2024 年 12 月 23 日

你不知道的 Python

The Python You Don't Know

Author: *Nobody*

大多数书籍会告诉我们 Python 如何简单，却少有书籍会告诉我们：**其实 Python 是一门易学难精的语言**。

Python 因其语法容易上手、动态类型、内存安全、生态丰富等诸多优点而圈粉无数。现如今跻身于程序开发圈的子里，似乎身边到处都是会 Python 的人。

Python 的基础语法跟其它编程语言比起来，确实可以说是“新手友好型”，但是其高级特性学起来也是相当抽象。笔者觉得，C 语言的语法比 Python 更简单，大多数没能学进去 C 语言的人，不是因为 C 语言本身难，而是因为 C 语言假设用户知道计算机是如何工作的；C 语言真正的难，在于它设施上的“一穷二白”，以及将计算机工作方式的复杂性直接暴露给了用户。然而 Python 并不假设用户知道计算机是如何工作的，所以小学生也可以在经过简单的学习后立刻上手；随着对 Python 的深入了解，大家会发现 Python 的闭包、匿名函数、自由变量、装饰器、元素编程等概念相当抽象。

大多数 Python 开发者只用到了 Python 最基础的功能，用 Python 的语法写另一种 C、C++，并没能有效发挥出 Python 的优势。Python 的语法提供了很多“语法糖”，如多值、匿名函数、解包、元编程等；Python 的标准库提供了相当多的常用功能，被大家称为“自带电池”，意思是说只用其标准库就可以做很多事情。善用 Python 的语法特性和标准库，能够有效帮助我们把代码写得更快更优雅。

自带电池：self operated

很多人说 Python 的性能不行，Python 的易用性确实是以性能为代价的，但现代的 Python 生态大多只是借用了 Python 语言作“外壳”，底层性能悠关的环节实际运行的是 C、C++、Fortran 等性能更好的语言编官写的库。语言本身的速度确实没法跟编译型语言媲美，但是也没有外界传的相差几百几千倍。笔者在实践中做过很多次将现有的 Python 算法移植到 C 语言，结果发现时间复杂度为 $O(n)$ 的算法，单核性能会得到大约 50–100 倍左右的提升。优化得好的 Python 代码，会在内部运行其 C 语言的内联实现[1]，性能直追 C++。当然，GIL[2]确实是一个较大的性能束缚，幸运的是，从 Python3.13 开始，已经能够以 no-gil 模式运行了，可以实现多核多线程。

[1] CPython 是这样的，其它实现要看情况了

[2] Global Interpreter Lock：全局解释器锁，会导致 Python 的多线程只能跑在单核上，无法发恢出多核 CPU 的优势。

Python 的定位就是鼓励大家善用现有的设施，好好做个调包侠，没有什么可耻的，毕竟——**人生苦短，我用 Python**。

同样是 Python3，编写本书时的 3.12 相比 3.5 已经产生了翻天覆地的变化，Python 本身也在跟着时代进步。本书中 Python 代码案例采用 Python3.12 进行测试，如有版本上的要求，会特别指出。

这并不是一本正而八经编写的的书籍，只是是笔者对工作中踩过的坑进行的记录。本书的内容假设您有一定的 Python 开发经验，如果您有一定的 C、C++、C#、Java 语言开发经验会更有助于理解。

笔者才疏学浅，如果您在阅读的过程中发现有错误的地方，感谢您的指正：ilieweili@tencent.com。

目录

第一部分

改善代码的诸多方法

语法糖

很多习惯于静态语言的程序员会在心中觉得，语法糖是奇技淫巧而已，无需当回事。但语法糖确实能够有效帮助人们避免给代码写得一团糟。

0.1 多值与解包

0.1.1 多值

多值

多值：multi-value

在传统的编程语言如 C 语言中，函数只能返回一个结果。比如一个数的平方根有正负两个解，但 C 语言中的函数只能返回一个结果。要解决这个问题，要不返回一个结构体；要不增加两个指针作为入参。

```c
typedef struct _Result {
    float pos, neg;
} Result;

Result MySqrt1(float num) {
    Result res;
    res.pos = sqrt(num);
    res.neg = -res.pos;
    return Result;
}

void MySqrt2(float num, float* pos, float* neg) {
    *pos = sqrt(num);
    *neg = -(*pos);
}
```

