第 3 章

变量与名称空间

本章的主要内容如下：

- 变量的声明与赋值；
- 类型批注；
- globals 函数与 locals 函数；
- 直接更新名称空间字典；
- global 语句与 nonlocal 语句的用法。

3.1 变量与赋值

案例 59 声 明 变 量

导 语

Python 是动态语言，声明变量时不需要指定变量类型。声明变量后必须进行赋值，否则在访问变量时会引发 NameError 异常。

变量的类型决定于它的值，例如：

```
a = 1000
b = 'xyz'
c = 0.0016
```

上面代码声明了四个变量，依据它们各自引用的值，变量 a 为整型(int)，变量 b 为字符串类型(str)，变量 c 为浮点数类型(float)。

在相同的代码范围内(例如在模块级别声明的变量)，向同一名称的变量赋值并不会产生同名的新变量，而是让变量引用最新赋值的对象，并解除对前一个对象的引用。例如下面代码中，变量 k 先是引用字符串案例，当执行第二行代码后，变量 k 解除对字符串对象的引用，并与新的整数值建立引用关系(可称为“绑定”)。

```
k = 'test'
k = 200
```

声明变量时还可以赋值为 None，它是一个内置的值，表示变量缺少有效的引用，即没有值的变量。None 值所对应的 Python 的内置类型为 NoneType。None 转换为布尔类型的值为 False，因此，None 值可以用于判断语句中，例如：

```
if None:
    …
else:
    …
```

以下代码会输出“False”。

```
print(bool(None))
```

操作流程

步骤 1：声明四个变量，分别进行赋值（必须赋值）。

```
v1 = True                                    # 布尔类型
v2 = 'start'                                 # 字符串类型
v3 = 3.14159                                 # 浮点数类型
v4 = [1, 2, 3]                               # 列表类型
```

步骤 2：依次输出上述四个变量的数据类型名称。

```
print(f'v1 的数据类型:{v1.__class__.__name__}')
print(f'v2 的数据类型:{v2.__class__.__name__}')
print(f'v3 的数据类型:{v3.__class__.__name__}')
print(f'v4 的数据类型:{v4.__class__.__name__}')
```

__class__属性可以获取指定对象的数据类型信息（type 类封装类型信息），再通过__name__属性获取类型名称的字符串表示方式。__class__与__name__都属于有特殊用途的属性。

注意：并非所有对象都有__name__属性，可以先调用 hasattr(obj, '__name__')进行检查，如果存在__name__属性，则 hasattr 函数返回 True，否则返回 False。

步骤 3：案例代码执行后，输出结果如下：

```
v1 的数据类型:bool
v2 的数据类型:str
v3 的数据类型:float
v4 的数据类型:list
```

案例 60　类 型 批 注

导　语

在声明变量时，可以显式地批注其应当使用的数据类型。例如：

```
num: int = 30
```

通过批注，表明变量 num 为 int 类型，赋值时应该使用整数值。不过，类型批注并不会影响 Python 语言的动态性——如果所赋的值不是 int 类型，在运行阶段是不会发生错误的。就像这样：

```
num: int = 'table'
```

尽管批注了 int 类型，使用字符串类型的案例进行赋值，也不会影响代码的执行。类型批注仅用于提示开发人员，尽可能为变量设置恰当的值。

操作流程

步骤 1：声明一个变量 xa，并批注为 str(字符串)类型，但赋值时使用 int(整数)类型的值。

```
xa: str = 1000
```

步骤 2：在屏幕上打印 xa 变量实际引用的类型。

```
print(f'变量的实际类型:{xa.__class__.__name__}')
```

步骤 3：打印当前模块中__annotations__属性的内容。

```
print(f'变量的批注类型:{__annotations__}')
```

变量的批注信息会存储到__annotations__属性中，以字典的形式记录变量与所批注的类型。

步骤 4：运行案例代码，屏幕输出内容如下：

```
变量的实际类型:int
变量的批注类型:{'xa': <class 'str'>}
```

案例 61　声明语句也是变量赋值

导　语

Python 代码在执行声明函数(使用 def 语句)和类(使用 class 语句)时，实际上也进行了变量赋值。函数或类的名称就是变量的名称，而变量引用的是函数或类的类型案例。在

