数重构为复合对象。
复合参数变化与函数分离。添加字段，或修改缺省值，不影响已有用户。

---

3. 返回值

函数没有返回值定义，具体返回什么，返回几个，全凭“任性”。

```
def test(n):
    if n > 0:
        return 1, 2, 3
    elif n < 0:
        return -1, -2

    return 0
```

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

>>> test(-1)
(-1, -2)

>>> test(0)
0
```

这种返回数量忽多忽少的写法，在有明确返回值定义的语言里是决不允许的。那么，使用时只能看帮助，或翻阅源代码？就没什么规矩么？

从实现角度看，只要返回数量多于一，编译器就将其打包成元组对象。如此一来，所谓忽多忽少，无非是单个元组里元素数量多寡，完全可以理解。

```
def test(a, b):
    return "hello", a + b
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 ('hello')
              2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_ADD
              8 BUILD_TUPLE              2             # 打包成元组，返回。
             10 RETURN_VALUE
```

```
>>> x = test(1, 2)
```

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

>>> x
('hello', 3)
```

即便什么都不做，也会返回 None。如此，可以说函数总是返回一个结果。

只有这样，函数才能保证作为表达式出现。

```
>>> def test(): pass

>>> dis.dis(test)
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

```
>>> def test(): return

>>> dis.dis(test)
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

至于用多变量接收返回值，实际是序列解包在起作用。约定俗成，习惯性表述而已。

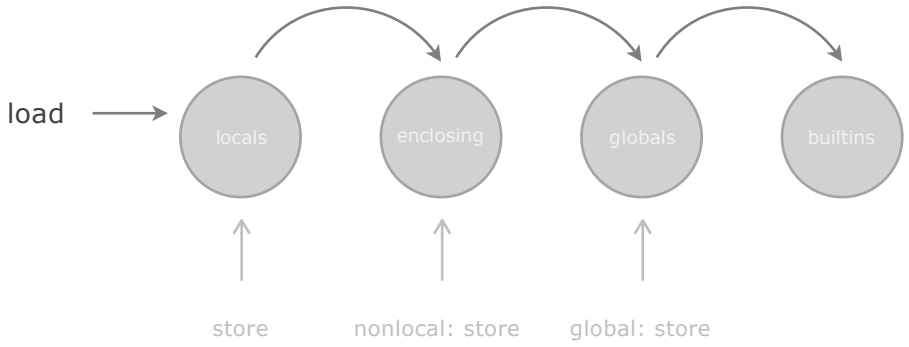
```
>>> dis.dis(compile("a, b = test()", "", "exec"))
  1           0 LOAD_NAME                0 (test)
              2 CALL_FUNCTION            0
              4 UNPACK_SEQUENCE          2                 # 解包
              6 STORE_NAME               1 (a)
              8 STORE_NAME               2 (b)
             10 LOAD_CONST               0 (None)
             12 RETURN_VALUE
```

4. 作用域

与赋值针对当前名字空间，或以 global、nonlocal 关键字作外部声明不同。在函数内访问变量，会以特定顺序依次查找不同层次作用域。

```
>>> import builtins
>>> builtins.B = "B"

>>> G = "G"

>>> def enclosing():
        E = "E"

        def test():
            L = "L"
            print(L, E, G, B)

        return test


>>> enclosing()()
L E G B
```

此规则，简称作 LEGB。



内存结构

函数每次调用，都会新建栈帧（stack frame），用于分配局部变量，以及执行过程存储。等执行结束，栈帧内存被回收，同时引发相关对象释放操作。

```
def test():
   print(id(locals()))
```

```
>>> test()
4460820664

>>> test()
4458753840
```

是否栈帧内存就是 locals 名字空间？

以字典实现的名字空间，虽然灵活，但存在访问效率低下等问题。对于使用频率相对较低的模块名字空间尚可，可对于有性能要求的函数调用，显然就是瓶颈所在。

为此，解释器划出专门的内存空间，改以效率最快的数组替代哈希字典。在函数指令执行前，先将包含参数在内的所有局部变量，以及要使用的外部变量复制（指针）到该数组。基于作用域不同，此内存区域可简单分作两部分：FAST 和 DEREF。

如此，操作指令只需用索引即可立即读取或存储目标对象，远比哈希查找过程高效许多。从前文反汇编开始，我们就看到大量类似 LOAD_FAST 的指令，其参数就是索引号。

```
def enclosing():

    E = "E"

    def test(a, b):
        c = a + b
        print(E, c)

    return test
```

```
>>> t = enclosing()                  # 返回 test 函数。

>>> t.__code__.co_varnames           # 局部变量列表（含参数）。与索引号对应。
('a', 'b', 'c')

>>> t.__code__.co_freevars           # 所引用外部变量列表。与索引号对应。
```

```
('E',)
```

```
>>> dis.dis(t)
  4           0 LOAD_FAST                0 (a)       # 从 FAST 区域，以索引号访问并载入。
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 STORE_FAST               2 (c)       # 将结果存入 FAST 区域。

  5           8 LOAD_GLOBAL              0 (print)
             10 LOAD_DEREF               0 (E)       # 从 DEREF 区域，访问并载入外部变量。
             12 LOAD_FAST                2 (c)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

FAST 和 DEREF 数组大小是统计参数和变量得来，对应索引值也是编译期所确定。所以，不能在运行期扩张。前文曾提及，global 关键字可向全局名字空间新建名字，但 nonlocal 不允许。其原因就是 nonlocal 代表外层函数，无法动态向其 FAST 数组插入或追加新元素。

另外，LEGB 中的 E 已被保存到 DEREF 数组，相应查找过程也被优化，无需费时费力去迭代调用堆栈（call stack），一级级扒拉外层函数。LEGB 是针对源码的说法，而非内部实现。

名字空间

现在的问题是，为何 locals 函数返回的是字典类型？实际上，除非调用该函数，否则函数执行期间，根本不会创建所谓名字空间字典。也就是说，其返回的字典是按需延迟创建，并从 FAST 区域复制相关信息而得来。

在了解此背景后，类似下面的问题就很好解释。

```
def test():
    locals()["x"] = 100                 # 运行期通过名字空间新建变量。
    print(x)                            # 编译此指令时，本地并没有 x 这个名字。
```

```
>>> test()                              # 失败。
NameError: name 'x' is not defined
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 (100)
              2 LOAD_GLOBAL              0 (locals)
              4 CALL_FUNCTION            0
              6 LOAD_CONST               2 ('x')
              8 STORE_SUBSCR

  3          10 LOAD_GLOBAL              1 (print)
             12 LOAD_GLOBAL              2 (x)          # 编译时确定，从全局而非 FAST 载入。
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

很显然，名字使用静态作用域。运行期行为，对此并无效果。另一方面，所谓 locals 名字空间不过是 FAST 复制品，对其变更不会同步到 FAST 区域。

```
def test():
    x = 100
    locals()["x"] = 999                    # 新建字典，复制。对复制品修改不会影响 FAST。

    print("fast.x =", x)
    print("locals.x =", locals()["x"])     # 从 FAST 刷新，修改丢失。
```

```
>>> test()
fast.x = 100
locals.x = 100
```

至于 globals 能新建并影响，是因为模块直接以字典实现名字空间，没有类似 FAST 机制。

在 Python 2 里，可通过插入 exec 语句影响名字作用域静态绑定。但随 Python 3 将其变成函数，此方法也告失效。

栈帧会保存 locals 函数所返回字典，避免每次新建。如此，可用它存储额外数据，比如向后续调用提供上下文状态等。但请注意，只有再次调用该函数，才会刷新字典数据。

```
def test():
    x = 1

    d = locals()
    print(d is locals())           # ① 每次返回同一字典对象。
```

```
    d["context"] = "hello"          # ② 存储额外数据。
    print(d)

    x = 999                         # ③ 修改 FAST 时，不会主动刷新 locals 字典。
    print(d)

    print(locals())                 # ④ 刷新操作由 locals 调用触发。
    print(d)
```

```
>>> test()
① True
② {'x': 1, 'context': 'hello'}
③ {'x': 1, 'context': 'hello'}
④ {'x': 999, 'context': 'hello'}
④ {'x': 999, 'context': 'hello'}
```

静态作用域

在对待作用域这个问题上，也许编译器比你想象的还要“笨”一些。

```
def test():
    if False: x = 100
    print(x)
```

```
>>> test()
UnboundLocalError: local variable 'x' referenced before assignment

>>> dis.dis(test)
  4           0 LOAD_GLOBAL              0 (print)
              2 LOAD_FAST                0 (x)            # x 作用域：FAST
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

虽然编译器将死代码剔除了，但其对 x 作用域的影响依然存在。

似乎也不能说编译器笨。既然程序猿在代码里写了赋值语句，那么在最初设想里，x 显然是本地变量。至于最后死代码执行与否，就是另一回事了。

```
def test():
    if False: global x
    x = 100
```

```
>>> dis.dis(test)
  4           0 LOAD_CONST               1 (100)
              2 STORE_GLOBAL             0 (x)          # x 作用域：GLOBAL
              4 LOAD_CONST               0 (None)
              6 RETURN_VALUE
```

建议

函数最好设计为纯函数，仅依赖参数、内部变量和自身属性。依赖外部状态，会给重构和测试带来诸多麻烦。或许可将外部依赖变成 keyword-only 参数，如此测试就可自定义依赖环境，以确保最终结果一致。

如必须依赖外部变量，那么尽可能不做修改，以返回值交由调用方决策。

纯函数（pure function）输出与输入以外的状态无关，没有任何隐式依赖。相同输入总是输出相同结果，且不对外部环境产生影响。

注意区分函数和方法的设计差异。函数以逻辑为核心，输入条件，“计算”结果。而方法则围绕实例状态，展示或修改。

---

5. 闭包

明面上，闭包（closure）是指函数离开生成环境后，依然可记住，并持续引用词法作用域里的外部变量。

```
def make():
    x = [1, 2]
    return lambda: print(x)
```

```
>>> a = make()
>>> a()
[1, 2]
```

> 如果不考虑闭包因素，这段代码就有很大问题。因为 x 生命周期是 make 调用栈帧。当 make 结束后，x 理应被销毁。但结果是，所返回匿名函数依然可以访问，这就是所谓闭包效应。

关于闭包，有很多学术解释。简单点说，就是函数和所引用环境变量的组合体。从这点上说，闭包不等于函数，只是形式上返回一个函数而已。因引用外部状态，闭包函数自然也不是纯函数。加上延长环境变量生命周期，理应慎重使用。

创建

既然闭包由两部分组成，那么其创建过程可分为：

1. 打包环境变量。
2. 将环境变量作为参数，新建函数对象。

> 同样因生命周期改变的缘故，环境变量作用域从 FAST 转移到 DEREF。
> 待返回闭包函数，可以是 lambda，或普通函数。

```
def make():
    x = 100
    return lambda: print(x)
```

```
>>> dis.dis(make)
  2           0 LOAD_CONST               1 (100)
              2 STORE_DEREF              0 (x)                          # 作用域改变。

  3           4 LOAD_CLOSURE             0 (x)                          # 打包。
              6 BUILD_TUPLE              1
              8 LOAD_CONST               2 (<code object <lambda>>)
             10 LOAD_CONST               3 ('make.<locals>.<lambda>')
             12 MAKE_FUNCTION            8                              # 创建函数对象。
             14 RETURN_VALUE
```

同样，闭包函数也得从 DEREF 读取环境变量。

```
>>> f = make()
>>> dis.dis(f)
  3           0 LOAD_GLOBAL              0 (print)
              2 LOAD_DEREF               0 (x)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

自由变量

我们将闭包所引用环境变量称为自由变量（free variable）。在创建函数对象时，它被保存到 __closure__ 属性。

```
def make():
    x = [1, 2]
    print(hex(id(x)))
    return lambda: print(x)
```

```
>>> f = make()
0x10eeae108

>>> f.__closure__
(<cell at 0x10ed47198: list object at 0x10eeae108>,)

>>> f()
[1, 2]
```

所引用自由变量名，可通过代码对象属性获取。

```
>>> f.__code__.co_freevars            # 所引用外部自由变量名列表。
('x',)

>>> make.__code__.co_cellvars         # 被闭包函数引用的变量名列表。
('x',)
```

自由变量保存在函数对象里，那多次调用是否被覆盖？自然不会，因为所返回函数对象也是每次新建。要知道，创建闭包等于“新建函数对象，附加自由变量”。

```
def make(x):
    return lambda: print(x)
```

```
>>> a = make([1, 2])
>>> b = make(100)

>>> a is b                                          # 每次返回新的函数对象实例。
False

>>> a.__closure__                                   # 自由变量保存在各自独立的函数实例。
(<cell at 0x10ed61c78: list object at 0x10ee7fcc8>,)

>>> b.__closure__
(<cell at 0x10ed47d68: int object at 0x10d0c6970>,)
```

多个函数可共享同一自由变量。

```
def queue():
    data = []

    push = lambda x: data.append(x)
    pop = lambda: data.pop(0) if data else None

    return push, pop
```

```
>>> push, pop = queue()

>>> push.__closure__
(<cell at 0x10ed612e8: list object at 0x10edcae48>,)   # 共享自由变量。
```

```
>>> pop.__closure__
(<cell at 0x10ed612e8: list object at 0x10edcae48>,)
```

```
>>> for i in range(10, 13):
        push(i)

>>> while True:
        x = pop()
        if not x: break
        print(x)

10
11
12
```

闭包让函数持有状态，可部分实现 class 功能。但应局限于特定小范围，避免隐式状态依赖对代码测试、阅读和维护造成麻烦。

自引用

在函数内引用自己，也可构成闭包。

原因很简单，def 创建函数对象，然后将其与函数名字关联。也就是说，该函数实例也属当前环境变量，自然可作为自由变量。

```
def make(x):

    def test():
        test.x = x              # 引用自己。
        print(test.x)           # 引用的是当前函数实例，其效果类似 this。

    return test
```

```
>>> a, b = make(1234), make([1, 2])
```

```
>>> a.__closure__
(<cell at 0x10ed61798: function object at 0x10ed50ea0>,    # 自由变量列表中包含自己。
 <cell at 0x10ed61318: int object at 0x10ee10890>)
```

```
>>> b.__closure__
(<cell at 0x10ed61618: function object at 0x10efb2840>,
 <cell at 0x10ed61918: list object at 0x10ee6d848>)
```

在函数内引用自己时，从 DEREF 载入。自然是专属实例，无须担心数据共享。

```
>>> dis.dis(a)
  3           0 LOAD_DEREF               1 (x)
              2 LOAD_DEREF               0 (test)
              4 STORE_ATTR               0 (x)

  4           6 LOAD_GLOBAL              1 (print)
              8 LOAD_DEREF               0 (test)
             10 LOAD_ATTR                0 (x)
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

```
>>> a()
1234

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

延迟绑定

闭包只是绑定自由变量（非引用目标），并不会立即计算其引用内容。只有当闭包函数执行时，才与所引用目标对象交互。如此，就有所谓延迟绑定（late binding）现象。

```
def make(n):
    x = []

    for i in range(n):
        x.append(lambda: print(i))

    return x
```

```
>>> a, b, c = make(3)
```

```
>>> a(), b(), c()
2
2
2
```

```
>>> a.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)

>>> b.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)

>>> c.__closure__
(<cell at 0x10efaf798: int object at 0x10d0c5d30>,)
```

输出结果不是 0, 1, 2 ?

整理一下执行次序：

1. make 创建并返回 3 个闭包函数，引用同一自由变量 i。
2. make 执行结束，i 等于 2。
3. 执行闭包函数，引用并输出 i 的值，自然都是 2。

就这么简单?

从 __closure__ 看，函数并不直接存储自由变量，而是 cell 包装对象，以此间接引用目标。每个自由变量都被打包成一个 cell。循环期间虽然也和 i 一样引用不同整数对象，但这对尚未执行的闭包函数没有影响。循环结束，cell 引用目标确定下来，这才是闭包函数执行时输出结果。

按其他语言经验，解决延迟绑定问题，要么立即复制，要么引用不同自由变量。但改成复制后，结果并不如人意。

```
def make(n):
    x = []
    for i in range(n):
        c = i                               # 复制对象? 其实并不是，这是个坑。
        x.append(lambda: print(c))

    return x
```

```
>>> a, b, c = make(3)
```

```
>>> a(), b(), c()
2
2
2
```

未能得到预期结果。原因并不复杂，变量 c 的作用域是 make 函数，而非 for 语句。也就是说，不管执行多少次循环，也仅有一个 c 存在。如此，闭包函数依然绑定同一自由变量，这与复制目标对象无关。

这是不同语言作用域规则不同而导致的经验错误。

```
>>> a.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)      # 仍是同一对象。

>>> b.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)

>>> c.__closure__
(<cell at 0x10ed47d38: int object at 0x10d0c5d30>,)
```

顺着这个思路，我们可换个坚决不共享的变量来，比如默认值。

```
def make(n):
    x = []
    for i in range(n):
        x.append(lambda o = i: print(o))

    return x
```

```
>>> a, b, c = make(3)
>>> a(), b(), c()
0
1
2
```

问题解决了。因为匿名函数参数 o 是私有变量，设置默认值时，复制当前 i 引用。虽然只有一个变量 i，但循环过程中，它指向不同整数对象。所以，复制的引用也就不同。

最重要的是，这样一来，就没有闭包了。^_^

```
>>> a.__closure__
>>> b.__closure__
>>> c.__closure__

>>> make.__code__.co_cellvars
()
```

优缺点

闭包的缺点可能和优点一样明显。

具备封装特征，可实现隐式上下文状态，并减少参数。在设计上，可部分替代全局变量，或将执行环境与调用接口分离。

首要缺点，对自由变量隐式依赖，会提升代码复杂度，直接影响测试和维护。其次，自由变量生命周期的提升，会提高内存占用。

应控制隐式依赖的范围和规模，能省则省。

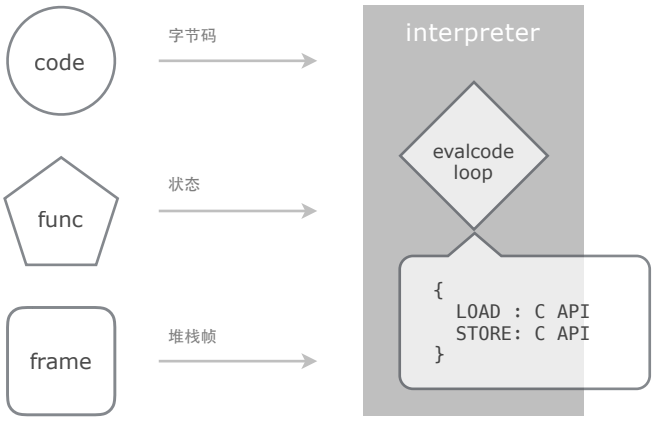
---

6. 调用

在前文不同章节里，多次提及字节码解释执行。那么当函数被调用时，具体情形如何？

整个系统核心是称为解释器（interpreter）的组件。这东西从代码看很是疯狂，重度肥胖函数，外加超大循环。让我们抛开技术名词，用个例子来说明其中的关系。

假设解释器是台 ATM 取款机。当储户发出“取款”指令（字节码），机器触发预置功能列表中与之对应的操作，以银行卡为参数，检查并修改账户数据，然后出钞。所谓指令不过是内部某个功能的“名字”而已，仅作为选择条件，并不参与机器执行。



在解释器内部，每条字节码指令对应一个完全由 C 实现的逻辑。

解释器运行在系统线程上，那如何处理内部系统代码和用户代码数据？从反汇编结果看，就算字节码指令被解释为内部调用，可依然有参数和返回值需要存储。

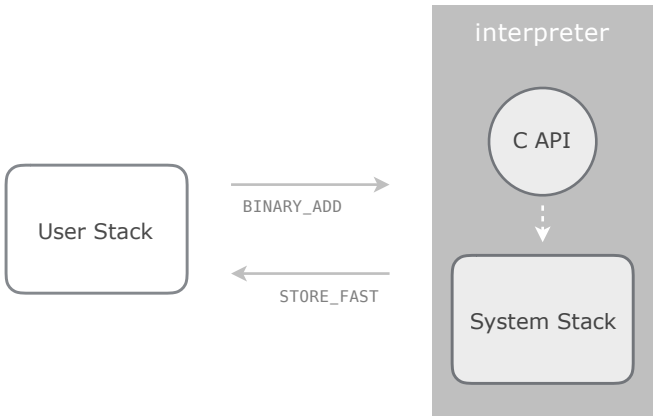
继续以上面例子解释，这里实际有两个存储空间，机器内部（系统栈）和储户钱包（用户栈）。取款时，银行卡从钱包传递到机器，最后连同钞票放回钱包。在操作完成后，机器准备下次交易，本次数据可被清除。与用户相关数据都在钱包内。所以说，系统栈用于机器执行，用户栈才是用户代码专用。

当函数被调用时，会专门为其分配用户栈内存。除用来存储变量外，还包括字节码参数和返回值所需。对系统指令来说，这里只能存放用户指令数据。如此，两方各有所属，确保数据互不影响。

```
def add(a, b):
```

```
    c = a + b
    return c
```

```
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (a)     # 从 FAST 读取参数 a，压入用户栈。
              2 LOAD_FAST                1 (b)     # 从 FAST 读取参数 b，压入用户栈。
              4 BINARY_ADD                         # 系统指令从用户栈读取操作数，执行加法操作。
              6 STORE_FAST               2 (c)     # 将结果写回 FAST。

  3           8 LOAD_FAST                2 (c)
             10 RETURN_VALUE
```



最后，编译器没有函数内联（inline），没有深度优化。即便是空函数，依旧生成字节码指令，需要解释器执行。

解释器命令行参数 -O 和 -OO，并非编译指令优化开关，而是为了移除调试代码和帮助文档。

```
>>> code ="""
    def test(): pass
    test()
"""

>>> dis.dis(compile(code, "", "exec", optimize = 2))          # 启用 -OO 优化方式。
  2           0 LOAD_CONST               0 (<code object>)
              2 LOAD_CONST               1 ('test')
              4 MAKE_FUNCTION            0                  # 创建函数。
              6 STORE_NAME               0 (test)

  3           8 LOAD_NAME                0 (test)
             10 CALL_FUNCTION            0                  # 调用函数。
             12 POP_TOP
             14 LOAD_CONST               2 (None)
```

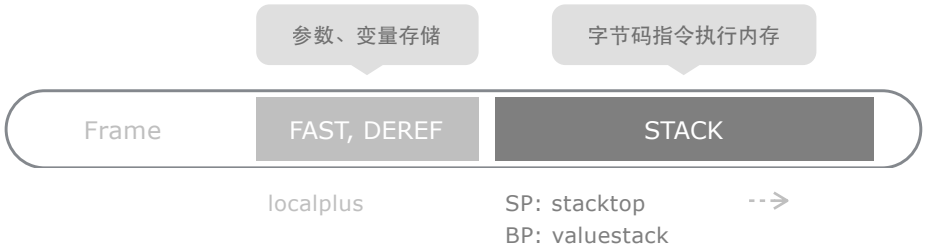
```
16 RETURN_VALUE
```

调用堆栈

我们通常将进程内存分作堆（heap）和栈（stack）两类。堆可自由申请，通过指针存储自由数据。而栈则用于指令执行，与线程相绑定。函数调用和执行都依赖线程栈存储上下文和执行状态。

在函数 A 内调用函数 B，须确保 B 结束后能回转到 A，继续执行后续指令。这就要求将 A 后续指令地址预先存储起来，调用堆栈（call stack）基本用途就在如此。

除返回地址外，还需为函数提供参数、局部变量存储空间。依不同调用约定，甚至要为被调用函数提供参数和返回值内存。显然，在线程栈这块内存里，每个被调用函数都划有一块保留地，我们将其称作栈帧（stack frame）。