但 Python 的函数能返回任意数量的结果，函数能返回多个结果的这个功能就叫做 “多值”。

```
def mysqrt(num):
    pos = math.sqrt(num)
    neg = -pos
    return pos, neg
```

0.1.2 解包

解包：unpacking

多值函数的返回值是什么

要了解什么是解包，还要先了解一个多值函数的返回值到底是什么。返回多值的函数，其返回值可以在同一个赋值语句中被多个变量接收。

```
pos, neg = mysqrt(4)
```

但是如果其返回值只赋值到一个变量上，得到的是一个元组。

```
res = mysqrt(4) # (2, -2)
```

这并不是什么隐式转换，而是 Python 的表达式语法，逗号分隔的连续多个子表达式构成元组的表达式，括号不是必须的 (空元组除外)。所以返回多值的函数，本质上是返回了一个元组。我们平时写元组都加括号，主要还是为了避免和其它语法结构产生冲突，比如放在函数的形参列表中，可能会被认为是函数的多个参数。

官方文档中有提到：Note that tuples are not formed by the parentheses , but rather by use of the comma operator. The exception is the empty tuple, for which parentheses are required —allowing unparenthesized “nothing”in expressions would cause ambiguities and allow common typos to pass uncaught.

```
1, 1+1, 4-1 # (1, 2, 3)
```

什么是解包

可迭代对象：iterable object

Python 中的可迭代对象 [3] 包括字符串、列表、元组、集合、字典、生成器等。所有能用`for ... in`语法进行迭代的对象都是可迭代对象。

[3] Python 中的可迭代对象都是 collections.abc.Iterable 的子类或虚子类，它们都实现了`__iter__`方法，可以用`for ... in`语法进行迭代，但是未必都有长度，比如一个可无限迭代的生成器。拥有长度的对象都是 collections.abc.Sized 的子类或虚子类，可用 len() 函数获取长度。list 和 tuple 就同时是 Iterable 和 Sized 的虚子类。虚子类是指其符合基类的必要协议，但并不直接集成自基类，这种常见于 CPython 的 C 语言内置对象。

可迭代对象可以展开，在赋值语句中同时赋值给多个变量，或变成另一个可迭代对象表达式上下文中的多个元素，这就叫做解包。

```
a, b = 1, 2
c, d = (1, 2)
e, f = [1, 2]
g, h = "12"
tup = (3, 4)
lst = [1, 2, *tup] # [1, 2, 3, 4]
```

可迭代对象解包

可迭代对象在赋值语句中同时对多个变量赋值，只需在赋值号左边提供的变量数量和可迭代对象的长度一致即可，变量名之间用逗号隔开。

```
pos = [1, 2]
x, y = pos
for name, age in [("小明", 24), ("小强", 25)]:
    print(f"{name}'s age is {age}")
```

在 for … in 语法中，迭代变量就是负责接收每次迭代时返回的结果的变量。可以理解为，每次迭代返回的值都被赋值给了迭代变量。如果每次迭代返回的值也是可迭代对象，那么当提供了多个迭代变量，也就被自然而然地解包给了每一个。

还可以利用解包的功能，轻松实现交换两个变量。其本质原理就是构造了一个新的元组，然后再按顺序解包给变量。

```
a, b = 1, 2
b, a = a, b
```

其实赋值号左边提供的变量数量和可迭代对象的长度不一致也可以。可以回想一个 Python 的函数定义中，当需要按顺序提供不确定数量的参数时的做法：星号。星号装饰过的变量，只能放在解包变量的尾部，它会在赋值时，“贪婪” 地将后续的一切内容包裹到一个元组中。

```
info = [
    ("李华", 25, 165.5, 55.3),
    ("小刚", 26, 170.1, 65.0),
    ("小明", 23, 172.0, 67.2),
]
for name, age, *_ in info: # 不知道起什么名称时，下划线是不错的选择
    print(f"{name}'s age is {age}")
```

接收数据的变量数量和解包的可迭代对象长度不一致的情况，可以考虑有这么一行数据，对用户有用的只有前两个，为剩下的数据起名实在是一件很头痛的事，毕竟起名实在是太难了。