Python 中，函数（包括 lambda 表达式）、类型（由 type 类描述）、方法等都被视为对象。

例如，下面代码声明了函数 test。

```
def test():
    pass
```

在模块代码运行时，无论 test 函数是否被调用，def 关键字所在的一行（即声明语句）都会被执行，而函数体（上面例子中的 pass 语句就是函数体）代码只有在函数被调用时才会执行。声明语句执行后，会在当前（模块）的名称空间中生成一个名为 test 的变量（与 test 函数名称相同），并把 test 函数的引用赋值给 test 变量。此过程类似于

```
test = test
```

赋值运算符（=）右边的 test 是函数引用，左边是变量名称，只是变量名与函数名相同而已。因此，还可以用其他变量去引用 test 函数，例如：

```
oth_test = test
```

赋值后，可以这样调用 test 函数

```
oth_test()
```

类似于为函数起一个别名。

对于类，也是相同的原理。

```
class A:
    pass

so_a = A
```

当案例化 A 类时，可以这样

```
var = so_a()
```

so_a 是个变量，它引用 A 类的类型对象，而 so_a()会调用 A 类的构造函数创建 A 的案例，再把 A 的案例赋给另一个变量 var。

操作流程

步骤 1：声明一个函数，命名为 num_sqr。

```
def num_sqr(x, n):
    return x ** n
```

对于当前模块的名称空间而言，此时全局变量中会存在一个名为 num_sqr 的变量，它引用了 num_sqr 函数，即变量名称与函数名称相同。

步骤 2：再声明一个变量，让它也引用 num_sqr 函数。

```
sqr_c = num_sqr
```

步骤 3：分别向屏幕打印 num_sqr 和 sqr_c 变量的值。

```
print(f'变量 num_sqr 所引用的对象:\n{num_sqr}')
print(f'变量 sqr_c 所引用的对象:\n{sqr_c}')
```

当这两行代码执行后，屏幕上会输出以下内容：

```
变量 num_sqr 所引用的对象:
<function num_sqr at 0x0000015F6FD580D0>

变量 sqr_c 所引用的对象:
<function num_sqr at 0x0000015F6FD580D0>
```

从输出结果中可以看到，两个变量所引用的对象具有相同的内存地址，表明它们都引用了同一个案例——num_sqr 函数。

步骤 4：下面代码获取在全局变量中引用了 num_sqr 函数的所有变量。

```
d = {k: v for k, v in globals().items() if v is num_sqr}
print('全局名称空间中,引用了 num_sqr 函数的变量有:')
for var, func in d.items():
    print(f'{var}: {func}')
```

调用 globals 函数返回存储全局变量的字典，并筛选出 value 为 num_sqr 对象引用的子项(通过串联的 if 语句)。

得到结果如下：

```
全局名称空间中,引用了 num_sqr 函数的变量有:
num_sqr: <function num_sqr at 0x0000015F6FD580D0>
sqr_c: <function num_sqr at 0x0000015F6FD580D0>
```

案例 62　as 关键字与赋值

在 import…或者 from … import …语句后面，都可以使用 as 关键字对导入的对象进行重新命名(分配别名)。例如：

```
from some_module import fun1 as fun2
```

从 some_module 模块中导入 fun1 函数，然后用 as 关键字分配一个别名——fun2。此过程类似于声明了新的变量 fun2 并让它引用 fun1 函数。或者为导入的模块分配一个别名

```
import some_module as new_name
```

