Python3 代码开发编程约定

> 🎉 **什么样的代码是好代码？如何评审别人的代码？**

一般来说，可以从如下 3 个方面评判一份代码是否是好代码。

1. **可读性**

代码的生命周期中，大部分时间都是被人阅读的，好的代码需要做到可以让别人快速无障碍地看懂你的代码，不需要太多的解释和注释，**做到代码即文档**。

所以在写代码的时候，不仅要考虑写的感受，**更要多考虑读代码人的感受**。

可阅读性可以提高代码的可维护性。

2. **可扩展性**

为了满足不同的需求，代码可以被更新或修改，并且同时不会破坏原有的功能。可扩展性可以帮助我们在不断变化的需求中保持代码的可用性。

3. **健壮性**

代码可以在不同的情况下正常工作，并且可以处理异常情况（允许抛出异常，但不允许不明不白的崩溃）。健壮性好的代码可以帮助防止程序崩溃和数据丢失。

编程规范正是从以上这些方面，来促进我们写出更高质量的代码。

依据约束性的强弱，此约定将约束依次分为【强制】、【推荐】、【参考】三类。在每条约定的解释中，说明对该条约定做了适当的扩展和解释，正例给出了提倡使用的方式，反例给出了需要避免使用的错误方式。

对于能够使用如 Pylint 等工具进行自动格式化的（如赋值符号 `=` 前后空格，本约定不再提及）。

> ✏️ There are only two hard things in Computer Science: cache invalidation and naming things.

> 在计算机科学领域只有两件难事：「缓存失效」 和 「给东西起名字」。
>
> -- Phil Karlton

一、编码约定

1. 【强制】代码的可读性非常重要。代码的生命周期中，绝大部份时间处于被人阅读的状态。因此，当有多个考量相互冲突时，请**优先**保证可读性。
2. 【强制】每行最多不超过 120 个字符。
   - 说明：过长的行会导致阅读障碍，使得缩进失效。
   - 确定单行字符个数时，应该考虑横屏，竖屏，git 对比 diff 等使用场景。
3. 【强制】模块中一级函数和类定义之间空 2 行，类中函数定义之间空 1 行，模块尾部有且仅有 1 个空行。
4. 【强制】在判断条件中应使用 is not，而不使用 not ... is。
   - 正例: if foo is not None:
   - 反例: if not foo is None:
5. 【强制】函数参数中，禁止使用可变类型变量（mutable）作为默认值。
   - 正例

```
def f(x=0, y=None, z=None):
    if y is None:
        y = []
    if z is None:
        z = {}
```

   - 反例

```
def f(x=0, y=[], z={}):
    pass

def f(a, b=time.time()):
    pass
```

6. 【强制】禁止定义了变量却不使用。
7. 【强制】禁止单个函数超过 35 行。

- 说明：人脑的阅读记忆能力有限。研究表明，人的短期记忆只能同时记住不超过 10 名字。因此，当某个函数过长（一般来说，超过一屏的函数被认为过长），应该及时把它拆分为多个小函数。

8. 【强制】禁止超过 2 个 for 语句或过滤器表达式，否则使用传统 for 循环语句替代。
   - 说明：当两个 for 语句的列表解析式比较复杂时候，也应该使用传统 for 循环语句替代。
   - 正例

```
number_list = [1, 2, 3, 10, 20, 55]
odd = [i for i in number_list if i % 2 == 1]

result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
```

   - 反例

```
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]
```

9. 【强制】列表推导式适用于简单场景。 如果语句过长，则不应该使用列表解析式。
   - 正例

```
fizzbuzz = []
for n in range(100):
    if n % 3 == 0 and n % 5 == 0:
        fizzbuzz.append(f'fizzbuzz {n}')
    elif n % 3 == 0:
        fizzbuzz.append(f'fizz {n}')
    elif n % 5 == 0:
        fizzbuzz.append(f'buzz {n}')
    else:
        fizzbuzz.append(n)
```