可迭代对象还可以解包为另一个可迭代对象中的多个元素，也是通过星号。值得注意的是函数的形式参数列表也可以算是一种可迭代对象[4]，所以也可以在参数列表中展开。

官方文档中关于单个星号解包语法的简短描述：An asterisk * denotes iterable unpacking. Its operand must be an iterable. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

```
# 在另一个可迭代对象中解包
lst1 = [1, 2, 3]
lst2 = [*lst1, 4, 5, 6] # [1, 2, 3, 4, 5, 6]
# 在函数参数列表中解包
def dist(x1, y1, x2, y2):
    xdiff = x2 - x1
    ydiff = y2 - y1
    return sqrt(xdiff ** 2 + ydiff ** 2)
pos1 = (1, 2)
pos2 = (3, 4)
dist(*pos1, *pos2)
```

对可迭代对象的解包顺序取决于可迭代对象本身迭代的顺序。对列表、元组、字符串这类有序对象，展开的顺序就是其自身元素储存的顺序；对集合、字典这类无序对象，因为其本身就是无序数据结构，迭代的顺序是无法保证的，因此展开后的元素顺序也无法保证。且字典的解包只能解包出来其键，因为字典每次迭代确实是只返回一个键。

对生成器展开后的顺序和生成器求值得到数据的顺序是一致的。如果是一个无限生成器，那么程序会卡死，因为解包的过程中会对可迭代对象不断迭代，直到迭代结束。

(i for i in range(3)) 是生成器初始式，并不是元组初始式，Python 中并没有元组初始式。

```
gen1 = (i for i in range(3))
[*gen1, 3, 4, 5] # [0, 1, 2, 3, 4, 5]

def gen2():
    i = 0
    while True:
        yield i
        i += 1
[*gen2, 3, 4, 5] # 程序会卡死
```

[4]CPython 的 C 代码中，对函数形式参数列表的访问，就是从元组对象身上通过`PyArg_ParseTuple`实现的，CPython 给函数对象的参数列表封装成了一个元组。

```
a = 1,
b = 2
c = 1 + 2
```

解包特性产生的 bug 也会极其难以排察。代码量大的情况下很难发现这个案例中，“1” 的后面有一个“,”，它会导致后续的代码中报出类型错误。乍一看这跟解包没啥关系，但这种 bug 多是由于代码编写的过程中有一段时期利用过解包特性，因此a曾经是一个元组，后期不再使用解包特性时，删除代码漏删了逗号。

字典解包

字典也是前面介绍的可迭代对象中的一种，但它有点特殊，所以拿出来单独讲。严格来说它是一种映射对象 [5] ，同时映射对象也是可迭代对象的子类。

映射对象：mapping object

前面说过字典的解包，只能得到其键，不能得到其值，因为字典迭代的时候确实每次只返回一个键。但有一种我们无时无刻不在用的字点键值对解包方法容易被忽视：`for k, v in yourdict.items()`。因为字典的`items`方法会返回一个`dict_items`可迭代对象，其每次迭代后返回一个元组，元组的内容正是一对键值对，此时迭代变量有两个，正好将元组内的键和值解包。

字典还有一种专用的解包语法：两个星号解包键值对。不过这个语法不能随便用，它有两个要求：字典的键都是字符串，只能在函数的形参列表中使用。它相当于是用字典的键去匹配函数的参数名称并将值输入进去。

官方文档中关于两个星号解包语法的简短描：A double asterisk ** denotes dictionary unpacking. Its operand must be a mapping. Each mapping item is added to the new dictionary. Later values replace values already set by earlier key/datum pairs and earlier dictionary unpackings.

```
1  info = [
2      {"name": "李华", "age":25, "height":165.5, "weight":55.3},
3      {"name": "小刚", "age":26, "height":170.1, "weight":65.0},
4  ]
5
6  def intro(name, age, height, weight):
7      print(f"{name}'s age is {age}, "
8            f"height - {height}, "
9            f"weight - {weight}")
10 for item in info:
11     intro(**item)
```

[5] Python 中的映射对象都是`collections.abc.Mapping`的子类或虚子类。

至此可能还是不会觉得有什么用，但是试想你有一个 json 字典，反序列化后直接就可以塞入函数中使用。

```
allinfo = json.load(your_json_file)
for info in allinfo:
    intro(**info)
```

0.2 闭包

0.2.1 什么是闭包

有时候某个函数中会临时用到一个临时函数，它只对当前作用域有意义，脱离当前环境后不会再用上。

在 C 语言中，我们通常定义一个 static 函数来限制该函数不会在编译后导出符号 [6] ，这样即使其它翻译单元也有一个同名函数，但是在链接时并不会产生冲突。

