

[笔记][黑马 Python 之 Python 基础 - 4]

Python

[笔记][黑马 Python 之 Python 基础 - 4]

- 116. 循环计算-01-思路分析
- 117. 循环计算-02-0到100数字累加
- 118. 循环计算-03-偶数求和-准备偶数
- 119. 循环计算-04-偶数求和-计算结果
- 120. break和continue-01-break关键字的应用场景
- 121. break和continue-02-break代码演练
- 122. break和continue-03-continue关键字的应用场景
- 123. break和continue-04-continue代码演练
- 124. 循环嵌套-01-基本语法
- 125. 循环嵌套-02-使用字符串运算直接输出小星星
- 126. 循环嵌套-03-[扩展]print函数的结尾处理
- 127. 循环嵌套小星星-01-输出行并确定思路
- 128. 循环嵌套小星星-02-嵌套循环完成案例
- 129. 九九乘法表-01-明确思路和步骤
- 130. 九九乘法表-02-打印9行小星星
- 131. 九九乘法表-03-九九乘法表数据输出
- 132. 九九乘法表-04-使用转义字符调整格式
- 133. 函数-01-明确学习目标
- 134. 函数-02-函数的概念以及作用
- 135. 函数-03-函数的快速体验
- 136. 函数基本使用-01-函数定义语法
- 137. 函数的基本使用-02-第一个函数演练
- 138. 函数基本使用-03-函数的定义以及调用执行线路图
- 139. 函数基本使用-04-应该先定义函数再调用函数
- 140. 函数基本使用-05-[扩展]单步越过和单步进入
- 141. 函数基本使用-06-函数的文档注释
- 142. 函数参数-01-没有参数的函数过于死板
- 143. 函数参数-02-函数参数的使用以及作用
- 144. 函数参数-03-形参和实参
- 145. 函数的返回值-01-返回值的应用场景和格式
- 146. 函数的返回值-02-改造求和函数
- 147. 函数的返回值-03-return关键字的注意事项
- 148. 函数的嵌套调用-01-函数嵌套调用的执行线路图
- 149. 函数的嵌套调用-02-[扩展]用百度网站举例说明函数的调用线路
- 150. 分隔线演练-01-利用参数增加分隔线的灵活度

- 151. 分隔线演练-02-打印多条分隔线
- 152. 分隔线演练-03-增加多行分隔线函数的参数
- 153. 分隔线演练-04-[扩展]PyCharm给函数增加文档注释

116. 循环计算-01-思路分析

在程序开发中，通常会遇到 **利用循环 重复计算** 的需求

遇到这种需求，可以：

1. 在 `while` 上方定义一个变量，用于 **存放最终计算结果**
2. 在循环体内部，每次循环都用 **最新的计算结果**，**更新** 之前定义的变量

117. 循环计算-02-0到100数字累加

需求

- 计算 0 ~ 100 之间所有数字的累计求和结果

`Ctrl + Shift + F10` 运行当前文件

```
# 计算 0 ~ 100 之间所有数字的累计求和结果
# 0. 定义最终结果的变量
result = 0

# 1. 定义一个整数的变量记录循环的次数
i = 0

# 2. 开始循环
while i <= 100:
    print(i)

    # 每一次循环，都让 result 这个变量和 i 这个计数器相加
    result += i

    # 处理计数器
    i += 1

print('0~100之间的数字求和结果 = %d' % result)
```

118. 循环计算-03-偶数求和-准备偶数

需求进阶

- 计算 0 ~ 100 之间 所有 **偶数** 的累计求和结果

开发步骤

1. 编写循环 **确认 要计算的数字**
2. 添加 **结果** 变量，在循环内部 **处理计算结果**

快捷键 **Shift + Esc** 隐藏下方控制台

```
# 计算 0 ~ 100 之间 所有 **偶数** 的累计求和结果

# 开发步骤

# 1. 编写循环 确认 要计算的数字
# 2. 添加 结果 变量，在循环内部 处理计算结果

i = 0
while i <= 100:

    # 判断变量 i 中的数值，是否是一个偶数
    # i % 2 == 0
    if i % 2 == 0:
        print(i)

    i += 1
```

