目錄

关于本书

本书主要关注一些先进的编写Python代码的做法，以“编写高效且优雅的Python代码”一文开始。

众所周知，Python是一门强大的语言，可以用她编写出非常有意思的实用代码。那么如何编写出高级代码，是本书关注的问题。

编写高效且优雅的 PYTHON 代码

Source/来源：编写高效且优雅的 PYTHON 代码

Python 作为一门入门极易并容易上瘾的语音，相信已经成为了很多人“写着玩”的标配脚本语言。但很多教材并没有教授 Python 的进阶和优化。本文作为进阶系列的文章，从基础的语法到函数、迭代器、类，还有之后系列的线程 / 进程、第三方库、网络编程等内容，共同学习如何写出更加 Pythonic 的代码部分提炼自书籍：《Effective Python》&《Python3 Cookbook》，但也做出了修改，并加上了我自己的理解和运用中的最佳实践。

Pythonic

列表切割

`list[start:end:step]`

- 如果从列表开头开始切割，那么忽略 start 位的 0，例如list[:4]
- 如果一直切到列表尾部，则忽略 end 位的 0，例如list[3:]
- 切割列表时，即便 start 或者 end 索引跨界也不会有问题
- 列表切片不会改变原列表。索引都留空时，会生成一份原列表的拷贝

```
b = a[:]

assert b==a and b is not a # true
```

列表推导式

- 使用列表推导式，来取代 `map` 和 `filter`

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

# use map
squares = map ( lambda x : x**2, a )

# use list comprehension
squares = [ x**2 for x in a ]

# 一个很大的好处是，列表推导式可以对值进行判断，比如
squares = [ x**2 for x in a if x%2 == 0 ]

# 而如果这种情况要用 map 或者 filter 方法实现的话，则要多写一些函数
```

- 不要使用含有两个以上表达式的列表推导式

```
# 有一个嵌套的列表，现在要把它里面的所有元素扁平化输出

list = [[
    [1,2,3],
    [4,5,6]
]]

# 使用列表推导式

flat_list = [ x for list0 in list for list1 in list0 for x in li
st1]
# [1, 2, 3, 4, 5, 6]

# 可读性太差，易出错。这种时候更建议使用普通的循环

flat_list = []

for list0 in list:
    for list1 in list0:
        flat_list.extend ( list1 )
```

- 数据多时，列表推导式可能会消耗大量内存，此时建议使用生成器表达式

```
# 在列表推导式的推导过程中，对于输入序列的每个值来说，都可能要创建仅含一项元素的全新列表。因此数据量大时很耗性能。

# 使用生成器表达式

list = ( x **2 for x in range(0, 1000000000) )

# 生成器表达式返回的迭代器，只有在每次调用时才生成值，从而避免了内存占用
```

迭代

- 需要获取index时使用enumerate
- enumerate可以接受第二个参数，作为迭代时加在index上的数值

```
list = ['a','b','c','d']

for index, value in enumerate(list):
    print(index)

# 0
# 1
# 2
# 3

for index, value in enumerate(list,2):
    print(index)

# 2
# 3
# 4
# 5
```

- 用zip同时遍历两个迭代器

```
list_a = [ 'a', 'b', 'c', 'd' ]
list_b = [ 1, 2, 3 ]

# 虽然列表长度不一样，但只要有一个列表耗尽，则迭代就会停止

for letter,number in zip(list_a, list_b):
    print(letter,number)
# a 1
# b 2
# c 3
```

- zip遍历时返回一个元组

```
a = [ 1, 2, 3]
b = [ 'w', 'x', 'y', 'z' ]

for i in zip ( a, b ):
    print(i)
# (1, ‘w’)
# (2, ‘x’)
# (3, ‘y’)
```

- 关于for和while循环后的else块
  - 循环正常结束之后会调用else内的代码
  - 循环里通过break跳出循环，则不会执行else
  - 要遍历的序列为空时，立即执行else

```
for i in range(2):
    print(i)
else:
    print('loop finish')

# 0
# 1
# loop finish

for i in range(2):
    print(i)

    if i%2 == 0:
        break

else:
    print('loop finish')

# 0
```

反向迭代

对于普通的序列（列表），我们可以通过内置的 `reversed()` 函数进行反向迭代：

```
list_example = [ i for i in range(5) ]
iter_example = ( i for i in range(5) ) # 迭代器
set_example = { i for i in range(5) } # 集合