> 基于栈式虚拟机（Stack-based VM）设计的字节码指令集，没有寄存器相关概念，内部使用变量实现类似 SP、BP、PC/IP 寄存器功能。

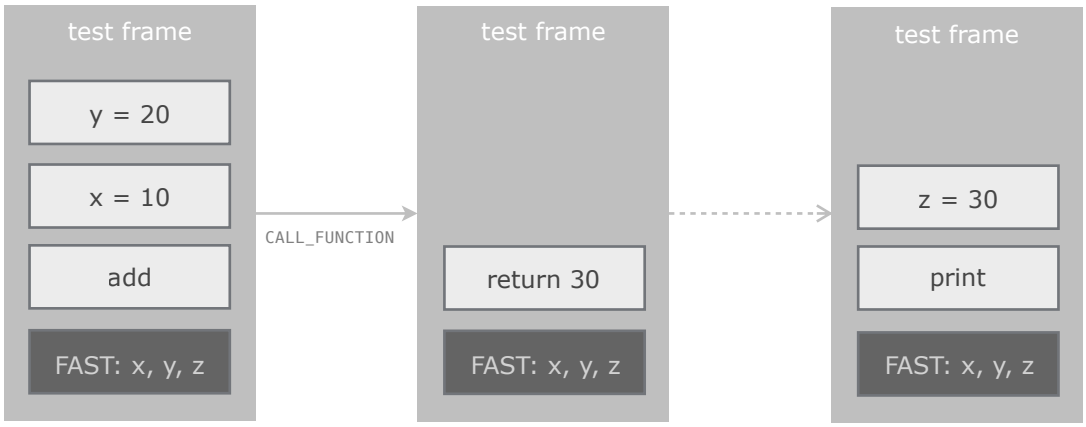
因解释执行缘故，字节码指令数据使用独立的用户栈空间。且与系统栈连续内存不同，用户栈帧由独立对象实现，以链表形式构成完整调用堆栈。好处是不受系统栈大小制约，缺点是性能要差上一些。但考虑到它只存储数据，实际执行过程依然在系统栈完成，倒也能接受。

> 栈帧使用频繁，系统最多缓存 200 个栈帧对象，并按函数实际所需调整其内存大小。
> 操作系统对栈大小限制可使用 ulimit -s 查看，最新 64 位系统通常为 8 MB。
> 一旦函数执行（比如递归）内存超出限制，就会引发堆栈溢出（stack overflow）错误。

栈帧内，变量内存通常固定不变，执行内存视具体指令重复使用。

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

def test():
    x = 10
    y = 20
    z = add(x, y)
    print(z)
```

```
>>> dis.dis(test)
  2           0 LOAD_CONST               1 (10)
              2 STORE_FAST               0 (x)

  3           4 LOAD_CONST               2 (20)
              6 STORE_FAST               1 (y)

  4           8 LOAD_GLOBAL              0 (add)       # 将待调用函数 add 入栈。
             10 LOAD_FAST                0 (x)         # 将变量 x 入栈。
             12 LOAD_FAST                1 (y)         # 将变量 y 入栈。
             14 CALL_FUNCTION            2             # 调用函数（2 为参数数量）。
             16 STORE_FAST               2 (z)         # 将返回值从栈保存到变量区。

  5          18 LOAD_GLOBAL              1 (print)
             20 LOAD_FAST                2 (z)
             22 CALL_FUNCTION            1
             24 POP_TOP                                # 清除 print 返回值，确保栈平衡。
             26 LOAD_CONST               0 (None)
             28 RETURN_VALUE
```



调用堆栈常出现调试工具中，用于检视调用过程，以及各级环境变量取值。当然，也可在代码中使用，比如获取上级函数设置的上下文信息。函数 sys._getframe 可访问调用堆栈内不同层级栈帧对象。参数 0 表示当前函数，1 为上一级函数，如此类推。

```
def A():
    x = "func A"
    B()

def B():
    C()

def C():
    f = sys._getframe(2)      # 向上 2 级，获取 A 栈帧。
    print(f.f_code)           # A 代码对象。
    print(f.f_locals)         # A 名字空间。（此时为运行期）
    print(f.f_lasti)          # A 最后执行指令偏移量。（以确定继续执行位置）
```

```
>>> A()
<code object A>
{'x': 'func A'}
6
```

```
>>> dis.dis(A)
  2           0 LOAD_CONST               1 ('func A')
              2 STORE_FAST               0 (x)

  3           4 LOAD_GLOBAL              0 (B)
              6 CALL_FUNCTION            0                  <------ A.lasti
              8 POP_TOP
             10 LOAD_CONST               0 (None)
             12 RETURN_VALUE
```

请注意，无论是在函数内调用 globals，还是通过 frame.f_globals 访问，总返回定义该函数的模块名字空间，而非调用处。

另有 sys._current_frame 返回所有线程当前栈帧，可用来确定解释器工作状态。只是标准库文档将这两个函数都标记为内部使用。在需要时，非调试性代码，可以标准库 inspect 替代，它拥有更完整的操作函数。

```
import inspect

def A(): B()
def B(): C()

def C():
```

```
    for f in inspect.stack():
        print(f.function, f.lineno)

A()
```

输出：

```
$ python test.py

C 7
B 4
A 3
<module> 10
```

如果只是简单查看调用过程，可直接用 traceback 模块，类似解释器输出错误信息那般形式。

递归

递归深度受限，可使用 sys.getrecursionlimit、setrecursionlimit 查看和调整。

```
import sys
sys.setrecursionlimit(50)

def test():
    test()                    # 递归调用。

test()
```

输出：

```
$ python demo.py
RecursionError: maximum recursion depth exceeded
```

递归常被用来改善循环操作，比如树状结构遍历。当然，它须承担函数调用的额外开销，类似栈帧创建等。尤其在不支持尾递归优化的情况下，这种负担尤为突出。

如函数 A 的最后动作是调用 B，并直接返回 B 结果。那么 A 的栈帧状态就无需保留，其内存可直接被 B 覆盖使用。另外，还可将函数调用指令优化跳转指令，大大提升执行性能。如此方式，被称作尾调用消除或尾调用优化（Tail Call Optimization，TCO）。

如果 A 尾调用自身，那么就成了尾递归。鉴于重复使用同一栈帧内存，可避免堆栈溢出。不过 Python 因实现方式不同，对此并不支持。

包装

对已有函数，可通过包装形式改变其参数列表，使其符合特定调用约束。

```
def test(a, b, c):
    print(locals())
```

```
>>> import functools

>>> t = functools.partial(test, b = 2, c = 3)

>>> t(1)
{'c': 3, 'b': 2, 'a': 1}
```

原理很简单，用包装函数合并相关参数后，再调用原目标即可。

示意伪码

```
def partial(func, *part_args, **part_kwargs):

    def wrap(*call_args, **call_kwargs):
        kwargs = part_kwargs.copy()                        # 复制包装键值参数。
        kwargs.update(call_kwargs)                         # 使用调用键值参数更新包装键值参数。
        return func(*part_args, *call_args, **kwargs)      # 按顺序展开参数。

    return wrap
```

按此流程，可确定基本合并规则：

1. 包装位置参数优先。
2. 调用键值参数覆盖包装键值参数。
3. 合并后不能对单个目标参数多次赋值。

```
>>> functools.partial(test, 1, 2)(3)                # 包装位置参数优先。
{'c': 3, 'b': 2, 'a': 1}
```

```
>>> functools.partial(test, 1, c = 3)(2, c = 100)   # 调用键值参数覆盖包装参数。
{'c': 100, 'b': 2, 'a': 1}
```

包装函数内存储有相关信息，可供查阅。

```
>>> t = functools.partial(test, 1, c = 3)

>>> t.func                                          # 原目标函数。
<function __main__.test>

>>> t.args                                          # 包装位置参数。
(1,)

>>> t.keywords                                      # 包装键值参数。
{'c': 3}
```

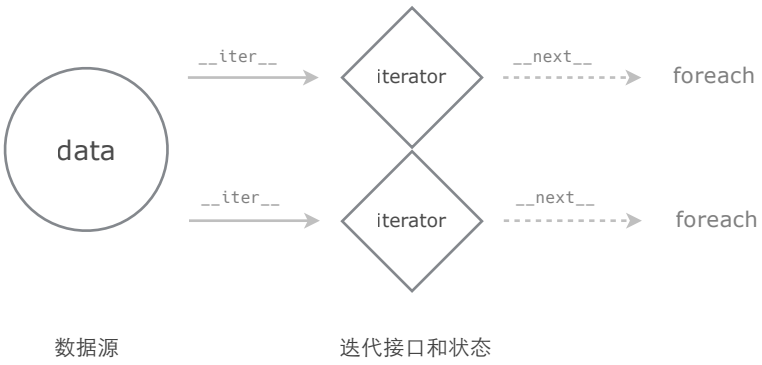
五. 迭代器

1. 迭代器

迭代是指重复从对象中获取数据，直至结束。而所谓迭代协议，概括起来就是用 __iter__ 方法返回一个实现了 __next__ 方法的迭代器对象。

实现 __iter__ 方法，表示目标为可迭代（iterable）类型，允许执行手动或自动迭代操作。该方法新建并返回一个具体的迭代器（iterator）实例。随后，通过调用 iterator.__next__ 依次返回结果，直至抛出 StopIteration 异常表示结束。

我们注意到，迭代器使用了分离设计。首先，对目标对象而言，迭代器只是一种与自身逻辑无关的用户接口，组合显然比内联更合适；其次，迭代分多次完成，需保存其进度。这其中还包括重复迭代，以及同时有多个迭代等情形。如何存储并管理这些状态？莫不如每次按需新建实例，单向存储当前进度。用完即毁，无需考虑太多。



> 可迭代类型未必就是序列数据，也可能是按前后顺序操作的栈、队列，随机返回键值的哈希表，甚至是未知终点的网络数据流。

内置容器类型，还有些常用函数均实现了迭代接口。

```
>>> issubclass(list, collections.Iterable)
True

>>> isinstance(range(3), collections.Iterable)
True
```

```
>>> isinstance(zip([1, 2]), collections.Iterable)
True
```

自定义类型

以 Python 包办家长的性格，肯定有便捷方式创建迭代器类型。但为更好理解协议流程和操作方式，我们先用笨方法逐一实现。

```
class Data:

    def __init__(self, n):
        self.data = list(range(n))

    def __iter__(self):
        return DataIter(self.data)                          # 返回新迭代器实例。
```

```
class DataIter:

    def __init__(self, data):
        self.data = data
        self.index = 0

    def __next__(self):
        if not self.data or self.index >= len(self.data):
            raise StopIteration                             # 抛出异常表示迭代结束。

        d = self.data[self.index]                           # 本次迭代返回数据。
        self.index += 1                                     # 存储迭代进度。
        return d
```

手工迭代

```
>>> d = Data(2)
>>> x = d.__iter__()

>>> x.__next__()
0

>>> x.__next__()
1

>>> x.__next__()
StopIteration
```

自动迭代

```
>>> for i in Data(2): print(i)
0
1
```

因无法通过 __next__ 方法传入参数，自然也就无从重置状态或调整进度。迭代器实例本质上属一次性消费，这也是每次在 __iter__ 里新建的理由。

按理说，迭代器本身也应该是可迭代类型。但因缺少 __iter__ 实现，无法使用。

```
>>> x = Data(2).__iter__()

>>> x.__iter__()
AttributeError: 'DataIter' object has no attribute '__iter__'
```

如此，可为其添加 __iter__ 方法，用于返回自身。

```
class DataIter:

    def __iter__(self):
        return self

    ...
```

辅助函数

前文示例有些笨拙，因为其内在列表容器已经实现迭代接口，直接返回便可，无需画蛇添足。当然，按照名字约定，我们不能直接调用 __iter__ 方法，而应改用 iter 函数。

```
class Data:

    def __init__(self, n):
        self.data = list(range(n))

    def __iter__(self):
        return iter(self.data)
```

```
>>> Data(2).__iter__()
<list_iterator at 0x106288b00>
```

作为辅助函数，iter 还可为序列对象（__getitem__）自动创建迭代器包装。

```
class Data:

    def __init__(self, n):
        self.n = n

    def __getitem__(self, index):
        if index < 0 or index >= self.n: raise IndexError
        return index + 100
```

```
>>> iter(Data(2))
<iterator at 0x1062a82e8>
```

更甚至于对函数、方法等可调用类型（callable）进行包装。

可用于网络和文件等 IO 数据接收，比起循环语句更优雅一些。

```
>>> x = lambda: input("n : ")                # 被 __next__ 调用，无参数。

>>> for i in iter(x, "end"): print(i)        # 函数 x 返回值等于 end 时结束。

n : 1
1
n : 2
2
n : end
```

与 __next__ 方法对应的则是 next 函数，可用于手动迭代。

```
>>> x = iter([1, 2])

>>> while True:
        try:
            print(next(x))
        except StopIteration:
            break
```

```
1
2
```

自动迭代

对于 for 循环语句，编译器会生成迭代相关指令，以实现对协议方法调用。

```
def test():
    for i in [1, 2]: print(i)
```

```
>>> dis.dis(test)
  2           0 SETUP_LOOP              20 (to 22)
              2 LOAD_CONST               3 ((1, 2))
              4 GET_ITER                               # 调用 __iter__ 返回迭代器对象（或包装）。
        >>    6 FOR_ITER                12 (to 20)     # 调用 __next__ 返回数据，结束则跳转。
              8 STORE_FAST               0 (i)
             10 LOAD_GLOBAL              0 (print)
             12 LOAD_FAST                0 (i)
             14 CALL_FUNCTION            1
             16 POP_TOP
             18 JUMP_ABSOLUTE            6             # 跳转，继续迭代。
        >>   20 POP_BLOCK                              # 迭代结束。
        >>   22 LOAD_CONST               0 (None)
             24 RETURN_VALUE
```

从解释器内部实现看，GET_ITER 指令行为与 iter 函数类似。如目标对象实现了 __iter__ 方法，则直接调用，否则尝试创建序列迭代包装。

设计意图

尽管列表、字典等容器类型实现了迭代器协议。但本质上，两者不属于同一层面。迭代器不仅是一种数据读取方法，而更多是一种设计模式。

容器核心是存储，围绕数据提供操作方法，是与用户逻辑无关的开放类型。而迭代器重点是逻辑控制。调用方发出请求，随后决策由迭代器决定。数据内敛，抽象和实现分离。

2. 生成器

生成器（generator）是迭代器的进化版本，改以函数和表达式替代接口方法实现。非但简化了编码过程，还提供更多控制用于复杂模型设计。

生成器函数特殊之处在于，其内部以 yield 返回迭代数据。与普通函数不同，无论内部逻辑如何，它总是返回生成器对象。随后，可以普通迭代器方式操作。

```
def test():
    yield 1
    yield 2
```

```
>>> test()
<generator object test at 0x1060cbf68>
```

```
>>> for i in test(): print(i)
1
2
```

每条 yield 语句对应一次 __next__ 调用。可分列多条，或出现在循环语句中。只要结束函数流程，就相当于抛出迭代终止异常。

```
def test():
    for i in range(10):
        yield i + 100
        if i >= 1: return
```

```
>>> x = test()

>>> next(x)
100

>>> next(x)
101

>>> next(x)
StopIteration
```

子迭代器

如果数据源本身就是可迭代对象，那么可使用 yield from 子迭代器语句。其本质和 for 循环内用 yield 并无不同，只是语法更加简练。

```
def test():
    yield from "ab"
    yield from range(3)
```

```
>>> for o in test(): print(o)
a
b
0
1
2
```

生成器表达式

至于生成器表达式，其规则与推导式完全相同，除了使用小括号。

```
>>> x = (i + 100 for i in range(8) if i % 2 == 0)

>>> x
<generator object <genexpr> at 0x1062f5990>
```

可直接用做函数调用参数。

> 如果不是唯一参数，为避免语法错误，不能省略小括号。

```
def test(x):
    print(x)
    for i in x: print(i)
```

```
>>> test(i for i in range(3))
<generator object <genexpr> at 0x105efff68>
0
```

```
1
2
```

执行

相比普通迭代器，生成器执行过程稍显复杂。

首先，编译器会为生成器函数添加标记。对此类函数，解释器并不直接执行。而是将栈帧和代码作为参数，创建生成器实例。

```
def test(n):
    print("gen.start")

    for i in range(n):
        print(f"gen.yield {i}")
        yield i
        print("gen.resume")
```

```
>>> test.__code__.co_flags                  # generator
99

>>> inspect.isgeneratorfunction(test)
True
```

简单点说，就是以通用生成器类型为模版，实现迭代器协议。至于用户定制部分，则由栈帧和代码对象完成。

```
>>> x = test(2)

>>> x.gi_frame.f_locals                     # 栈帧内存储函数调用参数。
{'n': 2}

>>> x.gi_code                               # 关联用户函数。
<code object test>
```

```
>>> x.__next__                              # 实现迭代器协议方法。
<method-wrapper '__next__' of generator>
```

所谓函数调用，不过是错觉。这也算解释执行的好处，起码不用插入额外的汇编代码。接下来，生成器对象在第一次 __next__ 调用时触发，进入用户函数执行。

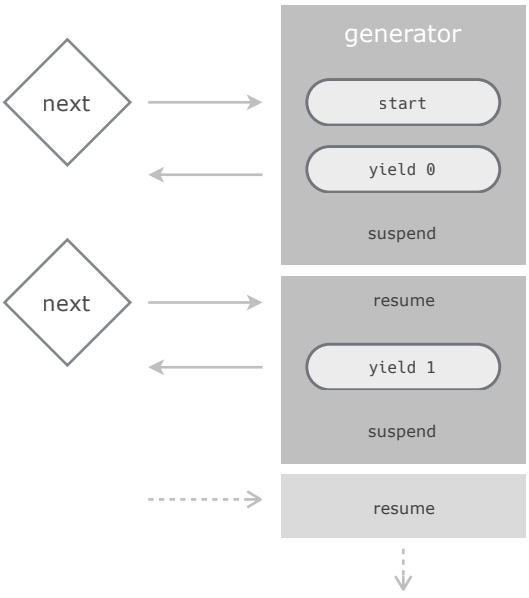
```
>>> next(x)
gen.start
gen.yield 0
0
```

当执行到 yield 指令，在设置好返回值后，解释器保存线程状态，并挂起当前函数流程。只有再次调用 __next__ 方法，才能恢复状态，继续执行。如此，以 yield 为切换分界线，往复交替，直到函数结束。

> 执行状态保存在用户栈帧内，系统线程算是无状态多路复用，切换操作自然很简单。

```
>>> next(x)
gen.resume
gen.yield 1
1
```

```
>>> next(x)
gen.resume
StopIteration
```



方法

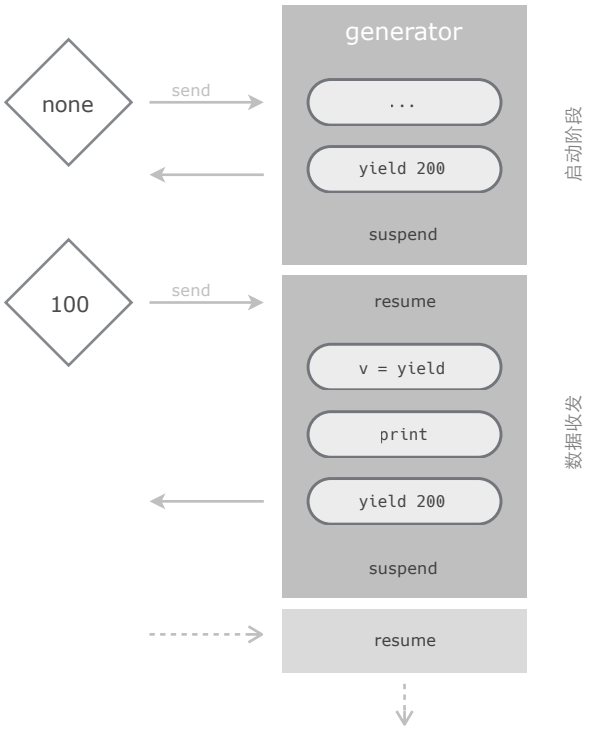
生成器另一进化特征，就是提供双向通信能力。生成器不再是简单的数据提供方，还可作为接收方存在。甚至可在外部停止，或发送信号实现重置等自定义行为。

方法 send 除可向 yield 传送数据外，其他与 next 完全一致。不过发送之前，须确保生成器已经启动。因为只有如此，才会进入函数执行流程，才会对外提供数据和交互。

```
def test():
    while True:
        v = yield 200
        print(f"resume {v}")
```

```
>>> x = test()

>>> x.send(None)              # 必须使用 next 或 send(None) 启动生成器。
200
```

```
>>> x.send(100)               # 可发送任何数据，包括 None，这与启动参数无关。
resume 100
200
```



对生成器函数而言，挂起点是个安全位置，相关状态被临时冻结。至于交互方式，只要将要发送数据，或者其他状态标记放置到栈帧指定位置，随后由解释器决定如何处理。比如说调用 close 方法，解释器将终止生成器迭代。

> 该方法在生成器函数内部引发 GeneratorExit 异常，通知解释器结束执行。
> 此异常无法捕获，但不影响 finally 执行。

```
def test():
    for i in range(10):
        try:
            yield i
       finally:
            print("fina.")
```

```
>>> x = test()

>>> next(x)                             # 启动生成器。
0

>>> x.close()                           # 终止生成器。
fina.

>>> next(x)                             # 已经终止。
StopIteration
```

还可像 send 发送数据那样，向生成器 throw 指定异常作为信号。

```
class ExitException(Exception): pass
class ResetException(Exception): pass

def test():
    while True:
        try:
            v = yield
            print(f"recv: {v}")
        except ResetException:
            print("reset.")
        except ExitException:
            print("exit.")
            return
```

```
>>> x = test()
```

```
>>> x.send(None)                    # 启动生成器。
```

```
>>> x.throw(ResetException)         # 发出重置信号。
reset.

>>> x.send(1)                       # 可继续发送数据。
recv: 1
```

```
>>> x.throw(ExitException)          # 发出终止信号。
exit.