此过程相当于声明新的变量 new_name，然后引用导入的 some_module 模块。

操作流程

步骤 1：新建 test_mod 模块，在模块中声明 trunc_str 函数。

```
def trunc_str(s, n = 8):
    # 获取字符串长度
    ln = len(s)
    # 如果字符串长度小于或等于 n,直接返回原字符串对象
    if ln <= n:
        return s
    # 如果字符串长度大于 n,就截断字符串
    ns = s[:n]
return ns + '…'
```

trunc_str 函数的功能是截断字符串，当字符串 s 的长度小于或者等于 n 时，不做处理，将原字符串返回；当字符串 s 的长度大于 n 时，截断字符串，保留 n 个字符，并在新字符串的末尾加上省略号。

步骤 2：在顶层代码模块中，导入 trunc_str 函数，然后使用 as 关键字分配一个别名 truncateStr。

```
from test_mod import trunc_str as truncateStr
```

步骤 3：随后通过变量 truncateStr 就可以访问 trunc_str 函数。

```
print(truncateStr('一二三四', 6))
print(truncateStr('曾经沧海难为水,除却巫山不是云',7))
```

步骤 4：运行案例代码，屏幕输出结果如下：

```
一二三四
曾经沧海难为水……
```

案例 63　组合赋值法

导　语

Python 允许在一个赋值语句中同时初始化多个变量。例如：

```
x, y, z = 7, 8, 9
```

相当于

```
x = 7
y = 8
z = 9
```

实际上，此赋值行为是对元组类型(tuple)的解包，即

```
x, y, z = (7, 8, 9)
```

赋值运算符(=)右边是元组对象，一对小括号可以省略。赋值语句执行时，会从左到右依次提取出元组中的元素，逐个赋值给左边的变量列表。如上述代码中，7 赋值给变量 x，8 赋值给变量 y，9 赋值给变量 z。

但下面代码会发生错误。

```
a, b = 1, 2, 3
```

因为用于赋值的元组有 3 个元素，但接收的变量只有两个，在数量上不匹配。若是希望把 1 赋值给变量 a，然后将剩余的 2、3 赋值给变量 b，可以这样写

```
a, *b = 1, 2, 3
```

赋值后，a 引用整数值 1，b 引用了一个列表对象，列表中包含 2、3 两个整数值。

操作流程

步骤 1：用两个浮点型数值来初始化两个变量。

```
v1, v2 = 0.002, 3.0025
```

步骤 2：用于初始化的变量值也可以是不同类型的。

```
m1, m2 = b'\x6e', 800
```

步骤 3：用 4 个字符串对象案例化 3 个变量，其中，s1、s2 依次引用前两个元素，sn 引用剩下的两个元素。

```
s1, s2, *sn = 'abc', 'opq', 'rst', 'lmn'
```

步骤 4：输出各个变量的值。

```
print(f'v1: {v1}\nv2: {v2}')
print(f'm1: {m1}\nm2: {m2}')

print(f's1: {s1}')
print(f's2: {s2}')
print(f'sn: {sn}')
```

步骤 5：运行案例代码，输出结果如下：

```
v1: 0.002
v2: 3.0025
m1: b'n'
```

```
m2: 800
s1: abc
s2: opq
sn: ['rst', 'lmn']
```

注意：sn 变量使用列表类型(list)来存储剩余的两个变量值，之所以使用列表类型，是为了允许代码修改 sn 变量中所包含的元素。

案例 64　组合赋值与表达式列表

导　语

组合赋值语句的实质是在赋值运算符(=)的两边各放置一个表达式列表。常规情况下，赋值运算符两边的表达式个数应当相等，并且从左到右依次赋值。例如：

```
v1, v2, v3 = 'a', 'b', 'c'
```

赋值运算符两边的表达式列表实为元组对象，因此，上述代码可以写成

```
(v1, v2, v3) = ('a', 'b', 'c')
```

或者，还可以这样写

```
v1, v2, v3, = 'a', 'b', 'c',                    # 两个表达式列表末尾都有一个逗号
```