119. 循环计算-04-偶数求和-计算结果

```
# 计算 0 ~ 100 之间 所有 **偶数** 的累计求和结果

# 开发步骤

# 1. 编写循环 确认 要计算的数字
# 2. 添加 结果 变量，在循环内部 处理计算结果

# 1> 定义一个记录最终结果的变量
result = 0
```

```
i = 0

while i <= 100:

    # 判断变量 i 中的数值，是否是一个偶数
    # 偶数 i % 2 == 0
    # 奇数 i % 2 != 0
    if i % 2 == 0:
        print(i)

    # 2> 当 i 这个变量是偶数时，才进行累加操作！
    result += i

    i += 1

print("0~100之间的偶数累加结果 = %d" % result)
```

120. break和continue-01-break关键字的应用场景

break 和 continue

break 和 **continue** 是专门在循环中使用的关键字

- **break** 某一条件满足时，退出循环，不再执行后续重复的代码
- **continue** 某一条件满足时，不执行后续重复的代码，跳转到循环开始的条件判断

break 和 **continue** 只针对 **当前所在循环** 有效

121. break和continue-02-break代码演练

```
i = 0

while i < 10:

    # break 某一条件满足时，退出循环，不再执行后续重复的代码
```

```
# i == 3
if i == 3:
    break

print(i)

i += 1

print('over')
```

122. break和continue-03-continue关键字的应用场景

- **continue** 某一条件满足时，不执行后续重复的代码，跳转到循环开始的条件判断

123. break和continue-04-continue代码演练

注意：**continue** 之前一定要处理计数器

```
i = 0

while i < 10:

    # continue 某一条件满足时，不执行后续重复的代码，跳转到循环开始的条件判断
    # i == 3
    if i == 3:
        # 注意：在循环中，如果使用 continue 关键字
        # 在使用关键字之前，需要确认循环的计数是否修改
        # 否则可能会导致死循环
        i += 1

        continue

    print(i)

    i += 1
```

124. 循环嵌套-01-基本语法

循环嵌套

- `while` 嵌套就是：`while` 里面还有 `while`

`while` 条件 1:

条件满足时，做的事情1

条件满足时，做的事情2

条件满足时，做的事情3

...(省略)...

`while` 条件 2:

条件满足时，做的事情1

条件满足时，做的事情2

条件满足时，做的事情3

...(省略)...

处理条件 2

处理条件 1

125. 循环嵌套-02-使用字符串运算直接输出小星星

用嵌套打印小星星

需求

- 在控制台连续输出五行 `*`，每一行星号的数量依次递增

```
*
**
***
****
*****
```

- 使用字符串乘法打印

```
# 在控制台连续输出五行 *，每一行星号的数量依次递增
# *
# **
# ***
# ****
# *****
```

```
# *****

# 1. 定义一个计数器变量，从数字 1 开始，循环会比较方便
row = 1

# 2. 开始循环
while row <= 5:

    print('*' * row)

    row += 1
```

126. 循环嵌套-03-[扩展]print函数的结尾处理

知识点 对 `print` 函数的使用做一个增强

- 在默认情况下，`print` 函数输出内容之后，会自动在内容末尾增加换行
- 如果不希望末尾增加换行，可以在 `print` 函数输出内容的后面增加 `, end=""`
- 其中 `""` 中间可以指定 `print` 函数输出内容之后，继续希望显示的内容
- 语法格式如下：

```
# 向控制台输出内容结束之后，不会换行
print("*", end="")

# 单纯的换行
print("")
```

`end=""` 表示向控制台输出内容结束之后，不会换行

127. 循环嵌套小星星-01-输出行并确定思路

假设 `Python` 没有提供 字符串的 `*` 操作 拼接字符串

需求

- 在控制台连续输出五行 `*`，每一行星号的数量依次递增

```
*
**
***
****
*****
```

开发步骤

- 1> 完成 5 行内容的简单输出
- 2> 分析每行内部的 * 应该如何处理？
 - 每行显示的星星和当前所在的行数是一致的
 - 嵌套一个小的循环，专门处理每一行中 列 的星星显示

128. 循环嵌套小星星-02-嵌套循环完成案例

col 是列， row 是行

```
# 需求
# 在控制台连续输出五行 *，每一行星号的数量依次递增
# *
# **
# ***
# ****
# *****
#
# 开发步骤
# 1> 完成 5 行内容的简单输出
# 2> 分析每行内部的 * 应该如何处理？
# 每行显示的星星和当前所在的行数是一致的
# 嵌套一个小的循环，专门处理每一行中 `列` 的星星显示

row = 1
while row <= 5:

    # 每一行要打印的星星就是和当前的行数是一致的
    # 增加一个小的循环，专门负责当前行中，每一 列 的星星显示
    # 1. 定义一个列计数器变量
    col = 1

    # 2. 开始循环
    """
    1 1
    2 2
    """
```



```

3 3
4 4
5 5
"""

while col <= row:

    # print('%d' % col)
    print('*', end='')

    col += 1

# print('第 %d 行' % row)
# 这行代码的目的，就是在一行星星输出完成之后，添加换行！
print('\n')

row += 1