# 普通的正向迭代
# for i in list_example

# 通过 reversed 进行反向迭代
for i in reversed(list_example):
    print(i)

# 4
# 3
# 2
# 1
# 0

# 但无法作用于 集合 和 迭代器
reversed(iter_example)# TypeError: argument to reversed() must b
e a sequence
```

除此以外，还可以通过实现类里的 `__reversed__` 方法，将类进行反向迭代：

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # 正向迭代
    def __iter__(self):
        n = self.start

        while n>0:
            yield n
            n -= 1

    # 反向迭代
    def __reversed__(self):
        n = 1

        while n<= self.start:
            yield n
            n += 1

for i in reversed( Countdown(4) ):
    print(i)

# 1
# 2
# 3
# 4

for i in Countdown(4):
    print(i)

# 4
# 3
# 2
# 1
```

`try` / `except` / `else` / `finally`

- 如果 `try` 内没有发生异常，则调用 `else` 内的代码
- `else` 会在 `finally` 之前运行
- 最终一定会执行 `finally` ，可以在其中进行清理工作

函数

使用装饰器

装饰器用于在不改变原函数代码的情况下修改已存在的函数。常见场景是增加一句调试，或者为已有的函数增加log监控

举个栗子：

```
def decorator_fun(fun):
    def new_fun(*args, **kwargs):
        print('current fun:', fun.__name__)
        print('position arguments:', args)
        print('key arguments:', **kwargs)
        result = fun(*args, **kwargs)
        print(result)
        return result

    return new_fun

@decorator_fun
def add(a, b):
    return a + b

add(3, 2)

# current fun: add
# position arguments: (3, 2)
# key arguments: {}

# 5
```

除此以外，还可以编写接收参数的装饰器，其实就是在原本的装饰器上的外层又嵌套了一个函数：

```
def read_file ( filename = 'results.txt' ):
    def decorator_fun(fun):
        def new_fun(*args, **kwargs):
            result = fun(*args, **kwargs)

            with open(filename, 'a') as f:
                f.write( result + '\n' )

            return result

        return new_fun

    return decorator_fun

# 使用装饰器时代入参数
@read_file(filename='log.txt')
def add(a, b):
    return a + b
```

但是像上面那样使用装饰器的话有一个问题：

```
@decorator_fun
def add( a, b ):
    return a + b

print(add.__name__)
# new_fun
```

也就是说原函数已经被装饰器里的 `new_fun` 函数替代掉了。调用经过装饰的函数，相当于调用一个新函数。查看原函数的参数、注释、甚至函数名的时候，只能看到装饰器的相关信息。为了解决这个问题，我们可以使用 Python 自带的 `functools.wraps` 方法。

stackoverflow: What does functools.wraps do?

`functools.wraps` 是个很 hack 的方法，它本身作为一个装饰器，做用在装饰器内部将要返回的函数上。也就是说，它是装饰器的装饰器，并且以原函数为参数，作用是保留原函数的各种信息，使得我们之后查看被装饰了的原函数的信息时，可

以保持跟原函数一模一样。

```
from functools import wraps

def decorator_fun(fun):
    @wraps(fun)
    def new_fun(*args, **kwargs):
        result = fun(*args, **kwargs)
        print(result)

        return result

    return new_fun

@decorator_fun
def add(a, b):
    return a+b

print(add.__name__)
# add
```

此外，有时候我们的装饰器里可能会干不止一个事情，此时应该把事件作为额外的函数分离出去。但是又因为它可能仅仅和该装饰器有关，所以此时可以构造一个装饰器类。原理很简单，主要就是编写类里的 `__call__` 方法，使类能够像函数一样的调用。

```
from functools import wraps

class logResult(object):

    def __init__(self, filename='results.txt'):
        self.filename = filename

    def __call__(self,fun):

        @wraps(fun)
        def new_fun(*args, **kwargs):

            result=fun(*args, **kwargs)

            with open(filename, 'a') as f:
                f.write(result + '\n')

            return result

        self.send_notification()

        returnnew_fun

    def send_notification(self):
        pass

@logResult('log.txt')
def add( a, b ):
    return a+b
```

使用生成器

考虑使用生成器来改写直接返回列表的函数

```
# 定义一个函数，其作用是检测字符串里所有 a 的索引位置，最终返回所有 index 组成的数组

def get_a_indexs(string):

    result=[]

    for index,letter in enumerate(string):
        if letter == 'a':
            result.append(index)

    return result
```

