#### 1.函数

前面我们学习了python的基础语法，掌握了判断和循环结构，目前为止，我们可以编写出大部分代码，但是如果我们从头往下写代码，你加入你有一个需求，你有一段代码在某个过程中需要执行1000遍，你真的把这个代码写1000遍吗？

**重点**：理解函数，会定义函数，会调用函数

**函数的作用**：函数是一种可重复使用的代码块，用于执行特定的任务。函数将一组相关的语句封装在一起，可以通过给定的名称进行调用，并且可以接受参数并返回值。函数可以帮助我们组织代码、提高代码的可读性和可维护性。

#### 2.定义函数

在 Python 中，使用 def 关键字来定义函数。格式如下：

def function\_name(parameters):  
 """函数的文档字符串"""  
 # 函数体代码  
 return expression

重点解析：首先需要写一个def；然后定义函数名，函数名需要满足标识符规格；紧接着要跟一个参数列表，参数列表可以为空，也可以有多个，多个之间使用，隔开；要有函数体，函数体可以使用pass代替；最后可以有返回值，也可以没有。

相信def和函数名大家都会写，我们下面分开介绍参数列表，函数体和返回值。

##### 2.1.参数列表

参数列表是函数定义中用于接收传递给函数的值的一组参数。在函数定义中，可以指定参数列表，以便函数在被调用时接受特定的值。参数列表中的参数是函数在执行时使用的局部变量。

1. 位置参数（Positional Arguments）：按照定义的顺序传递给函数的参数。在函数被调用时，传递的值将按照参数列表中的位置与参数一一对应。  
def sub(x,y):  
 return x-y  
print(sub(3,4))  
print(sub(4,3))  
  
2. 关键字参数（Keyword Arguments）：通过参数名称来传递值，可以不按照定义的顺序传递。关键字参数提供了更清晰的代码，因为函数调用时可以指定参数的名称。  
def sub(x,y):  
 return x-y  
print(sub(y=3,x=4))  
  
3. 默认参数（Default Arguments）：在函数定义中为参数指定默认值。如果没有提供对应参数的值，将使用默认值。默认参数使得函数调用更加灵活，可以选择性地提供部分参数的值。  
def helloHuahai(name="huahai"):  
 print(f"hello {name}")  
helloHuahai("Huahua Gou")  
helloHuahai()  
  
4. 可变数量的位置参数（Variable Number of Positional Arguments）：使用 `\*args` 语法，允许函数接受任意数量的位置参数。在函数内部，`args` 是一个包含所有位置参数的元组。（不常用）  
  
5. 可变数量的关键字参数（Variable Number of Keyword Arguments）：使用 `\*\*kwargs` 语法，允许函数接受任意数量的关键字参数。在函数内部，`kwargs` 是一个包含所有关键字参数的字典。（不常用）  
  
def greet(name, age=19, \*languages, \*\*skills):  
 print("Name:", name)  
 print("Age:", age)  
 print("Languages:", languages)  
 print("Skills:", skills)  
  
greet("Alice", 25, "English", "French", Python=5, Java=3)  
  
输出结果：  
Name: Alice  
Age: 25  
Languages: ('English', 'French')  
Skills: {'Python': 5, 'Java': 3}

通常我的函数定义是，使用关键字定义，然后给关键字添加默认值，后面调用函数使用关键字的方式进行调用。

##### 2.2.函数体

函数体是在函数定义中包含的一组语句，用于指定函数在被调用时要执行的操作。函数体是函数的实际执行部分，它包含了函数的逻辑和算法。

* 函数体必须在函数定义中缩进，并且通常使用四个空格进行缩进。Python 语言依赖于缩进来定义代码块，因此正确的缩进对于函数体的执行非常重要。
* 函数体中可以包含任意数量的语句，用于执行特定的任务。这些语句可以是赋值语句、条件语句、循环语句、函数调用等，以及其他 Python 支持的语句。
* 函数体中可以使用参数和局部变量来处理数据。参数是函数在调用时接收的值，而局部变量是函数内部定义的变量，其作用范围仅限于函数体内部。
* 函数体中可以使用控制流语句（如条件语句和循环语句）来实现不同的逻辑和算法。这些控制流语句可以根据条件决定执行的代码路径，或者重复执行一部分代码。
* 函数体中可以使用 return 语句来指定函数的返回值。return 语句用于将结果传递给函数的调用者，并将函数的执行控制权返回给调用者。

def add\_numbers(a, b):  
 result = a + b  
 if result > 10:  
 print("The result is greater than 10")  
 else:  
 print("The result is less than or equal to 10")  
 return result  
  
sum\_result = add\_numbers(5, 7)  
print("Sum:", sum\_result)  
  