   - 反例

```
# 条件过于复杂，应该采用for语句展开
```

```
fizzbuzz = [
    f'fizzbuzz {n}' if n % 3 == 0 and n % 5 == 0
    else f'fizz {n}' if n % 3 == 0
    else f'buzz {n}' if n % 5 == 0
    else n
    for n in range(100)
]
```

a. 【强制】尽量避免使用 `from x imoprt *`。

10. 【推荐】变量名要有描述性，不能太宽泛，好的变量名可以极大提高代码的整体可读性。
    - 说明：在**可接受的长度范围内**，变量名能把它所指向的内容描述的越精确越好。因此，尽量不要用过于宽泛的词来作为变量名。
    - 反例：day, host, cards, temp
    - 正例：day_of_week, hosts_to_reboot, expired_cards
11. 【推荐】变量名最好让人能猜出类型。
    - 说明：即使在 PEP 484 出现之后，合适的命名也能够更有效的帮助阅读者的知道变量的类型。
    - 布尔类型变量的最大特点是：它只存在两个可能的值「是」或「不是」，因此推荐使用 is、has 等非黑即白的词修饰的变量名。
    - 正例：is_superuser，has_error，allow_vip，use_msgpack，debug 【约定俗成】
    - 阅读者看到和数字相关的名词时，都会默认想到它们是 int、float 类型。
    - 正例：释义为数字的所有单词：port（端口号）、age（年龄）、radius（半径）
    - 正例：使用 length、count 开头或者结尾的单词： length_of_username、max_length 。不要使用普通的复数来表示一个 int 类型变量，比如 apples、trips，最好用 number_of_apples、trips_count 来替代。
12. 【推荐】变量名不能过长，也不能过短，在 1～5 个单词之内较为合适。
    - 说明：绝大多数情况下，都应该**避免使用**只有一两个字母的短名字。如数组索引三剑客 i、j、k，用有明确含义的名字，比如 person_index 来代替它们总是会更好一些。
13. 【推荐】不要使用带否定含义的变量名。
    - 正例：is_special
    - 反例：is_not_normal
14. 【推荐】变量使用应该保持一致性。
    - 说明：如果你在一个方法内里面把图片变量叫做 photo，在其他的地方就不要把它改成 image，这样只会让代码的阅读者困惑：image 和 photo 到底是不是同一个东西？

15. 【推荐】变量定义尽量靠近使用。
    - 说明：很多人在刚开始学习编程时，会有一个习惯。就是把所有的变量定义写在一起，放在函数或方法的最前面。这样做只会让你的代码「看上去很整洁」，但是对提高代码可读性没有任何帮助。
    - 更好的做法是，**让变量定义尽量靠近使用**。那样当你阅读代码时，可以更好的理解代码的逻辑，而不是费劲的去想这个变量到底是什么、哪里定义的。
16. 合理定义临时变量，提升可读性。
    - 说明：代码里的复杂表达式可以通过增加临时变量的方式明确含义。
    - 反例：

```python
# 为所有性别为女性，或者级别大于 3 的活跃用户发放 10000 个金币
if user.is_active and (user.sex == 'female' or user.level > 3):
    user.add_coins(10000)
    return
```

    - 正例：

```python
# 为所有性别为女性，或者级别大于 3 的活跃用户发放 10000 个金币
user_is_eligible = user.is_active and (user.sex == 'female' or user.level > 3

if user_is_eligible:
    user.add_coins(10000)
    return
```

    - 反例：

```python
def get_best_trip_by_user_id(user_id):
    # 心理活动：「嗯，这个值未来说不定会修改/二次使用」，让我们先把它定义成变量吧!
    user = get_user(user_id)
    trip = get_best_trip(user_id)
    result = {
        'user': user,
        'trip': trip
    }
    return result
```

    正例：

其实，你所想的「未来」永远不会来，这段代码里的三个临时变量完全可以去掉，变成这样：

```
def get_best_trip_by_user_id(user_id):
    return {
        'user': get_user(user_id),
        'trip': get_best_trip(user_id)
    }
```

17. 【推荐】如非必要，不要用 `globals()`、`locals()`。
    - 说明：也许你第一次发现 `globals()`、`locals()` 这对内建函数时很兴奋，迫不及待的写下下面这种极端「简洁」的代码：
    - 反例：

```
def render_trip_page(request, user_id, trip_id):
    user = User.objects.get(id=user_id)
    trip = get_object_or_404(Trip, pk=trip_id)
    is_suggested = judg_is_suggested(user, trip)
    # 利用 locals() 节约了三行代码，我是个天才！
    return render(request, 'trip.html', locals())
```