>>> x.send(2)                       # 生成器已终止。
StopIteration
```

异常属于合理流程控制，不能和错误等同起来。

---

3. 模式

借助生成器切换功能，改善程序结构设计。

生产消费模型

在不借助并发框架的情况下，轻松实现生产、消费协作。

> 消费者启动后，使用 yield 将执行权限交给生产者，等待其发送数据后激活处理。
> 如果有多个消费者，或数据处理时间较长，依然建议使用专业并发方案。

```
def consumer():
    while True:
        v = yield
        print(f"consumer: {v}")


def producer(c):
    for i in range(10, 13):
        c.send(i)
```

```
c = consumer()                  # 创建消费者
c.send(None)                    # 启动消费者

producer(c)                     # 生产者发送数据
c.close()                       # 关闭消费者
```

输出：

```
consumer: 10
consumer: 11
consumer: 12
```

消除回调

回调函数（callback）是常见异步接口设计方式。调用者在发起请求后，不再阻塞等待结果返回，改由异步服务调用预先注册的函数来完成后续处理。

尽管回调函数使用广泛，但不太招人喜欢，甚至有“callback hell”的说法。究其原因，主要是回调方式让代码和逻辑碎片化，不利于阅读和维护。

设计一个简单异步服务示例，然后尝试用生成器清除回调函数。

```
import threading
import time


def target(request, callback):
    s = time.time()
    request()                                                  # 调用请求函数。
    time.sleep(2)                                              # 模拟阻塞。
    callback(f"done: {time.time() - s}")                       # 调用回调函数，传入结果。


def service(request, callback):
    threading.Thread(target=target, args=(request, callback)).start()
```

```
def request():                                                 # 任务请求模拟。
    print("start")

def callback(x):                                               # 任务结束通知。
    print(x)

service(request, callback)
```

输出：

```
start
done: 2.003718137741089
```

首先，以生成器函数替代原来分离的两个函数。用 yield 分隔请求和返回代码，以便服务可介入其中。

```
def request():
    print("start")                                             # 请求部分。

    x = yield                                                  # 接收服务返回结果。
    print(x)
```

接下来，改造服务框架，以生成器方式调用任务。

```
def target(fn):
    try:
        s = time.time()

        g = fn()                                        # 调用目标生成器函数。
        g.send(None)                                    # 启动，开始执行请求代码。

        time.sleep(2)
        g.send(f"done: {time.time() - s}")              # 阻塞结束后，将结果传回任务。
    except StopIteration:                               # 目标结束，拦截异常。
        pass


def service(fn):
    threading.Thread(target=target, args=(fn,)).start()
```

如此，既没有影响异步执行，也不再碎片化。

```
service(request)
print("do something")
```

输出：

```
start
do something                                            # 发起请求后，并未阻塞。
done: 2.0059099197387695
```

协程

协程（coroutine）以协作调度方式，在单个线程上切换执行并发任务。配合异步接口，可将原本 IO 阻塞时间用来执行更多任务。因在用户空间实现，也被称作用户线程。

> 理论上，任何具备 yield 功能的语言都能轻松实现协程。
> 当然，基于性能考虑，专业异步框架会选用 greenlet 之类的协程实现。

```
def sched(*tasks):
    tasks = list(map(lambda t: t(), tasks))      # 调用所有任务函数，生成器列表。
    while tasks:                                 # 循环调用任务。
        try:
```

```
            t = tasks.pop(0)                    # 从列表头部弹出任务。
            t.send(None)                        # 开始执行该任务。

            tasks.append(t)                     # 如果任务没有结束，则放回列表尾部。
        except StopIteration:                   # 任务结束，则放弃。
            pass
```

```
from functools import partial


def task(id, n, m):                             # 模拟任务模版。
    for i in range(n, m):
        print(f"{id}: {i}")
        yield                                   # 主动调度。


t1 = partial(task, 1, 10, 13)
t2 = partial(task, 2, 30, 33)
sched(t1, t2)
```

输出:

```
1: 10
2: 30
1: 11
2: 31
1: 12
2: 32
```

4. 函数式编程

函数式编程（Functional Programming）中的“函数”更趋近于数学概念，以计算表达式替代命令式语句。强调逐级结果推导，而非执行过程。

要求函数为第一类型，可作为参数和返回值传递。需要时，可用闭包构成带有上下文状态的逻辑返回。通常不使用变量，所有状态以参数传递，用嵌套或链式调用代替过程语句。最好是没有外部依赖的纯函数，且参数不可变，仅以结果带入下一级运算。

命令式

```
x = a + b
y = x * 2
```

函数式

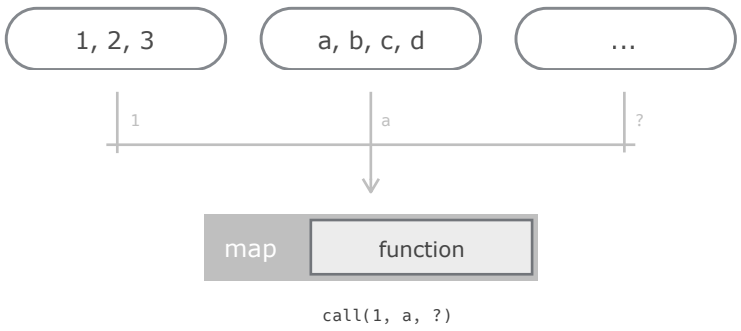
```
mul(add(a, b), 2)

add(a, b).mul(2)
```

相比命令式代码，函数式风格更精简，易组合，可替换。对单元测试和并发编程友好，引用透明且没有副作用。

> 三大特征：第一类型函数，不可变数据，尾递归优化。函数式编程只是一种编程范式，多数语言都支持或部分支持，这与计算机架构无关。实际上，现有冯·诺伊曼结构就是按命令序列执行。

迭代

以迭代器方式，用指定函数处理数据源。相比推导式，它能从多个数据源中平行接收参数，直至最短数据源迭代完成。



```
>>> x = map(lambda a, b: (a, b), [1, 2, 3], "abcd")

>>> list(x)                                                    # 将迭代器对象转换为列表。
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Python 3 以最短数据源为截止，而不是像 Python 2 那样以 None 补位。
如果参数已经完成并行组合，可使用 itertools.starmap 代替。

聚合

平行从多个数据源接收参数，聚合成元组，直到最短数据源迭代结束。

```
>>> x = zip([1, 2, 3], "abcd", (1.1, 1.2))

>>> list(x)
[(1, 'a', 1.1), (2, 'b', 1.2)]
```

常被用来构造字典。

```
>>> kv = zip("abcd", range(100, 200))

>>> dict(kv)
{'a': 100, 'b': 101, 'c': 102, 'd': 103}
```

如果以最长数据源为准，可使用 itertools.zip_longest。

累积

迭代数据源，将结果带入下次计算，适合完成统计或过滤操作。

```
>>> import functools

>>> def calc(ret, x):
        print(f"ret = {ret}, x = {x}")
        return ret + x
```

```
>>> functools.reduce(calc, [1, 2, 3])              # 没有初始化值，直接将第一数据当结果。
ret = 1, x = 2
ret = 3, x = 3
6
```

```
>>> functools.reduce(calc, [1, 2, 3], 100)         # 需要对第一数据和初始化值进行计算。
ret = 100, x = 1
ret = 101, x = 2
ret = 103, x = 3
106
```

如果没有初始值，则跳过第一数据，直接将其当作下次计算参数。
同 Python 2 内置函数 reduce。

过滤

使用指定函数对数据进行迭代过滤。

函数返回布尔值决定数据去留。
为 None 时相当于 bool 函数，移除所有 False 元素。

```
>>> x = filter(lambda n: n % 2 == 0, range(10))

>>> list(x)
[0, 2, 4, 6, 8]
```

```
>>> list(filter(None, [0, 1, "", "a", [], (1,)]))
[1, 'a', (1,)]
```

判断

判断一系列数据中，全部或某个元素为真值。

```
>>> all([1, "a", (1,2)])
True

>>> any([0, "", (1,2)])
True
```

标准库 itertools 拥有大量针对迭代器的操作函数，推荐使用。

六. 模块

1. 模块

模块（module）是顶层代码组织单元，提供大粒度的封装和复用。

通常情况下，每个模块对应一个源码文件。从某些角度看，模块像是更大规模的类。其中定义的变量、函数、类型等，都属于其私有成员。

> 也可用 C 或 Cython 编写模块，以绕开解释器和 GIL 限制，获得更好的执行性能与并发支持。

在首次导入（import）时，被编译成字节码。随后创建模块实例，执行初始化语句，构建内部成员。模块不仅是代码组织形式，还是运行期对象，为成员提供全局名字空间。

无论是在同一文件内，还是在不同文件多次导入，整个解释器进程内仅有一个目标模块实例存在。导入后，亦不会监测目标模块源文件变动。重复导入只是引用已存在实例，并不会再次执行初始化过程。

demo.py

```
x = 1234

def hello(): pass
class User: pass
```

```
>>> import demo

>>> isinstance(demo, types.ModuleType)
True
```

当然，也可像普通类型那样直接创建模块实例，未必非得有源码文件。

```
>>> types.ModuleType("abc")
<module 'abc'>
```

初始化

所谓初始化过程，就是将模块里的代码按序执行一遍。正好，已编译的字节码缓存文件可作为反汇编目标。

```
>>> import io, marshal, dis
>>> import demo

>>> f = open(demo.__cached__, "rb")      # 以二进制方式打开缓存文件。
>>> f.seek(12, io.SEEK_SET)              # 跳过 magic 等头部信息。
>>> code = marshal.load(f)               # 反序列化，还原代码对象。
```

```
>>> dis.dis(code)
  3           0 LOAD_CONST               0 (1234)
              2 STORE_NAME               0 (x)

  6           4 LOAD_CONST               1 (<code object hello>)
              6 LOAD_CONST               2 ('hello')
              8 MAKE_FUNCTION            0
             10 STORE_NAME               1 (hello)

  9          12 LOAD_BUILD_CLASS
             14 LOAD_CONST               3 (<code object User>)
             16 LOAD_CONST               4 ('User')
             18 MAKE_FUNCTION            0
             20 LOAD_CONST               4 ('User')
             22 CALL_FUNCTION            2
             24 STORE_NAME               2 (User)
             26 LOAD_CONST               5 (None)
             28 RETURN_VALUE
```

很简单，普通语句直接执行，类似 def、class 则创建函数和类型对象。最终这些成员都被保存到模块全局名字空间内。

```
>>> vars(demo)
{ ...
  '__cached__': '__pycache__/demo.cpython-36.pyc',
  '__file__': 'demo.py',
  '__name__': 'demo',
  'User': demo.User,
  'hello': <function demo.hello>,
  'x': 1234
 }
```

以示例中全局变量 x 为例，初始化过程为其赋予一个初始值。而后执行过程中，它可能会发生变更。如果每次导入都重新执行初始化过程，那么势必导致该变更状态丢失。所以说，重复导入只是引用，不会再次执行。

名字空间

模块的全局名字空间对应 __dict__ 属性，在内部不能直接访问，且不会被 dir 输出。

```
>>> vars(demo) is demo.__dict__
True
```

函数 vars 返回目标 __dict__ 属性。参数为空时，类似 locals 返回当前名字空间。
而 dir 则返回目标，或当前名字空间可访问名字列表。

当 def 创建函数对象时，会将所在模块名字空间作为构造参数。这意味着，无论后续将该函数传递到何处，或“绑定”给其他模块，均不能改变函数内部 globals 总是返回原生地名字空间。

demo.py

```
def test():
    return globals()
```

```
>>> demo.test() is vars(demo)              # 在任何地方调用，函数依然返回原生地名字空间。
True
```

```
>>> m = types.ModuleType("abc")
>>> m.test = demo.test                     # 即便“绑定”给其他模块，依然无法改变。

>>> m.test() is vars(demo)
True

>>> m.test.__module__
'demo'
```

名字

在源文件中可通过 __name__、__file__ 获知所在模块信息。或者，利用 __module__ 属性返回目标定义模块名称。

```
>>> x = demo.hello
>>> x.__module__
'demo'
```

> 建议使用 inspect.getmodule 获取目标对象定义模块引用。

正常情况下，模块名字对应源文件名（不含扩展名）。仅有一个例外，就是当模块作为程序入口时，会被赋予“__main__”名字。

demo.py

```
print(__name__)
```

```
$ python demo.py                    # 入口文件。
__main__

$ python -c "import demo"           # 普通导入。
demo
```

如此，我们只要编写如下语句，就可让其自动适应身份变化。

```
def main():
    test()

if __name__ == "__main__":
    main()
```

> 创建语句 def 并不会执行 main 函数，只有作为入口时才会进入流程。如果作为普通模块导入，因名字不匹配而被忽略。另外，不建议将 main 函数内容直接写到 if 语句块内，因为这会直接操作全局名字空间，可能引发不必要的错误。

2. 导入

在多数语言里，多源码文件共享全局环境。但在 Python 里则不同，每个模块文件都有自己独立的全局名字空间，是其内部代码访问边界。

依照 LEGB 规则，除包含内置函数的 __builtins__ 模块外，所有名字搜索都不能超出当前模块。所以任何外部目标，都必须提前导入到当前名字空间，否则无法访问。

完整导入步骤：

1. 搜索目标模块文件。
2. 按需编译目标模块。
3. 创建模块实例，执行初始化。
4. 将模块实例保存到全局列表。
5. 在当前名字空间建立引用。

模块实例一旦创建，则保存到 sys.modules 内，后续导入直接引用。
内置模块 __builtins__ 由解释器自动导入。

```
>>> for i in range(3):
        import sys                                  # 重复导入，不会新建实例。
        print(id(sys))

4480974584
4480974584
4480974584
```

```
>>> dis.dis(compile("import sys", "", "exec"))
        4 IMPORT_NAME              0 (sys)          # 导入模块。
        6 STORE_NAME               0 (sys)          # 在当前名字空间建立关联。
```

2.1 搜索

导入所使用模块名不包含路径信息，这需要系统提供搜索方式和匹配规则。其中搜索路径，由解释器在启动时，按优先级整理到 sys.path 列表中。

搜索路径列表：

1. 程序根目录。
2. 环境变量（PYTHONPATH）路径列表。
3. 标准库目录。
4. 第三方扩展库等附加路径（site-specific）。

附加路径由 site 模块添加，在解释器启动时自动执行（可用参数 -S 禁用）。
除系统 site-packages 外，还包括用户相关目录。

```
$ python3 -c "import sys, pprint; pprint.pprint(sys.path)"
['',                                                        # 应用程序根目录。
 '/.../lib/python36.zip',                                   # 标准库
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload',
 '/.../lib/python3.6/site-packages']                        # 系统扩展库目录。
```

```
$ python3 -S -c "..."                                       # 禁用 site，缺少 site-packages.
['',
 '/.../lib/python36.zip',
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload']
```

```
$ PYTHONPATH=/usr/local/a:/usr/local/b python3 -c "..."     # 环境变量，注意优先级。
['',
 '/usr/local/a',
 '/usr/local/b',
 '/.../lib/python36.zip',
 '/.../lib/python3.6',
 '/.../lib/python3.6/lib-dynload',
 '/.../lib/python3.6/site-packages']
```

虽然可在运行期向 sys.path 添加搜索路径，但对扩展库而言，更简便方式是路径配置文件。将所有要添加路径按行保存在 .pth 文本文件内，并放到 site-packages 等目录，剩余的事情由 site 完成。

```
$ cd site-packages
$ mkdir a
```

```
$ cat demo.pth                                  # 路径配置文件（任意文件名）。

# comment                                       # 注释。
/usr/local/go                                   # 绝对路径。
a                                               # 相对路径。
```

```
$ python -m site                                # 查看完整搜索路径列表。

sys.path = [
    '/Users/qyuhen/test',                       # 应用程序根目录。
    '/.../lib/python36.zip',                    # 标准库。
    '/.../lib/python3.6',
    '/.../lib/python3.6/lib-dynload',
    '/.../lib/python3.6/site-packages',         # 系统扩展库目录。
    '/usr/local/go',                            # 路径配置文件添加目录。
    '/.../lib/python3.6/site-packages/a',
]

USER_BASE: '/Users/qyuhen/.local' (doesn't exist)
USER_SITE: '/Users/qyuhen/.local/lib/python3.6/site-packages' (doesn't exist)
ENABLE_USER_SITE: False
```

提示：site 会过滤掉配置文件中重复，以及不存在的路径。

在整理好搜索路径列表后，接下来需要进一步匹配模块文件名。但问题是，语言规范并未定义不同扩展名的同名模块匹配次序，也就无法保证不同解释器的选择结果一致。安全起见，不要在同一路径下放置同名模块。

- 源码或字节码文件（.py，.pyc）。
- 包目录名。
- 其他语言编写的扩展模块（.dll，.so 等）。

2.2 编译

确定模块所在路径后，解释器优先选择已编译过的字节码文件，这有助于提升载入性能。当然，执行之前，需读取相关头信息，确认源文件是否更新，以便重新编译。

从 Python 3.2 开始，字节码缓存文件被整体转移到 __pycache__ 子目录。相比以前版本，这避免了对源码目录的污染，有利于管理和维护。

缓存文件名中包含解释器、版本和优化信息，且不再使用 .pyo 扩展名。如此，不同版本的解释器缓存文件可和谐共处。

> 可使用 -B 参数阻止生成字节码缓存文件。

```
$ python -c "import demo"

$ ls __pycache__
demo.cpython-36.pyc
```

```
$ python -OO -c "import demo"

$ ls __pycache__
demo.cpython-36.opt-2.pyc
```

无源码部署

单就运行而言，源码文件不是必须的。因为解释器需要的字节码及相关元数据，要么编译后存储在内存，要么就保存在字节码文件中。

可忽略源文件，改为直接部署字节码文件（.pyc），算是一种轻度保护。

> 不能直接使用 __pycache__ 下的文件，它的文件命名方式不适合作为直接部署使用。
> 缓存文件很容易被 dis 反汇编，并不能有效保护代码安全。

main.py

```
import demo
demo.hello()
```

demo.py

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

直接使用 compileall 编译所有源文件。

```
$ python -m compileall -b -l *.py        # 编译所有文件（以原文件名保存，不包括自目录）。
Compiling 'demo.py'...
Compiling 'main.py'...

$ mv *.pyc release/                      # 将生成的 .pyc 文件转移到独立目录。
$ cd release/
$ ls
demo.pyc main.pyc

$ python main.pyc                        # 使用解释器执行。
hello, world!
```

因为 .pyc 是二进制文件，所以就算 main.py 头部头 Shebang 也无法被系统 shell 识别，只能以解释器参数方式执行。

虽然字节码是平台中立格式，但 .pyc 头部 magic number 却保存了解释器版本，所以要使用相同版本（大版本号）的解释器执行。

2.3 导入

可一次导入多个模块，或模块中的多个成员。当名字发生冲突时，可使用 as 命名别名。

```
>>> import sys, inspect as reflect
>>> from string import ascii_letters as letters, ascii_lowercase as lowercase
```

```
>>> %whos                                # iPython 命令，查看当前环境变量信息。

Variable    Type      Data/Info
--------------------------------
letters     str       abcde...
lowercase   str       abcde...
reflect     module    <module 'inspect'>
sys         module    <module 'sys'>
```

别名仅在当前名字空间有效，并不影响被导入模块或其成员。
还可用别名做缩写处理。
如果导入成员过多，可用括号分成多行。

在相关编码规范中，通常建议每行仅导入一个模块，且不在全局名字空间导入模块成员。导入语句通常放在文件头部，以“标准库、扩展库，当前程序模块”分块排列。

```
import sys                  # 标准库
import inspect

import psutil               # 第三方扩展库
import tornado

import db                   # 应用程序模块
import logic
```

至于成员导入，其名字和其他模块，或当前模块成员冲突的可能性较大。且对模块重新载入有所影响（见后文），故不建议在模块级别使用。至于以星号导入目标模块“全部”成员的行为，更是被很多规范所禁止。

对成员而言，合适的名字通常就是些常用单词。如果是以模块为前缀的全名，通常可避免冲突。但直接以成员导入，则可能被“同名覆盖”。

建议参考 《Google Python Style Guide》等文档，养成规范而统一的编码风格。
另外，可使用 isort 整理导入语句。

```
from sys import version
print(version)

version = "ver: 1.0"              # 覆盖，同一名字仅有一个关联。
print(version)
```

在函数内部，因其名字空间范围小，且生命周期较短，故影响也小。如此，使用成员导入可减少编码，还能在一定程度上提升性能。

Python 3 不允许在函数内使用星号导入，只能用于模块级别。

```
def test():
    from sys import version
    print(version)
```

对比模块导入和成员导入指令差异。

```
def a():
    import sys
    print(sys.version)

def b():
    from sys import version
    print(version)
```

```
>>> dis.dis(a)
  3           8 LOAD_GLOBAL              1 (print)
             10 LOAD_FAST                0 (sys)             # 需要 2 条指令引用 version。
             12 LOAD_ATTR                2 (version)
             14 CALL_FUNCTION            1
```

```
>>> dis.dis(b)
  7          12 LOAD_GLOBAL              2 (print)
             14 LOAD_FAST                0 (version)         # 单条指令，且是 FAST。
             16 CALL_FUNCTION            1
```

如果是多次调用，那么对比效果就比较明显。

```
def a():
    import sys
    for i in range(100):
        s = sys.version
    return s

def b():
    from sys import version
    for i in range(100):
        s = version
    return s
```

```
>>> %timeit a()
6.02 µs ± 253 ns per loop          # µs 微秒。

>>> %timeit b()
4.18 µs ± 261 ns per loop
```

导出列表

星号导入是将目标模块名字空间内所有成员全部引入，这里面必然有很多用不到的东西。可作为模块开发者，我们无法阻止用户使用星号。结果就是调用方名字空间被污染，这显然不是我们愿意看到的。还有，我们编写的模块也会导入其他模块，它们一同被裹挟着进入调用方名字空间。

main.py

```
from demo import *
print(dir())
```

demo.py

```
import sys

x = 1234
def hello(): pass
```

```
$ python main.py            # demo 成员，以及它导入的 sys 模块，都进入调用者 main 名字空间。
['hello', 'sys', 'x']
```

简单做法是模块成员私有化，也就是为名字添加下划线前缀。私有成员不会被星号导入，但我们导入的其他模块呢？私有别名？

demo.py

```
import sys as _sys          # 私有别名?

_x = 1234                   # 私有成员
def hello(): pass
```

```
$ python main.py
['hello']
```

显然这种写法既不方便也不美观。正确做法是在模块内添加 __all__ 声明，指定那些可以被星号导入的成员名字列表。为空时，表示不会被导入任何成员。

demo.py

```
__all__ = ["hello"]
```

```
import sys

x = 1234
def hello(): pass
```

```
$ python main.py                # 仅 __all__ 指定成员被星号导入。
['hello']
```

当然，无论是私有成员，还是 __all__ 导出列表，都不影响显式成员导入。

demo.py

```
__all__ = []

import sys

_x = 1234
def hello(): pass
```

main.py

```
from demo import _x, hello
print(dir())
```

```
$ python main.py
['_x', 'hello']
```

> Python 很多“私有”规则只是调用约定，并不是真正意义上的权限设置。

动态导入

使用 import 的前提是明确知道目标模块名字。但某些时候，只有在运行期才能动态获知，这就需要以其他方式导入。

方法一：使用 exec 动态执行，随后从 sys.modules 中获取模块实例。

```
import sys
```

```
def test(name):
    exec(f"import {name}")
    m = sys.modules[name]
    print(m)

test("inspect")
```

方法二：使用 importlib 库。

```
import importlib

def test(name):
    m = importlib.import_module(name)
    print(m)
```

两种方法都会将模块实例保存到 sys.modules 内。

不建议使用 __import__ 函数。作为 import 语句的内在实现，其行为在 Python 2 和 3 下存在差异，可能导致意外错误。

```
/
|
+--- main.py                    # import lib.demo
|
+--- m.py                       # print(__file__)
|
+--- lib/
      |
      +--- __init__.py          # 空文件（包初始化文件）
      |
      +--- demo.py              # import m
      |
      +--- m.py                 # print(__file__)
```

```
$ python2 main.py
/lib/m.py

$ python3 main.py
/m.py
```

其原因是两个版本搜索起始路径不同。Python 2 从当前目录开始，那么 demo 优先选择的自然是同一目录下的 m，而 Python 3 无论何时何地都从根目录开始。好在 import_module 在两个版本里行为一致，都从根目录开始。

某些时候，模块需要同时支持 Python 2 和 3，可用异常保护处理导入语句差异。

```
try:
    from urlparse import urljoin
    from urllib2 import urlopen
except ImportError:
    # Python 3
    from urllib.parse import urljoin
    from urllib.request import urlopen
```

重新载入

模块被编译导入后，解释器就不再关心源文件是否被修改过，这对简单或者已趋于稳定的程序并没什么影响。但那些需要在线测试，或频繁变更业务逻辑的互联网应用，我们就不得不设法在不重启进程的情况下，对某些模块实现热更新。