输出结果：  
The result is less than or equal to 10  
Sum: 12

函数体可以暂时使用pass来定义。

##### 2.3.返回值

返回值是函数执行完毕后，通过 return 语句将结果返回给函数的调用者。返回值可以是任意类型的数据，如整数、浮点数、字符串、列表、字典等。

* 返回值是函数执行结果的一部分，它是函数与外部环境之间的一种通信方式。返回值允许函数将计算结果传递给调用者，以便后续的处理或使用。
* 函数可以返回单个值，也可以返回多个值。多个返回值可以使用元组、列表或字典等数据结构来表示。
* 函数可以有多个 return 语句，但一旦执行到某个 return 语句，函数将立即结束，并将指定的值返回给调用者。后续的语句将不会被执行。

def addNumbers(a, b):  
 result = a + b  
 return result  
sumResult = add\_numbers(5, 7)  
print("Sum:", sumResult)  
  
输出结果：  
Sum: 12  
  
  
下面是一个函数返回多个值的示例：  
def divide\_numbers(a, b):  
 quotient = a / b  
 remainder = a % b  
 return quotient, remainder  
  
result = divide\_numbers(10, 3)  
print("Quotient:", result[0])  
print("Remainder:", result[1])  
  
输出结果：  
Quotient: 3.3333333333333335  
Remainder: 1  
  
在上面的示例中，`divide\_numbers()` 函数接受两个参数，并使用除法运算符计算商和余数。然后，通过 `return` 语句返回这两个值。在函数调用中，返回的结果被赋给 `result` 变量，它是一个包含两个元素的元组。通过索引访问元组中的各个元素，并将它们打印出来。

#### 3.调用函数

调用函数时，要着重注意参数问题，首先参数数量上要匹配，其次参数的数据类型上要匹配。

def calculate\_product(a, b, c):  
 return a \* b \* c  
  
result = calculate\_product(2, 3)  
print("Product:", result)  
运行上面的代码会产生一个错误：  
TypeError: calculate\_product() missing 1 required positional argument: 'c'  
错误消息指出，`calculate\_product()` 函数缺少一个必需的位置参数 `'c'`。这是因为在函数调用中，我们只提供了两个参数值，而函数期望接收三个参数。  
  
  
为了解决这个问题，我们需要在函数调用中提供正确数量的参数值。  
def calculate\_product(a, b, c):  
 return a \* b \* c  
  
result = calculate\_product(2, 3, 4)  
print("Product:", result)  
输出结果：  
Product: 24

当函数期望接收特定数据类型的参数，但实际传入的参数数据类型不匹配时，可能会导致错误或意外结果。  
def calculate\_square(number):  
 return number \*\* 2  
  
result = calculate\_square("5")  
print("Square:", result)  
  
在上述示例中，函数 `calculate\_square()` 期望接收一个数字作为参数，并计算该数字的平方。然而，在函数调用中，我们传入了一个字符串 `"5"` 作为参数。  
  
这将导致以下错误：  
TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'  
  
  
错误消息指出，字符串类型和整数类型之间不支持平方运算。因此，无法将字符串 `"5"` 作为参数传递给函数 `calculate\_square()`。  
  
def calculate\_square(number):  
 return number \*\* 2  
  
result = calculate\_square(5)  
print("Square:", result)  
  