    - 千万不要这么做，这样只会让读到这段代码的人「包括三个月后的你自己」痛恨你。
    - 因为他需要记住这个函数内定义的所有变量，更别提 `locals()` 还会把一些不必要的变量传递出去。
    - The Zen of Python（Python 之禅）说的清清楚楚：**Explicit is better than implicit.（显式优于隐式）**。因此，推荐下面的方式：
    - 正例：

```
return render(request, 'trip.html', {
        'user': user,
        'trip': trip,
        'is_suggested': is_suggested
    })
```

18. 【推荐】避免多层分支嵌套。
    - 说明：要竭尽所能的避免分支嵌套，**提前结束**可以优化很多情况下的分支嵌套问题。
    - 反例

```python
def buy_fruit(nerd, store):
    """去水果店买苹果

    - 先得看看店是不是在营业
    - 如果有苹果的话，就买 1 个
    - 如果钱不够，就回家取钱再来
    """
    if store.is_open():
        if store.has_stocks("apple"):
            if nerd.can_afford(store.price("apple", amount=1)):
                nerd.buy(store, "apple", amount=1)
                return
            else:
                nerd.go_home_and_get_money()
                return buy_fruit(nerd, store)
        else:
            raise MadAtNoFruit("no apple in store!")
    else:
        raise MadAtNoFruit("store is closed!")
```

- 正例

```python
def buy_fruit(nerd, store):
    if not store.is_open():
        raise MadAtNoFruit("store is closed!")

    if not store.has_stocks("apple"):
        raise MadAtNoFruit("no apple in store!")

    if nerd.can_afford(store.price("apple", amount=1)):
        nerd.buy(store, "apple", amount=1)
        return
    else:
        nerd.go_home_and_get_money()
        return buy_fruit(nerd, store)
```

19. 【推荐】封装过于复杂的逻辑判断。
    - 说明：如果条件分支里的表达式过于复杂，出现了太多 `not`、`and`、`or`，那么此代码的可读性就会大大降低。
    - 反例

```python
# 如果活动还在开放，并且活动剩余名额大于 10，为所有性别为女性，或者级别大于 3
```

```
# 的活跃用户发放 10000 个金币
if activity.is_active and activity.remaining > 10 and \
        user.is_active and (user.sex == 'female' or user.level > 3):
    user.add_coins(10000)
    return
```

- 正例：可以通过将具体的分支逻辑**封装**成函数或者方法，来达到**简化代码**的目的。

```
if activity.allow_new_user() and user.match_activity_condition():
    user.add_coins(10000)
    return
```

20. 【推荐】总是注意不同分支下的重复代码

- 说明：重复代码是代码质量的天敌，而条件分支语句又非常容易成为重复代码的重灾区。因此，当我们编写条件分支语句时，需要特别留意，**不要生产**不必要的重复代码。
- 反例

```
# 对于新用户，创建新的用户资料，否则更新旧资料
if user.no_profile_exists:
    create_user_profile(
        username=user.username,
        email=user.email,
        age=user.age,
        address=user.address,
        # 对于新建用户，将用户的积分置为 0
        points=0,
        created=now(),
    )
else:
    update_user_profile(
        username=user.username,
        email=user.email,
        age=user.age,
        address=user.address,
        updated=now(),
    )
```

- 正例：

```
if user.no_profile_exists:
```

```
    profile_func = create_user_profile
    extra_args = {'points': 0, 'created': now()}
else:
    profile_func = update_user_profile
    extra_args = {'updated': now()}

profile_func(
    username=user.username,
    email=user.email,
    age=user.age,
    address=user.address,
    **extra_args
)
```

21. 【推荐】多个条件判断： `or` 条件表达式应该将值为真可能性较高的写在前面， `and` 则应该写在后面。

```
abbreviations = ["cf.", "e.g.", "ex.", "etc.", "flg."]

for w in ("Mr.", "Hat", "is", "chasing", "."):
    if w in abbreviations and w[-1]=='.': # 这句性能较差
    # if w[-1] == '.' and w in abbreviations: # 性能好
        pass
```

22. 【推荐】使用德摩根定律： `not A or not B` 等价于 `not (A and B)` ，推荐使用前者。
    - 说明：人类思维不擅长处理过多的【否定】以及【或】这种逻辑关系。
    - 正例

```
# 如果用户没有登录或者用户没有使用 chrome，拒绝提供服务
if not user.has_logged_in or not user.is_from_chrome:
    return "our service is only available for chrome logged in user
```

