# Code Info Statistics 代码行数统计

**完全依赖于正则表达式**的代码行数统计工具

## 1. 问题描述

本题需要完成的主要工作有：

- 各类行数统计
- 缩进层级统计
- if, for等语句数量统计
- 函数数量及行数统计
- 变量名及长度统计
- 结果输出

## 2. 问题分析

在本小节**由简到易**介绍在完成过程中遇到较大困难的部分。
以下所有的分析都建立在：

1. 目标文件中**只拥有N个函数**的情况。
2. 所有功能都基于**正则表达式**
   目标文件的示例代码如下：(详细代码见路径 `'/target_py.py` )

```
def func1():
    var1 = '变量1'
    var2 = '变量2'
    var3 = '变量3'

    var1 = 'a'
    var2 = 'b'

def func2():
...
...
def func_n()
```

## 2.1 最大缩进层级

在python中，缩进层级以4个空格为一级，因此需要用正则表达式，正确匹配n×4个空格，并保存该行的缩进层级。难点主要在于如何正确匹配。

## 2.2 统计注释行数

在python中，注释分为单行注释(以"#"符号开头), 注释块(以'''开头, '''结尾)以多行形式出现，因此在统计过程中，需要考虑两种不同情况的注释，综合统计。
下面是一个简单的注释示例：

```
# 这是一个单行注释

'''
这是一个多行注释，行数:1
这是一个多行注释，行数:2
'''

```

## 2.3 变量名出现次数统计

### 2.3.1 赋值操作与变量名首次出现可能混淆

```
var1 = '变量1'
var2 = '变量2'
var3 = '变量3'

var1 = 'a'
var2 = 'b'
```

以上述的例子举例，变量名的**正确数量为3**，如果仅简单的用正则表达式如下:

```
var_num = re.findall('^(\w)+\s*=\s*.*', content)
print('变量名出现次数:', len(var_num))

>>> 变量名出现次数:5

```

显然，这样的方法是错误的，正确的计数应当仅考虑变量名首次出现的情况

### 2.3.2 不同函数中的同名变量

```
def func1():
    var1 = '变量1'
    var2 = '变量2'

def func2():
    var1 = '变量1'
    var2 = '变量2'
```

与上述情况相悖，该情况下变量名的**正确数量为4**, 因此正确的计数应当考虑在不同函数中的同名变量

### 2.3.3 注释中的变量名

```
def func1():

    var1 = '变量1'
    var2 = '变量2'
    # var3 = 'a'
    # var4 = 'b'

```

在该情况下，变量名的**正确数量为2**, 需要正确考虑注释中出现的变量名

### 2.3.4 单行中的多个变量

```
def func1():
    var1, var2, var3 = 'a', 'b', 'c'
```

该情况下变量名的**正确数量为3**，因此不能简单地通过判断存在=号来确定该行的变量名数量

### 2.3.5 函数使用时的参数名易与变量混淆

```
def func1(var1, var2):

    print(var1, var2)
    return 'hello'

def func2():

    a = '变量1'
    b = '变量2'
    result = func1(var1=a, var2=b)

```

### 2.3.6 以 _ 命名的变量名

```
def func1():
    return 1, 2, 3

a, _, _=func1()
```

该情况下变量名的**正确数量为3**，虽然_重复出现了两次，但是由于_是python中的特殊变量，因此每次出现都要被单独算作一次变量名

## 2.4 带全空白行的函数体的确定

```
def func():

    print('普通函数体')

```

普通函数体的范围确定较为简单，仅需检测以def开头的行直至不再出现\n+空格的行即可。

```
def func():

    print('带有全空白行的函数体')
```

在函数体中，很可能出现完全空白的行，这是一个非常非常麻烦的问题，如何正确判断一个完全空白的行是在函数内，还是函数外，直接影响到了函数体行数的确定。

# 3. 解决方案

## 3.1 最大缩进层级

使用正则表达式匹配每一行的空格数，并通过取余操作，获取该行的缩进层级，最后取最大值即可。

```
indent_match = re.match(r'^(\s*)\S', line) # 匹配n个空格后接非空字符的行
if indent_match:

    indent = indent_match.group(1)
    max_indent = max(max_indent, len(indent) // 4)

```

# 3.2 统计注释行数

## 3.2.1 单行注释

单行注释较为简单，仅需匹配以"#"开头的行即可。

```
re.match(r'^\s*#', line)
```

## 3.2.2 多行注释

此处的解决方案较为繁琐，具体阐述如下：

1. 定义标志位comment_flag，用于标记当前行是否在注释块中
   定义数组comment_index，用于保存注释块的起点和终点

> comment_flag的目的是在后续变量名，if语句等统计中，**避免将注释块中的内容也统计进去**

1. 遍历每一行，当遇到以'''开头的行时，将comment_flag置为True，同时将该行的行号保存到comment_index中
2. 当comment_index数组中的个数为奇数时，说明当前的注释块还没有找到终点，自动置comment_flag为True
3. 遍历结束后，通过计算数组中的终点-起点，即可获得全部注释块的行数