使用列表(list)的格式来进行组合赋值也是允许的，例如：

```
[v1, v2, v3] = ['a', 'b', 'c']
```

以下赋值语句，左边是列表格式，右边是元组格式，这种情况也是允许的。

```
[a, b, c] = ('a', 'b', 'c')
```

如果赋值运算符两边的元素个数不相等，会发生语法错误。

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

第一行代码中，左边有两个变量，而右边有三个值；第二行代码则相反，左边有三个变量，右边有两个值。

变量名称前可以加上星号(*)，表示该变量接收多个值。这些值是从序列对象(列表、元组等)中“解包”(unpack)出来的，然后组合为新的序列赋值给带星号的变量。请看下面代码。

```
a, *b = 'a', 'b', 'c'
```

上面代码先将字符“a”赋值给变量 a，再把剩下的字符“b”和“c”组成新的列表赋值给变量 b。把顺序反过来也是可行的。

```
*a, b = 'a', 'b', 'c'
```

此时，变量 a 前面带有星号，赋值过程为：先将最后一个字符“c”赋值给变量 b，再将剩下的两个字符“a”和“b”组成新的列表赋值给变量 a。

但是，同一个变量列表中不能出现多个带星号的变量，下面语句会发生错误。

```
*a, *b = 'a', 'b', 'c', 'd'
```

因为带星号的变量所引用的值列表在元素个数上是不确定的，同时使用多个带星号的变量，会导致赋值运算符右边的值列表无法进行准确的分配。

如果赋值运算符左边只有一个变量，也是可以使用带星号的变量的，以下三条语句都是有效的。

```
*x, = 1, 2, 3
(*x) = 1, 2, 3
[*x] = 1, 2, 3
```

第一行与第二行都是把变量列表组成元组对象，由于第一行中省略了小括号，所以变量后面要加上一个逗号，表示 x 不是单个值，而是一个元组。第三行代码中变量 x 表示一个列表对象。

综上所述，不管是赋值语句的左边还是右边，表达式列表可以包装为元组(tuple)和列表(list)两种序列格式。

操作流程

步骤 1：声明 n1、n2、n3 三个变量，并进行组合赋值。

```
n1, n2, n3 = 100, 101, 103
```

此赋值语句中，赋值运算符两边的表达式列表均为元组结构。

步骤 2：声明 m1、m2、m3 变量，并用五个浮点数值赋值。

```
m1, *m2, m3 = 0.1, 0.2, 0.3, 0.4, 0.5
```

此语句先将数值 0.1 赋值给 m1 变量，再把 0.5 赋值给 m3 变量，剩下的 0.2、0.3、0.4 三个数值组成新的列表赋值给 m2 变量。

步骤 3：将字符串案例赋值给变量 q。

```
[*q] = 'abcdefg'
```

此语句与 q = 'abcdefg' 语句不同，如果 q 前面不带星号，那么 q 直接引用字符串案例。而上面代码中变量 q 前有星号，而且放到一对中括号中，表明 q 接收的是字符列表（即将字符串“abcdefg”，解包为由“a”“b”“c”“d”“e”“f”“g”七个单独字符组成的列表）。

步骤 4：在屏幕上打印上述各个变量的值。

```
print(f'n1: {n1}\nn2: {n2}\nn3: {n3}\n')
print(f'm1: {m1}\nm2: {m2}\nm3: {m3}\n')
print(f'q: {q}')
```

步骤 5：运行案例代码，屏幕将输出以下内容：

```
n1: 100
n2: 101
n3: 103

m1: 0.1
m2: [0.2, 0.3, 0.4]
m3: 0.5

q: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

3.2　名称空间

案例 65　获取全局名称空间的字典

导　语

在 Python 中，变量以及它所引用的对象都以字典格式存储在内存中，并且由 Python 进程动态维护。当新的变量产生时会实时地将此变量作为新记录添加到字典中；当变量被销毁（使用 del 语句删除变量或应用程序退出）后会从字典中将此变量记录删除。

依据名称空间的范围，Python 程序在运行后会产生两种类型的名称空间字典——全局字典与局部字典。在模块级别定义的所有变量（标识符）都会存放到全局字典中，而局部字典常存在于函数或者类的内部——即存放函数（或类）内部所定义的变量。

globals 函数可以获取存放全局变量的字典引用。