用这种方法有几个小问题：

- 每次获取到符合条件的结果，都要调用 `append` 方法。但实际上我们的关注点根本不在这个方法，它只是我们达成目的的手段，实际上只需要 `index` 就好了
- 返回的 `result` 可以继续优化
- 数据都存在 `result` 里面，如果数据量很大的话，会比较占用内存

因此，使用生成器 `generator` 会更好。生成器是使用 `yield` 表达式的函数，调用生成器时，它不会真的执行，而是返回一个迭代器，每次在迭代器上调用内置的 `next` 函数时，迭代器会把生成器推进到下一个 `yield` 表达式：

```
def get_a_indexs(string):

    for index,letter in enumerate(string):
        if letter == 'a':
            yield index
```

获取到一个生成器以后，可以正常的遍历它：

```
string = 'this is a test to find a\' index'

indexs = get_a_indexs(string)

# 可以这样遍历
for i in indexs:
    print(i)

# 或者这样
try:
    while True:
        print(next(indexs))

except StopIteration:
    print('finish!')

# 生成器在获取完之后如果继续通过 `next()` 取值，则会触发 `StopIteration
` 错误但通过 `for` 循环遍历时会自动捕获到这个错误
```

如果你还是需要一个列表，那么可以将函数的调用结果作为参数，再调用 `list` 方法

```
results = get_a_indexs('this is a test to check a')
results_list = list(results)
```

可迭代对象

需要注意的是，普通的迭代器只能迭代一轮，一轮之后重复调用是无效的。解决这种问题的方法是，你可以定义一个可迭代的容器类：

```
class LoopIter(object):

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

    # 必须在 __iter__ 中 yield 结果
    def __iter__(self):
        for index,letter in enumerate(self.data):
            if letter == 'a':
                yield index
```

这样的话，将类的实例迭代重复多少次都没问题：

```
string = 'this is a test to find a\' index'

indexs = LoopIter(string)
print('loop 1')

for _ in indexs:
    print(_)

# loop 1
# 8
# 23


print('loop 2')

for _ in indexs:
    print(_)

# loop 2
# 8
# 23
```

但要注意的是，仅仅是实现 `__iter__` 方法的迭代器，只能通过 `for` 循环来迭代；想要通过 `next` 方法迭代的话则需要使用 `iter` 方法：

```
string = 'this is a test to find a\' index'

indexs = LoopIter(string)

next(indexs)# TypeError: ‘LoopIter’ object is not an iterator

iter_indexs = iter(indexs)

next(iter_indexs) # 8
```

使用位置参数

有时候，方法接收的参数数目可能不一定，比如定义一个求和的方法，至少要接收两个参数：

```
def sum( a, b ):
    return a + b

# 正常使用
sum ( 1, 2 ) # 3

# 但如果我想求很多数的总和，而将参数全部代入是会报错的，而一次一次代入又太麻烦

sum ( 1, 2, 3, 4, 5 ) # sum() takes 2 positional arguments but 5
 were given
```

对于这种接收参数数目不一定，而且不在乎参数传入顺序的函数，则应该利用位置参数 `*args` ：

```
def sum( *args ):

    result=0

    fornum in args:
        result += num

    return result

sum(1,2) # 3
sum(1,2,3,4,5) # 15

# 同时，也可以直接把一个数组带入，在带入时使用 * 进行解构
sum ( *[1, 2, 3, 4, 5] ) # 15
```

但要注意的是，不定长度的参数 `*args` 在传递给函数时，需要先转换成元组 tuple。这意味着，如果你将一个生成器作为参数带入到函数中，生成器将会先遍历一遍，转换为元组。这可能会消耗大量内存：

```
def get_nums():

    for num in range(10):
        yield num

nums = get_nums()

sum ( *nums ) # 45
# 但在需要遍历的数目较多时，会占用大量内存
```

使用关键字参数

- 关键字参数可提高代码可读性
- 可以通过关键字参数给函数提供默认值
- 便于扩充函数参数

定义只能使用关键字参数的函数

- 普通的方式，在调用时不会强制要求使用关键字参数

```
# 定义一个方法，它的作用是遍历一个数组，找出等于(或不等于)目标元素的 `index`


def get_indexs( array, target='', judge=True):

    for index,item in enumerate(array):
        if judge and item==target:
            yield index

        elif not judge and item!=target:
            yield index

array = [ 1, 2, 3, 4, 1 ]

# 下面这些都是可行的
result = get_indexs( array, target=1, judge=True)

print( list(result) ) # [0, 4]

result = get_indexs( array, 1, True )

print( list(result) ) # [0, 4]

result = get_indexs( array, 1 )

print( list(result) ) # [0, 4]
```