    - 反例

```
if not (user.has_logged_in and user.is_from_chrome):
    return "our service is only available for chrome logged in user"
```

23. 【推荐】Ask forgiveness not permission。

- 说明：如果代码执行只有极少数情况会产生异常，此时建议使用 `try/exception` 而不是提前做条件判断。

24. 【推荐】在判断条件中使用 `all` 和 `any` 内置函数，可以使代码简洁，高效。

25. 【推荐】对序列（字符串、列表 、元组），空序列为 `False` 的情况判断，不要使用 `len` 函数。

    - 正例

```
if not seq:
    pass

if seq:
    pass
```

    - 反例

```
if len(seq):
    pass

if not len(seq):
    pass
```

26. 【推荐】使用 `def` 定义简短函数而不是使用 `Lambda` 匿名函数。

    - 说明：使用 `def` 定义函数有助于在 `trackbacks` 中打印有效的类型信息。
    - Python 曾经想要移除 Lambda，后来妥协了，参考这里。

27. 【推荐】经常改动的列表定义、字典定义、函数参数，建议每行一个元素，并且每行增加一个 `,`。

    - 正例

```
yes = ('y', 'Y', 'yes', 'TRUE', 'True', 'true', 'On', 'on', '1')  # 基本不再改

kwlist = [
    'False',
    ...
    'yield',  # 最后一个元素也增加一个逗号 ，方便以后diff不显示此行
]

person = {
    'name': 'bob',
```

```
    'age': 12,      # 可能经常增加字段
    }
```

- 反例

```
kwlist = ['False', 'None', 'True', 'and', 'as',
          'assert', 'async', ...
]

person = {'name': 'bob', 'age': 12} # 经常增加字段，不利于 git 做 diff 比较
```

28. 【推荐】过长的计算表达式，操作符应该在换行符之后出现。

- 正例

```
# YES: 易于将运算符与操作数匹配，可读性高
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

- 反例

```
# No: 运算符的位置远离其操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

29. 【推荐】公共函数、类、模块需要包含文档字符串。内部使用的函数，在函数名不能让使用者一眼就明白作用的情况下，需要提供注释。

- 说明：一个函数如果被设计为可以直接被其他开发者使用，那么请提供详细文档明确其含义，明确指出输入、输出、以及异常内容。

30. 【推荐】文档字符串（docstring）必须使用三重双引号 """ 。

31. 【推荐】在使用文档字符串时，推荐使用 reStructuredText 风格类型。

- 前三引号后不应该换行，应该紧接着在后面概括性的说明模块、函数、类、方法的作用，然后再空一行进行详细的说明。后三引号应该单独占一行。
- 其中函数的 docstring 推荐大致顺序是：概述、详细描述、参数、返回值、异常。
- 一般不要求描述实现细节，除非其中涉及非常复杂的算法。
- 示例

```
def fetch_bigtable_rows(big_table: Table, keys: list[str], other_silly_variab
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    :param big_table: An open Bigtable Table instance.
    :param keys: A sequence of strings representing the key of each table row
        to fetch.
    :param other_silly_variable: Another optional variable, that has a much
        longer name than the other args, and which does nothing.

    :return: A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    :raises ValueError: if `keys` is empty.
    :raises IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

32. 【推荐】在 `except` 子句中重新抛出原有异常时，不能用 `raise ex`，而是用 `raise`。
- 正例

```
try:
        int("hello")
  except ValueError:
        raise # 可以保留原始的 traceback
```

```
5
6  Traceback (most recent call last):
7  File "~/temp.py", line 2, in <module>
8      int("hello")
9  ValueError: invalid literal for int() with base 10: 'hello'
10
11   try:
12       raise MyException()
13   except MyException as ex:
14       raise AnotherException(str(ex))   # 允许：建议保留之前的异常栈信息，用于定位
```

- 反例

```
1  try:
2      int("hello")
3    except ValueError as e:
4      raise e # 异常栈信息从这里开始
5  Traceback (most recent call last):
6    File "~/temp.py", line 2, in <module>
7      raise e
8    File "~/temp.py", line 2, in <module>
9      int("hello")
10 ValueError: invalid literal for int() with base 10: 'hello'
```

33. 【推荐】所有 `try/except` 子句的代码要尽可的少，以免屏蔽其他的错误。