至于如何监测源文件变化，如何启动热更新，均不在本文讨论之列。这里，我们尝试不同模块重载方案，探知其实现机制和存在的问题。

既然模块实例保存在 sys.modules 列表里，那么是否可通过移除实例引用，使其被回收后再重新冷导入（首次导入）？

demo.py

```
x = 1234
```

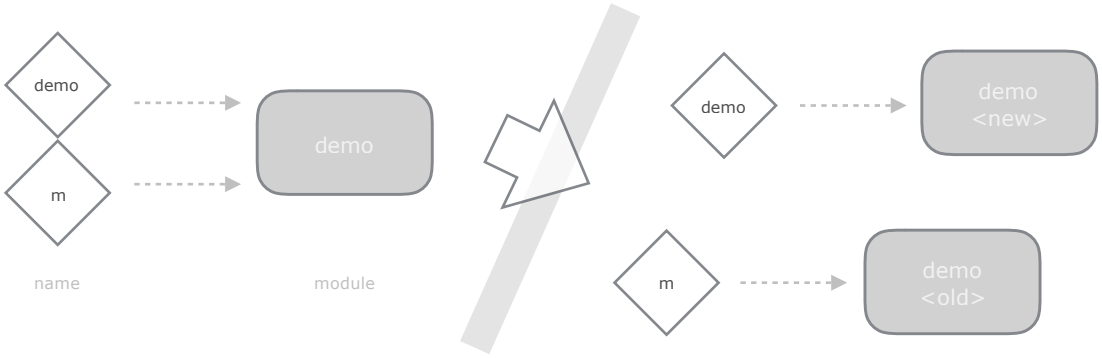
```
>>> import demo

>>> demo.x
1234

>>> del sys.modules["demo"]             # 从 sys.modules 移除，使其回收。
>>> !echo "x = 999" > demo.py           # 修改 demo.py。

>>> import demo                         # 重新导入。
>>> demo.x                              # 更改生效。
999
```

从结果看，似乎没有什么问题。可如果有其他引用存在呢？

```
>>> import demo

>>> m = demo                        # 另一引用。
>>> m.x
999

>>> del sys.modules["demo"]         # 移除。
>>> !echo "x = 888" > demo.py       # 修改源文件。

>>> import demo                     # 重新导入。
>>> demo.x
888

>>> m.x                             # 对 m 没有影响，热更新失败。
999
```

失败原因很简单。因为 m 的存在，导致旧 demo 模块实例不会被回收。如此，更新后它们各自引用不同的模块实例，即便旧模块已不在 sys.modules 列表里。

```
>>> m is demo
False
```



标准库为我们提供了另外一种方案 importlib.reload。它直接原址（memory in-place）替换模块内容，这样就可让所有引用都指向新的模块实例。

```
>>> import demo

>>> m = demo
>>> m.x
888

>>> !echo "x = 666" > demo.py       # 修改源文件。
>>> importlib.reload(demo)          # 重载模块。
```

```
>>> demo.x                              # 热更新成功。
666

>>> m.x
666

>>> demo is m
True
```

此种方案不变动内存地址，直接替换内容，这就不会影响原有引用。但问题是，这种方法只对模块引用有效，对成员引用无法更新，因为 reload 不会递归修改成员。

```
>>> import demo

>>> x = demo.x
>>> x
666

>>> !echo "x = 777" > demo.py
>>> importlib.reload(demo)

>>> demo.x
777

>>> x                                   # 热更新失败。
666

>>> demo.x is x
False
```

有鉴于此，应避免直接引用其他模块成员，总是透过模块间接访问。当然，在函数内部，成员引用生命周期较短，对热更新影响有限。

另外，模块重载会重新初始化，可能导致状态丢失，甚至引发连带错误，需慎重对待。

3. 包

如果说模块用于组织代码，那么包就是组织模块。

将相同用途的多个源文件放置于同一目录，这就构成了包。包能隐藏内部文件组织结构，仅暴露必要的用户接口，毕竟不是所有模块都对外提供服务。还有，包为模块提供更详细的名字前缀，避免模块同名冲突。

> 与模块类似，包名直接对应目录名。包名可以是产品名称，公司或组织结构名称等。
> 相比模块名 db，有包前缀的全名 orm.db 更不易冲突。

虽然只是一个目录，包也是运行期对象，有自己的名字空间。

```
/
|
+--- main.py
|
+--- lib/                        (package)
      |
      +--- demo.py               (package.module)
```

```
>>> import lib

>>> lib
<module 'lib' (namespace)>

>>> vars(lib)
{'__name__': 'lib',
 '__package__': 'lib',
 '__path__': _NamespacePath(['/Users/qyuhen/test/lib']), ...
}
```

从包名字空间可以看出，仅导入包并不能直接访问其内部模块，需显式导入。

```
>>> import lib.demo

>>> lib.demo.__package__
'lib'

>>> lib.demo.__name__
'lib.demo'
```

3.1 初始化

包算是种另类的模块实例，对应源文件 __init__.py。虽然该文件在 Python 3 里不再是必须的，但可用来执行某些初始化操作。比如导入对外服务接口，解除内部模块依赖。

```
lib/
 |
 +--- __init__.py
 |
 +--- demo.py
```

```
>>> import lib

>>> lib.__file__
'/Users/qyuhen/test/lib/__init__.py'
```

初始化文件在包或其内部模块首次导入时自动执行，且仅执行一次。

__init__.py

```
print("init")
```

demo.py

```
print("demo")
```

```
>>> import lib.demo
init
demo

>>> import lib
```

> 重载（reload）包内模块，不会再次执行初始化文件，但重载包会。

还可在包内创建 __main__.py 文件，作为直接运行时的入口。

__main__.py

```
print("main")
```

```
$ python -m lib             # 以包方式运行，自动执行初始化文件。
init                        # 先导入包，后执行 __main__。
main

$ python lib                # 以普通程序方式运行，对应 __main__ 入口模块。
main                        # 不会自动执行初始化文件，除非显式导入 __init__。
```

这对包普通操作没有任何影响，仅对应 python -m 执行方式。

隐藏结构

既然初始化文件构成包名字空间，那么只要将要公开给用户的模块或成员导入进来，就可以解除用户对具体模块的依赖。

demo.py

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

__init__.py

```
from .demo import hello
```

```
>>> import lib

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

如此，用户只依赖包和初始化文件中导入的特定成员。无论我们如何重构代码和重新组织模块文件，都不影响用户调用接口。即便将公开成员升级到新版本，也可用别名方式映射到原名字，实现多版本单一名称。

demo.py

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

def hello2():
    from sys import version
    print(version)
```

```
    hello()
```

__init__.py

```
from .demo import hello2 as hello
```

还可更进一步，将整个包压缩成 ZIP 文件分发。

```
lib.v1.zip                              任意文件名，可添加版本等信息。
 |
 +--- lib/                              必须包含包目录。
       |
       +--- __init__.py
       |
       +--- demo.py
```

> 如果只打包模块，则直接放到压缩包根目录下。

```
$ zip -r -o lib.v1.zip lib                      # 压缩后，可将原目录重命名，避免影响下面的测试。
  adding: lib/ (stored 0%)
  adding: lib/__init__.py (deflated 9%)
  adding: lib/demo.py (deflated 37%)
```

使用前，需将 .zip 文件路径添加到 sys.path，可动态添加或使用路径配置文件。

```
>>> sys.path.append("./lib.v1.zip")

>>> import lib
>>> lib.__file__                                # 指向压缩包内文件。
'./lib.v1.zip/lib/__init__.py'
```

> 因解释器不会向 .zip 中写入缓存文件，故建议连带 .pyc 一起打包，以加快导入速度。

```
$ python -m compileall -b lib/*.py              # 不能使用 __pycache__。
$ zip -r -o lib.v1.zip lib/*.pyc                # 仅打包 .pyc 文件，注意包含包目录。
```

也可将应用程序打包，不过须在根目录下放置 __main__.py 作为执行入口。

```
app.zip
 |
 +--- __main__.py
 |
 +--- lib/
       |
       +--- __init__.py
       |
       +--- demo.py
```

__main__.py

```
print("main")

import lib
lib.hello()
```

```
$ zip -r -o app.zip __main__.py lib
  adding: __main__.py (stored 0%)
  adding: lib/ (stored 0%)
  adding: lib/__init__.py (stored 0%)
  adding: lib/demo.py (deflated 7%)
```

```
$ python app.zip
main
hello, world!
```

导出列表

同样可在初始化文件中添加 __all__ 星号导出成员列表。除当前文件成员外，还可指定要导出的模块名字。

不用 import 语句，只需在列表中添加要导出模块名字即可。
无论是隐藏结构（包括 .zip），还是 __all__ 导出列表，都不影响用户显式导入包内模块。

__init__.py

```
__all__ = ["x", "demo"]             # 可被星号导出的模块和本地成员。

x = 100
y = 200
```

```
>>> from lib import *

>>> dir()
[..., "demo", "x"]
```

3.2 相对导入

包被放置于应用程序根目录，或其他系统搜索路径中。但其中并不包括包目录自身，这就导致在包内访问同级模块时，发生找不到文件的状况。

__init__.py

```
import demo
demo.hello()
```

demo.py

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

```
>>> import lib

lib/__init__.py in <module>
      1
----> 2 import demo
      3 demo.hello()

ModuleNotFoundError: No module named 'demo'
```

就算在包内导入，import 语句也严格按照 sys.path 查找。会搜索应用程序根目录，但不检查包目录。这就是引发错误的原因，除非全名导入。

__init__.py

```
import lib.demo
lib.demo.hello()
```

Python 2 优先搜索当前路径，也就是 relative-then-absolute 顺序。
Python 3 移除了该设定，严格按照 sys.path 搜索。能找到 lib，但找不到 demo。

在包内使用全名这种绝对路径自然不够友好。为此，Python 3 引入相对路径概念，以点前缀表达当前或上级包。

```
/
|
+--- lib/
       |
       +--- __init__.py
       |
       +--- demo.py
       |
       +--- sub/
              |
              +--- test.py
```

sub/test.py

```
from ..demo import hello             # 从上级包的 demo 模块导入 hello。
```

__init__.py

```
from . import demo                   # 从当前包导入 demo 模块。
from .demo import hello              # 从当前包的 demo 模块导入 hello。
from .sub import test                # 从当前包的 sub 子包导入 test 模块。
```

> 相对导入只能用于 from 子句，可导入模块或成员。
> 单个 . 表示当前包，.. 上级包，... 更上一级，如此类推。

相对路径解除了对包名的依赖，还可用来解决名字冲突。比如导入 string 时，如果使用相对导入则明确表示选择包内（或上级）模块，否则选择标准库同名模块。

相比 Python 2 优先选择当前路径的隐式设定，Python 3 的显式做法要更清晰一些。避免了在 Python 2 下导入标准库 string，反而优先选择本地同名模块的尴尬。

> 显式规则总要好过隐式设定。

优先级

同名除让解释器困扰外，也给代码维护造成潜在风险。当相同名字的包和模块出现在同一路径下，那么解释器将按如下顺序匹配。

1. 如果包有初始化文件（包括空文件），导入包。
2. 没有初始化文件，则 .py 或 .pyc 模块优先。

没有同名冲突，那么还是按搜索路径规则走。所以说，应尽可能避免同名问题。

```
(1)                             (2)

 /                               /
 |                               |
 +--- lib.py                     +--- lib.py ⋆
 |                               |
 +--- lib/                       +--- lib/
       |                               |
       +--- __init__.py ⋆              +--- demo.py
       |
       +--- demo.py
```

3.3 拆分

当包内模块文件过多时，可建立子包分组维护，但需要调整内部相对导入路径。还有一种方法，同样是将文件分散到多个子目录下，但它们依然属于同一级别包成员。

先将原本放在同一目录下的模块文件分别转移到各自分组子目录中。
为让它们继续以包成员存在，须在 __init__.py 中修改 __path__ 属性。

包属性 __path__ 作用类似 sys.path，实现包内搜索路径列表，并使用相同匹配规则。
该搜索列表中默认为包全路径，只需将子目录全路径添加进去即可。
因子目录中各模块依然属于原包级别，故模块间的相对导入无须更改。

```
 /                               /
 |                               |
 +--- lib/                       +--- lib/
       |                               |
       +--- __init__.py                +--- __init__.py
       |                               |
       +--- demo.py                    +--- a/
       |                               |    |
       +--- hello.py                   |    +--- demo.py
                                       |
                                       +--- b/
                                            |
                                            +--- hello.py
```

__init__.py

```
import os.path
__path__.extend(os.path.join(__path__[0], d) for d in ("a", "b"))
```

```
from . import demo                          # 包内各模块依然视作同一级别。
from . import hello

print(__path__)
print(demo.__file__)
print(hello.__file__)
```

```
>>> import lib
['/Users/qyuhen/test/lib',                  # __path__
 '/Users/qyuhen/test/lib/a',
 '/Users/qyuhen/test/lib/b']

/Users/qyuhen/test/lib/a/demo.py
/Users/qyuhen/test/lib/b/hello.py
```

因为 __path__ 添加的是全路径，意味着可将子目录放到任意位置。甚至是将其他包内容引入进来。但这有混乱嫌疑，并不推荐。

另，对包内文件迭代可使用 pkgutil 相关函数。

七. 类

1. 定义

类（class）封装一组相关数据，使之形成一个整体，并使用方法持续展示和维护。这有点像把零件组装成整车提供给用户。驾驶人无须了解汽车内部结构和工作原理，只要知道方向盘、刹车和油门这些外在接口就可以让其正常行驶。

与函数类似，类也是一种小粒度复用单位，但行为特征更为复杂。函数具有单一入口和出口，完成一次计算过程。类从构造开始，就面临多个方法。这些方法在不同调用次序下，产生不同结果。就好像同一辆车，不同习惯的驾驶人会有不同运动轨迹。反之，函数就像自动烤面包机，放入和拿回不会因人有太大差异。

如果说函数是数学，那么类就是生物学。擅长对有持续状态，有生命周期，有遗传特征的物体进行抽象模拟。无论模拟的是动物还是植物，在保留种族共性的前提下，个体总是有属于自己的体貌特征和行为差异。如此，函数是机械加工，着重于处理过程；类则关注于数据本身，使其苏醒并活过来。

类存在两种关系：继承（inhertiance，is-a）自某个族类，组合（composition，has-a）了哪些部件。前者可用来表达本车属于某厂的哪个车族系列，除继承原车系的技术和优势外，还基于哪些环境进行了改进。而后者则是该车使用了哪些零部件，比如最新的发动机。

作为一种复合结构，类与模块有相似之处。但其不同之处在于：

- 类可生成多实例。
- 类可继承和扩展。
- 类实例生命周期可控。
- 类支持运算符，可按需重载。

这些是模块没有或不需要的。同时，模块粒度大，用来提供游戏场景级别解决方案，而类则是该场景下的特定家族和演员。

类可以有多个外在方法，但应专注于单个目标。即便用户类需要加密手段保存信用卡等敏感数据，也不应有加密相关代码，而是选择组合安全数据类型，以便替换和分离测试。

在某些语言中，类的用途被放大，形成一种名字空间概念。比如将多个函数附加到一个静态类上，形成函数套装。这与本书所提及的类有所不同，此处特指自定义复合数据类型，是一个缩小概念。至于静态类这种用途，完全可用模块替代。

建议将类与逻辑分离。逻辑代表一个连续处理过程，通常被设计成无状态，以请求上下文传递数据。讲究效率和并发性，以及相对固定的执行过程。从这点看，适合以函数实现分段，按特定顺序组装，并以模块存储。这些特点，都不适合以注重家族遗传，用多实例体现个性的类来实现。毕竟，逻辑以组装和替换实现持续改良和升级，有稳固测试流程，传承并非重点，也没有多实例需求。

创建

定义类，以此为工厂制造个体实例。

```
class User:
    pass
```

可在函数内定义，以限制其作用范围。

关键字 class 同样是运行期指令，完成类型对象创建。我们准备相对“完整”的示例，查看具体创建过程。

```
def test():
    class X:
        data = 100
        def get(self): return self.data
```

```
>>> dis.dis(test)
  2           0 LOAD_BUILD_CLASS
              2 LOAD_CONST               1 (<code X>)       # test.__code__.co_consts[1]
              4 LOAD_CONST               2 ('X')
              6 MAKE_FUNCTION            0                  # 创建 X 函数。
              8 LOAD_CONST               2 ('X')
             10 CALL_FUNCTION            2                  # 调用 __build_class__ 函数。
             12 STORE_FAST               0 (X)
```

从反汇编结果看，先创建 X 函数，其内容是属性设置和方法创建。随后，该函数被当作参数传递给 builtins.__build_class__ 调用，这里其实就是元类（metaclass）执行所在。

在 __build_class__ 调用 X 时，会提供一个字典作为其堆栈帧名字空间，用于保存类型成员。完成后，将该字典连同名字和基类一起传递给元类，最终生成目标类型对象。

提示：元类用于创建类型对象。详细情形，请参考后续章节。

```
>>> dis.dis(test.__code__.co_consts[1])
  2           0 LOAD_NAME                0 (__name__)
              2 STORE_NAME               1 (__module__)
              4 LOAD_CONST               0 ('test.<locals>.X')
              6 STORE_NAME               2 (__qualname__)

  3           8 LOAD_CONST               1 (100)
             10 STORE_NAME               3 (data)

  4          12 LOAD_CONST               2 (<code object test>)
             14 LOAD_CONST               3 ('test.<locals>.X.get')
             16 MAKE_FUNCTION            0
             18 STORE_NAME               4 (get)
```

类型与实例

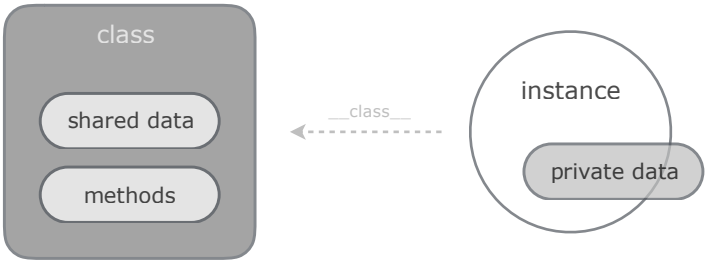
如果类型在模块中定义，那么其生命周期与模块等同。如果放在函数内，那么每次都是新建。即便名字和内容相同，也属于不同类型。

```
def test():
    class X: pass
    return X()
```

```
>>> a, b = test(), test()

>>> a.__class__ is b.__class__
False
```

函数内定义的类型对象，在所有实例死亡后，会被垃圾回收。

类型对象除用来创建实例外，也为所有实例定义了基本操作接口，负责管理整个家族共享数据和行为模版。而实例只保存私有特征，以内部引用从所属类型或其它祖先类查找所需方法，用以驱动和展现个体面貌。



类型所创建多个实例之间，除家族归属外，并无直接关系。无论从诞生到消亡，还是私有数据变更，都不影响类型或其他兄弟姐妹。

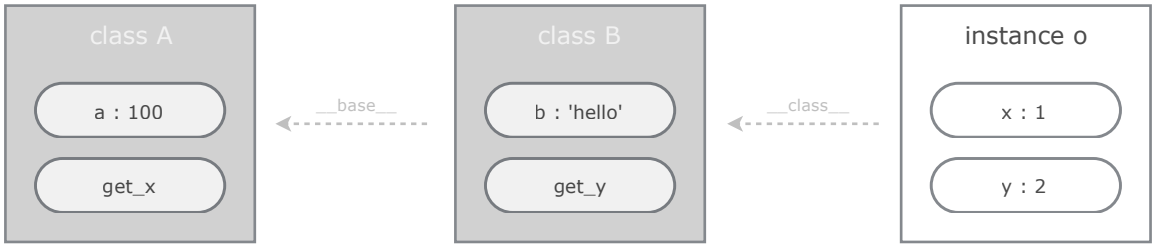
名字空间

类型有自己的名字空间，存储为当前类型定义的字段和方法。这其中并不包括所继承的祖先成员。同样以引用关联祖先类型，无需复制到本地。

```
class A:
    a = 100                             # 类字段。

    def __init__(self, x):              # 实例初始化方法。
        self.x = x                      # 实例字段。

    def get_x(self):                    # 实例方法。
        return self.x
```

```
class B(A):
    b = "hello"

    def __init__(self, x, y):
        super().__init__(x)             # 调用父类初始化方法。
        self.y = y

    def get_y(self):
        return self.y
```

```
>>> o = B(1, 2)
```



实例则存储所有继承层次的实例字段，因为这些都属于其私有数据。方法算是函数变体，本身并无状态，所以为大家共享。但即便是从远古祖先那里继承的眉眼肤色，到了本人这里也是私有特征。如果也放到类型里共享，岂不是千人一面？也正因为这些私有数据驱动，才让方法展现出不同结果来。

```
>>> A.__dict__
mappingproxy({'__init__': <function A.__init__>,
              'a': 100,
              'get_x': <function A.get_x>})
```

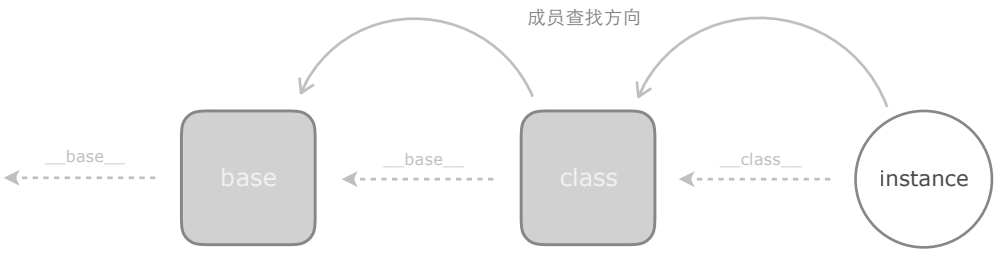
```
>>> B.__dict__
mappingproxy({'__init__': <function B.__init__>,
              'b': 'hello',
              'get_y': <function B.get_y>})
```

类型名字空间返回 mappingproxy 只读视图，不允许修改。

```
>>> o.__dict__
{'x': 1, 'y': 2}

>>> dir(o)
['a', 'b', 'get_x', 'get_y', 'x', 'y']
```

实例名字空间是普通字典，可直接修改。
函数 dir 按查找方向搜索所有可访问成员名字，vars 直接返回 __dict__ 属性。

当透过实例或类型访问某个成员时，会从当前对象开始，依次由近到远向祖先类查找。如此做的好处，就是祖先类新增功能可直接“广播”给所有子孙。



> 在继承层次的不同名字空间中允许有同名成员，按顺序优先命中。

成员查找规则和 LEGB 不同。前者基于继承体系，后者基于执行作用域。

```
x = 100

class A:
    def __init__(self, x):
        self.x = x

    def test(self):
        print(x)                    # LEGB。
        print(self.x)               # 明确以 self 指定搜索目标。
```

```
>>> o = A("abc")
>>> o.test()
100
abc
```

```
>>> dis.dis(o.test)
  8           0 LOAD_GLOBAL              0 (print)
              2 LOAD_GLOBAL              1 (x)            # LEGB
              4 CALL_FUNCTION            1

  9           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                0 (self)         # 先载入 self，
             12 LOAD_ATTR                1 (x)            # 然后 getattr(self, x)。
             14 CALL_FUNCTION            1
```

当两者同时出现时，可能会引发小小的混乱。

前文分析类型创建时提到过，class 在内部以函数方式执行，接收类型名字空间字典作为堆栈帧执行名字空间。如此，成员定义作用域内的 locals 实际指向 class.__dict__（可读写版）。