- 使用 Python3 中强制关键字参数的方式

```
# 定义一个方法，它的作用是遍历一个数组，找出等于(或不等于)目标元素的 `index`


def get_indexs ( array, *, target='', judge=True ):

    for index,item in enumerate(array):
        if judge and item==target:
            yield index

        elif not judge and item!=target:
            yield index

array = [ 1, 2, 3, 4, 1 ]

# 这样可行
result = get_indexs ( array, target=1, judge=True)

print( list(result) ) # [0, 4]

# 也可以忽略有默认值的参数
result = get_indexs( array, target=1 )

print( list(result) ) # [0, 4]

# 但不指定关键字参数则报错
get_indexs ( array, 1, True)
# TypeError: get_indexs() takes 1 positional argument but 3 were
 given
```

- 使用 Python2 中强制关键字参数的方式

```
# 定义一个方法，它的作用是遍历一个数组，找出等于(或不等于)目标元素的 `index`


# 使用 `**kwargs`，代表接收关键字参数，函数内的 `kwargs` 则是一个字典，
传入的关键字参数作为键值对的形式存在

def get_indexs ( array, **kwargs ):

    target = kwargs.pop( 'target', '')

    judge = kwargs.pop( 'judge', True )

    for index,item in enumerate(array):
        if judge and item==target:
            yield index

        elif not judge and item!=target:
            yield index

array = [ 1, 2, 3, 4, 1 ]

# 这样可行
result = get_indexs ( array, target=1, judge=True)

print( list(result) ) # [0, 4]

# 也可以忽略有默认值的参数
result = get_indexs ( array, target=1 )

print( list(result) ) # [0, 4]
# 但不指定关键字参数则报错

get_indexs ( array, 1, True )
# TypeError: get_indexs() takes 1 positional argument but 3 were
 given
```

关于参数的默认值

算是老生常谈了：函数的默认值只会在程序加载模块并读取到该函数的定义时设置一次

也就是说，如果给某参数赋予动态的值（比如 `[]` 或者 `{}` ），则如果之后在调用函数的时候给参数赋予了其他参数，则以后再调用这个函数的时候，之前定义的默认值将会改变，成为上一次调用时赋予的值：

```
def get_default ( value=[] ):
    return value

result = get_default()
result.append(1)

result2=get_default()
result2.append(2)

print(result) # [1, 2]

print(result2) # [1, 2]
```

因此，更推荐使用 `None` 作为默认参数，在函数内进行判断之后赋值：

```
def get_default ( value=None ):

    if value is None:
        return []

    return value

result = get_default()
result.append(1)

result2 = get_default()
result2.append(2)

print(result) # [1]
print(result2) # [2]
```

类

`__slots__`

默认情况下，Python 用一个字典来保存一个对象的实例属性。这使得我们可以在运行的时候动态的给类的实例添加新的属性：

```
test = Test()
test.new_key = 'new_value'
```

然而这个字典浪费了多余的空间 — 很多时候我们不会创建那么多的属性。因此通过 `__slots__` 可以告诉 Python 不要使用字典而是固定集合来分配空间。

```
class Test(object):

    # 用列表罗列所有的属性
    __slots__ = [ 'name', 'value']

    def __init__(self, name='test', value='0'):
        self.name = name
        self.value = value

test = Test()

# 此时再增加新的属性则会报错
test.new_key = 'new_value'
# AttributeError: 'Test' object has no attribute 'new_key'
```

`__call__`

通过定义类中的 `__call__` 方法，可以使该类的实例能够像普通函数一样调用。

```
class AddNumber(object):

    def __init__(self):
        self.num=0

    def __call__(self, num=1):
        self.num += num

add_number = AddNumber()

print( add_number.num ) # 0

add_number() # 像方法一样的调用

print(add_number.num) # 1

add_number(3)

print(add_number.num) # 4
```

通过这种方式实现的好处是，可以通过类的属性来保存状态，而不必创建一个闭包或者全局变量。

`@classmethod` 与 `@staticmethod`

资料：

- Python @classmethod and @staticmethod for beginner
- Difference between staticmethod and classmethod in python