在《C Primer Plus》中关于翻译单元的描述：一个源代码文件和它所包含的头文件。大体就是指，预处理完成后、编译前这个时期的产物。

```
static max_of_3(int a, int b, int c) {
    int ret = a;
    if (b > ret) ret = b;
    if (c > ret) ret = c;
    return ret;
}
```

但这样做还是会有漏洞，万一被该翻译单元内的其它函数误用了怎么办。通常为了方便后续维护，会在代码中写上详细的使用说明。

Python 运行时的对象具有一切皆数据的特性。无论普通的数据类型如整数、字符串，还是函数和类，它们都是数据，没有指令和数据的区分，都可以在程序运行过程中动态生成。所以我们可以在函数中定义函数，甚至在函数中定义类，这种行为叫做闭包。

[6] C 语言默认情况下函数或全局变量如果没写 static 或 extern 修饰符，都是 extern 来修饰。如果被 extern 修饰过，那么编译后函数、全局变量的名称就具有全局链接属性，可以和其它库进行链接；如果被 static 修饰过，就具有静态链接属性，只能在当前翻译单元内使用。保留函数、全局变量名称到二进制数据中的这个行为就叫做导出符号。如果两个目标文件中拥有同名符号，那么链接时就会产生符号重定义的错误。

在 Python 中我们可以直接在一个函数内部定义另一个函数，这样在内部定义的函数就成了其所在函数中的一个局部变量，确保无法在更广阔的外部作用域中访问到，也不容易对后续的开发造成干扰。

```
def max_of_tup3s(tup3s):
    def max_of_3(a, b, c):
        ret = a
        if b > ret: ret = b
        if c > ret: ret = c
        return ret
    return [max_of_3(*tup3) for tup3 in tup3s]

tup3s = ((1, 2, 3), (4, 5, 6))
max_of_tup3s(tup3s) # (3, 6)
```

这种在函数中定义函数来进行功能隔离的行为，就叫做闭包。

0.2.2 循环体内的闭包函数会被多次编译吗

由于 Python 中的函数可以在程序运行中被动态生成，所以如果闭包函数定义被写在了循环体内，会导致反复生成函数对象。话说到这可能着急的读者已经开始改代码了，但其实大多数情况下根本不用着急。

```
def outter():
    for i in range(100):
        def pow2(n): return n * n
        pow2(i)
```

可能一些读者曾经在各种书籍、视频教程中看到这样一种描述——“Python 是一行一行边解释边执行的”。其实这么说并不准确，Python 虽然是一种脚本语言，但它也有自己 “编译” 的过程，Python 实际执行的是其编译后的字节码。

解释器可以在导入模块时对整个模块进行一次性编译产生字节码，并在本地写入其编译得到的字节码为缓存文件供后续使用，这样下次导入就不用再次编译。这种导入模块时进行的一次性编译是一行一行进行的，但其也是兼顾上下文的，并非所谓的行行独立，中间偶尔还会触发解释器的常量

折叠或代码优化。当全部编译完成后，解释器会去逐个指令执行编译完成后的字节码。

解释器也可以在运行过程中将一行字符串编译为字节码，然后立即执行，这种情况下则比较像“一行一行边解释边执行”，典型的案例是就是咱们的 REPL。

如果说 Python 是“一行一行边解释边执行”，那么是不是意味着循环体中的闭包函数会被反复重新定义。我们可以用标准库的`dis`模块中的`dis`方法对前面的 outter 函数进行“反编译”，看到函数编译后的字节码。

```
import dis
dis.dis(outter)
# 1     0 RESUME         0
#
# 2     2 LOAD_GLOBAL    1 (NULL + range)
#      12 LOAD_CONST     1 (100)
#      14 CALL           1
#      22 GET_ITER
#    >> 24 FOR_ITER      13 (to 54)
#      28 STORE_FAST     0 (i)
#
# 3    30 LOAD_CONST     2 (<code object pow2 at 0x102d4bab0>)
#      32 MAKE_FUNCTION  0
#      34 STORE_FAST     1 (pow2)
#
# 4    36 PUSH_NULL
#      38 LOAD_FAST      1 (pow2)
#      40 LOAD_FAST      0 (i)
#      42 CALL           1
#      50 POP_TOP
#      52 JUMP_BACKWARD 15 (to 24)
#
# 2 >> 54 END_FOR
#      56 RETURN_CONST   0 (None)
#
# Disassembly of <code object pow2 at 0x102d4bab0>:
# 3     0 RESUME         0
#       2 LOAD_FAST      0 (n)
#       4 LOAD_FAST      0 (n)
```

```
#        6 BINARY_OP      5 (*)
#       10 RETURN_VALUE
```