输出结果：  
Square: 25

#### 4.变量作用域

变量的作用域是指变量在程序中可访问的范围。作用域规定了变量在不同部分或层次结构中的可见性和可用性。在不同的作用域内，变量可能具有不同的生命周期和可访问性。

* 全局作用域（Global scope）：全局作用域中定义的变量可以在整个程序中访问。这意味着变量在函数内部和外部都是可见的。如果在函数外部定义了一个变量，则该变量在整个程序中可见，称为全局变量。全局变量在程序的任何地方都可以使用。
* 局部作用域（Local scope）：局部作用域中定义的变量只能在其所在的特定代码块（如函数、循环或条件语句）内部访问。这些变量在代码块执行结束后会被销毁，无法在外部访问。如果在函数内部定义了一个变量，则该变量在函数内部可见，称为局部变量。局部变量只在函数内部有效，函数执行结束后会被销毁。

**如果在函数内部使用了与全局作用域中同名的变量，则函数内部的变量会覆盖全局变量。这被称为变量的屏蔽（shadowing）。**

def my\_function():  
 x = 5 # 局部变量  
  
 def inner\_function():  
 y = 20 # 局部变量  
 print("Inner:", x, y)  
  
 inner\_function()  
 print("Outer:", x)  
x=10  
print("out:",x)  
my\_function()  
print("out:",x)  
  
输出结果：  
out: 10  
Inner: 5 20  
Outer: 5  
out: 10

def my\_function():  
 global x  
 x=5 # 局部变量  
 def inner\_function():  
 global y  
 y = 20 # 局部变量  
 print("Inner:", x, y)  
  
 inner\_function()  
 print("Outer:", x)  
x=10  
print("out:",x)  
my\_function()  
print("out:",x)  
print("global:",y)  
  
输出结果：  
out: 10  
Inner: 5 20  
Outer: 5  
out: 5  
global: 20

#### 5.lambda函数（不常用）

Lambda函数是一种匿名函数，也称为内联函数。它是一种在需要函数对象的地方**快速定义函数**的方法，而无需使用def关键字来定义常规函数。

以下是Lambda函数的一般语法：

lambda arguments: expression

Lambda函数的主要特点是它们是单行函数，并且可以在一行代码中定义和使用。它们通常用于需要一个简单的函数，而不想定义一个完整的函数来实现它。

下面是一个使用Lambda函数的简单例子，它接收两个参数并返回它们的和：

addition = lambda x, y: x + y  
result = addition(3, 5)  
print("Result:", result)

输出结果：

Result: 8

在上述示例中，我们使用Lambda函数定义了一个函数对象addition，它接收两个参数x和y，并返回它们的和。然后，我们通过调用addition(3, 5)来使用这个Lambda函数并计算结果，将结果存储在变量result中。最后，我们打印出结果。

Lambda函数的优点之一是可以直接在需要函数对象的地方使用它，而不必将其分配给变量。例如，你可以在map()、filter()和reduce()等函数中使用Lambda函数来进行快速的迭代、筛选和聚合操作。

##### 5.1lambda+map()

以下是一个使用Lambda函数和map()函数的例子，它将一个列表中的每个元素加倍：

numbers = [1, 2, 3, 4, 5]  
doubled\_numbers = list(map(lambda x: x \* 2, numbers)) #将map()函数返回的可迭代对象转换为列表  
print("Doubled Numbers:", doubled\_numbers)

输出结果：

Doubled Numbers: [2, 4, 6, 8, 10]

在这个示例中，我们使用map()函数将Lambda函数应用于列表numbers中的每个元素。Lambda函数将每个元素加倍，并返回一个新的列表doubled\_numbers，其中包含加倍后的结果。

##### 5.2lambda+filter()

使用Lambda函数和filter()筛选出符合条件的元素：

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
even\_numbers = list(filter(lambda x: x % 2 == 0, numbers)) #将filter()函数返回的可迭代对象转换为列表  
print("Even Numbers:", even\_numbers)

输出结果：

Even Numbers: [2, 4, 6, 8, 10]

##### 5.3lambda+reduce()

使用Lambda函数和reduce()计算列表中的累积值：

from functools import reduce  
  
numbers = [1, 2, 3, 4, 5]  
product = reduce(lambda x, y: x \* y, numbers)  
print("Product:", product)

输出结果：

Product: 120

使用Lambda函数和reduce()函数计算了列表numbers中所有元素的乘积。Lambda函数接收两个参数x和y，并返回它们的乘积。reduce()函数将Lambda函数应用于列表中的每对元素，不断地将结果累积，直到计算出最终的乘积。