```
vardic = globals()
```

不过，以迭代方式访问字典元素时会发生错误，例如 for 循环。

```
for name,value in vardic.items():
    print(f'{name}: {value}')
```

在对字典进行迭代的过程中，由于 for 语句产生了新的变量 name 和 value，存放全局变

量的字典被更新，从而引发错误。解决方法有以下 3 种。

第一种方法是调用字典案例的 copy 方法复制出一个新的字典案例，因为这个案例不是 globals 函数所返回的案例，在全局变量发生变更时不会被更新，所以能顺利完成 for 循环。代码如下：

```
vardic = globals()
other_dict = vardic.copy()
for name,value in other_dict.items():
    print(f'{name}: {value}')
```

第二种方法是使用推导语句来从原来的字典案例中产生新的字典案例，因为执行推导语句不会在全局名称空间中出现新的变量。

```
vardic = globals()
other_dict = {k:v for k,v in vardic.items()}
for name,value in other_dict.items():
    print(f'{name}: {value}')
```

第三种方法与推导句式类似，即通过对原字典对象进行解包操作，然后再组成一个新的字典案例。字典案例的解包运算符是两个星号（**）。

```
vardic = globals()
other_dict = { **vardic}
for name,value in other_dict.items():
    print(f'{name}: {value}')
```

操作流程

步骤 1：声明四个变量并初始化。

```
var1 = 'label'
var2 = None
var3 = 3500
var4 = True
```

步骤 2：定义两个函数。

```
def func_1():
    pass

def func_2():
    pass
```

步骤 3：定义一个类。

```
class members:
    pass
```

步骤 4：获取存放全局变量的字典的引用。

```
gdic = globals()
```

步骤 5：使用推导句式重新构造一个字典案例，并且过滤掉名称中以下画线开头的变量。

```
_dic = {key: value for key, value in gdic.items() if not key.startswith('_')}
```

推导语句可以将 for 语句与 if 语句串联使用。首先以 for 语句从原来的字典中提取一个子项，然后再用 if 语句后的条件验证一下变量名是否以下画线开头，如果不以下画线开头就把 key 和 value 组成“键/值”对添加到新的字典案例中。

步骤 6：在屏幕上输出变量名和变量值。

```
for name in _dic:
    print(f'{name}: {_dic[name]}')
```

步骤 7：运行案例代码，输出结果如下：

```
var1: label
var2: None
var3: 3500
var4: True
func_1: <function func_1 at 0x000002F304486BF8>
func_2: <function func_2 at 0x000002F3047FCE18>
members: <class '__main__.members'>
gdic: {…}
```

注意：不管是在模块级别调用 globals 函数，还是在函数(或类)内部调用 globals 函数，该函数始终返回全局名称空间的字典。

案例 66　获取局部名称空间的字典

导　语

locals 函数的功能(与 globals 函数相对)是返回局部名称空间中存放变量的字典案例。当 locals 函数在模块级别调用时，它返回的内容与 globals 函数相同。

locals 函数一般在函数或者类的内部调用，以获取局部的字典案例引用。例如下面代码定义了一个 work 函数，在函数内声明了三个变量，然后调用 locals 函数获取存放这些变量的字典案例。

```
def work():
    x = 100
```

```
    y = 200.03
    z = b'6fc3'
    dic = locals()
    # 解包原字典,组成新的字典案例
    # 避免因原字典动态更新而导致 for 语句出错
    _d = { ** dic}
    for k, v in _d.items():
        print(f'{k}: {v}')
```

存放局部变量的字典与全局字典一样,会实时更新,因此,避免在原来的字典案例上直接使用 for 循环,需要复制一个新的案例再进行迭代运算。

在类(class)内部调用 locals 函数所返回的字典案例中,只会包含用户代码中显式声明过的变量(不包括__module__、__qualname__等特殊成员),动态属性以及存储动态属性的__dict__成员也没有包含在字典中。

操作流程

步骤 1:定义 test 函数,并在函数内部声明四个变量和一个嵌套函数。