`@classmethod` 和 `@staticmethod` 很像，但他们的使用场景并不一样。

类内部普通的方法，都是以 `self` 作为第一个参数，代表着通过实例调用时，将实例的作用域传入方法内；

- `@classmethod` 以 `cls` 作为第一个参数，代表将类本身的作用域传入。无论通过类来调用，还是通过类的实例调用，默认传入的第一个参数都将是类本身
- `@staticmethod` 不需要传入默认参数，类似于一个普通的函数

来通过实例了解它们的使用场景：

假设我们需要创建一个名为 `Date` 的类，用于储存 年/月/日 三个数据

```
class Date(object):

    def __init__( self, year=0, month=0, day=0 ):
        self.year = year
        self.month = month
        self.day = day

    @property
    def time(self):
        return "{year}-{month}-{day}".format(
            year = self.year,
            month = self.month,
            day = self.day
        )
```

上述代码创建了 `Date` 类，该类会在初始化时设置day/month/year属性，并且通过 `property` 设置了一个getter，可以在实例化之后，通过 `time` 获取存储的时间：

```
date=Date('2016', '11', '09')
date.time # 2016-11-09
```

但如果我们想改变属性传入的方式呢？毕竟，在初始化时就要传入年/月/日三个属性还是很烦人的。能否找到一个方法，在不改变现有接口和方法的情况下，可以通过传入 `2016-11-09` 这样的字符串来创建一个 `Date` 实例？

你可能会想到这样的方法：

```
date_string = '2016-11-09'
year, month, day = map(str, date_string.split('-'))
date = Date(year, month, day)
```

但不够好：

- 在类外额外多写了一个方法，每次还得格式化以后获取参数
- 这个方法也只跟 `Date` 类有关
- 没有解决传入参数过多的问题

此时就可以利用 `@classmethod` ，在类的内部新建一个格式化字符串，并返回类的实例的方法：

```
# 在 Date 内新增一个 classmethod

@classmethod
def from_string(cls, string):
    year, month, day = map( str, string.split('-') )

    # 在 `classmethod` 内可以通过 `cls` 来调用到类的方法，甚至创建实例
    date = cls ( year, month, day )

    return date
```

这样，我们就可以通过 `Date` 类来调用 `from_string` 方法创建实例，并且不侵略、修改旧的实例化方式：

```
date = Date.from_string('2016-11-09')

# 旧的实例化方式仍可以使用
date_old = Date ( '2016', '11', '09' )
```

好处：

- 在 `@classmethod` 内，可以通过 `cls` 参数，获取到跟外部调用类时一样的便利
- 可以在其中进一步封装该方法，提高复用性
- 更加符合面向对象的编程方式

而 `@staticmethod` ，因为其本身类似于普通的函数，所以可以把和这个类相关的 `helper` 方法作为 `@staticmethod` ，放在类里，然后直接通过类来调用这个方法。

```
@staticmethod
def is_month_validate(month):
    return int(month)<=12 and int(month)>=1
```

将与日期相关的辅助类函数作为 `@staticmethod` 方法放在 `Date` 类内后，可以通过类来调用这些方法：

```
month = '08'

if not Date.is_month_validate(month):
    print( '{} is a validate month number'.format(month) )
```

创建上下文管理器

上下文管理器，通俗的介绍就是：在代码块执行前，先进行准备工作；在代码块执行完成后，做收尾的处理工作。 `with` 语句常伴随上下文管理器一起出现，经典场景有：

```
with open('test.txt', 'r') as file:
    for line in file.readlines():
        print(line)
```

通过 `with` 语句，代码完成了文件打开操作，并在调用结束，或者读取发生异常时自动关闭文件，即完成了文件读写之后的处理工作。如果不通过上下文管理器的话，则会是这样的代码：

```
file = open ('test.txt', 'r')

try:
    for line in file.readlines():
        print(line)

finally:
    file.close()
```

比较繁琐吧？所以说使用上下文管理器的好处就是，通过调用我们预先设置好的回调，自动帮我们处理代码块开始执行和执行完毕时的工作。而通过自定义类的 `__enter__` 和 `__exit__` 方法，我们可以自定义一个上下文管理器。

```
class ReadFile(object):

    def __init__( self, filename ):
        self.file = open(filename, 'r')

    def __enter__( self ):
        return self.file

    def __exit__(self, type, value, traceback):

        # type, value, traceback 分别代表错误的类型、值、追踪栈
        self.file.close()

        # 返回 True 代表不抛出错误
        # 否则错误会被 with 语句抛出
        return True
```