```
1  try:
2      value = collection[key]
3  except KeyError:
4      return key_not_found(key)
5  else:
6      return handle_value(value)
7  反例
8  try:
9      # 范围太广
10     return handle_value(collection[key])
11 except KeyError:
12     # 会捕捉到 handle_value() 中的 KeyError
13     return key_not_found(key)
```

34. 【推荐】函数的返回类型要单一，不能返回不同的类型【允许返回 `None`】。

- 正例

```
def add(a, b):
      if isinstance(a, int) and isinstance(b, int):
          return a + b
      else:
          raise Exception("Only int are allowed")
```

- 反例

```
 def add(a, b):
      if isinstance(a, int) and isinstance(b, int):
          return a + b
      else:
          return str(a) + str(b)
```

35. 【推荐】对于未知的条件分支或不应进入的分支，应该抛出异常，而不是返回一个值【如 `None`、`False`】。

- 正例

```
def foo(x):
    if x in ('SUCCESS',):
        return True
    else:
        # 如果一定不会走到的条件，应该增加异常，防止将来未知的语句执行。
        raise Exception()
```

- 反例

```
def foo(x):
    if x in ('SUCCESS',):
        return True
    return None
```

36. 【参考】应该充分使用 Python 的语法，但是不应该过度的使用 Python 的奇技淫巧

- 说明：比如 Python 的 Slice 语法

- 正例

```
a = [1, 2, 3, 4]
c = 'abcdef'
print(list(reversed(a)))
print(list(reversed(c)))
```

- 反例

```
a = [1, 2, 3, 4]
c = 'abcdef'
print(a[::-1])
print(c[::-1])
```

37. 【参考】使用 `dict` 来返回多个值。

```
# 一个函数返回多个值
def latlon_to_address(lat, lon):
    return country, province, city

# 但是，这样的用法会产生一个小问题：
# 如果某一天，latlon_to_address 函数需要增加返回城区「District」时怎么办？
# 如果是上面这种写法，你需要找到所有调用 latlon_to_address 的地方，
# 补上多出来的这个变量，否则就会抛出 ValueError: too many values to unpack 异常。
```

- 对于这种可能变动的多返回值函数，使用 `dict` 会更方便一些。当你新增返回值时，不会对之前的函数调用产生任何破坏性的影响

```
def latlon_to_address(lat, lon):
    return {
        'country': country,
        'province': province,
        'city': city
    }

addr_dict = latlon_to_address(lat, lon)
```

38. 【参考】多参考标准库，标准库中有很多很好的功能。如 operator.methodcaller() 等。

39. 【参考】使用 `[]`，`{}` 而不是 `list`，`dict`（速度更快）。
40. 【参考】使用 `for/else` 简化异常处理。
    - 以下两段代码等价
    - 我们借助了一个标志量 `found` 来判断循环结束是不是由 `break` 语句引起的

```python
def find_index(nums: list[int], target: int) -> None:
    found = False
    for index, num in enumerate(nums):
        if num == target:
            found = True
            break

    if found:
        print("not find {target}")
    else:
        print("find {target} at index {index}")

```

```python
def find_index(nums: list[int], target: int) -> None:
    for index, num in enumerate(nums):
        if num == target:
            print("find {target} at index {index}")
            break
    else:
        print("not find {target}")
```

二、编码原则

此部分提供一些编码的指导，特别是在多种写法在语法上都正确的时候，应该如何抉择。

1. **【推荐】进攻式实现、防御式调用**
    - 对于实现：没有人能保证自己的代码「完美」地处理所有 case。
    - 对于调用：没有不出错的代码，但没有人希望自己的系统因别人的手残挂掉（假定外部系统是沙子）。
2. **【推荐】不要为了正确而正确，隐藏错误**

- 如果对于一个字典 `my_dict`，其中键 `my_key` 是一个必要的 `key`。 就永远不要写 `my_dict.get(my_key)`。这样的程序只会**莫名其妙**的**对了**。根据墨菲定律，总有一天，这段代码会坑了整个团队。
- 必要的情况用异常通知上层发生了不可预知的错误，异常需要带有必要的信息，要辅助快速定位问题
- 反例