通过对字节码的观察，我们不难发现，闭包函数pow2的字节码只产生了一次编译结果。但是解释器会在循环中多次通过指令LOAD_CONST加载编译好的字节码，然后通过指令MAKE_FUNCTION产生一个新的函数对象。这也就证明了，Python 并非“一行一行边解释边执行”。

虽然LOAD_CONST和MAKE_FUNCTION会被冗余执行，如果循环本身次数不那么多，每次循环体内多这两个指令也无伤大雅，但是 Python 的循环语句本身就比较慢，但是如果循环的次数多了，叠加效应就会变得显著。

但无论如何，这都不是一个好的编程习惯。

0.2.3 捕获与自由变量

闭包函数能够访问其外部函数的局部变量，且在其外部函数执行完成后，仍然能够继续访问该变量。

```python
def return_a_number(n):
    num = n
    def impl():
        return num
    return impl
```

闭包函数访问外部函数作用域内的局部变量，这种行为叫作捕获；被闭包函数访问的外部函数作用域内的变量叫自由变量，在一些编程语言中也叫上值。

上值：up-value

由于自由变量本身也是外部函数的局部变量，所以外部函数是可重入的。外部函数多次执行返回的闭包函数，即使在代码里访问的都是同一个自由变量，但它们都是外部函数执行时的局部变量，所以互不影响。

可重入：re-entrant
是指不同时序下多次调用互不影响。

```python
r3 = return_a_number(3)
r4 = return_a_number(4)
r3() # 3
r4() # 4
```

Python 中为变量赋值，由于不需要声明，所以在函数中为变量赋值时，到底是为全局变量、自由变量还是局部变量赋值就容易产生歧意。为了消除歧意，如果要为全局变量赋值，要在赋值前使用`global`命令在函数内声明全局变量；如果要为自由变量赋值，要在赋值前使用`nonlocal`命令在闭包函数内声明自由变量

函数内变量只是读取而不写入的话就生歧意，只有需要赋值时才会产生此声明 global 或 nonlocal 的话，默局部变量。

```
total_counter = 0
def counter(n):
    closure_counter = 0
    def impl():
        global total_counter
        nonlocal closure_counter
        total_counter += 1
        closure_counter += 1
        return closure_counter, total_counter
    return impl

c1 = counter()
c2 = counter()
c1() # 1, 1
c1() # 2, 2
c2() # 1, 3
c2() # 2, 4
```

0.3 匿名函数

有时候需要临时用到一个简单函数的时候，定义一个闭包函数也觉得大动干戈了，还可以定义一个匿名函数，直接到一个变量上。

匿名函数形式：`lambda <arg1>, <arg2>, <...>: <expression>`

使用`lambda`关键字进行定义，冒号分隔参数列表和函数体。参数列表可以没有参数或任意数量的参数，使用逗号分隔，参数列表前后不需要括号。函数体是一个表达式，所以也意味着其必然得有返回值，即使是返回 None。

Python 中的函数即使没使用 return 语句，也会默认返回 None。我们可以用这个特性，使用 or 结构将不返回内容的函数结合起来，形成链式调用。

0.4 高阶函数

0.4.1 map

0.4.2 filter

0.4.3 reduce

0.5 装饰器

0.5.1 装饰器有什么用

0.5.2 自定义装饰器

0.5.3 自定义带参数的装饰器

0.5.4 手动实现 property 装饰器

第二部分

类

0.6 魔法方法

第三部分

模块与包

模块与包各自怎么理解

0.7 模块是什么

0.8 包是什么

0.9 相对导入

导入模块

xxx 导入模块

第四部分

标准库的设施

数据结构

路径操作

枚举

数据类

第五部分

高级特性

元类

描述符协议

类型注解

第六部分

高性能编程

与 C 语言混合编程

numpy

cython