```

129. 九九乘法表-01-明确思路和步骤

需求 输出 九九乘法表，格式如下：

```

1 * 1 = 1
1 * 2 = 2   2 * 2 = 4
1 * 3 = 3   2 * 3 = 6   3 * 3 = 9
1 * 4 = 4   2 * 4 = 8   3 * 4 = 12   4 * 4 = 16
1 * 5 = 5   2 * 5 = 10  3 * 5 = 15   4 * 5 = 20   5 * 5 = 25
1 * 6 = 6   2 * 6 = 12  3 * 6 = 18   4 * 6 = 24   5 * 6 = 30   6 * 6 = 36
1 * 7 = 7   2 * 7 = 14  3 * 7 = 21   4 * 7 = 28   5 * 7 = 35   6 * 7 = 42   7
* 7 = 49
1 * 8 = 8   2 * 8 = 16  3 * 8 = 24   4 * 8 = 32   5 * 8 = 40   6 * 8 = 48   7
* 8 = 56   8 * 8 = 64
1 * 9 = 9   2 * 9 = 18  3 * 9 = 27   4 * 9 = 36   5 * 9 = 45   6 * 9 = 54   7
* 9 = 63   8 * 9 = 72   9 * 9 = 81

```

开发步骤

- 打印 9 行小星星

```

*
**
***
****
*****

```

```
*****
*****
*****
*****
```

- 将每一个 `*` 替换成对应的行与列相乘

130. 九九乘法表-02-打印9行小星星

```
row = 1

while row <= 9:

    col = 1

    while col <= row:

        print('*', end='')

        col += 1

    # print('%d' % row)
    print('')

    row += 1
```

131. 九九乘法表-03-九九乘法表数据输出

```
# 1. 打印 9 行小星星
row = 1

while row <= 9:

    col = 1

    while col <= row:

        # print('*', end='')
        print('%d * %d = %d' % (col, row, col * row), end=' ')

    row += 1
```

```
col += 1

# print('%d' % row)
print('')

row += 1
```

132. 九九乘法表-04-使用转义字符调整格式

字符串中的转义字符

- `\t` 在控制台输出一个 **制表符**，协助在输出文本时 **垂直方向** 保持对齐
- `\n` 在控制台输出一个 **换行符**

制表符 的功能是在不使用表格的情况下在 **垂直方向** 按列对齐文本

转义字符	描述
<code>\\</code>	反斜杠符号
<code>\'</code>	单引号
<code>\"</code>	双引号
<code>\n</code>	换行
<code>\t</code>	横向制表符
<code>\r</code>	回车

133. 函数-01-明确学习目标

目标

- 函数的快速体验
- 函数的基本使用
- 函数的参数
- 函数的返回值
- 函数的嵌套调用
- 在模块中定义函数

134. 函数-02-函数的概念以及作用

- 所谓**函数**，就是把 **具有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**
- 函数的使用包含两个步骤：
 1. 定义函数 —— **封装** 独立的功能
 2. 调用函数 —— 享受 **封装** 的成果
- **函数的作用**，在开发程序时，使用函数可以提高编写的效率以及代码的 **重用**

135. 函数-03-函数的快速体验

演练步骤

1. 新建 **04_函数** 项目
2. 复制之前完成的 **乘法表** 文件
3. 修改文件，增加函数定义 **multiple_table()**:
4. 新建另外一个文件，使用 **import** 导入并且调用函数

```
import hm_01_九九乘法表

hm_01_九九乘法表.multiple_table()
```

136. 函数基本使用-01-函数定义语法

函数的定义

定义函数的格式如下：

```
def 函数名():

    函数封装的代码

    .....
```

1. **def** 是英文 **define** 的缩写
2. **函数名称** 应该能够表达 **函数封装代码** 的功能，方便后续的调用
3. **函数名称** 的命名应该 **符合标识符的命名规则**
 - 可以由 **字母**、**下划线** 和 **数字** 组成

