Python基础语法文档以及配套练习（三）

Author: Limzh

从第一天的`规范性`和`快捷性`，我们初探Python编程特性;再到第二天的`变量机制`和`作用域机制`，我们已经为理解Python最核心概念，`层次性`或者更加准确地用术语说，`面向对象性`打下了很好的铺垫。昨天我们还额外介绍了`鸭子类型`的概念，即:如果一个对象它具有鸭子的属性和鸭子的方法，那么它无论逻辑上是什么，它都可以当作鸭子使用。当我们进入函数和类的学习的时候，我们会惊喜地发现，我们昨天所讨论的`鸭子类型`和`变量机制`等概念其实都从某一个侧面为学习`面向对象`提供了理解，他们根本上是一体的。

接下来，进入`函数`和`类`的学习。但首先，在学习所有东西之前请先记住

> Python中万物都是对象（objects）

因此，我要提前给出以下论断：

- 函数是一个对象，是一个类的实例 （万物都是对象）
- 任何类的实例都可以成为一个函数，只要它拥有函数所必须具备的属性和方法。（鸭子类型）

第一点或许已经很难理解，但是第二点会使得你更加讶异。但是，这就是Python令人着迷的地方，实际上，类和函数是没有本质差异的，正如各种数据类型之间没有本质差异一样，他们都彻彻底底地作为一个个对象存在。Python内几乎任何事物，只是某些特殊的类的实例罢了，他们的本质都是对象。

虽然我已经迫不及待向你证明这两个论断，但首先我们还是需要从基本的语法开始入手。

Section 1.函数的语法和类的语法

1. 函数的基本语法

```
def func(parameters):
    """This is a function."""
    x = 10
    return x
```

函数的基本语法如上，包括：

- 必要的关键字： `def` 说明你要进行一个函数的定义
- 函数名字: `func`
- 函数的输入参数(optional): `parameters`
- 函数主体
- 函数的返回值(optional): `return x` （当返回值为`None`的时候,可以不写,Python会自动补全）
- 函数描述(optional): `"""This is function."""`，如果你要写一个函数描述的话，请一定要把该字符串写在函数主体的一开始。（之后会说明这个规范的意义）

除此之外，函数必须拥有自己本地的`作用域`,因此需要进行一个级别的缩进.

2. 函数的高级语法和特性

所谓函数的高级语法和特性，大部分只是对函数的各个组成部分进行剖析。

2.1 参数(parameters)

学习Python的函数语法，不如先横向类比`C++`和`matlab`的函数定义语法，`C++`的函数定义要求声明变量的类型，而`matlab`并没有这个规定。

结合`变量引用机制`知识,我们知道Python传入的函数参数是一个个`对象的引用`。 因此和`C++`不同，它不需要对参数的类型进行提前说明，只需要你告诉它参数的数目即可。

```
def func(x,y):
    pass
```

（比如这个代码就是指要求有两个参数的函数，`pass`是一个占位符，它不起任何作用。它存在的意义是帮助你完整语法，就当你定义一个函数但还没想好如何实现它，就可以先用一个pass来使得语法完整，之后再来实现。）

但我们遇到了这样几个难题：

- 如果我想传入任意个参数该怎么办？
- 如果要传入的参数较多，我怎么设置参数的默认值？

2.1.1 常规传参

按照数目和相应的顺序传入参数。

```
def func(x,y):
    print(x,y)
func(1,2) # 1,2
```

2.1.2 位置传参

如果我们忘记了应当传入参数的`顺序`，我们也可以通过传参时指定对应参数名来进行传参。定义函数时还是按照常规传参的定义去定义的。

```
def func(x,y):
    print(x,y)
func(y = 2, x = 1) # 1,2
```

2.1.3 任意个参数的传入（`不定参数`）

`不定参数`： 定义函数的时候，不定参数的引用关系不是一对一，而是一对多。函数会将额外传入的n个一对一引用关系的参数打包成一个一对n引用关系的参数进行处理。我们发现，元组和字典就满足这样的要求，即：把n个一对一引用关系的变量打包成一个一对n引用关系的变量。因此实际上，不定参数只是元组和字典在参数里面的一层外衣。

`语法`： `*args, **kwargs`（`args`或者`kwargs`这个名字可以任意起，重要的是`*`代表元组，`**`代表字典）以下是实例：

```
def func(var1, *args):
    print(var1)
    print(args)
func(1,2,3,4)         # 1  (2,3,4)
func(1,2,3,4,5,6,7,8) # 1 (2,3,4,5,6,7,8)
func("hahah", "sdsd", 2, 3, 10) # "hahah" ("sdsd", 2, 3, 10)
```

注： 元组和列表唯一的区别在于，元组是只读的（不可修改）的列表

当然，如果是(`**kwargs`)的话，

```
def func(var1, **kwargs):
    print(var1)
    print(kwargs)
func(1,2,3)     # error
func(1, k = 1, x = 2) # 1, {'k':1, 'x':2}
```

注：字典指的是存储键值对的容器，你可以通过一个名字（键）来获取它的值。实际上，我们可以认为列表和元就是index和值的键值对。因此，字典的传入必须要包含名字，必须形如`k = 1`这样传递。

其实，就算没有不定参数的规定，我们也完全可以通过下面这种方式来实现上述需求,所以我们也可以认为这种不定参数的设计是一种对程序员友善的`语法糖`。

```
def func(var1, args):
    print(var1)
    print(args)
x = 1
y = (2,3,4,5)
func(x,y)
```

（这种方式就是我传了两个参数，只不过其中一个参数是一个元组。结果是完全一样的。）

2.1.4 默认值 (default values)

我可以通过这样的定义来为传入参数附上默认值

```
def func(x = 1, y = 2, z = 3):
    print(x,y,z)
func() # 1,2,3 （使用默认值）
func("x","y") # "x", "y", 3  （有就使用传值的值，没有就使用默认值）
func(10,20,30) # 10,20,30
func(x = 10, y = 30, z = 40) # 10, 30, 40
func(z = 40, y = 30, x = 10) # 10, 30, 40 (当我们用键值对传递参数值的时候，顺序就不重要了)
func(z = 40, x = "ha") # "ha", 2, 40
```

2.2 返回值

`return`关键字表征着函数的执行结束。也即，函数运行到`return`时就结束并返回相应的值。