```
value = my_dict.get(my_key)
```

- 正例

```
value = my_dict[my_key] # my_key must exists

# 或者
if my_key not in my_dict:
  raise KeyError
```

3. **【推荐】不放大错误**
   - raise 不是用来推卸责任。 合理的 `assert`、`raise` 只是向外声明：我的代码发生了一个不可忽视的错误。代码的调用者必须负责处理这个错误（因为我声明了它不可忽视）。
   - 维护者： `raise` 不是推卸责任，维护者有义务抛出并且只抛出被认为不对的 case。
   - 调用者：调用别人的函数，就应该遵守别人的规范。调用方需要自己判断，是否需要继续 raise 给上层。
4. **【推荐】严厉排查死循环**
   - 无论多么小的 `while true` 循环，当 `break` 条件无法触发时（因为各种 bug），我们的服务就死掉了。
5. **【推荐】合理的处理异常**

代码不可能永不出错，合理的处理错误，能提高代码的健壮性。对于出错的处理情况，可以大致分为三类。

- 调用者不关心的异常：降级函数代替（确保降级函数不会出错）
- 关键功能异常：使用 `raise` 抛出
- 重要但是不应该影响主流程的异常：降级函数 + 日志报警

6. **【推荐】接口返回类型统一**
   - 接口的返回类型尽量统一，减少 `None` 值的返回。

```python
def get_number_list(num: int):
    if num < 10:
        return 0
    elif 10 <= num < 100:
        return [10 for i in range(10)]
    else:
        return None
```

- 一个命名为 `get_number_list` 的函数应当总是返回数组或者抛出异常。

7. **【推荐】只做自己该做的事情**

```python
def pickup_slotcard(feeds):
    ...
    for feed in feeds:
        if feed.type == slot_card:
            return format_feed(slot_card)
    return None
```

- pickup_slotcard：找出 feed 中的slot card ，只要负责找就行了，不要干别的事情。
- format_feed 不需要做。

8. **【推荐】尽量避免默认参数**

默认意味着参数可以不传，容易给调用者造成困扰：是传递还是不传？

- 不负责的调用者：直接就不传了。
- 负责的调用者：看下文档，代码才知道怎么传。
- 代码维护者：如果对默认值进行修改，则必须排查所有调用方。

9. **【推荐】合理分层，判断参数是否合理**

对于那些只是为了控制调用流程的参数，可以考虑拆分单独函数。

```python
def func(control, a, b ,c):
    if control == 1:
        pass
    elif control == 2:
        pass
    else:
        pass
return

```

```
# 只实现功能逻辑， 调用掌管业务逻辑
def f_control_1(a, b, c)
def f_control_2(a, b, c)
def f_control_3(a, b, c)
```

10. **【推荐】类设计，少用继承，多用组合。**
    - 过深的继承链会导致代码维护成本成倍的增加。
11. **【推荐】压缩类属性**

    如果一个类有很多相互关联的属性，其中某些熟悉由其他属性计算而出。那么这样的属性应该被去除，转而使用 get_xx 方法获取，或者使用 `property` 。

    没有衍生属性，类的 init 变得简单，代码的可测试性得到提高，可维护性得到提高。

```python
class Sample:

    def __init__(self, price: float, amount: int):
        self.price = price
        self.amount = amount

    @property
    def profit(self):
        return self.price * self.amount
```

12. **【推荐】禁止改变函数签名【里氏替换原则】**
    - 所有用到父类方法的地方，都可以用子类替换的同时而不产生代码语法错误，类型错误。

```python
class Base:
    def func(self, a: int):
        raise NotImplementedError()


class ClassA(Base):
    def func(self, a: int):
        pass


class ClassB(Base):
    def func(self, a: int):
        pass

```

```python
class ClassC(Base):
    def func(self, b: list[str]):  # 作者觉得参数不满足
        pass