但从语法上说，class 是类型定义，而非函数定义，这与内部执行方式无关。因此，class 不构成 E/enclosing 作用域。

```
def enclosing():
    a = "enclosing.a"

    class A:
        a = "A.a"                               # L --> A.__dict__

        def test(self):
            print("E.a =", a)                   # E --> enclosing.a ③

        print("A.locals =", locals())           # L --> A.__dict__ ①
        print("A.a =", a)                       # L --> A.__dict__ ②


    A().test()
```

```
>>> enclosing()

① A.locals = {'a': 'A.a', 'test': <function A.test>}
② A.a = A.a
③ E.a = enclosing.a
```

2. 字段

依照所处名字空间不同，我们将字段分为类型字段和实例字段两类。

> 官方文档将成员统称为 Attribute，不过本书按惯例将数据当作字段。

类型字段在 class 语句块直接定义，而实例字段必须通过实例引用（self）赋值定义。仅从执行方式看，无论实例方法存在于哪级类型，其隐式参数 self 总指向当前调用实例。那么透过它创建的字段，也必然存在于该实例名字空间中。

```
class A:
    def test(self):                     # self 总是引用当前实例，与继承层次无关。
        print(self)
        self.x = 100                    # 向当前实例字段赋值。

class B(A):
    pass
```

```
>>> o = B()

>>> hex(id(o))
0x1092244a8

>>> o.test()                            # 观察实例 id。
<__main__.B object at 0x1092244a8>
```

> 实例参数 self 只是约定俗成的名字，类似其他语言中的 this。

可站在语法和设计角度，实例字段乃是所在类型的组成部分，与该类型相关方法存在关联。所以，我们依然称其为某某类型的实例字段。

字段赋值

可使用赋值语句添加新的类型或实例字段。

```
>>> class X: pass
```

```
>>> X.a = 100                       # 新增类型字段。

>>> vars(X)
mappingproxy({'a': 100, ...})
```

```
>>> o = X()
>>> o.b = 200                       # 新增实例字段。

>>> vars(o)
{'b': 200}
```

可一旦以子类或实例重新赋值，那么将会在其名字空间建立同名字段，遮蔽原字段。这与“赋值总是在当前名字空间建立关联”规则一致。

```
>>> class X: pass
>>> o = X()

>>> X.data = 100
>>> o.data                          # 按搜索规则，命中 X.data。
100
```

```
>>> o.data = 200                    # 在 o 名字空间新建 data 实例字段。
>>> o.data                          # 按搜索顺序，命中 o.data。
200

>>> vars(o)
{'data': 200}
```

```
>>> X.data                          # 不影响原类型字段。
100

>>> vars(X)
mappingproxy({'data': 100, ...})
```

删除操作仅针对当前名字空间，不会按搜索顺序查找类型或基类。

```
>>> class X: pass
>>> X.data = 100

>>> o = X()
```

```
>>> del o.data                    # 当前实例名字空间并没有 data 成员。
AttributeError: data
```

成员访问总是按搜索规则进行，这与普通变量的静态作用域不同。

> 实质上是 LOAD_ATTR 和 LOAD_GLOBAL/LOAD_FAST 的区别。
> 有关静态作用域，请阅读“函数”章节。

```
class X:
    data = "X.data"


def test():
    o = X()
    print(o.data)                  # 找到 X.data。

    o.data = "o.data"              # 新增 o.data。
    print(o.data)                  # 找到 o.data。

    del o.data                     # 删除 o.data。
    print(o.data)                  # 找到 X.data。
```

```
>>> test()
X.data
o.data
X.data
```

私有字段

将私有字段暴露给用户是很危险的。因为无论是修改还是删除都无法被截获，由此可能引发意外错误。因没有严格意义上的访问权限设置，只好将它们“隐藏”起来。

如果成员名字（字段和方法等）以双下划线开头，那么编译器会自动对其重命名。

```
class X:
    __table = "user"

    def __init__(self, name):
```

```
        self.__name = name

    def get_name(self):
        return self.__name
```

重命名不包括系统方法，比如 __init__、__hash__ 等。

这些字段被编译器附加了类型名称前缀。虽然这种做法不能真正阻止用户访问，但基于名字约定也算是一种提示。

```
>>> vars(X)
mappingproxy({'_X__table': 'user', ...})

>>> vars(X("user1"))
{'_X__name': 'user1'}
```

```
>>> dis.dis(o.get_name)
  6           0 LOAD_FAST                0 (self)
              2 LOAD_ATTR                0 (_X__name)
              4 RETURN_VALUE
```

只是有个小问题，这种方式让继承类也无法“访问”。

重命名机制总是针对当前类型，所以在继承类型里，它所访问的并不是基类字段名称。只能将双下划线前缀改为单下划线，虽然不能自动重命名，不过提示作用依旧。

```
class A:
    __name = "user"

class B(A):
    def test(self):
        print(self.__name)           # 被当作当前类型私有字段重命名：_B__name。
```

```
>>> B().test()
AttributeError: 'B' object has no attribute '_B__name'
```

相关函数

几个与类成员相关的内置函数。

```
class A:
    data = "A.data"

class B(A):
    pass
```

```
>>> hasattr(B(), "data")                # 按搜索规则查找整个继承层次。
True
```

```
>>> getattr(B(), "data")                # 相当于 B().data，同样是按搜索规则进行。
'A.data'

>>> getattr(B(), "name", "abc")         # 返回默认值。
'abc'
```

```
>>> o = B()
>>> setattr(o, "data", "o.data")        # 等同 o.data = 'o.data'，针对当前名字空间。

>>> o.data
'o.data'
```

```
>>> o = B()
>>> delattr(o, "data")                  # 等同 del o.data，针对当前名字空间。
AttributeError: data
```

它们算是相关语句的函数版本，接收动态成员名称，还可用于 lambda 等环境。

```
>>> (lambda: getattr(B(), "data"))()
'A.data'
```

3. 属性

对私有字段进行重命名保护，那么公开字段呢?

问题核心是访问拦截，必须由内部逻辑决定如何返回结果。而属性（property）机制就是将读、写和删除操作映射到指定的方法调用上，从而实现操作控制。

```
class X:
    def __init__(self, name):
        self.__name = name

    @property                                                   # 读
    def name(self):
        return self.__name

    @name.setter                                                # 写（注意语法格式）
    def name(self, value):
        self.__name = value

    @name.deleter                                               # 删除
    def name(self):
        raise AttributeError("can't delete attribute")
```

```
>>> o = X("user1")
>>> o.name
'user1'

>>> o.name = "abc"
>>> o.name
'abc'

>>> del o.name
AttributeError: can't delete attribute
```

这种 @ 语法被称作装饰器（decorator），详情参考本书第九章。
方法名必须相同。默认从读方法开始定义属性，随后以属性名定义写和删除操作。
如果实现只读，或禁止删除，只需去掉对应的方法即可。

```
>>> vars(X)
mappingproxy({'name': <property>, ...})
```

当然，也可以 lambda 简化代码。

```
class X:
    name = property(
        fget = lambda self: getattr(self, "_name", None),
        fset = lambda self, value: setattr(self, "_name", value),
    )
```

以方法应对操作，可主动判断是否继续执行。例如，判断赋值数据是否合法，或阻止创建同名实例字段。

```
class X:
    __data = "X.data"

    @property
    def data(self):
        return self.__data

    @data.setter
    def data(self, value):
        if not isinstance(value, str):                    # 检查数据类型。
            raise ValueError("value type must be str")

        X.__data = value                                  # 直接修改类型字段。
```

```
>>> o = X()

>>> o.data = 1
ValueError: value type must be str

>>> o.data = "abc"
>>> vars(o)
{}
```

属性可实现延迟初始化，无需提前准备数据。且属性方法调用也未必绑定字段，可按需要从任意数据源获取。

优先级

尽管属性保存在类型名字空间，但其优先级高于同名实例字段。

```
class X:
    data = property(lambda self: "X.data")
```

```
>>> o = X()

>>> o.data = 123                            # 属性优先，data 没有指定赋值方法。
AttributeError: can't set attribute

>>> o.__dict__["data"] = 123                # 绕开限制，直接对 __dict__ 操作。
>>> vars(o)
{'data': 123}

>>> o.data                                  # 依旧优先返回 data 属性，而非实例字段。
'X.data'
```

这种情形是由描述符（descriptor）规则导致。

4. 方法

方法是种特殊函数，与特定对象绑定，用来获取或修改对象状态。

实际上，无论是对象构造、初始化、析构，还是运算符都以方法实现。按绑定目标和调用方式不同，可分为实例方法、类型方法，以及静态方法。

> 名字以双下划线开始和结束的方法，通常是特殊用途，由解释器和内部机制调用。

顾名思义，实例方法与实例对象绑定。其方法签名中，须将绑定对象作为第一参数，以便在方法中读取或修改数据状态。以实例引用调用时，无需显式传入第一实参，交由解释器自动完成。

> 官方（PEP8）建议参数名用 self，同样以 cls 作为类型方法第一参数名。
> 当然，也可使用其他命名，解释器并不以参数名字区分方法类别。

```
class X:
    def __init__(self, name):
        self.__name = name

    def get(self):                      # 以实例引用调用，自动传入 self 参数。
        return self.__name

    def set(self, value):
        self.__name = value
```

```
>>> o = X("Q.yuhen")

>>> o.get()                             # 忽略第一参数。
'Q.yuhen'

>>> o.set("qyuhen")
```

类型方法用来维护类型状态，面向族群提供服务接口。除约定的第一参数名称不同外，还需添加专门的装饰器，以便解释器将其与实例方法区分开来。

```
class X:
    __data = "X.data"
```

```
    @classmethod                        # 定义为类型方法。
    def get(cls):                       # 解释器自动将类型对象（X）作为 cls 传入。
        return cls.__data

    @classmethod
    def set(cls, value):
        cls.__data = value
```

```
>>> X.get()                             # 同样忽略 cls 参数。
'X.data'

>>> X.set('hello')
```

至于静态方法，更像是普通函数。既不接收实例引用，也不参与类型处理，所以也就没有自动传入的第一参数。使用静态方法，更多原因是将类型作为一个作用域，或者为当前类型添加便捷接口。

例如，在同一个模块中，分别定义了 AES、DES 等加密类型。那么，基于类型创建静态便捷方法，除避免同名冲突外，还更利于代码维护。

```
class DES:
    def __init__(self, key):
        self.__key = key

    def encrypt_bytes(self, value):                         # 实例方法。
        pass

    @staticmethod                                           # 定义为静态方法。
    def encrypt(key, s):                                    # 所有参数都需显式传入。
        des = DES(key)
        return DES(key).encrypt_bytes(s.encode("utf-8"))
```

```
>>> DES.encrypt("key", "abc")
```

且不论是哪种方法，它们都保存在类型名字空间中，所以不能重名。装饰器和参数差异，并不能改变在同一字典中对同一主键重复赋值的现实。

```
class X:
    def test(self):
        pass
```

```
    @classmethod
    def test(cls, a):
        pass

    @staticmethod
    def test(x):
        pass
```

```
>>> vars(X)
mappingproxy({'test': <staticmethod at 0x1023d5cf8>, ...})
```

绑定

所谓方法绑定的真实面目如何？

```
class X:
    def test(self): pass
```

```
>>> o = X()
>>> o.test
<bound method X.test>

>>> X.test
<function X.test>
```

同一个方法，仅仅是引用方式不同，结果就有绑定方法和函数的区别。那么奥妙在哪？这其中依然是描述符（descriptor）机制在起作用。它会将实例附加到方法的 __self__ 属性，当解释器执行时就可隐式自动传入 self 参数。

```
>>> o.test.__self__ is o
True

>>> X.test.__self__
AttributeError: 'function' object has no attribute '__self__'
```

当然，就算不是绑定方法，依然可像函数那样调用，无非是显式传入 self 实参而已。

```
>>> X.test(o)               # 显式传入 self 参数。
```

可使用 method.__func__ 获取 __code__ 相关信息。
另外，Python 3 已没有 unbound method 说法，直接当作 function。

自带 __self__ 属性后，无论作为参数传递，或赋值给变量，总是能正确引用原实例。

```
class X:
    def __init__(self, name):
        self.__name = name

    def test(self):
        return self.__name
```

```
>>> a, b = X("a").test, X("b").test          # 将方法赋值给变量。

>>> a()
'a'

>>> b()
'b'
```

与之类似的还有类型方法。实现方式与实例方法完全相同，甚至连存储类型引用的字段名都还是 __self__。至于孤零零被剩下的静态方法，不傍他人，总以函数身份出现。

```
class X:
    @classmethod
    def class_method(cls): pass

    @staticmethod
    def static_method(): pass
```

```
>>> X.class_method.__self__ is X
True

>>> X.static_method
<function X.static_method>
```

特殊方法

几个自动调用，与对象生命周期相关的方法。

- __new__ : 构造方法，创建对象实例。
- __init__ : 初始化方法，设置实例相关属性。
- __del__ : 析构方法，垃圾回收时调用。

创建实例时，会先后调用构造和初始化方法。

```
class X:
    def __new__(cls, *args):              # 与 __init__ 接收相同调用参数。
        print("__new__", args)
        return super().__new__(cls)

    def __init__(self, *args):            # self 由 __new__ 创建并返回。
        print("__init__", args)
```

```
>>> X(1, 2)
__new__  (1, 2)
__init__ (1, 2)
```

如果 __new__ 方法返回实例与 cls 类型不符，将导致 __init__ 无法执行。

```
class X:
    def __new__(cls):
        print("__new__", cls)
        return [1, 2]                      # 返回结果与 cls 类型不符。

    def __init__(self):
        print("__init__")
```

```
>>> X()
__new__ <class 'X'>
```

更多详情，请参考第九章《元类》。

---

5. 继承

面向对象有三个基本特征：封装、继承和多态。

封装讲究结构复用，逻辑内敛，以固定接口对外提供服务。遵循单一职责，规定每个类型仅有一个引发变化的原因。多于一个的耦合设计会导致脆弱类型，任何职责变更都可能引发连带变故。单一封装的核心是解耦和内聚，让设计更简单清晰，代码更易测试和冻结，避免不确定性。

继承并非要复制原有类型，而是一种增量进化。在遵循原有设计和不改变既有代码的前提下，添加新功能，或改进算法。对应开闭原则，对修改封闭，对扩展开放。因为修改任何已完成测试，或正被用户使用的类型都存在风险。除此之外，还不能违背里氏替换原则，也就是多态特征，所有继承子类应该能直接用于引用父类的场合。保持类型家族整体行为统一是必要的，诸多设计模式以此为基础，同时保证平滑升级。

继承给设计带来便利的同时，也带来维护上的麻烦。其天生基于实现的依赖方式导致耦合大范围存在，且过多继承层次直接提升了代码复杂度。我们应该分清楚是功能扩展，还是使用某些功能。如果是后者，那么应该以组合替代。因为组合仅依赖对方外部接口，没有任何实现上的关联，更无需介入内部运作，很容易替换。当然，组合无法自动获得祖先能力，需要添加相应方法才能实现相同操作接口，更无法直接支持多态。两者非替代关系，都有其适用场景。

另外，我们也习惯将复杂类型的公用部分剥离出来，形成稳固的抽象基类。其他引发变化的相似因素则被分离成多个子类，确保单一职责得到遵守，并能相互替换。常见例子就是类型里的选择分支。先将整体框架被抽象成基类，然后每个子类仅保留单一分支逻辑。

> 封装是将不同部件藏到车壳里，仅向用户提供方向盘、油门、刹车等有限操控接口，以车控电脑算法间接控制汽车。
>
> 继承则是设计师提供一个通用框架，从技术、生产、维修等整体方向考虑。然后选用不同等级配件和控制算法，以局部变更快速形成手动、自动，以及舒适和豪华等版本。当然，设计师也可以将一个久经考验的成熟车型设计独立出来，以此为基础发展出不同版本。
>
> 可无论哪样，每款车都可以车族身份出场，代表整体车型设计。但在行驶过程中又能体现出其自身的能力和特点，这便是多态。

在理解面向对象基本规则后，我们可写出更优雅的代码，更自然的设计。

5.1 统一类型

Python 3 总算甩掉陈旧包袱，统一了两种互不兼容的类型体系。这让元编程有了更大发挥余地，也避免了以往需要处理不同类型的麻烦。

Python 2.7

```
>>> class A: pass                    # classic class
>>> class B(object): pass            # new-style class

>>> type(A)
<type 'classobj'>

>>> type(B)
<type 'type'>

>>> issubclass(A, object)
False

>>> A.__class__
AttributeError: class A has no attribute '__class__'
```

Python 3 不再支持 Classic Class，所有类型都是 New-Style Class。

```
class A    : pass                  # 默认继承自 object。
class B(A) : pass
class C(B) : pass
```

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

>>> type(A) is A.__class__
True
```

除子类以 __base__ 引用基类外，基类也可用 __subclasses__ 获知所有直接子类。

```
>>> B.__base__
A

>>> A.__subclasses__()               # 仅返回直接子类。
[B]
```

甚至可介入子类型创建过程。

__init_subclass__ 是个隐式类型方法，在所有层次子类型创建时被调用，甚至可接收键值参数。

```
class A:
    def __init_subclass__(cls, **kwargs):
        print(cls, kwargs)
```

```
>>> class B(A): pass
<class 'B'>  {}

>>> class C(B, data = "hello"): pass              # 向基类传递参数。
<class 'C'>  {'data': 'hello'}
```

5.2 初始化

初始化方法 __init__ 是可选的。

如果子类没有新构造参数，或者新的初始化逻辑，那么没必要创建该方法。因为按照搜索顺序，解释器会找到基类方法并执行。也因同样缘故，我们须在子类初始化方法中显式调用基类方法。

```
class A:
    def __init__(self):
        print("A.init")

class B(A): pass
```

```
>>> B()
A.init
```

可使用名字引用基类方法，或调用 super 返回基类代理，以解除对类型名字的直接依赖。

```
class A:
    def __init__(self, x):
        self.x = x
```

```
class B(A):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y
```

```
>>> o = B(1, 2)

>>> vars(o)
{'x': 1, 'y': 2}
```

无论是 A.__init__，还是 B.__init__，其 self 参数都引用同一实例对象，所以在不同继承层次里创建的实例字段都存储在同一名字空间中，不会出现多个同名成员。

5.3 覆盖

覆盖（override）是指子类重新定义基类方法，从而实现功能变更。在其他某些语言中，仅有虚方法可被覆盖，但 Python 并无此概念，只要在搜索优先级更高的名字空间中定义同名方法即可。

覆盖不能改变方法参数列表和返回值类型，确保不影响原有调用代码和相关文档，这与名字遮蔽不同。无论是多态机制，还是名字搜索顺序，都能确保新定义的方法被执行。

```
class A:
    def m(self) : print("A.m")
    def do(self): self.m()                # 从 self 开始搜索。

class B(A):
    def m(self) : print("B.m")            # override
```

```
>>> A().do()
A.m

>>> B().do()
B.m
```

方法搜索发生在运行期，不会静态绑定具体类型方法。

```
>>> dis.dis(A.do)
  6           0 LOAD_FAST                0 (self)
              2 LOAD_ATTR                0 (m)             # 运行期搜索 self.m。
              4 CALL_FUNCTION            0
```

> 因方法没有属性（property）那样的描述符优先级策略，应注意避免被实例同名成员遮蔽。

5.4 多继承

多重继承允许类型有多个基类。

这是曾被疯狂追捧，后来被嫌弃的语言特性。从优点上说，它提供一种混入（mixin）机制，让既有体系的类型轻松扩展出其他体系的功能。但另一方面，它会导致严重的混乱。试想一个类型同时继承猫和狗会有什么后果？是汪汪叫，还是喵喵？这种含糊性直接让设计和代码复杂度被提升到非常麻烦的程度。

新语言大多会避开多继承，改为单继承体系，并严格遵守相关设计原则。至于混入需求，则多以接口实现。如此，类型是什么是确定的，能做什么则以组合构建。

> 继承表达是什么，而接口体现能做什么。是什么对接口并不重要，它也不关心起源后续。

作为老古董，Python 依然保留对多继承支持，不过使用时需慎重，能免则免。

```
class A:
    def a(self): pass

class B:
    def b(self): pass

class X(A, B): pass
```

```
>>> dir(X)                          # 可访问所有基类成员。
['a', 'b']
```

要获取全部基类，须使用 __bases__。

```
>>> X.__base__
A

>>> X.__bases__
(A, B)
```

不同于 __base__ 仅返回单个基类的只读属性，__bases__ 按继承顺序返回全部基类。且能用来调整基类顺序，或动态插入新的基类，实现功能混入。

```
class C:
    def c(self): pass
```

```
>>> X.__bases__ = (B, A, C)             # 调整继承顺序，并添加新的基类。

>>> dir(X)                              # 可访问新基类成员。
['a', 'b', 'c']
```

除动态横向扩展外，还可直接注入到继承体系中。

```
class A:
    def test(self): print("A.test")

class B(A):
    pass
```

```
class Proxy(A):
    def test(self):                      # 拦截 A.test 调用。
        print("proxy")
        return super().test()
```

```
>>> B.__bases__ = (Proxy,)              # 修改 B 基类，形成 B -> Proxy -> A 方式。

>>> B().test()
proxy
A.test
```

__mro__

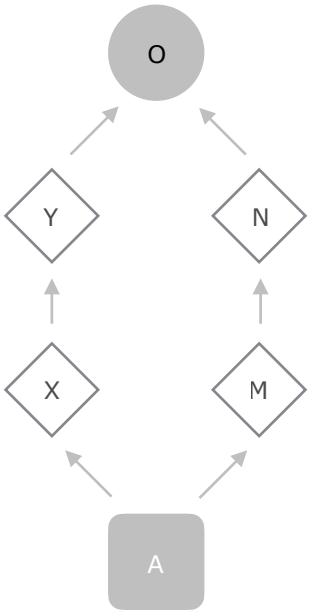
任何类型最终基类都是 object，所以多重继承会出现菱形布局。可最大的问题是，当多条继承线上出现相同名字的成员时，如何选择？

这涉及成员搜索基本规则：MRO（method resolution order）。对于单继承，由近到远向祖先类依次查找。而复杂的在于，多继承在深度和广度间的排列。

> 在 Python 2 时期，Classic Class 使用深度优先的 DFLR（depth-first, left-to-right）模式。
> 后 New-Style Class 因单根继承的关系，进化为 MRO（breadth-first than depth-first）算法。

MRO：

1. 按“深度优先，从左到右”顺序获取类型列表。
2. 移除列表中重复类型，仅保留最后一个。
3. 确保子类总在基类前，并保留多继承定义顺序。




```
DFLR [A,X,Y,O,M,N,O]
MRO  [A,X,Y,  M,N,O]
```

```
class Y:
    n = "Y.n"
    def test(self): print("Y.test")

class X(Y): pass
```

```
class N: pass

class M(N):
    n = "M.n"
    def test(self): print("M.test")
```

```
class A(X, M): pass

>>> A.__mro__
(A, X, Y, M, N, object)
```

```
>>> A().n
'Y.n'

>>> A().test()
Y.test
```

简单点说，就是“左侧优先、子类优先”。当然，这不违背实例优先级最高的原则。
另外，__mro__ 是只读属性，且不能通过实例访问。

按算法规则，对 __bases__ 的调整会直接影响 __mro__ 结果。

```
>>> A.__bases__ = (M, X)

>>> A.__mro__
(A, M, N, X, Y, object)

>>> A().test()
M.test
```

任何时候，都应避免多条继承线存在交叉现象，并竭力控制多继承和继承深度。这有助于梳理其相互关系，为代码可读和可维护性提供保障。

super

该函数返回基类型代理，完成对基类成员的委托访问。

有两个参数，第二参数提供 __mro__ 列表，第一参数则指定起点。函数总是返回起点位置后一类型。这要求第二参数必须是首参数的实例或子类型，否则首参数不会出现在列表中。同时，第二参数还是调用实例和类型方法所需的绑定对象。

算法伪码

```
def super(t, o):
    mro = getattr(o, "__mro__", type(o).__mro__)          # 返回实例或类型的 mro 列表。
    index = mro.index(t) + 1                              # 返回列表中下一类型。
    return mro[index]
```

对单继承，通常省略参数，默认使用当前类型和实例，返回直接基类。但对多继承，则须明确指定起点和列表提供者。

```
class A:
    @classmethod
    def demo(cls): pass
    def test(self): print("A")

class B(A):
    def test(self): print("B")

class C(B):
    def test(self): print("C")
```

```
>>> C.__mro__
(C, B, A)

>>> super(C, C).test
<function B.test>

>>> super(B, C).test
<function A.test>
```

依旧使用 __self__ 作为绑定属性，存储第二参数引用。

```
>>> o = C()

>>> super(C, o).__self__ is o
True
```

```
>>> super(C, o).test
<bound method B.test>

>>> super(C, C).demo
<bound method A.demo>
```

不建议直接使用 __class__.__base__ 访问基类，这可能引发一些意外错误。

```
class A:
    def test(self):
        print("A.test")

class B(A):
    def test(self):
        print("B.test")
        self.__class__.__base__.test(self)   # 站在 B 角度，self.__class__.__base__ 似乎指向 A。
                                             # 但实际上，self 可能是 B 的子类实例，比如 C。
                                             # 如此，self.__class__.__base__ 实际指向的依然是 B。
                                             # 其结果就是导致 B.test 被递归。
class C(B): pass
```

```
>>> C().test()
RecursionError: maximum recursion depth exceeded in comparison
```

正确方法是显式类型名字访问，或用 super 函数。即便省略 super 参数，它默认也使用当前类型 B 作为搜索起点。如此，可确保访问 A.test 方法，避免递归错误。

5.5 抽象类

抽象类表示部分完成，且不能被实例化的类型。

作为一种设计方式，抽象类可用来分离主体框架和局部实现，或将共用和定制解耦。不同于接口纯粹的调用声明，抽象类属继承树组成部分，允许有实现代码。从抽象类继承，必须实现所有层级未被实现的抽象方法，否则无法创建实例。

> 抽象类就像一个准系统。汽车厂家在完成整体设计和主要部件生产后，再由不同地区的分公司依照当地路面、油料以及法规选用不同配件，最终完成适合当地销售策略的成品。

定义抽象类，需继承 ABC，或使用 ABCMeta 元类。

```
from abc import ABCMeta, abstractmethod

class Store(metaclass = ABCMeta):

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

    def save(self, data):
        with open(self.name, "wb") as f:
            d = self.dump(data)
            f.write(d)

    def read(self):
        with open(self.name, "rb") as f:
            return self.load(f.read())

    @abstractmethod
    def dump(self, data): ...