然后可以以这样的方式进行调用：

```
with ReadFile('test.txt') as file_read:

    for line in file_read.readlines():
        print(line)
```

在调用的时候：

1. `with` 语句先暂存了 `ReadFile` 类的 `__exit__` 方法
2. 然后调用 `ReadFile` 类的 `__enter__` 方法
3. `__enter__` 方法打开文件，并将结果返回给 `with` 语句
4. 上一步的结果被传递给 `file_read` 参数
5. 在 `with` 语句内对 `file_read` 参数进行操作，读取每一行
6. 读取完成之后， `with` 语句调用之前暂存的 `__exit__` 方法
7. `__exit__` 方法关闭了文件

要注意的是，在 `__exit__` 方法内，我们关闭了文件，但最后返回 `True` ，所以错误不会被 `with` 语句抛出。否则 `with` 语句会抛出一个对应的错误。

Python使用断言的最佳时机

Source/来源：Python 使用断言的最佳时机, When to use assert

使用断言的最佳时机偶尔会被提起，通常是因为有人误用，因此我觉得有必要写一篇文章来阐述一下什么时候应该用断言，为什么应该用，什么时候不该用。

对那些没有意识到用断言的最佳时机的人来说，Python的断言就是检测一个条件，如果条件为真，它什么都不做；反之它触发一个带可选错误信息的 `AssertionError` 。如下例所示：

```
>>> x = 23
>>> assert x > 0, "x is not zero or negative"
>>> assert x%2 == 0, "x is not an even number"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: x is not an even number
```

很多人将断言作为当传递了错误的参数值时的一种快速而简便的触发异常的方式。但实际上这是错误的，而且是非常危险的错误，原因有两点。首先， `AssertionError` 通常是在测试函数参数时给出的错误。你不会像下面这样编码：

```
if not isinstance(x, int):
    raise AssertionError("not and int")
```

你应该用 `TypeError` 来替代，“断言”解决了错误的异常类型。

但是对断言来说更危险也更纠结的是：如果你执行Python时使用了 `-O` 或 `-OO` 优化标识，这能够通过编译却从来不会被执行，实际上就是说并不能保证断言会被执行。当恰当地使用了断言，这非常好的，但当不恰当地使用了断言，在使用 `-O` 标识执行时它将导致代码被彻底中断。

那么我们什么时候应该使用断言呢？如果没有特别的目的，断言应该用于如下情况：

- 防御性的编程

- 运行时对程序逻辑的检测
- 合约性检查（checking contracts）（比如前置条件及后置条件，e.g. pre-conditions and post-conditions）
- 程序中的常量
- 检查文档

（断言也可以用于代码测试，用作一个做事毛手毛脚的开发人员的单元测试，只要能你接受当使用 `-O` 标志时这个测试什么都不做。我有时也会在代码中用 `assert Fasle` 来对还没有实现的分支作标记，当然我希望他们失败。如果稍微更细节一些，或许触发 `NotImplementedError` 是更好的选择）

因为程序员是对于代码正确性表现出的信心不同，因此对于什么时候使用断言的意见各不相同。如果你确信代码是正确的，那么断言没有任何意义，因为它们从不会失败，因此你可以放心地移除它们。如果你确信它们会失败（例如对用户输入的数据的检测），你不敢用断言，这样编译就能通过，但你跳过了你的检查。

在以上两种情况之间的情况就显得特别有趣了，那就是当你相信代码是正确的，但又不是特别确定的时候。或许你忘记了一些奇怪的边角情况（因为我们都是人），在这种情况下，额外的运行时检查将帮助你尽可能早地捕获错误，而不是写了一大堆代码之后。

（这就是为什么使用断言的时机会不同。因为我们对代码正确性的信息不同，对于一个人有用的断言，对于另一个人来说却是无用的运行时测试。）

另一个断言用得好的地方就是检查程序中的不变量。一个不变量是一些你能相信为真的条件，除非一个缺陷导致它变成假。如果有一个缺陷，越早发现越好，因此我们需要对其进行测试，但我们不想因为这些测试而影响代码执行速度。因此采用断言，它能在开发时生效而在产品中失效。