- 不能以数字开头
- 不能与关键字重名

137. 函数的基本使用-02-第一个函数演练

函数调用

调用函数很简单的，通过 `函数名()` 即可完成对函数的调用

第一个函数演练

需求

- 编写一个打招呼 `say_hello` 的函数，封装三行打招呼的代码
- 在函数下方调用打招呼的代码

```
# 注意：定义好函数之后，只表示这个函数封装了一段代码而已
# 如果不主动调用函数，函数是不会主动执行的
def say_hello():

    print('hello 1')
    print('hello 2')
    print('hello 3')

say_hello()
```

138. 函数基本使用-03-函数的定义以及调用执行线路图

```
name = '小明'

# Python 解释器知道下方定义了一个函数
def say_hello():
    print('hello 1')
    print('hello 2')
    print('hello 3')

print(name)
```

```
# 只有在程序中，主动调用函数，才会让函数执行
say_hello()

print(name)
```

用 **单步执行 F8 和 F7** 观察以下代码的执行过程

- 定义好函数之后，只表示这个函数封装了一段代码而已
- 如果不主动调用函数，函数是不会主动执行的

139. 函数基本使用-04-应该先定义函数再调用函数

思考

- 能否将 **函数调用** 放在 **函数定义** 的上方？
 - 不能！
 - 因为在 **使用函数名** 调用函数之前，必须要保证 **Python** 已经知道函数的存在
 - 否则控制台会提示 **NameError: name 'say_hello' is not defined** (名称错误：say_hello 这个名字没有被定义)