    @abstractmethod
    def load(self, data): ...
```

```
>>> Store("test.dat")
TypeError: Can't instantiate abstract class Store with abstract methods dump, load
```

该示例定义存储框架，并实现了通用部分接口方法。至于具体序列化过程，则交由子类完成。如此，可按需创建不同版本。

```
import pickle

class PickleStore(Store):

    def dump(self, data):
        return pickle.dumps(data)
```

```
    def load(self, data):
        return pickle.loads(data)
```

```
>>> s = PickleStore("test.dat")

>>> s.save({"x":1, "y":2})

>>> s.read()
{'x': 1, 'y': 2}
```

如果从抽象类继承，未实现全部方法，或添加新的抽象定义，那么该类型依然是抽象类。另外，抽象装饰器还可用于属性、类型方法和静态方法。

```
from abc import ABCMeta, abstractmethod

class A(metaclass = ABCMeta):

    @classmethod                                    # 注意两个装饰器顺序。
    @abstractmethod
    def hello(cls): ...

class B(A): pass
```

```
>>> B()
TypeError: Can't instantiate abstract class B with abstract methods hello
```

即便是抽象类型方法，依然需要实现，否则无法创建实例。

```
class B(A):
    @classmethod
    def hello(cls): pass
```

```
>>> B()
<B at 0x105021940>
```

6. 开放类

在运行期间，可动态向实例或类型添加新成员，其中自然也包括方法。

有别于实例字段，方法需要添加到类型名字空间，因为要作用于所有实例和后续类型。另外，只有添加到类型名字空间才能自动构成绑定，这是解释器运行规则决定的。

将已绑定方法添加到实例名字空间，只能算是字段赋值，算不上添加方法。
而将一个函数添加到实例，即便手工设定 __self__ 也无法构成绑定。

```
>>> class X: pass

>>> o = X()
>>> o.test = lambda self: None                # 向实例添加函数字段。
>>> o.test.__self__ = o                       # 尝试手工设定绑定。

>>> o.test                                    # 依然无法构成绑定，除非重写描述符相关方法。
<function <lambda>>

>>> o.test()
TypeError: <lambda>() missing 1 required positional argument: 'self'
```

考虑到类型 __dict__ 是个只读代理，只能以字段赋值方式添加新的方法。如果是类型和静态方法，还需调用装饰器函数。

受益于动态成员查找方式，新增的方法对已创建实例同样有效。

```
>>> class X: pass
>>> o = X()                                    # 预先创建的实例。
```

```
>>> X.a = lambda self: print(f"instance method: {self}")
>>> X.b = classmethod(lambda cls: print(f"class method: {cls}"))
>>> X.c = staticmethod(lambda: print("static method"))
```

```
>>> o.a()
instance method: <X object at 0x111688908>

>>> o.b()
```

```
class method: <class 'X'>

>>> o.c()
static method
```

object

作为祖先类的 object，有点特殊，我们无法向其类型和实例添加任何成员。

```
>>> object.x = 1
TypeError: can't set attributes of built-in/extension type 'object'

>>> o = object()
>>> o.x = 1
AttributeError: 'object' object has no attribute 'x'
```

实例甚至没有 __dict__ 属性。

```
>>> object().__dict__
AttributeError: 'object' object has no attribute '__dict__'
```

对此，可用 SimpleNamespace 替代。

> SimpleNamespace 简单继承自 object，用于替代某些“class X: pass”语句。
> 不同于 namedtuple 创建结构化类型，SimpleNamespace 创建实例。

```
>>> o = types.SimpleNamespace(a = 1, b = "abc")
>>> o.c = [1, 2]
>>> o.__dict__
{'a': 1, 'b': 'abc', 'c': [1, 2]}
```

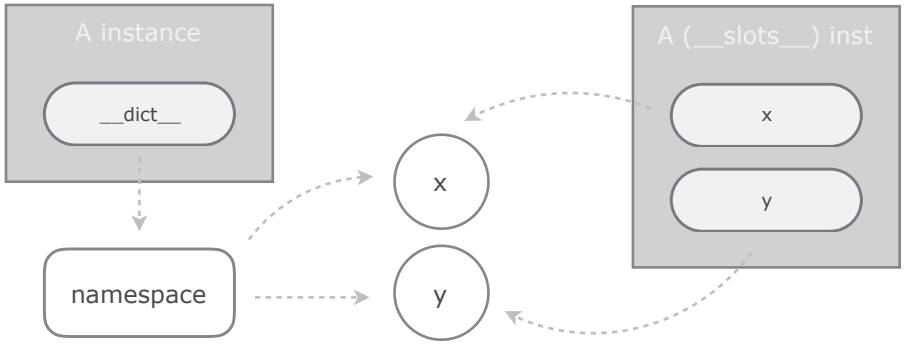
__slots__

名字空间字典带来便利的同时，也会造成内存和性能问题。

对那些需要创建海量实例的类型，须做“固化”处理。通过设置 __slots__，阻止实例创建 __dict__ 等成员。解释器仅为指定成员分配空间，添加任何非预置属性都会引发异常。

对于有 __slots__ 设置的类型，解释器在创建类型对象时，直接将指定成员包装成描述符后静态分配在类型对象尾部。实例字段也不再通过 __dict__ 存储，改为直接分配引用内存。这样一来，只能替换引用内容，无法改变成员名字，更无法分配新的成员空间。

成员后续操作均由描述符完成。可间接修改或删除字段内容，但无法改变描述符本身。




```
class A:
    __slots__ = ("x", "y")
```

```
>>> o = A()

>>> o.z = 100                                          # 不能新增属性，因为无法为其分配引用内存。
AttributeError: 'A' object has no attribute 'z'

>>> o.x = 1                                            # 只能为预定字段赋值。
>>> o.y = 2
```

```
>>> o.x = "abc"                                        # 通过描述符修改字段内容。
>>> o.x
'abc'
```

```
>>> del o.x                                            # 删除字段内容，但描述符依然存在。
>>> o.x = 100                                          # 引用内存也在，可重新赋值。
```

对 __slots__ 的修改并不会影响类型创建时设定的内存分配策略。加上预定成员以描述符实现，所以也无法更换其名称。

```
>>> A.__slots__ = ("x", "y", "z")

>>> o = A(1, 2)
>>> o.z = 3
AttributeError: 'A' object has no attribute 'z'
```

```
>>> A.__slots__ = ("a", "b")

>>> dir(A)
['x', 'y']
```

注意，__slots__ 限制的是实例，而非类型。仍然可以为类型添加新成员。
如果在 __slots__ 中添加 __dict__，可回归“原来”样子，不过那就没什么意义了。

继承有 __slots__ 设置的类型，同样需要添加该设置。可为空，或是新增字段，以指示解释器继续使用特定分配策略。

```
class A:
    __slots__ = ("x", "y")

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

```
class X(A):                                 # 没有 __slots__，会创建 __dict__。

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


class Y(A):
    __slots__ = ("z",)                      # 新增字段列表，或为空。

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z
```

```
>>> x = X(1, 2, 3)

>>> x.__dict__                                  # 包含 __dict__，用于存储非 __slots__ 新增成员。
{'z': 3}
```

```
>>> y = Y(1, 2, 3)

>>> y.__dict__
AttributeError: 'Y' object has no attribute '__dict__'

>>> dir(y)
['x', 'y', 'z']
```

在了解功能细节后，测试我们所关注的内存和性能问题。先用 pympler 详细统计实例及其成员的全部内存开销，以做初步对比。

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class B:
    __slots__ = ("x", "y")

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

从结果看，__dict__ 引发了“巨大”内存开销。对单个或少量实例或许毫不起眼，可一旦海量叠加就是大麻烦。

```
>>> from pympler.asizeof import asized
```

```
>>> print(asized(A(1, 2), detail = 2).format())

<A> size=344 flat=56
    __dict__ size=288 flat=112
        [K] x    size=56 flat=56
        [K] y    size=56 flat=56
        [V] x: 1 size=32 flat=32
        [V] y: 2 size=32 flat=32
```

```
    __class__ size=0 flat=0
```

```
>>> print(asized(B(1, 2), detail = 2).format())

<B> size=192 flat=56
    __slots__ size=72 flat=72
            x size=32 flat=32
            y size=32 flat=32
    __class__ size=0  flat=0
```

继续模拟百万级别“海量”需求，观察创建实例所需内存和时间开销。

test.py

```
class A:
    def __init__(self):
        self.data = 100


class B:
    __slots__ = ("data")

    def __init__(self):
        self.data = 100


@profile
def test():
    x = list(range(1000000))
    a = [A() for i in x]
    b = [B() for i in x]


if __name__ == '__main__':
    test()
```

创建时间差距不算太大，姑且标记为中低优先级。但内存开销相差三倍，就亟待解决。毕竟，相比大多真实案例，百万真算不上什么大数字。

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

Hits         Time    Per Hit    % Time      Line Contents
==============================================================
   1      1389059   1389059.0     53.9      a = [A() for i in x]
   1      1153548   1153548.0     44.7      b = [B() for i in x]
```

```
$ python -m memory_profiler test.py

  Mem usage      Increment       Line Contents
================================================
233.246 MiB    161.895 MiB       a = [A() for i in x]
287.824 MiB     54.578 MiB       b = [B() for i in x]
```

测试结果仅供参考，请按实际需求组织测试用例。

---

7. 运算符重载

每种运算符都对应一个有特殊名称的方法。

解释器会将运算符指令转换成方法调用，方法名可参考标准库 operator 文档。
运算符重载就是定义目标方法。但不限于运算符，还包括内置函数和某些语句。

```
class X:
    def __init__(self, data):
        self.data = data

    def __repr__(self):
        return f"{self.__class__} {self.data!r}"

    def __add__(self, other):                           # 加法 +
        return X(self.data + other.data)

    def __iadd__(self, other):                          # 增量赋值 +=
        self.data.extend(other.data)
        return self
```

```
>>> a, b = X([1, 2]), X([3, 4])

>>> a + b
<class 'X'> [1, 2, 3, 4]

>>> a += b
>>> a
<class 'X'> [1, 2, 3, 4]
```

__repr__

函数 repr、str 都用于输出对象字符串格式信息。

方法 __repr__ 倾向于输出运行期状态，比如类型、id，以及关键性内容，适合调试和观察。而 __str__ 通常返回内容数据，面向用户，易阅读，可输出到终端或日志。

```
class X:
    def __init__(self, x):
        self._x = x
```

```
    def __repr__(self):
        return f"<X:{hex(id(self))}>"

    def __str__(self):
        return str(self._x)
```

```
>>> repr(X(1))
'<X:0x10664d518>'

>>> str(X(1))
'1'
```

__item__

为对象添加索引（index）或主键（key）访问。

```
class X:
    def __init__(self, data = None):
        self.data = data or []

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        if index >= len(self.data):
            self.data.append(value)
            return

        self.data[index] = value

    def __delitem__(self, index):
        return self.data.pop(index)
```

```
>>> o = X([1, 2, 3, 4])

>>> o[2] = 300

>>> del o[2]
>>> o.data
[1, 2, 4]
```

__call__

让对象可以像函数那样被调用。

实现该方法的对象被称作 callable，函数也是其中一种。
用实例对象替代函数，让逻辑自带内部状态（封装性比闭包更好）。

```
class X:
    def __init__(self):
        self.count = 0

    def __call__(self, s):
        self.count += 1
        print(s)


def test(f):
    f("abc")
```

```
>>> test(lambda s: print(s))
abc

>>> test(X())
abc

>>> o = X()
>>> o("abc")
abc
```

__dir__

定制 dir 函数返回值，隐藏部分成员。

```
class X:
    def __dir__(self):
        return (m for m in vars(self).keys() if not m.startswith("__"))
```

```
>>> o = X()
>>> o.a = 1
>>> o.b = 2
```

```
>>> dir(o)
['a', 'b']
```

__getattr__

几个与实例属性访问相关的方法。

- __getattr__ : 当整个搜索路径都找不到目标属性时触发。
- __setattr__ : 拦截对任何属性的赋值操作。
- __delattr__ : 拦截对任何属性的删除操作。

拦截到赋值和删除操作后，由拦截方法负责处理赋值和删除行为。忽略则视为放弃该操作。

在拦截方法内部，通过属性或 setattr 等函数调用都可能再次被拦截，甚至引发递归调用错误。应直接操作 __dict__，或使用基类 object.__setattr__ 方法。

```
class A:
    a = 1234


class B(A):
    def __init__(self, x):
        self.x = x

    def __getattr__(self, name):
        print(f"get: {name}")
        return self.__dict__.get(name)

    def __setattr__(self, name, value):
        print(f"set: {name} = {value}")
        self.__dict__[name] = value

    def __delattr__(self, name):
        print(f"del: {name}")
        self.__dict__.pop(name, None)
```

```
>>> o = B(1)
set: x = 1                              # 拦截 B.__init__ 里对 self.x 的赋值操作。

>>> o.a                                 # 搜索路径里能找到的成员，不会触发 __getattr__。
1234
```

```
>>> o.xxx                               # 找不到，才会触发。
get: xxx

>>> o.x = 100                           # 任何赋值操作都会被 __setattr__ 拦截，无论是否存在。
set: x = 100

>>> del o.x                             # 拦截任何删除操作，无论是否存在。
del: x
```

方法 __getattribute__ 拦截任何实例属性访问，无论是否存在。

拦截目标包括 __dict__，这意味着只能通过基类方法进行操作。
访问不存在成员时，__getattribute__ 拦截后不再触发 __getattr__，除非显式调用或触发异常。

此例中，因 object.__getattribute__ 找不到目标属性，所以会触发 A.__getattr__ 调用，继而还会拦截到其内部对 __dict__ 的访问。

```
class A:
    def __init__(self, x):
        self.x = x

    def __getattr__(self, name):
        print(f"getattr: {name}")
        return self.__dict__.get(name)

    def __getattribute__(self, name):                # 返回结果，或引发 AttributeError。
        print(f"getattribute: {name}")
        return object.__getattribute__(self, name)
```

```
>>> o = A(1)

>>> o.x                                             # 无论是否存在，均被拦截。
getattribute: x
1

>>> o.s
getattribute: s                                     # __getattribute__ 拦截。
getattr: s                                          # 因 object 找不到目标，触发 __getattr__。
getattribute: __dict__                              # __getattr__ 访问 __dict__ 被拦截。
```

八. 异常

1. 异常

当程序执行出现故障，或返回非预期结果时，通常会抛出一个异常对象用来表达非正常状态。但不能因此就将异常等同于错误，因为异常只是一种运行期状态值。

常见错误处理方式有两类：其一是函数返回错误码，由调用方决定如何处理。另一类是以专用语句保护代码块，当异常发生时，跳转到指定处理单元。前者有较好的性能，而后者可分离正常逻辑与故障处理代码。

将错误码作为函数签名的组成部分，可让调用方预先知道可能会发生什么。常见于系统编程，尤其是系统调用函数，因为错误码本身就是结果预选项之一。且该机制无需处理调用堆栈信息，无需额外状态保存和流程控制机制，简单高效。但也正因为执行流程不会主动中断，所以必须检查全部返回值，任何忽略行为都可能引发连续错误。

这对应用开发是很难忍受的，逻辑处理中充斥冗长的错误处理代码，直接导致可读性变差，更不便于维护和重构。相比而言，结构化异常要更优雅一些。虽然需要付出额外的运行期代价，但可以统一错误代码。毕竟，错误处理仅是意外分支。当然，对异常的捕获同样需要倍加小心，任何隐式行为都存在风险。

执行机制

在系统内部，解释器使用一种称作“块栈”（block stack）的结构处理异常逻辑。它和执行栈一起被栈帧管理。区别在于，块栈专门用来处理循环和异常跳转。在运行期，相关指令会提前将跳转位置信息存储到块栈，需要时直接从中获取即可。

```
def test():
    try:
        raise Exception
    except:
        print("except")
```

```
>>> dis.dis(test)
```

```
      0 SETUP_EXCEPT             8 (to 10)        # 向块栈添加 except 跳转位置设置。

      2 LOAD_GLOBAL              0 (Exception)    # 创建并引发异常。
      4 RAISE_VARARGS            1                # 解释器从块栈弹出设置，按参数跳转。

      6 ...
      8 ...

>>   10 POP_TOP                                   # 跳转到此处。
     12 POP_TOP                                   # 清除 exc 参数。
     14 POP_TOP

     16 LOAD_GLOBAL              1 (print)        # 执行 except 代码。
     18 LOAD_CONST               1 ('except')
     20 CALL_FUNCTION            1
     22 POP_TOP
     24 POP_EXCEPT                                # 移除块栈内的设置。
     26 JUMP_FORWARD             2 (to 30)        # 处理后，函数正常返回。
     28 END_FINALLY

>>   30 LOAD_CONST               0 (None)
     32 RETURN_VALUE
```

无论异常是主动抛出，还是函数调用引发，都会导致解释器转入错误处理逻辑。解释器会检查当前块栈内是否有匹配的处理逻辑，如果有则跳转并执行相应指令。如果没有，那么当前函数被终止，并返回 NULL。该返回值会导致上一级函数再次进入错误处理逻辑，其结果要么异常被捕获，要么沿调用堆栈向外传递，直至进程崩溃。

解释器执行伪码

```
EvalFrame (Frame *f)
{
    for(;;)
    {
        RAISE_VARARGS:
            do_raise()                      // 保存异常对象。
            goto error

        CALL_FUNCTION:
            res = call_function()
            if (res == NULL) goto error     // 返回 NULL 表示有错误发生。

        RETURN_VALUE:
            retval = POP()
            why = WHY_RETURN                // 正常结束标志。
            goto end

error:
        why = WHY_EXCEPTION                 // 异常发生标志。

end:
```

```
        /* 如果有异常发生，则检查当前块栈是否有匹配处理逻辑。
         * 如果有对应设置，那么跳转执行处理指令。
         * 被捕获处理后，必然以 RETURN_VALUE 指令结束，从而解除异常。
         * 鉴于可能有多层嵌套，需要循环检查。*/
        while (why != WHY_NOT && iblock > 0)
        {
            // 从块栈弹出设置。
            b = f->block[iblock - 1]
            f->iblock--

            // 跳转执行。
            if (why == WHY_EXCEPTION && (SETUP_EXCEPT || SETUP_FINALLY)) {
                why = WHY_NOT
                JUMPTO(handler)
                break
            }
        }

        // 如果异常未被处理，则终止当前函数执行。
        if (why != WHY_NOT) break
    }

    // 非正常结束，函数返回 NULL。
    if (why != WHY_RETURN) retval = NULL

exit:
    /* 终止当前函数执行，回到上一级函数调用。
     * 如果异常未被处理，那么上一级函数获得的返回值就是 NULL，会再次进入 error 逻辑。*/
    tstate->frame = f->f_back
    return retval
}
```

注意：NULL != None。None 是正常对象，有值有地址。

异常对象被保存到当前线程状态里，可使用 sys.exc_info 获取。

```
def test():
    try:
        raise Exception("err")
    except:
        print(sys.exc_info())
```

```
>>> test()
(<class 'Exception'>, Exception('err',), <traceback object at 0x106c6b248>)
```

用线程保存异常对象，仅能确保外层调用能获取异常状态。但依旧缺少一份重要数据，就是在什么地方，因为什么原因导致异常发生，这对代码调试尤为重要。为此，当异常发生时，解释器会逐级为每个未能捕获异常的函数调用创建关联跟踪对象（traceback），它负责存储栈帧、最后指令位置，以及源码行等信息。如此，就算引发异常的函数结束，在外层调用中依然可以查询到详细执行信息。

与栈帧类似，对应函数的跟踪对象也构成链表结构，如此就能还原异常发生时的完整调用堆栈状态。可用来恢复现场，进入调试模式，或输出相关数据。

```
def div(a, b):
    return a/b

def main():
    try:
        print(div(10, 0))
    except:
        _, _, tb = sys.exc_info()
        print(traceback.print_tb(tb))
```

```
>>> main()
File "<python>", line 6, in main
  print(div(10, 0))
File "<python>", line 2, in div
  return a/b
```

异常一旦被捕获处理，保存在线程状态内的 exc_type、exc_value、exc_traceback 都将被清除。

异常处理

完整异常处理结构包含四个组成部分。

1. **try** : 需要保护的代码块。
2. **except** : 异常发生时，按所属类型捕获并处理。
3. **else** : 异常未发生时执行，但最少有一个 except 语句。
4. **finally** : 无论异常是否发生，总是执行。

处理单元可选，但必须按顺序排列，否则会导致语法错误（SyntaxError）。

```
def test(n):
    try:
        print("try")
        if not n: raise Exception
    except:
        print("except")
    else:
        print("else")
    finally:
        print("finally")
```

```
>>> test(1)
try
else
finally
```

```
>>> test(0)
try
except
finally
```

当被保护代码块发生异常时，会终止其流程，立即跳转到相关异常处理单元。即便异常被捕获并修复，也不再恢复原有流程，而是从处理单元后继续执行。如此，我们应该细心选择保护范围。