一个关于不变量的例子可能是这样的情况。如果你的函数在开始的时候期望一个打开的数据库连接，并且在函数返回后该数据库连接依然是打开的，这是一个函数的不变量：

```
def some_function(arg):
    assert not DB.closed()
    ... # code goes here
    assert not DB.closed()
    return result
```

断言也是一个很好的检查点注释。为了替代如下注释：

```
#当我们执行到这里，我们知道n>2
```

你可以确保在运行时用以下断言：

```
assert n > 2
```

断言也是一种防御性的编程形式。你不是在防范当前代码发生错误，而防范由于以后的代码变更发生错误。理想情况下，单元测试应该直到这个作用，但是让我们面对这样一个现实：即使存在单元测试，他们在通常情况下也不是很完备。内建的机器人可能没有工作，但数周以来也没有人注意到它，或者人们在提交代码之前忘记了执行测试。内部检查将是防止错误渗入的另一道防线，尤其对于那些悄悄地失败，但会引起代码功能错误并返回错误结果的情况有效。

假设你有一系列的 `if...elif` 代码块，你预先知道变量期望的值：

```
# target期望x, y或z中的一个值
if target == x:
    run_x_code()
elif target == y:
    run_y_code()
else:
    run_z_code()
```

假设这段代码现在完全正确。但它会一直正确吗？需求变更，代码变更。如果需求变为允许 `target == w` ，并关联到 `run_w_code` ，那将会发生什么情况？如果我们变更了设置 `target` 的代码，但是忘记了改变这个代码块，它就会错误地调用 `run_z_code()` ，错误就会发生。对于这段代码最好的方法就是编写一些防御性的检查，这样它的执行，即使在变更以后，要么正确，要么马上失败。

在代码开始添加注释是个好的开端，但是人们都不太喜欢读和更新这些注释，这些注释会很快变得过时。但对于断言，我们可以同时对这块代码编写文档，如果这些断言被违反了，会直接引起一个简单而又直接的失败。

```
assert target in (x, y, z)
if target == x:
    run_x_code()
elif target == y:
    run_y_code()
else:
    assert target == z
    run_z_code()
```

这里的断言同时用于防御性编程和检查文档。我认为这是最优的解决方案：

```
if target == x:
    run_x_code()
elif target == y:
    run_y_code()
elif target == z:
    run_z_code()
else:
    # 这个不会发生，但是以防万一
    raise RuntimeError("an unexpected error occurred")
```

这诱使开发者去不理代码，移除像 `value == c` 这类不必要的测试，以及 `RuntimeError` 的“死代码”。另外，当 `unexpected error` 错误发生时这个消息将非常窘迫，确实会发生。

合约式设计是断言另一个用得好的地方。在合约式设计中，我们认为函数与其他调用者遵循合约，例如像这样的情况：

“如果你传给我一个非空字符串，我保证返回转换成大写的首字母。”

如果合约被破坏了，不管是被函数本身还是调用者，这都会产生缺陷。我们说这个函数需要有前置条件（对期望的参数的限制）和后置条件（对返回结果的约束）。因此这个函数可能是这样的：

```
def first_upper(astring):
    assert isinstance(astring, str) and len(astring) > 0
    result = astring[0].upper()
    assert isinstance(result, str) and len(result) == 1
    assert result == result.upper()
    return result
```

合约式设计的目的是，在一个正确的程序里，所有的前置条件和后置条件都将得到处理。这是断言的经典应用，自（这个想法持续）我们发布无缺陷的程序并且将其放入产品，程序将是正确的并且我们可以放心地移除检查。

这里是我建议不使用断言的情况：

- 不要用于测试用户提供的数据，或者那些需要在所有情况下需要改变检查的地方
- 不要用于检查你认为在通常使用中可能失败的地方。断言用于非常特别的失败条件。你的用户绝不看到一个 `AssertionError` ，如果看到了，那就是个必须修复的缺陷。
- 特别地不要因为断言只是比一个明确的测试加一个触发异常矮小而使用它。断言不是懒惰的代码编写者的捷径。
- 不要将断言用于公共函数库输入参数的检查，因为你不能控制调用者，并且不能保证它不破坏函数的合约（the functions's contract）。
- 不要将断言用于你期望修改的任何错误。换句话，你没有任何理由在产品代码捕获一个 `AssertionError` 异常。
- 不要太多使用断言，它们使代码变得晦涩难懂。