```
def test():
    item1 = b'f36e03'
    item2 = 'the str'
    item3 = 0.00025
    item4 = 95
    # 嵌套函数
    def inner():
        pass

    _dic = { ** locals()}

    print('test 函数中的局部变量:')
    for key in _dic:
        print(f'{key}: {_dic[key]}')
```

声明完变量和函数后,调用 locals 函数获取存放变量的字典案例,接着输出字典中的内容。

步骤 2:尝试调用 test 函数。

```
test()
```

步骤 3:调用 test 函数后,输出以下信息:

```
item1: b'f36e03'
item2: the str
item3: 0.00025
```

```
item4: 95
inner: <function test.<locals>.inner at 0x0000024D022DA0D0>
```

步骤 4：定义一个 demo 类，在类中声明两个数据成员和一个案例方法。

```
class demo:
    m_1 = 1
    m_2 = 2
    def do_something(self):
        pass
    # 获取局部变量列表
    _dic = {**locals()}
```

从 locals 函数返回的结果中获取的字典案例由_dic 成员引用，以便在类外部进行访问。

步骤 5：案例化 demo 类，并设置两个动态属性(a 和 b)。

```
v = demo()
v.a = 0.2
v.b = 0.75
```

步骤 6：使用 for 循环输出_dic 成员中的子项。

```
for k, v in v._dic.items():
    print(f'{k}: {v}')
```

输出结果如下：

```
__module__: __main__
__qualname__: demo
m_1: 1
m_2: 2
do_something: <function demo.do_something at 0x0000024D022DA158>
```

可以看出，动态属性 a、b 并没有存放到局部名称空间的字典中(动态属性存放到__dict__成员中，它也是字典结构)，而且，连存放动态属性的__dict__成员也没有被添加到局部名称空间的字典中。除了特殊成员__module__和__qualname__外，局部字典中只包含代码中明确声明过的 m_1、m_2、do_something 三个成员。

案例 67　直接更新名称空间字典

导　语

由于 Python 应用程序是通过动态维护全局或局部字典案例来存储变量的，因此，直接修改字典案例来管理变量也是可行的。尽管这种做法在代码阅读上显得不太友好，但有时候还是有实用价值的——例如以编程方式动态生成变量列表。

本案例将以编程方式向全局变量字典添加三条记录，实际效果会产生三个新的变量，随后可以在代码中直接访问这些变量。

操作流程

步骤 1：获取存储全局变量的字典案例引用。

```
g_dic = globals()
```

步骤 2：向字典案例添加三条记录。

```
g_dic['var_x'] = 'coding'
g_dic['var_y'] = 6000
g_dic['var_z'] = b'af7c32'
```

此举相当于声明了三个变量 var_x、var_y 和 var_z，并完成初始化。

步骤 3：将三个变量的值打印到屏幕上。

```
print(f'var_x: {var_x}')
print(f'var_y: {var_y}')
print(f'var_z: {var_z}')
```

从代码上看，这三个变量似乎未经过声明就访问了，但它们确实已经存在于全局变量字典中，因此在运行阶段是能够被访问的。

步骤 4：运行案例代码，屏幕输出内容如下：

```
var_x: coding
var_y: 6000
var_z: b'af7c32'
```

案例 68　使用 global 关键字声明变量

导　语

Python 的名称空间存在全局与局部两个范围，每个范围都有存在独立的字典案例来管理变量。不同名称空间下的变量字典具有相对独立性，比较典型的问题如下面代码所示。

```
ch = 'ab'                    # 全局变量

def change():
    ch = 'cd'

change()                     # 调用函数
print(f'ch: {ch}')
```

在 change 函数中向 ch 变量赋值字符串“cd”，我们期望的输出结果是：

