Python Code Compression Guide

By *Albert Lin*

2 July, 2023

Foreword

An accidental idea[1] that makes me want to try to compress Python code (like html, js, css). I have consulted relevant information on the Internet, and there is not much information. Perhaps the pythonors prefer coding pythonic, but it does not block my `min.py` plan.

> **Why compress Python code?**
>
> Answer: Because no one does it, I want to do it!

Attention, all these methods somewhat violate *The Zen of Python* and some other writing specifications. In addition, the readability will reduce. If it is not for the convenience of storage, I definitely do not recommend this, because it will affect subsequent maintenance; Or if you want to give someone a "surprise" when they say "*Talk is cheap, show me the code*", I suggest adding a code obfuscation (just for the sake of the trick someone) in addition to the following methods.

If you think it's useful, please ~~read and memorize it~~, **share it** with others, with more Pythonors, as this is one of the few tutorials on manually compressing Python code.

If you have any questions, please **contact me** through my email: `gkcoll@111.com`.

MAIN

Now, I will show many ways to compress or minify your Python code in different levels.

All these ways including:

- **Delete Comments & Blank Lines**
- **Refactoring**
- **Hidden Syntax Property**(*Semicolon*).

> Suggested execution order: From top to bottom according to this article.

Delete Comments & Blank Lines

> Difficulty: ★☆☆☆☆

It is best to have comments when the programmer hands over work, so that the receiver can quickly adapt to the work. But if there is no need for work handover and the work is already completed (or if you want to cause trouble while others are asking for code), simply removing all comments and blank lines can significantly reduce the number of lines of your code.

Find all the content after the hash sign (`#`) or between the triple quotations mark (`'''` or `"""`), remove them along with the symbols, and then delete all blank lines. As you can see, your code has also been compressed to a very compact size, and its readability has significantly decreased.

Refactoring

In some textbooks of Python, refactoring is an important way to make code more readable and reduce maintenance costs.

But in this section, refactoring plays a more important role: **abstraction**.

Merge Import Statements

> Difficulty: ★☆☆☆☆

While we have Python programming, we often need to import various packages. Like this:

```
from time import sleep as s
from time import time as t
import requests
import json
from langid import classify as c
```

In most of time, these import ~~expressions~~ statements can be compressed:

```
from time import sleep as s, time as t
import requests, json
from langid import classify as c
```

Although this may not seem short enough, if all import statements are only in the format of the `import module` (or `import module as alias`), they can all be compressed into one line.

```
import time, requests, json, langid
```

Merge Assignment Expressions

> Difficulty: ☆☆☆☆☆

This method should be known and proficiently used in both class and self-study:

```
a = 1
b = 2
c = 3
```

Just write it directly as `a, b, c = 1, 2, 3`.

Merge With Statements [2]

> Difficulty: ☆☆☆☆☆

When you see:

```
with a:
    with b:
        pass
```

Nice, replace with `with a,b: pass` is OK.

Modify variable names

> Difficulty: ★★☆☆☆

Ignoring the meaning, in order to ensure the degree of compression in your code, it is recommended to replace those overly long variable names with unreadable names like `a,b,c`.

Optimize For Loop Structures

Generator

> Difficulty: ★★★★★

In this part, I hope you have some knowledge about **Generator**.

Like the following code:

```
lst = list()
for i in range(1, 6):
    lst.append(i)
```

We can try this (Same operation effect):

```
lst = [i for i in range(1, 6)]
```

The effect of the example above maybe is not significant enough, but in some situations, a line of generator can compress your code by a wide margin.

Give an exaggerated example (A special function for calculating the length of a string, unlike the built-in `len` function, it recognizes *Hanzi* and treats them as 2 units of length):

```
def l(s: str) -> int:
    return sum([2 if '\u4e00' <= i <= '\u9fff' else 1 for i in s])
```

Cartesian Product [3]

Difficulty: ★★★☆☆

Sometimes, we may need to traverse multiple iterable objects at the same time, which may result in annoying nesting of for loops.

For multiple objects that involve traversing the same class or similar functionality, using for loop nesting is often redundant. In addition, for languages like Python that do not require a for loop exit declaration, the code may also be very unsightly without JS like triangle principle.

Like this:

```
# Non original
phone = ['iPhone', 'HuaWei', 'Mi']
number = [1, 2, 3]
color = ['White', 'Black']
for p in phone:
    for n in number:
        for c in color:
            print(f'{p}{n}{c}', end=' ')
```

Before giving further suggestions, let's first understand a mathematical knowledge - Cartesian Product.