核心伪代码如下

```
for index, line in enumerate(lines):

    if (len(comment_index)%2 !=0): # 当comment_index数组中的个数为奇数时，说明当前的注释块还没
        comment_flag=0
    if re.match(r"^\s*'''", line)
        comment_flag = True
        comment_index.append(index)

comment_lines=0
for i in range(0, len(comment_index), 2):

    comment_lines += comment_index[i+1] - comment_index[i]

```

## 3.3 统计变量名

### 3.3.1 解决：赋值操作与变量名首次出现可能混淆

采用字典记录已经出现过的变量名，出现变量名时先判断是否已经存在在字典中。
核心伪代码如下

```
# v代表变量名
if v not in variable_dict:
    variable_dict[v] = 1
    result['variable_len'] += len(v)
    result['variable_num'] += 1
```

### 3.3.2 解决：不同函数中的同名变量

将每一个函数视为一段独立的代码块，统计变量时仅在该函数体中进行统计。
与上述的逐行方法( `f.readlines()` )不同，为了正确的确定函数体，采用 `f.read()` 获取文件的全部内容进行正则匹配。(这是因为 `f.read()` 得到的内容带有换行符 `\n` )
核心伪代码如下

```
content= 直接读取的py文件的全部内容
pattern = r'\n?[ ]*def\s+\w+\(.*?\):(?:\n[ ]+.+)+'

# 使用正则表达式匹配整个函数体

matches = re.findall(pattern, content)
for func in matches:

    寻找变量

```

### 3.3.3 解决：注释中的变量名

在寻找形如 `a = 1` 的赋值语句时，将 `#` 考虑在内，在后续的校验中，如果该行存在 `#` 则continue，不进行变量搜寻操作。

核心伪代码如下:

```
#* 在函数体内寻找赋值语句 通过\#?匹配0或1个#的情况
let_sentence = re.findall(r'\s*\#?\s*(?:\w+,?\s)+=\s+.*[^\)]', func)
if let_sentence:
    for sentence in let_sentence:
        if "#" in sentence:
            continue
```

### 3.3.4 解决：单行中的多个变量

在该情况下需要考虑, 紧跟着的变量名，因此首先在函数体内找到赋值语句，再通过字符串 `split(',')` 操作，分隔出变量名。为了避免2.3.5的情况，额外使用了 `split('=')` 提取了赋值语句的前半部分。

核心伪代码如下：

```
let_sentence = re.findall(r'\s*\#?\s*(?:\w+, ?\s)+=\s+.*[^\)]', func)
if let_sentence:

    for sentence in let_sentence:
        variable = sentence.strip().replace(' ', '').split('=')[0].split(',')

```

### 3.3.5 半解决：函数使用时的参数名易与变量混淆

解决该问题的前提为规范的python代码，我本人使用的代码格式化工具为yapf, yapf会自动将变量赋值操作使用 隔开，具体示例如下：

```
a = 1

b = func(var=1) #传参赋值=周边无空格
```

因此在统计过程中只需在正则表达式中特地指定 的出现次数，即可区分传参的赋值与变量赋值。

核心代码伪代码同3.3.4。

### 3.3.6 未解决：以 _ 命名的变量名

由于时间问题以及本人代码实力，目前没有想到简洁明了的解决方式，这也是有待提高的部分

## 3.4 未解决：带全空白行的函数体的确定

经过一定时间的探索，本人暂时没有想到基于以下条件的全空白行的函数体确定方式

1. 完全依赖正则表达式
2. 简洁 (所有的代码信息统计仅循环一次所有行，不做重复扫行操作)

# 4 结果展示

1. 终端输出结果

```
总行数：149

注释行数：19

空行数：28

有效代码行数：102

代码行平均长度：39

最大缩进层级：2

if语句数：3

for语句数量：2

while语句数量：0

try语句数量：0

except语句数量：0

函数数量：13

函数平均行数：8

变量数：63

变量名平均长度：6
```

```
函数view_straight定义变量:
    start_x
    start_y
    end_x
    end_y
    _

函数view_circle定义变量:
    start_x
    start_y
    new_end_x
    new_end_y

函数view_Rect定义变量:
    start_x
    start_y
    end_x
    end_y
    _
```

2. 写入txt文件

```
总行数: 149

注释行数: 19

空行数: 28

有效代码行数: 102

代码行平均长度: 39

最大缩进层级: 2

if语句数: 3

for语句数量: 2

while语句数量: 0

try语句数量: 0

except语句数量: 0

函数数量: 13

函数平均行数: 8

变量数: 63

变量名平均长度: 6


函数view_straight定义变量:
    start_x
    start_y
    end_x
    end_y
    _

函数view_circle定义变量:
    start_x
    start_y
    new_end_x
    new_end_y

函数view_Rect定义变量:
    start_x
    start_y
    end_x
    end_y
```