# 代码调用者
ClassA().func(a)   # no bug
ClassB().func(a)   # no bug
ClassC().func(a)   # bug?  why?  怎么会挂？？？代码维护者脑子抽了?
```

13. **【推荐】考虑使用者的感受**
    - **写代码时，【考虑的是怎么写，还是怎么用】**是区分是否是优秀工程师的一个重要指标。只对作者友好的代码绝对称不上是优秀的代码。面向工程，面向合作，要让用的人舒服，让用的人**心理成本低。**
14. **【推荐】稍后等于用不【**LeBlanc （勒布朗）法则**】**
    - 如果旧的架构不能满足新的需求，那就需要调整，而且是尽快调整。
    - 合理的时间进行合理的重构才能让架构足够的活力，而不是「下次再说」。所有的新功能都是「应该在哪里」，而不是「狗皮膏药」漫天飞。
15. **【推荐】尽量无状态设计**
    - 有状态的代码，可维护性，可测试性都会大大降低。测试代码时，维护者需要痛苦的构造上下文，debug 场景难以复现
16. **【推荐】删除无用代码**（YAGNI 原则）

    You Ain't Gonna Need It（你不会需要它），好处：
    - 节约工程成本，避免编写不必要代码，无需维护这些代码。
    - 提高代码质量，不会让暂时无用甚至永远无用的代码，污染项目。

      开发者不应该编写无用逻辑，也**不应该**允许无用逻辑的存在。所有下线的逻辑（被注释的代码、没有调用的函数、永远为 False 的判断）必须干掉代码！

      如果将来某天需要，可以依赖 git 版本控制、仓库打 tag、文档维护 commit 节点等等，会有无数个比注释掉代码更高效的方案。
17. **【推荐】好的代码胜过注释【代码本身即注释】**
    - 良好的变量、函数命名是最好的注释。
    - 因为注释没有任何机制保证强制更新，不及时的注释容易更其他人带来更多的理解上的困惑。依赖注释的代码很可能由「工程导向」恶化成「文档导向」。漫天都是解释这个几百行函数在干嘛，可是没人能看懂。
18. **【推荐】减少变量，让变量尽量局部**

- 变量的中间赋值、写逻辑，往往是代码千奇百怪 bug 的导火索。让变量的使用限制在尽量小的范围。

a. 可以通过使用子函数、代码块等手段，来减少变量漫天飞的情况。

19. **【推荐】接口隔离原则【**Interface Segregation Principle】
    - 一个类，方法，不应该依赖于它不需要的接口（与单一职责原则有相似之处）
20. **【推荐】依赖倒置原则** （**D**ependence Inversion）
    - 高层方法不依赖于低层方法，二者都应该依赖于抽象。

三、类型注释

> 🍰 **为什么 Python 类型注释（Type Annotation）越来越重要且逐渐被广泛使用?**

1. 提高代码可读性：类型注释使得开发者更容易理解函数、类和方法的预期输入和输出，从而大大提高代码的可读性，使得其他人更容易维护和扩展代码。
2. 更好的文档：类型注释提供了关于函数的输入和输出类型的清晰说明，而无需额外的注释（代码即文档）。
3. 工具支持：许多 IDE 工具和 linter 可以基于类型注释提供代码的改进提示，错误检查和重构支持。
4. 提高代码质量：类型注释可以帮助开发者在开发过程中尽早发现潜在错误，因为类型不匹配能在编译时检测到。
5. 更好的性能：类型注释也可以被工具用来生成优化的代码，从而产生更快、更有效的程序。
6. 相关文档：**参考1**，**参考2**，**参考3**，**参考4**，**参考5**，**PEP 484**

> 📌 因此，在 Python 编程过程中，**强烈推荐使用类型注解**。
>
> 从 Python 3.5 版本开始，Python 将 typing 作为标准库引入。

四、代码审核

> 🎨 **提交 PR 时请求他人审核代码的目的**

1. 发现代码不合理的结构设计，提高代码质量，提高代码可读性，可维护性，可测试性。

2. 降低开发者之间的沟通成本，鼓励开发者之间互相帮助学习。
3. Code Review 不是大家来找茬，不要刻意找 Bug，不要鸡蛋里挑骨头，不要把代码审查搞成批判会。
4. 遇到不合常理的情况，先了解项目背景，可能是原有架构不合理。要做到**不妥协，不执拗，共同改善框架**。
5. 有争议的问题可暂时搁置【记下问题，整理思路，开会讨论】。

> 🏖️ **评审代码时，可以对 Code Review 进行分级**

1. 【request】xxxxxxx　　此条评论的代码**必须**修改才能予以通过
2. 【advise】xxxxxxxx　　此条评论的代码**建议**修改，但不修改也可以通过
3. 【question】xxxxxx　　此条评论的代码有疑问，需代码提交者进一步解释

五、扩展阅读

一些 Python 规范文档和检查工具。

1. Python Enhancement Proposals
2. The Zen of Python
3. PreCommit
4. Pylint
5. Pyright
6. Mypy