Cartesian product refers to the combination of objects in two sets X and Y in mathematics, with the first object being a member of X and the second object being a member of Y, completing all possible ordered pairs. The Cartesian product, also known as the direct product, is represented by X × Y. For example, if set X = {a, b} and set Y = {0, 1, 2}, then the Cartesian product of the two sets is {(a, 0), (a, 1), (a, 2), (b, 0), (b, 1), (b, 2)}.

In Python's built-in module `itertools`, a higher-order class product() is provided to implement the combination of elements in multiple iterable objects (lists, strings, etc.), returning the Cartesian product of the combination of elements in the iterable object, which is equivalent to a nested loop. To see the effect more intuitively, let's compare the code below to see the effect of `product`.

```
# Calling `product` to implement element combination instead of a for loop
import itertools
for p, n, c in itertools.product(phone, number, color):
    print(f'{p}{n}{c}', end=' ')
```

The results are the same:

```
iPhone1White iPhone1Black iPhone2White iPhone2Black iPhone3White iPhone3Black
HuaWei1White HuaWei1Black HuaWei2White HuaWei2Black HuaWei3White HuaWei3Black
Mi1White Mi1Black Mi2White Mi2Black Mi3White Mi3Black RedMi1White RedMi1Black
RedMi2White RedMi2Black RedMi3White RedMi3Black
```

As can be seen, to achieve the same functionality, the for loop uses three layers of nesting, while `product()` only requires one layer of loop.

Convenience goes without saying. It is mainly helpful for us to further compress Python code. As for other advanced usage, please refer to relevant materials on your own.

Optimize Functions definition

> Difficulty: ★★★★☆

In most of time, define a function in Python need:

```
def func():
    """A little function."""
    # Do something...
    pass
```

If you are a senior Python programmer, you know the **lambda** expression [4]. It's a way to create an **Anonymous Function**. It only has the function body, without function name, without `def` and `return`. And in fact, a lambda function can be transferred like a function defined by `def`.

Just like this linear function: `def f(x): return 2*x + 1`, if the function will be called at only one place, you can replace it in where you want to call by this `lambda x: 2*x + 1`.

For example:

```
def calc(func, x):
    """
    Income a function and the parameter x,
    return the result calculated by the `func`.
    """
    return func(x)

y = calc(lambda x: 2*x + 1, 2)
print(y)  # 5
```

And for more advanced, this is a function for compute hex-digest MD5 value:

```python
import hashlib

def md5(content):
    hash = hashlib.md5(content.encode('utf-8'))
    return hash.hexdigest()
```

After have compression, we can use a *lambda* function, and the definition will solve in one line:

```python
md5 = lambda s: __import__("hashlib").md5(s.encode('utf-8')).hexdigest()
```

Attention

According to the official instructions of **PEP8**[5], it is not recommended to write lambda functions like this.

- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier:

```python
# Correct:
def f(x): return 2*x
```

```python
# Wrong:
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic '<lambda>'. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

Because this not only loses the anonymity of lambda (The example assigns it to a variable, which means it has a name), but also makes it difficult to maintain because the error displayed by the interpreter when an error occurs is limited and cannot be traced back to the function, it will only tell you that caused by a lambda.

> This reflects its **lack of flexibility**, **poor readability**, and **not suitable for complex logic**. So, it is not necessary to abuse it, especially for the project being maintained.

Therefore, if you are confident that the function you write will only be used once throughout the entire code file, make sure to use lambda functions only where needed. For example, in the above code, you can directly pass it in the desired location without assigning it to a variable:

```
print((lambda s: __import__("hashlib").md5(s.encode('utf-8')).hexdigest())
(input('>>>')))
```

In other cases, although defining a function that only contains one expression, as long as the function is called multiple times in different places, I suggest that you continue to use the `def` keyword (`def func():pass`).

> It's great to trick others, but don't trick yourself the same way you trick others.

Hidden Syntax Property

> Difficulty: ★★★☆☆

As is well known, Python utilizes indentation to ensure its code readability. We also know that in many cases, errors in indentation can have an impact on the results or successful operation. But this is not a big problem, our code still needs to be compressed, and Python does provide a way for us to compress them like JavaScript. That's a semicolon(`;`).

Unlike other languages, running Python with a semicolon after a single line of **expression** is meaningless. In Python, the semicolon is just for split expression in one line.

For example:

```
print('Hello World!');print('Viva Geek Collection!')
```

Let's upgrade the difficulty level: This is an example code that demonstrates how to read a file and print the content of it line by line:

```
with open('test.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(f)
```

> I guess you may know how to do now, but you may get a `SyntaxError` soon.

Now, let's compress it:

```
with open('test.txt', 'r', encoding='utf-8') as f:for line in f:print(f)
```

Run it... Okay, the expected bug has occurred.

Before solving this trouble, you should have noticed the bolded word closest to this in the above text. Yes, it's "**expression**". I didn't tell you that **Control Flow** cannot be compressed this way.

Actually, all the control flow (including `if-elif-else` structure, for-loop, while-loop, `with` structure and `try-except-finally` structure) cannot be compress directly by this way. Unless they are in the first position of a line.

So, the example should have a change:

```
with open('test.txt', 'r', encoding='utf-8') as f:print("\n".join(l for l in
f)
# OR
f=open('test.txt', 'r', encoding='utf-8');print("\n".join(l for l in
f);f.close()
```

In summary, it is

- All expressions of the **same level** can be merged directly with semicolons;
- When encountering control flow structures such as loops, **line breaks** are required (ensuring that they are at the beginning of a line), expressions within the control flow **add like above**, and expressions outside the control flow need to be **line breaks**;
- If the control flow structure is **nested**, the original **indentation** principle needs to be followed.

Difficult to understand? See this example:

```
if x>y:x+=y;print(y);
else:y+=x;print(x);
```

THE END

Have a Practice

> Difficulty: ★★★☆☆

Before the successful conclusion, I want to set a question to test everyone.

This is a function to return the titleize text of incomed English text, please compress it:

```
import nltk # Natural languages toolkit package
import string

CAPITALIZE_TAGS = set(['NN', 'PRP', 'VB', 'RB', 'JJ', 'WDT', 'WP', 'WRB'])
SPECIAL_SITUATION_TAGS = set(["IN", "UH", "CC", "DT", "TO"])

def titleize(obj: str) -> str:
    """
    Core function: Titleize the incomed sentence.
    :param obj: The sentence need to process.
    """
    words = nltk.pos_tag(nltk.word_tokenize(obj))
    newwords = []
    for i, word in enumerate(words):
        if i == 0 or i == len(words) - 1 or word[0][-1] in string.punctuation:
            newwords.append(word[0].capitalize())
        elif word[1] in CAPITALIZE_TAGS:
            newwords.append(word[0].capitalize())
        else:
            if word[1] in SPECIAL_SITUATION_TAGS and len(word[0]) < 5:
                newwords.append(word[0])
            else:
                newwords.append(word[0].capitalize())
    return " ".join(newwords)
```

In fact, there is one thing I didn't mention earlier, which is that before compressing the code, you can try carefully rereading the code again to see if there is any logic that can be optimized, this is also important for the compression effect.

Just like the content of the for loop in the above function appears redundant, because it can be written directly like this:

```
def titleize(obj: str) -> str:
    """
    Core function: Titleize the incomed sentence.
    :param obj: The sentence need to process.
    """
    words = nltk.pos_tag(nltk.word_tokenize(obj))
    newwords = []

    for word in words:
        if word[1] in SPECIAL_SITUATION_TAGS and len(word[0]) < 5:
            newwords.append(word[0])
        else:
            newwords.append(word[0].capitalize())

    return " ".join(newwords)
```

Then you may notice some details - the variable `CAPITALIZE_TAGS` and the module `string` are useless now, we can delete them directly. In addition, the variable which type of set - `SPECIAL_SITUATION_TAGS` is not very useful, because it needs to be traversed (Just use tuple or list directly).

Combining with the main text, we can compress it into this:

```
from nltk import pos_tag as a, word_tokenize as b;s,l=["IN", "UH", "CC", "DT",
"TO"],len  # The first line
def t(obj):return " ".join([w[0] if w[1] in s and l(w[0]) < 5 else
w[0].title() for w in a(b(obj))])  # The second line
```

Conclusion

Now, you've learned all the skills to compress Python code by reading this article. But in fact, the author me, is still a **Novice** - A novice who has been in the field for three years! So, if there is any unreasonable content or something wrong, please be sure to raise it, and I will study and modify it accordingly. If there are additional suggestions for relevant methods, you can also contribute, let's maintain this **revolutionary** document together!

---

1. performance - Can Python code be compressed like Javascript? - Stack Overflow ↩

2. Refer to: mnfy — minify/obfuscate Python 3 source code — mnfy 33.0.0 documentation ↩

3. Refer to: Explanation of the Permutation and Combination Generator in Python ↩

4. Refer to Summary of Lambda Function Usage - Zhihu and Python Lambda Function Deep Summary - Juejin (Content language is Chinese) ↩

5. See details in: https://pep8.org/#programming-recommendations ↩