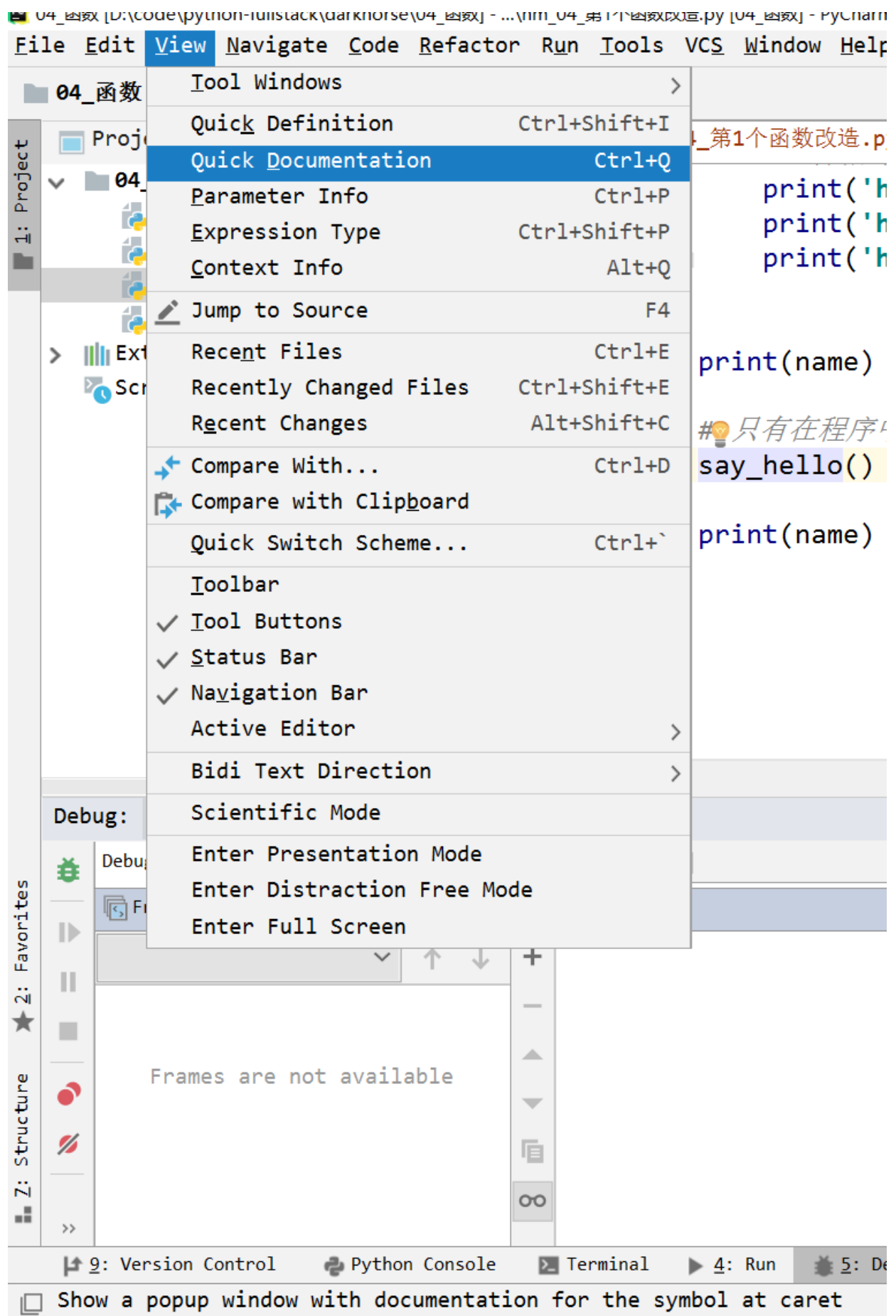
140. 函数基本使用-05-[扩展]单步越过和单步进入

- **F8 Step Over** 单步越过执行代码，会把函数调用看作是一行代码直接执行
- **F7 Step Into** 单步进入执行代码，如果是函数，会进入函数内部

141. 函数基本使用-06-函数的文档注释

函数、类定义的上方和下方要有两个空行。

查看文档，快捷键 **Ctrl + Q**



```
hm_04_第1个函数改造
n def say_hello() -> None

打招呼
```

函数的文档注释

- 在开发中，如果希望给函数添加注释，应该在 **定义函数** 的下方，使用 **连续的三对引号**
- 在 **连续的三对引号** 之间编写对函数的说明文字
- 在 **函数调用** 位置，使用快捷键 **CTRL + Q** 可以查看函数的说明信息

注意：因为 **函数体相对比较独立**，**函数定义的上方**，应该和其他代码（包括注释）保留 **两个空行**

142. 函数参数-01-没有参数的函数过于死板

演练需求

1. 开发一个 `sum_2_num` 的函数
2. 函数能够实现 **两个数字的求和** 功能

```
def sum_2_num():
    """对两个数字的求和"""

    num1 = 10
    num2 = 20

    result = num1 + num2

    print("%d + %d = %d" % (num1, num2, result))

sum_2_num()
```

思考一下存在什么问题

函数只能处理 **固定数值** 的相加

如何解决？

- 如果能够把需要计算的数字，在调用函数时，传递到函数内部就好了！

143. 函数参数-02-函数参数的使用以及作用

- 在函数名的后面的小括号内部填写 **参数**
- 多个参数之间使用 **,** 分隔

```
def sum_2_num(num1, num2):  
  
    result = num1 + num2  
  
    print("%d + %d = %d" % (num1, num2, result))  
  
sum_2_num(50, 20)
```

想看函数内部执行情况，可以在函数调用处打断点，然后 **F7** 单步进入调试。

参数的作用

- **函数**，把 **具有独立功能的代码块** 组织为一个小模块，在需要的时候 **调用**
- **函数的参数**，增加函数的 **通用性**，针对 **相同的数据处理逻辑**，能够 **适应更多的数据**
 1. 在函数 **内部**，把参数当做 **变量** 使用，进行需要的数据处理
 2. 函数调用时，按照函数定义的**参数顺序**，把 **希望在函数内部处理的数据**，**通过参数** 传递

144. 函数参数-03-形参和实参

- **形参**：**定义** 函数时，小括号中的参数，是用来接收参数用的，在函数内部 **作为变量使用**
- **实参**：**调用** 函数时，小括号中的参数，是用来把数据传递到 **函数内部** 用的

145. 函数的返回值-01-返回值的应用场景和格式

- 在程序开发中，有时候，会希望 **一个函数执行结束后**，**告诉调用者一个结果**，以便调用者针对具体的结果做后续的处理

- **返回值** 是函数 **完成工作后**，**最后** 给调用者的 **一个结果**
- 在函数中使用 `return` 关键字可以返回结果
- 调用函数一方，可以 **使用变量** 来 **接收** 函数的返回结果

注意：`return` 表示返回，后续的代码都不会被执行

```
def sum_2_num(num1, num2):  
    """对两个数字的求和"""  
  
    return num1 + num2  
  
# 调用函数，并使用 result 变量接收计算结果  
result = sum_2_num(10, 20)  
  
print("计算结果是 %d" % result)
```

【!】如果不写或者写成 `return`，则默认 `return None`

146. 函数的返回值-02-改造求和函数

```
def sum_2_num(num1, num2):  
    """对两个数字的求和"""  
  
    result = num1 + num2  
  
    # 可以使用返回值，告诉调用函数一方计算的结果  
    return result  
  
# 可以使用变量，来接收函数执行的返回结果  
sum_result = sum_2_num(10, 20)  
print('计算结果: %d' % sum_result)
```