```
def test():
    try:
        print("try")
        raise Exception
        print("try2")                  # 此行代码不会执行。
    except:
        print("except")

    print("exit")                      # 异常处理完成后，继续后续代码执行。
```

```
>>> test()
try
except
exit
```

使用 raise 语句主动抛出异常，但必须是 BaseException 子类或实例。

```
>>> raise "abc"
TypeError: exceptions must derive from BaseException
```

通常我们会选择 Exception，可直接使用类型，交由解释器创建实例。

多个 except 语句按顺序匹配，所以应像选择语句那样，优先处理更具体的异常类型。除非必要，不建议捕获所有异常（Exception）。其一，可能隐式忽略需要修复的错误；其次，意外拦截其他用途的异常信息。

```
def test():
    try:
        raise IndexError
    except Exception:                           # 按顺序匹配，所以在此处捕获。
        print("Exception")
    except IndexError:                          # 永远不会执行，与原意不符。
        print("IndexError")
```

```
>>> test()
Exception
```

可同时匹配多种类型，并导出异常对象实例。

```
def test():
    try:
        raise KeyError("key")
    except (IndexError, KeyError) as exc:       # 与 sys.exc_info 返回相同实例。
        print(repr(exc), exc.__traceback__)     # 输出异常及其跟踪对象信息。
```

```
>>> test()
KeyError('key') <traceback object at 0x102ed6248>
```

注意：Python 3 不再支持 Python 2 的某些语法。

被捕获的异常可原样重新抛出。比如说仅记录不处理，或者将异常作为广播消息继续向外传播等等。当然，也可抛出新异常对象。

```
def div(a, b):
    return a/b

def test():
    try:
        div(10, 0)
    except Exception as exc:
        print(f"[log] {exc}")              # 仅做日志，不做处理。
        raise                              # 原样抛出。

def main():
    try:
        test()
    except:
        _, val, tb = sys.exc_info()        # 重新抛出的异常，不会改变跟踪信息。
        print(f"[exc] {val}")
        traceback.print_tb(tb)
```

```
>>> main()
[log] division by zero
[exc] division by zero

  File "<python>", line 21, in main
    test()
  File "<python>", line 13, in test
    div(10, 0)
  File "<python>", line 8, in div
    return a/b
```

所导出的异常实例名字在语句块结束时，会被自动删除。这与普通名字作用域不同。

Python 2 不会删除，但可能会导致循环引用，垃圾回收延迟等负面效应。

```
def test():
    try:
        x = 1                              # 整个函数作用域。
        raise Exception("err")
    except Exception as exc:               # 异常变量 exc 仅在当前 except 内有效。
        print("except:", locals())
    finally:
        print("finally:", locals())
```

```
    print("function:", locals())
```

```
>>> test()
except   : {'exc': Exception('err',), 'x': 1}
finally  : {'x': 1}
function : {'x': 1}
```

因 finally 总被执行，所以常用来执行清理操作，确保相关资源被及时回收。

```
def test():
    try:
        raise Exception("error")
    finally:                              # 即便异常发生，也会先执行 finally 语句。
        print("dispose")

def main():
    try:
        test()
    except Exception as exc:              # 随后，才是外层拦截捕获。
        print(exc)
```

```
>>> test()
dispose
error
```

即便在 try 语句块内执行 return、break 等逻辑跳转指令， 解释器也确保 finally 被执行。

当然，如果 try 语句未被执行，那么 finally 也同样不会执行，这是一个整体。

```
def test(n):
    if not n: return

    try:
        print("try")
        return
    finally:
        print("finally")
```

```
>>> test(1)
try
```

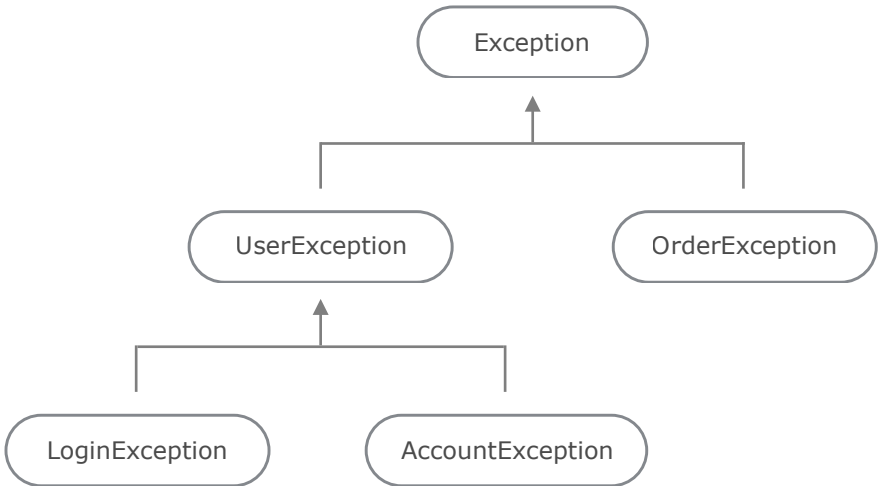
```
finally

>>> test(0)
```

自定义类型

鉴于 excpet 语句以类型作为匹配条件，那么我们应该针对性抛出具体异常类型，而非用字符串或错误码以做区别。

异常字符串或错误码通常作为日志，或便于阅读的附加内容。
在某些规范中，建议异常字符串不包含结束标点，以便嵌入到其他格式化文本中。

使用自定义异常类型，可以存储更多格式化信息用于诊断和修复。或者按逻辑分类构建异常类型树，以便通过不同基类以不同粒度捕获异常。同时，与逻辑扩展相对应，派生出更多详细类型。



自定义异常类型直接或间接继承自 Exception，通常以 Error 和 Exception 为名称后缀。

内置异常多以 Error 结尾，但建议以 Exception、Error 后缀区分可修复异常和不可修复错误。

日志和显示用信息依旧使用 Exception.args，以便通用模块在不了解异常对象结构的情况下，记录和输出可阅读信息。自定义属性可用于诊断和修复。尽管 Exception.args 也可存储多个数据，但会造成异常处理逻辑依赖参数顺序和索引序号，不如显式属性合理。

```
class UserException(Exception):
    def __init__(self, uid):
        self.uid = uid                                              # 用于用户修复逻辑。
        super().__init__(f"an exception about user {uid}", uid)     # 用于日志和显示。

def test():
    try:
        raise UserException(1001)
    except UserException as exc:
        logging.error(exc.args)                                     # 日志记录。
        print(exc.uid)                                              # 修复或重新抛出。
```

```
>>> test()
ERROR: ('an exception about user 1001', 1001)
```

异常链

使用 raise 抛出异常时，还可用 from 子句接收另一异常，形成链式结构。但这不意味着会引发多次异常处理，每个线程仅有最后一个异常可被捕获。其最大用途在于，在保留原异常状态的情况下，包装并抛出新的异常。

```
def test():
    try:
        raise Exception("err")
    except Exception as exc:
        raise Exception("wrap") from exc            # 使用链保留原异常状态。

def main():
    try:
        test()
    except Exception as exc:
        while True:                                 # 循环访问链上所有异常。
            print(repr(exc), exc.__traceback__)
            if not exc.__cause__: break
            exc = exc.__cause__
```

```
>>> main()
Exception('wrap',) <traceback object at 0x102ec9e48>
Exception('err',) <traceback object at 0x102ec9b48>
```

反过来，我们也可主动打断异常链构造。

```
def test():
    try:
        try:
            raise Exception("bbb") from Exception("aaa")
        except Exception as exc:
            raise exc from None                             # 打断异常链。
    except Exception as exc:
        print(repr(exc))
        print(repr(exc.__cause__))
```

```
>>> test()
Exception('bbb',)
None
```

实际上，即便不构造异常链，解释器也会以 __context__ 保存前一异常对象。只是不能像 from 语句那样，可主动构造和选择目标。

```
def test():
    try:
        try:
            raise Exception("err")
        except Exception as exc:
            raise Exception("wrap")          # 抛出新异常时，自动使用 __context__ 保存前异常对象。
    except Exception as exc:
        print(repr(exc.__context__))
```

```
>>> test()
Exception('err',)
```

如果异常未被捕获，解释器会输出 __cause__、__context__ 里所有异常信息。

使用模式

除用于错误和非正常状态处理外，异常还可用来控制执行流程。比如，从多重嵌套循环中跳出，或终止整个调用堆栈。

```
def test():
    class LabelException(Exception): pass

    try:
        while True:
            while True:
                while True:
                    print("loop")
                    raise LabelException
    except LabelException:
        pass

    print("exit")
```

```
>>> test()
loop
exit
```

不同于 return 只能终止当前函数，异常会沿调用堆栈向外传递。只需在合适的地点拦截，即可终止多级函数调用过程。这也是不建议拦截全部异常的理由之一。

```
class CancelException(Exception): pass

def db():
    raise CancelException

def logic():
    db()
    print("logic end")                # 不会执行。

def request():
    logic()
    print("request end")              # 不会执行。

def main():
    try:
        request()
    except CancelException:
        print("request cancel")
```

```
>>> main()
request cancel
```

还可用来插入调试干预代码。

```
def div(x, y):
    z = x / y
    return z

def test():
    try:
        x, y = 10, 0
        div(x, y)
    except Exception:                          # 如果出现未被处理异常，则启动调试器。
        if __debug__: pdb.post_mortem()
```

```
>>> test()
(PDB) w
  <python>(8) test()
-> div(x, y)
> <python>(2) div()
-> z = x / y

(PDB) locals()
{'y': 0, 'x': 10}
```

除此之外，我们还可向生成器对象发送异常，作为信号干预其执行流程。相关内容请参考前文章节，此处不再赘述。

警告提示

标准库另有一个 warnings 模块，用于输出警告信息。比如，提示开发人员当前算法已过时，或当前为测试版等。默认输出不会干扰程序执行，但可配置忽略，或转换为异常。

test.py

```
import warnings
class BetaWarning(FutureWarning): pass

def test():
    warnings.warn("This BETA version will expire on 2018-01-01", BetaWarning)
    print("hello, world!")

if __name__ == '__main__':
    test()
```

```
$ python test.py                          # 默认输出警告信息到 stderr。

BetaWarning: This BETA version will expire on 2018-01-01
hello, world!
```

```
$ python -Wi test.py                      # 忽略警告信息。
hello, world!
```

```
$ python -We test.py                      # 将警告信息转换为异常。

Traceback (most recent call last):
  File "test.py", line 15, in <module>
    test()
  File "test.py", line 11, in test
    warnings.warn("this BETA version will expire on 2018-01-01", BetaWarning)
```

虽然 Warning 继承自 Exception，但默认只输出相关信息，并不会做为异常抛出。只有开启 -We 参数设置时，才会被引发，从而被拦截捕获。

内置有 DeprecationWarning，无需自行定义。不过默认会被忽略，需 -Wd 参数开启显示。

---

2. 断言

我们通常会在代码中插入一些防御性（defensive）代码，用来检查数据（参数）是否是指定类型，是否在允许的取值范围内，以避免逻辑发生错误。这对以名字引用，没有类型限制的 Python 似乎更为重要。

只是，在经过严格测试和试运行，确保数据和调用安全后，这些防御性代码基本不再有正面作用，反而会拖累执行性能。如此，我们需要将其从源码中删除？当然不能！因为代码会随产品持续变更，进入下一个开发和测试迭代周期。最好的方式是通过条件编译，让编译器能自动忽略这些代码。

防御性编程涉及诸多内容，但其根本目标是保护代码本身，而非用户逻辑。所以不适合完成身份验证、权限检查之类的工作。

> 验证用户输入属于业务逻辑，这与面向代码，检查调用方是否按协议传参的防御编程不同。即便是公共类库，也应慎重考虑是否有必要对每次调用参数进行检查，是否要为了万分之一的错误而拖累海量正确。或许，严格的测试覆盖，外加详细的文档和示例会更友好。当然，敏感服务的安全检查另论。

断言

多数语言都会提供断言（assert）机制，用于检查所定义第一条件表达式是否成立。如果失败，则输出或执行第二条件。

```
assert expression1, expression2
```

其实际行为如下：

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

通常为每个检查条件单独提供断言语句。既便于阅读，也有利于调整检查条件，输出针对性提示信息。另外，第二表达式作为异常构造参数，可以是简单字符串内容，或者其他执行表达式。

```
def int_div(a, b):
    assert isinstance(a, int), "must be an integer"
    assert isinstance(b, int) and bool(b), "must be a non-zero integer"

    return a/b
```

```
>>> int_div(1, 0)
AssertionError: must be a non-zero integer
```

除检查参数外，有时也用来检查上下文状态。比如，检查数据库连接是否已经建立，以确定执行环境符合预期。或替代 print 用作临时测试，即便忘了删除也不会影响执行。

开启优化模式，编译器会直接忽略断言语句，甚至不会为其生成字节码。这远比在运行期进行判断好得多。这有点类似条件编译机制，仅针对调试版本运行，检查并约束调用行为和数据格式。

```
>>> dis.dis(compile("assert True", "", "exec", optimize = 0))
  1           0 LOAD_CONST               0 (True)
              2 POP_JUMP_IF_TRUE         8
              4 LOAD_GLOBAL              0 (AssertionError)
              6 RAISE_VARARGS            1
        >>    8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

```
>>> dis.dis(compile("assert True", "", "exec", optimize = 1))
  1           0 LOAD_CONST               0 (None)
              2 RETURN_VALUE
```

在命令行下以 -O 或 -OO 参数启动解释器优化模式。

当然，也正因为可被忽略，所以不适合用作固定检查条件，或替代 NotImplementedError 标记那些尚未实现的算法。从这点上说，断言和异常用途各异。归根结底，断言面向开发阶段的代码，以及开发人员，应避免作为运行期逻辑组成部分。

调试

不仅是断言，实际上任何 __debug__ 逻辑都可能被编译器忽略。

默认 __debug__ = True，开启优化模式后为 False。无法通过赋值修改。

```
code = """
if __debug__:
    print("debug")

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

```
>>> dis.dis(compile(code, "", "exec"))
  3           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('debug')
              4 CALL_FUNCTION            1
              6 POP_TOP

  5           8 LOAD_NAME                0 (print)
             10 LOAD_CONST               1 ('hello, world')
             12 CALL_FUNCTION            1
             14 POP_TOP
             16 LOAD_CONST               2 (None)
             18 RETURN_VALUE
```

```
>>> dis.dis(compile(code, "", "exec", optimize = 1))
  5           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('hello, world')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

利用这种特性，我们可以编写比断言更复杂的调试版代码。但请注意，无论编译器忽略与否，它都可能会造成附带影响。

test.py

```
def test():
    if __debug__:
        print("debug")
        global x                    # 即便被忽略，依然让代码分析器认为 x 是全局变量。
```

```
    else:
        print("release")

    x = 100

test()
print(x)

import dis
dis.dis(test)
```

```
$ python test.py                                        # 调试版。

debug
100

  6           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('debug')
              4 CALL_FUNCTION            1
              6 POP_TOP

 11           8 LOAD_CONST               2 (100)
             10 STORE_GLOBAL             1 (x)             # 全局变量。
             12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

```
$ python -O test.py                                     # 正式版。

release
100

  9           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('release')
              4 CALL_FUNCTION            1
              6 POP_TOP

 11           8 LOAD_CONST               2 (100)
             10 STORE_GLOBAL             1 (x)            # 全局变量。
             12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

3. 上下文

上下文管理协议（Context Management Protocol）是对异常处理结构的一种包装，相比分散的处理逻辑，它显得更加优雅，也更利于重用。

```
with expression [as var]:
    suite
```

表达式所创建对象必须包含协议方法 __enter__ 和 __exit__，具体执行流程如下：

伪码

```
o = expression()                          # 1. 创建上下文对象。
x = o.__enter__()                         # 2. 执行 __enter__，将返回值导出给本地变量 x。
                                          #    如果发生异常，则 suite、__exit__ 不会执行。
try:
    suite                                 # 3. 执行用户代码。
except:
    typ, val, tb = sys.exc_info()         # 4. 拦截异常，传递给 __exit__。
finally:
    if not o.__exit__(typ, val, tb):      # 5. 确保 __exit__ 总被执行。
        raise val                         #    如果 __exit__ 返回 False，则重新抛出异常。
```

设想一下，编写一个数据库上下文对象，需要注意什么？

首先，需在用户逻辑执行前初始化并建立连接；其次，在执行中提供相关操作接口；最后，关闭连接并清理现场。这其中，如何建立连接，如何处理连接池，包括如何清理现场等等都与用户逻辑无关。同样，当用户逻辑发生异常时，上下文对象自身如何应对也与用户逻辑无关。

很显然，我们需要将上下文与用户逻辑解耦，让其各自为政。不但用户逻辑无需关心上下文状态，上下文对象也可不理会用户逻辑是否会抛出异常，是否会捕获异常，确保总能适时结束上下文状态即可。

这样一来，上下文状态开启和结束都由其自身负责。无论当前环境有多少上下文对象，都无需担心。既简化用户逻辑，又利于在不同场合重用。

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

```
    def __enter__(self):
        print(f"__enter__: {self.name}.open")
        return self

    def __exit__(self, typ, val, tb):
        print(f"__exit__: {self.name}.close; exception: {val}")
        return False


def logic():
    with DB("mysql") as db:
        print(f"exec in {db.name}")
        raise Exception("logic error")
```

```
>>> logic()

__enter__: mysql.open
  exec in mysql
__exit__: mysql.close; exception: logic error
--------------------------------------------------------------------------
Exception: logic error
```

可同时使用多个上下文对象。

```
def logic():
    with DB("mysql") as mq, DB("postgres") as pg:
        print(f"exec in {mq.name}")
        print(f"exec in {pg.name}")
        raise Exception("logic error")
```

```
>>> logic()
__enter__: mysql.open
__enter__: postgres.open
  exec in mysql
  exec in postgres
__exit__: postgres.close; exception: logic error
__exit__: mysql.close;    exception: logic error
--------------------------------------------------------------------------
Exception: logic error
```

如果某个上下文对象 __exit__ 方法返回 True，那么表示它处理了该异常，后续上下文对象将接收不到该异常信息。请注意多个__exit__ 按 FILO 顺序执行。

当然，上下文对象未必就要参与用户逻辑。可仅仅用来测量执行性能，记录日志后重新抛出，启动调试器，甚至压制异常等等。

标准库里很多对象都实现了上下文协议，应优先使用此模式。

```
with open("test.py", "r") as f:           # 确保文件总被关闭。
    print(f.read())
```

```
lock = threading.Lock()

for i in range(10):
    with lock:                            # 确保锁总被释放，避免死锁。
        print(i)
```

contextlib

上下文对象需要创建类型，并实现协议方法。但对于已有代码如何改造？这点可以借鉴标准库 contextmanager 的做法。

先准备一个示例，其中混杂上下文和用户逻辑代码。

```
def test(db):
    print(f"open: {db}")                 # 初始化。
    try:
        print(f"exec: {db}")             # 逻辑。
    finally:
        print(f"close: {db}")            # 关闭。
```

```
>>> test("mysql")
open: mysql
exec: mysql
close: mysql
```

我们要对其重构，将上下文剥离出来，以便分离职责并实现重用。

```
@contextlib.contextmanager
```

```
def context(db):
    try:
        print(f"open: {db}")              # 初始化。
        yield db
    finally:
        print(f"close: {db}")             # 关闭。


def test(db):
    with context(db):
        print(f"exec: {db}")              # 逻辑。
        raise Exception("error")          # 模拟用户逻辑异常。
```

```
>>> test("mysql")
open: mysql
exec: mysql
close: mysql
--------------------------------
Exception: error
```

从结果看，我们成功分离了上下文和逻辑代码，实现最初设想。而且仅仅是拆分函数，并未创建上下文类型。显然起作用的是 contextmanager，且看看精简后的示意代码。

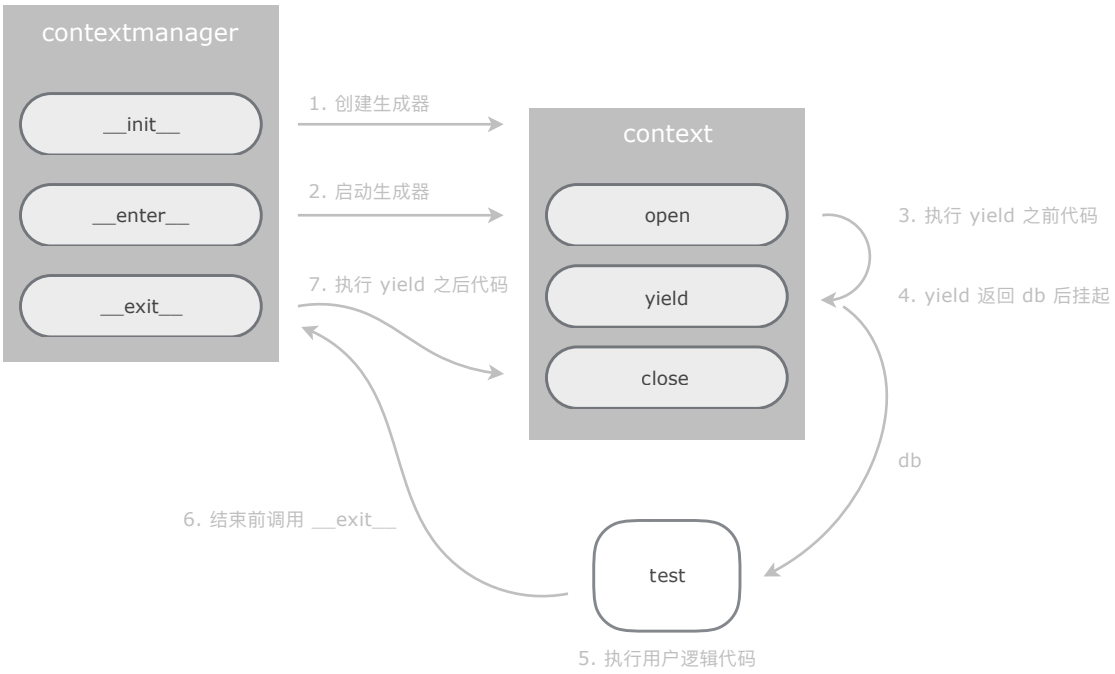
contextmanager

```
class GeneratorContextManager:

    def __init__(self, func):
        self.gen = func()                     # 执行原函数（context），返回生成器对象。

    def __enter__(self):
        return next(self.gen)                 # 启动生成器，返回 yield 结果作为导出变量。
                                              # 实际上就是执行 yield 之前的代码。
    def __exit__(self, typ, val, tb):
        if typ is None:
            next(self.gen)                    # 继续执行 yield 后面的代码。
        else:
            self.gen.throw(typ, val, tb)      # 如果有异常发生，则将异常抛回给生成器。
```

整体设计利用了生成器分步执行能力。函数 context 用 yield 完成了三个目标：首先，yield 之前代码作为 __enter__ 内容；其次，yield 返回上下文导出变量；最后，yield 后面代码作为 __exit__ 内容。

装饰器则创建通用上下文对象，在相关方法中切换执行生成器，完成三个步骤。

在用户函数中，with 语句首先调用 __enter__ 方法完成了第一、二两个步骤，此时生成器因 yield 返回导出变量被挂起，所以改为执行用户逻辑代码。直到后面 __exit__ 被调用，生成器得以重新激活，完成第三步骤。



该模块中，还基于 contextmanager 实现了 closing、suppress 等通用上下文函数。

九. 元编程

元编程将程序当做数据，或在运行期完成编译期工作。

> Write code that writes code. —— MetaProgramming

1. 装饰器

对于下面这样一段代码，如果要介入其调用过程，该如何做?