```
ch: cd
```

然而上述代码实际输出结果是：

```
ch: ab
```

也就是说，change 函数调用后，ch 变量并没有被更新。

问题的根源是模块级别声明的 ch 与 change 函数内部声明的 ch 不是同一个变量，尽管它们的名称相同，但它们分属不同的名称空间范围。模块级别的 ch 变量存储在全局变量字典中，而 change 函数内的 ch 变量只存储于函数内的局部变量字典中。

此现象是因 Python 语言的动态特性而产生的。在 Python 代码中声明变量不需要指定类型，所以赋值语句直接完成变量的声明。若要让函数内部的 ch 变量与模块中的 ch 变量合为同一个变量，那么在 change 函数中要先用 global 关键字声明一下变量，然后再赋值。即

```
def change():
    global ch
    ch = 'cd'
```

这时，change 函数内部的变量字典中不会添加 ch 变量的记录，ch = 'cd' 修改的是模块级别的 ch 变量。在 change 函数调用之后，ch 变量的值就会变成“cd”。

不过，如果不修改变量的值（仅仅是读取），此种情况下，函数内部是不需要使用 global 关键字的。例如：

```
label = 'news'                # 声明全局变量

def output():
    # 访问全局变量
    print(f'label: {label}')

output()                      # 调用函数
```

output 函数内部只是读取变量的值，没有使用赋值语句，就不会产生新的变量，所以它所访问的就是全局的 label 变量。

操作流程

步骤 1：在代码模块中声明 number 变量，初始化为 10。

```
number = 10
```

步骤 2：定义两个函数，依次将 number 变量的值加上 20 和 30。

```
def add_20():
    global number
```

```
    number += 20

def add_30():
    global number
    number += 30
```

为了能顺利修改全局变量 number 的值，在函数内部需要使用 global 关键字再次声明 number 变量，以表明修改的是全局变量而不是声明局部变量。

步骤 3：依次调用上述两个函数。

```
add_20()
add_30()
```

步骤 4：打印 number 变量的值。

```
print(f'number: {number}')
```

步骤 5：运行案例代码，输出结果如下：

```
number: 60
```

案例 69　使用 nonlocal 关键字声明变量

导　语

nonlocal 关键字的用法与 global 关键字相似，但 nonlocal 关键字一般用于嵌套函数中（即函数内包含函数）。例如：

```
def test():
    nvar = 1
    def inner():
        nvar = 5
    inner()                    # 调用内嵌函数
    return nvar                # 返回变量的值

print(test())
```

nvar 变量是在 test 函数中声明的，嵌套的 inner 函数试图将 nvar 变量的值修改为 5。但调用 test 函数后，返回的值仍然是 1，而不是 5。原因在于 inner 函数中声明了新的局部变量 nvar，被赋值的是这个局部变量而不是 test 函数中的 nvar 变量。解决方法是在 inner 函数中用 nonlocal 关键字声明 nvar 变量。

```
def test():
    nvar = 1
```

```
    def inner():
        nonlocal nvar
        nvar = 5
    inner()                    # 调用内嵌函数
    return nvar                # 返回变量的值
```

操作流程

步骤 1：定义一个 somework 函数，在函数内声明变量 a，并以字符串案例初始化。然后在此函数内部定义两个函数——inner1 和 inner2，分别修改变量 a 的值。接着调用 inner1 和 inner2 两个函数，再把变量 a 的值返回。

```
def somework():
    a = 'a'
    def inner1():
        nonlocal a
        a = 'b - ' + a
    def inner2():
        nonlocal a
        a = 'c - ' + a
    # 调用以上两个函数
    inner1()
    inner2()
    # 返回变量的值
return a
```

为了能让嵌套的函数内部可以顺利修改变量 a 的值，需要使用 nonlocal 关键字进行一次声明。

步骤 2：调用 somework 函数，并打印它返回的内容。

```
print(somework())
```

步骤 3：运行案例代码，屏幕输出结果如下：

```
c - b - a
```