147. 函数的返回值-03-return关键字的注意事项

注意：`return` 表示返回，后续的代码都不会被执行

148. 函数的嵌套调用-01-函数嵌套调用的执行线路图

```
def test1():  
  
    print('*' * 50)  
  
def test2():  
  
    print('-' * 50)  
  
    # 函数的嵌套调用  
    test1()  
  
    print('+ ' * 50)  
  
test2()
```

如果函数 `test2` 中，调用了另外一个函数 `test1`

- 那么执行到调用 `test1` 函数时，会先把函数 `test1` 中的任务都执行完
- 才会回到 `test2` 中调用函数 `test1` 的位置，继续执行后续的代码

149. 函数的嵌套调用-02-[扩展]用百度网站举例说明函数的调用线路

150. 分隔线演练-01-利用参数增加分隔线的灵活度

函数嵌套的演练 —— 打印分隔线

体会一下工作中 需求是多变的

需求 1

- 定义一个 `print_line` 函数能够打印 `*` 组成的 **一条分隔线**

```
def print_line():  
  
    print('*' * 50)  
  
print_line()
```

需求 2

- 定义一个函数能够打印 **由任意字符组成** 的分隔线

```
def print_line(char):  
  
    print(char * 50)  
  
print_line('-')
```

需求 3

- 定义一个函数能够打印 **任意重复次数** 的分隔线

```
def print_line(char, times):  
  
    print(char * times)  
  
print_line('hi', 40)
```

151. 分隔线演练-02-打印多条分隔线

需求 4

- 定义一个函数能够打印 **5 行** 的分隔线，分隔线要求符合需求 3

提示：工作中针对需求的变化，应该冷静思考，**不要轻易修改之前已经完成的，能够正常执行的函数！**

```
def print_line(char, times):  
  
    print(char * times)  
  
def print_lines():  
  
    row = 0  
  
    while row < 5:  
  
        print_line('-', 50)  
  
        row += 1  
  
print_lines()
```

152. 分隔线演练-03-增加多行分隔线函数的参数

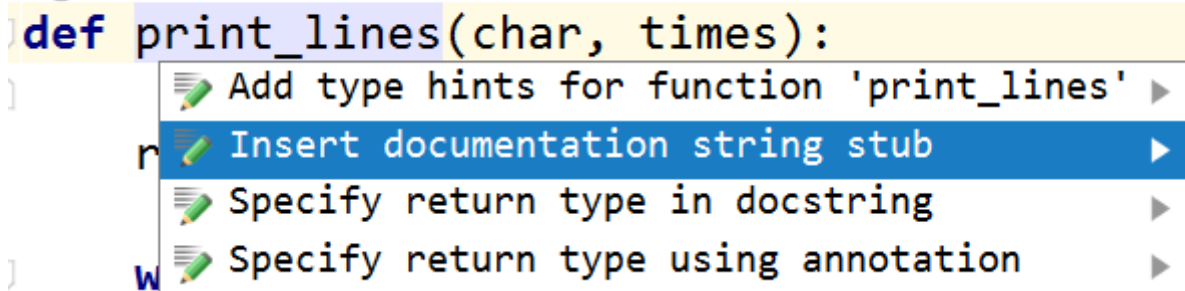
```
def print_line(char, times):  
  
    print(char * times)  
  
def print_lines(char, times):  
  
    row = 0  
  
    while row < 5:  
  
        print_line(char, times)  
  
        row += 1  
  
print_lines('-', 20)
```

153. 分隔线演练-04-[扩展]PyCharm给函数增加文档注释

Ctrl + Q 查看光标所在位置的函数文档

如何快速生成函数文档？

光标放在函数名，点击灯泡，选择第二个
或者 **Alt + Enter**，选择第二个



```
def print_line(char, times):  
    """打印单行分隔线  
  
    :param char: 分隔字符  
    :param times: 重复次数  
    """  
    print(char * times)  
  
def print_lines(char, times):  
    """打印多行分隔线  
  
    :param char: 分隔线使用的分隔字符  
    :param times: 分隔线重复的次数  
    """  
    row = 0  
  
    while row < 5:  
  
        print_line(char, times)  
  
        row += 1  
  
print_lines('-', 20)
```

然后再使用 `Ctrl + Q` 就可以查看到参数说明。

`print_lines(char, times):`


hm_09_打印多条分隔线

```
def print_lines(char: {__mul__},
                    times: Any) -> None
```

打印多行分隔线

Params: char - 分隔线使用的分隔字符

times - 分隔线重复的次数



完成于 201809290902