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

def test():
    print(add(1, 2))
```

利用装饰器（decorator），可在不侵入内部实现，甚至不知情的情况下，插入扩展逻辑。

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

因为编译器会将装饰器语法处理成如下模式。

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

add = log(add)
```

对按名字搜索执行的调用函数来说，根本不会察觉被偷天换日。而被调用函数，只要能接收到正确的参数即可。如此，我们可在任何时候，添加额外处理过程。从装饰器语法和实现效果看，非常类似 AOP 编程范式。

举例来说，日志、缓存、代理等都不是逻辑必须的功能，按职责分离原则被封装到不同类型，但如何在没有耦合依赖的情况下组装？或者在不修改逻辑和调用的情况下进行调整？除此之外，类似权限检查、事务管理等是否可从逻辑中剥离，以按需配置形式动态绑定。毕竟，单元测试并不希望有复杂的链式过程。

AOP（aspect-oriented programming），面向切面编程。就是在不修改目标源码的前提下，添加功能的技术手段或设计模式，是对 OOP 的补充。其效果就像为相机镜头添加滤镜。

面向对象虽然可通过接口解除对直接类型的依赖，但接口调用依然会出现在代码实现里。这本质上也是一种耦合，功能耦合。试想，在信用卡模块里出现日志调用怎么说都是突兀，且对单元测试不友好的行为。面向切面不同，它介入的是执行而非实现环节。通过技术手段，拦截调用，提取数据，或者改变执行流程，以低耦合的阻隔方式实现功能变更。从这点上说，它是基于组装和按需配置的，每个都像一片滤镜插到逻辑和调用之间。

1.1 实现

基于装饰器的实际执行方式，可直接以函数实现。

```
def log(fn):
    def wrap(*args, **kwargs):                # 通过包装函数间接调用原函数。
        print(f"log: {args}, {kwargs}")
        return fn(*args, **kwargs)

    return wrap                               # 返回包装函数替代原函数与名字关联。

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

```
>>> add(1, 2)
log: (1, 2), {}
3
```

返回包装代理，或仅添加一些额外属性后，原样返回。

```
def log(fn):
    fn.log_func = lambda *args, **kwargs: print(f"log: {args}, {kwargs}")
    return fn
```

同理，任何可调用对象（callable）都可用来实现装饰器模式。

```
class log:
    def __init__(self, fn):
        self.fn = fn

    def __call__(self, *args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return self.fn(*args, **kwargs)

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

因每条“@log”都会被处理成“log(add)”，也就是每次都会新建一个实例。所以，应用于多个目标函数完全没问题。

相比函数，使用类实现似乎可以构建更复杂的装饰器，但存在一些麻烦。类实现的装饰器应用于实例方法时，会导致方法绑定丢失。

```
class log:
    def __init__(self, fn):
        self.fn = fn

    def __call__(self, *args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return self.fn(*args, **kwargs)

class X:
    @log
    def test(self): pass
```

```
>>> x = X()
>>> x.test                                  # 方法被装饰器实例替代。
<log at 0x103a05160>

>>> x.test()
TypeError: test() missing 1 required positional argument: 'self'
```

因为装饰器实例替换了方法，结果导致实现绑定的描述符方法被隐藏，无法自动调用。要么让装饰器也实现描述符协议，要么显式传递参数。反不如用函数来得简单，因为函数对象默认实现了描述符协议和绑定规则。至于状态维持，函数一样可以添加属性。

```
def log(fn):
    def wrap(*args, **kwargs):
        print(f"log: {args}, {kwargs}")
        return fn(*args, **kwargs)

    return wrap

class X:
    @log
    def test(self): pass
```

```
>>> x = X()
>>> x.test
<bound method log.<locals>.wrap of <X object>>
```

```
>>> log.__get__                                    # 函数默认实现了描述符协议。
<method-wrapper '__get__' of function object>
```

嵌套

可对同一目标使用多个装饰器。

```
@a
@b
def test(): pass
```

最终效果是多次嵌套调用。

```
test = a(b(test))
```

装饰器接收前一装饰器返回值，包装对象或原函数等。如此，就需注意排列顺序，因为每个装饰器返回值并不相同。比如，我们须确保类型方法装饰器是最外面的一个，因为无法确定内层装饰器如何实现。可能也会因描述符问题，导致方法绑定失效。

```
class X:
```

```
    @classmethod                                    # 注意放在最外层。
    @log
    def test(cls): pass
```

参数

除被装饰目标外，还可向装饰器传递其他参数，以实现更多定制特性。

```
@log("demo")
def test(): pass
```

含参数的装饰器，相当于在原装饰器外面套一个专用来接收参数的外壳。

```
decorator = log("demo")
test = decorator(test)
```

这样，就多一次处理过程。

```
def log(name = "default"):                      # 外层函数接收参数。
    print(f"args: {name}")

    def decorator(fn):                          # 装饰器。
        print(f"decorator: {fn}")
        return fn                               # 返回包装函数或原函数。

    return decorator
```

```
>>> @log("demo")
    def test(): pass

args: demo
decorator: <function test>
```

```
>>> @log()                                      # 对于有参数的装饰器，即便是默认值也需要括号。
    def test(): pass
```

```
args: default
decorator: <function test>
```

需要注意，@log 和 @log() 并不相同。后者依然表示含参外壳函数调用，无非实参数量为零。

属性

我们应该让包装函数更像原函数一些，比如拥有某些相同的属性。

```
def log(fn):

    @functools.wraps(fn)                              # 将 fn 相关属性复制到 wrap。
    def wrap(*args, **kwargs):
        return fn(*args, **kwargs)

    print(f"wrap: {id(wrap)}, func: {id(fn)}")
    return wrap


@log
def add(x: int, y: int) -> int:
    return x + y
```

```
>>> add.__name__
add

>>> add.__annotations__
{'return': int, 'x': int, 'y': int}
```

装饰器 functools.wrap 从原函数复制 __module__、__name__、__doc__、__annotations__ 属性，添加到包装函数。另外，还会用 __wrapped__ 存储原始函数或上一装饰器返回值。可据此判断并绕开装饰器对单元测试的干扰。

```
wrap: 4520685640, func: 4520686456

>>> id(add), id(add.__wrapped__)
(4520685640, 4520686456)
```

类型装饰器

装饰器可同样用于类型，区别无非是接收参数不同。

```
def log(cls):

    class wrapper:
        def __init__(self, *args, **kwargs):
            self.__dict__["inst"] = cls(*args, **kwargs)

        def __getattr__(self, name):
            value = getattr(self.inst, name)
            print(f"get: {name} = {value}")
            return value

        def __setattr__(self, name, value):
            print(f"set: {name} = {value}")
            return setattr(self.inst, name, value)

    return wrapper


@log
class X: pass
```

```
>>> x = X()

>>> x.a = 1
set: a = 1

>>> x.a
get: a = 1
```

因为每次都新建 wrapper 类型，所以可应用于不同类型目标。

当然，未必就要返回包装类。也可用函数代替，间接调用目标构造方法创建实例。

```
def log(cls):
    def wrap(*args, **kwargs):
        o = cls(*args, **kwargs)
        print(f"log: {o}")
        return o

    return wrap
```

```
@log
class X: pass
```

```
>>> X()
log: <__main__.X object at 0x10d7dd208>
     <__main__.X at 0x10d7dd208>
```

1.2 应用

利用装饰器功能，我们可以编写各种辅助开发工具，完成诸如调用跟踪、性能测试、内存检测等任务。当然，更多时候用于模式设计，改善代码结构。

调用跟踪

记录目标调用参数、返回值，以及执行次数和执行时间等信息。

```
def call_count(fn):                             # 调用计数器。

    def counter(*args, **kwargs):
        counter.__count__ += 1
        return fn(*args, **kwargs)

    counter.__count__ = 0
    return counter


@call_count
def a(): pass

@call_count
def b(): pass
```

```
>>> a(); a(); a.__count__
2

>>> b(); b(); b(); b.__count__
3
```

在标准库中有类似应用，通过缓存结果减少目标执行次数。

```
>>> @functools.lru_cache(10)
    def test(x):
        time.sleep(x)

>>> %%time
    for i in range(1000): test(1)

CPU times: user 668 µs, sys: 1.44 ms, total: 2.1 ms
Wall time: 1 s
```

包装 cProfile，像 line_profiler、memory_profiler 那样输出性能测试结果。

属性管理

为目标添加额外属性，在原有设计上以装配方式混入（mixin）其他功能组。

```
def pet(cls):
    cls.dosomething = lambda self: None
    return cls

@pet                                            # 添加宠物功能。
class Parrot: pass
```

比起前面章节提及的直接添加基类（__bases__）方式，装饰器显然更优雅一点。
还可用代理类拦截（__getattr__、__setattr__）对目标访问，比如实现只读功能。

实例管理

替代目标构造方法，拦截实例创建。可用于实现对象缓存，或单例模式。

```
def singleton(cls):
    inst = None

    def wrap(*args, **kwargs):
        nonlocal inst
```

```
        if not inst: inst = cls(*args, **kwargs)
        return inst

    return wrap


@singleton
class X: pass
```

```
>>> X() is X()
True
```

部件注册

在很多 Web 框架里，用装饰器替代配置文件实现路由注册。

```
class App:
    def __init__(self):
        self.routers = {}

    def route(self, url):                       # 实现为带参数的装饰器，用于注册路由配置。
        def register(fn):
            self.routers[url] = fn
            return fn

        return register
```

```
app = App()

@app.route("/")
def index(): pass

@app.route("/help")
def help(): pass
```

```
>>> app.routers
{'/': <function index>, '/help': <function help>}
```

2. 描述符

描述符（descriptor）一直被解释器当做秘密武器使用。前文提及的属性（property）、绑定方法等内部机制都是描述符在起作用。

不同于实例通用拦截方法（__getattr__），描述符以单个属性出现，并针对该属性的不同访问行为作出响应。最重要的是，描述符能“感知”通过什么引用该属性，从而和目标建立绑定关联。

下面是一个完整描述符实现。

```
class descriptor:

    def __set_name__(self, owner, name):
        print(f"name: {owner.__name__}.{name}")
        self.name = f"__{name}__"

    def __get__(self, instance, owner):
        print(f"get: {instance}, {owner}")
        return getattr(instance, self.name, None)

    def __set__(self, instance, value):
        print(f"set: {instance}, {value}")
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        print(f"del: {instance}")
        raise AttributeError("delete is disabled")
```

描述符属性必须定义为类型成员，所以其自身不适合存储实例相关状态。在创建属性时，__set_name__ 方法被调用，可通过参数获知目标类型（owner），以及属性名称。

```
>>> class X:
        data = descriptor()

name: X.data
```

以类型或实例访问描述符属性时，__get__ 方法被调用，且会接收到类型和实例引用。

> 如此，可针对目标进行操作。比如，创建存储字段。

```
>>> x = X()

>>> x.data = 100
set: <X object at 0x1063c6fd0>, 100

>>> x.data
get: <X object at 0x1063c6fd0>, <class 'X'>
100
```

方法 __set__、__delete__ 仅在实例引用时被调用。以类型引用进行赋值或删除操作，会导致描述符属性被替换或删除。

```
>>> X.data
get: None, <class 'X'>

>>> X.data = 100                    # 以类型引用赋值，导致描述符属性被替换。
>>> X.data
100
```

还有，将描述符属性赋值给变量或传参时，实际结果是 __get__ 方法的返回值。

```
>>> x = X()

>>> x.data = 100
set: <X object at 0x10627a898>, 100

>>> o = x.data
get: <X object at 0x10627a898>, <class 'X'>

>>> o
100
```

数据描述符

如果定义了 __set__ 或 __delete__ 方法，那么我们便称其为数据描述符（data descriptor）。而仅有 __get__ 的则是非数据描述符（non-data descriptor）。两者区别在于，数据描述符属性优先级高于实例名字空间中的同名成员。

```
class descriptor:
```

```
    def __get__(self, instance, owner):
        print("__get__")

    def __set__(self, instance, value):
        print("__set__")


class X:
     data = descriptor()
```

```
>>> x = X()
>>> x.__dict__["data"] = 0

>>> x.data = 100                          # 即便实例名字空间有同名成员，数据描述符依然优先。
__set__

>>> x.data
__get__
```

如果注释掉 __set__ ，使其成为非数据描述符。再执行时，结果就不一样了。

```
>>> x = X()

>>> x.data = 10                           # 同名实例成员优先级高于非数据描述符。
>>> x.data
10

>>> vars(x)
{"data": 10}
```

属性（property）就是数据描述符。就算没有提供 setter 方法，但 __set__ 依然存在，所以其优先级总是高于同名实例成员。

```
>>> p = property()

>>> p.__get__
<method-wrapper '__get__' of property object>

>>> p.__set__
<method-wrapper '__set__' of property object>

>>> p.__delete__
<method-wrapper '__delete__' of property object>
```

方法绑定

因为函数默认实现了描述符协议，当以实例或类型访问时，__get__ 首先被调用。

类型和实例作为参数被传入 __get__，从而截获绑定目标（__self__），如此就就将函数包装成绑定方法对象返回。实际被执行的，就是这个会隐式传入第一参数的包装品。

```
class X:
    def test(self, o): print(o)
```

```
>>> x = X()

>>> x.test
<bound method X.test of <X object at 0x1063e9ac8>>
```

也就是说方法执行实际分成两个步骤：

```
    x.test(123)  --->

            m = x.test.__get__(x, type(x))          # 将函数包装为绑定方法。
            m(123)                                  # 执行时，隐式将 self/cls 参数传给目标函数。
                    --->  X.test(m.__self__, 123)
```

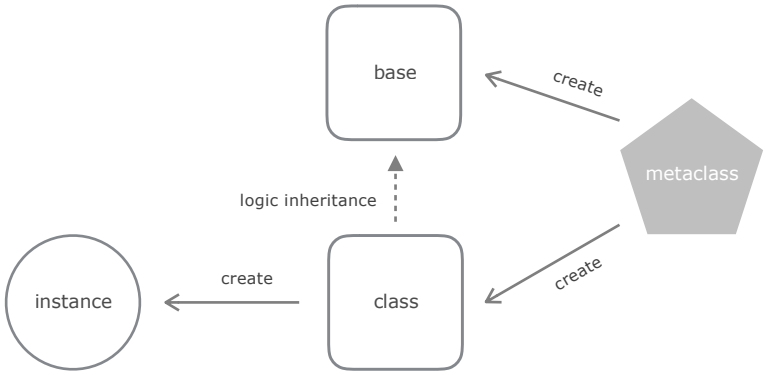
在绑定方法对象内，__self__ 和 __func__ 存储了执行所需信息。

```
>>> m = x.test.__get__(x, X)

>>> m
<bound method X.test of <X object at 0x1063e9ac8>>

>>> m.__self__, m.__func__
(<X at 0x1063e9ac8>, <function X.test>)
```

3. 元类

我们已经知道，类型是实例的创建工厂。那么，又是谁创造了类型对象呢？是父类？

确切说，元类（metaclass）制造了所有的类型对象，并将其与逻辑上的父类关联起来。所以，这里存在两条线：创建和逻辑。



系统默认元类是 type，所以我们可以描绘出如下过程。

```
base = type("base", object, ...)
class = type("class", base, ...)
instance = class(...)
```

```
instance.__class__ is class   and   isinstance(instance, class)
class.__class__ is type       and   isinstance(class, type)
base.__class__ is type        and   isinstance(base, type)
```

属性 __class__ 表明该对象由何种类型创建，可用 type(o) 返回。

实际上，用 type 直接创建类型对象也无不妥。

```
>>> User = type("User", (object,), {
    "__init__" : lambda self, name: setattr(self, "name", name),
    "test"     : lambda self: print(self.name),
    "table"    : "user",
})

>>> User.__dict__
```

```
mappingproxy({'__init__': <function <lambda>>,
              'table': 'user',
              'test': <function <lambda>>,
              ... })
```

```
>>> u = User("yuhen")
>>> u.test()
yuhen

>>> u.__dict__
{'name': 'yuhen'}
```

3.1 自定义

可自定义元类，以控制类型对象的生成过程。通常自 type 继承，以 Meta 为后缀名。

```
class DemoMeta(type): pass

class X(metaclass = DemoMeta): pass            # 使用自定义元类。
```

```
>>> X.__class__
DemoMeta
```

Python 3 不再支持 Python 2 __metaclass__ 语法。

与普通类型构造过程类似，可覆盖构造和初始化方法定制创建过程。

```
class DemoMeta(type):

    @classmethod
    def __prepare__(cls, name, bases):
        print(f"__prepare__: {name}")
        return {"__make__": "make in DemoMeta"}            # 定制名字空间。

    def __new__(cls, name, bases, attrs):
        print(f"__new__: {name}, {bases}, {attrs}")
        return type.__new__(cls, name, bases, attrs)       # 创建并返回类型对象。

    def __init__(self, name, bases, attrs):
        print(f"__init__: {self}")
```

```
        return type.__init__(self, name, bases, attrs)     # 初始化后，返回类型对象。

    def __call__(cls, *args, **kwargs):
        print(f"__call__: {cls}, {args}, {kwargs}")
        return type.__call__(cls, *args, **kwargs)         # 调用类型对象创建实例过程，返回实例。
```

按需去实现相关方法。

__prepare__ 用来创建类型对象名字空间（X.__dict__），可往里添加点什么或使用自定义字典类型。随后，该名字空间会填充其他属性成员，继续传给 __new__ 和 __init__ 方法。

```
    class X(metaclass = DemoMeta)   --->

            namespace = DemoMeta.__prepare__(name, ...)
                    X = DemoMeta.__new__(name, bases, namespace{attrs})
                        DemoMeta.__init__(X, ...)
```

__call__ 在类型对象（X）创建其所属实例时被调用。实际上 X.__new__、X.__init__ 方法就是由此调用，可用于拦截实例创建。

```
    o = X(1, 2)   --->

            o = DemoMeta.__call__(X, 1, 2)   --->

                    o = X.__new__(...)
                        X.__init__(o, ...)
```

```
>>> class X(metaclass = DemoMeta):
        data = 100
        def __init__(self, x, y): pass
        def test(self): pass


__prepare__ : X

__new__     : X, (), {'__make__': 'make in DemoMeta',
                      '__module__': '__main__',
                      '__qualname__': 'X',
                      '__init__': <function X.__init__ at 0x10f815620>,
                      'data': 100,
                      'test': <function X.test at 0x10f815730>}

__init__    : <class '__main__.X'>
```

```
>>> o = X(1, 2)
__call__ : <class '__main__.X'>, (1, 2), {}
```

```
>>> o
<__main__.X at 0x10f840a90>
```

当然，用函数或其他可调用对象（callable）代替也未尝不可。

```
def demo_meta(name, bases, attrs):
    print(f"{name}, {bases}, {attrs}")
    return type(name, bases, attrs)
```

```
>>> class X(metaclass = demo_meta):
    data = 100
    def __init__(self, x, y): pass
    def test(self): pass


X, (), {'__module__': '__main__',
        '__qualname__': 'X',
        '__init__': <function X.__init__ at 0x10f8682f0>,
        'data': 100,
        'test': <function X.test at 0x10f868268>}
```

> 函数只是拦截调用，类型对象创建依然使用 type 完成。但多数时候，这就足够。

参数

还可向元类传递参数，实现功能定制。

```
class DemoMeta(type):

    def __new__(meta, name, bases, attrs, **kwargs):
        print(kwargs)
        return type.__new__(meta, name, bases, attrs)
```

```
>>> class X(metaclass = DemoMeta, a = 1, b = "abc"):
        def test(self): pass

{'a': 1, 'b': 'abc'}
```

继承

类型对象的元类设置顺序：

1. 用 metaclass 显式指定。
2. 从基类继承。
3. 默认元类 type。

```
class DemoMeta(type): pass

class X(metaclass = DemoMeta): pass
class Y(X): pass                            # 从基类继承元类。
```

```
>>> type(Y)
__main__.DemoMeta
```

如果是多继承，必须保证能继承所有祖先元类。

```
class AMeta(type): pass
class BMeta(type): pass

class A(metaclass = AMeta): pass
class B(metaclass = BMeta): pass
```

```
>>> class C(A, B): pass

TypeError: metaclass conflict:
     the metaclass of a derived class must be a (non-strict) subclass of
     the metaclasses of all its bases
```

因 __class__ 只能指向一个元类，除非祖先元类之间存在继承关系，否则必然因无法访问而导致错误发生。为此，我们需要新建一个元类，让其继承所有祖先元类。

```
class CMeta(AMeta, BMeta): pass
class C(A, B, metaclass = CMeta): pass
```

3.2 应用

基于元类，我们可以实现很多魔法，让对象拥有很高的隐式“智能”，但这会大大提升代码复杂度。除非必要，否则不建议这么做。

另外，元类虽然能像普通类型那样为自己的实例提供共享成员，但依旧要避免。就让元类专注于类型创建和管理，不要掺合逻辑为好。

> Metaclasses are deeper magic that 99% of users should never worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
>
> —— Python Guru Tim Peters on metaclasses.

静态类

阻止类型创建实例对象。

> 直接用 __call__ 拦截实例创建就行了。

```
class StaticClassMeta(type):
    def __call__(cls, *args, **kwargs):
        raise RuntimeError("can't create object for static class")
```

```
>>> class X(metaclass = StaticClassMeta): pass
RuntimeError: can't create object for static class
```

密封类

阻止类型被继承。

> 将使用该元类，也就是不能被继承的类型添加到集合里，作为后续基类判断条件即可。

```
class SealedClassMeta(type):
    types = set()

    def __init__(cls, name, bases, attrs):
        if cls.types & set(bases): raise RuntimeError("can't inherit from sealed class")
        cls.types.add(cls)
```

```
>>> class A(metaclass = SealedClassMeta): pass

>>> class B(A): pass
RuntimeError: can't inherit from sealed class
```

4. 注解

注解（annotation）为函数参数、返回值，以及模块和类型属性添加额外元数据。

```
def add(x: int, y: int) -> int:
    return x + y
```

```
>>> add.__annotations__
{'return': int, 'x': int, 'y': int}
```

其本质上仅是一种可编程和可执行注释。在编译期被提取，并与对象相关联。在运行期，对解释器指令执行没有任何影响和约束。

```
>>> add("abc", "d")                                    # 并不受注解 int 类型约束。
'abcd'
```

注解内容可以是任何对象或表达式。可应用于变量，但不能用于 lambda 函数。

```
>>> x: int = 123                                       # 注意和等号后的初始化值分开。

>>> __annotations__
{'x': int}
```

```
>>> def test(x: {'type': int, 'range':(0, 10)} = 5):   # 等号后为参数默认值。
        pass


>>> test.__annotations__
{'x': {'range': (0, 10), 'type': int}}

>>> test.__defaults__
(5,)
```

```
>>> class X:
        data: int = 10
        def test(self, o: str) -> str: pass
```

```
>>> X.__annotations__
{'data': int}

>>> X.test.__annotations__
{'o': str, 'return': str}
```

用途

注解最常见用途是类型和取值范围检查，这在 ORM 框架里很常见。

```
def test(x: (int, 0, 10)):
    if __debug__:
        ann_x = test.__annotations__["x"]
        assert isinstance(x, ann_x[0])
        assert ann_x[1] <= x <= ann_x[2]

    return
```

> 更优雅的做法是使用装饰器完成，成为一个通用检查器。

还可为代码编辑器（PyCharm）和帮助生成工具（pydoc）提供更详细的分类信息。另有 Mypy 等实验型解释器，借助注解实现编译期静态类型检查。

十. 进阶

1. 解释器

2. 扩展

十一. 测试

1. 单元测试

2. 性能测试

十二. 工具

1. 调试器

pdb

2. 包管理

pip, wheel

3. 增强环境

ipython, notebook

4. 虚拟环境

virtualenv