从小白到大神，一文掌握 Python 密集知识点

第一个问题，什么是 Python ？根据 Python 之父 Guido van Rossum 的话，Python 是：

> 一种高级程序语言，其核心设计哲学是代码可读性和语法，能够让程序员用很少的代码来表达自己的想法。

对于我来说，学习 Python 的首要原因是，Python 是一种可以优雅编程的语言。它能够简单自然地写出代码和实现我的想法。

另一个原因是我们可以将 Python 用在很多地方：数据科学、Web 开发和机器学习等都可以使用 Python 来开发。Quora、Pinterest 和 Spotify 都使用 Python 来进行他们的后端 Web 开发。那么让我们来学习一下 Python 吧。

Python 基础

1. 变量

你可以把变量想象成一个用来存储值的单词。我们看个例子。

Python 中定义一个变量并为它赋值是很容易的。假如你想存储数字 1 到变量 “one” ，让我们试试看：

```
one = 1
```

超级简单吧？你只需要把值 1 分配给变量 “one” 。

```
two = 2

some_number = 10000
```

只要你想，你可以把任意的值赋给任何其他的变量。正如你从上面看到的那样，变量 “two” 存储整型变量 2 ，变量 “some_number” 存储 10000 。

除了整型，我们还可以使用布尔值（True/Flase）、字符串、浮点型和其他数据类型。

```
# booleanstrue_boolean = Truefalse_boolean = False# stringmy_name = "Leandro Tk"# floatbook_price = 15.80
```

2. 控制流程：条件语句

“If” 使用一个表达式来判断一个语句是 True 还是 False ，如果是 True ，那么执行 if 内的代码，例子如下：

```
if True:

print("Hello Python If")if 2 > 1:

print("2 is greater than 1")
```

2 比 1 大，所以 print 代码被执行。

当“if”里面的表达式是 false 时，“else” 语句将会执行。

```
if 1 > 2:

print("1 is greater than 2")else:

print("1 is not greater than 2")
```

1 比 2 小，所以 “else” 里面的代码会执行。

你也可以使用 “elif” 语句：

```
if 1 > 2:

print("1 is greater than 2")elif 2 > 1:

print("1 is not greater than 2")else:

print("1 is equal to 2")
```

3. 循环和迭代

在 Python 中，我们可以用不同的形式进行迭代。我会说下 while 和 for。

While 循环：当语句是 True 时，while 内部的代码块会执行。所以下面这段代码会打印出 1 到 10 。

```
num = 1while num <= 10:
    print(num)
    num += 1
```

while 循环需要循环条件，如果条件一直是 True ，它将会一直迭代，当 num 的值为 11 时，循环条件为 false 。

另一段代码可以帮你更好的理解 while 语句的用法：

```
loop_condition = Truewhile loop_condition:
    print("Loop Condition keeps: %s" %(loop_condition))
    loop_condition = False
```

循环条件是 True 所以会一直迭代，直到为 False 。

For 循环：你可以在代码块上应用变量 “num” ，而 “for” 语句将为你迭代它。此代码将打印与 while 中相同的代码：从 1 到 10 。

```
for i in range(1, 11):
    print(i)
```

瞧见没？这太简单了。i 的范围从 1 开始一直到第 11 个元素（10是第十个元素）

List：集合 | 数组 | 数据结构

假如你想要在一个变量里存储整数 1 ，但是你也要存储 2 和 3，4，5 ...

不是用成百上千个变量，我有别的方法存储这些我想要存储的整数吗？你已经猜到了，确实有别的存储它们的方法。

列表是一个集合，它能够存储一列值（就像你想要存储的这些），那么让我们来用一下它：

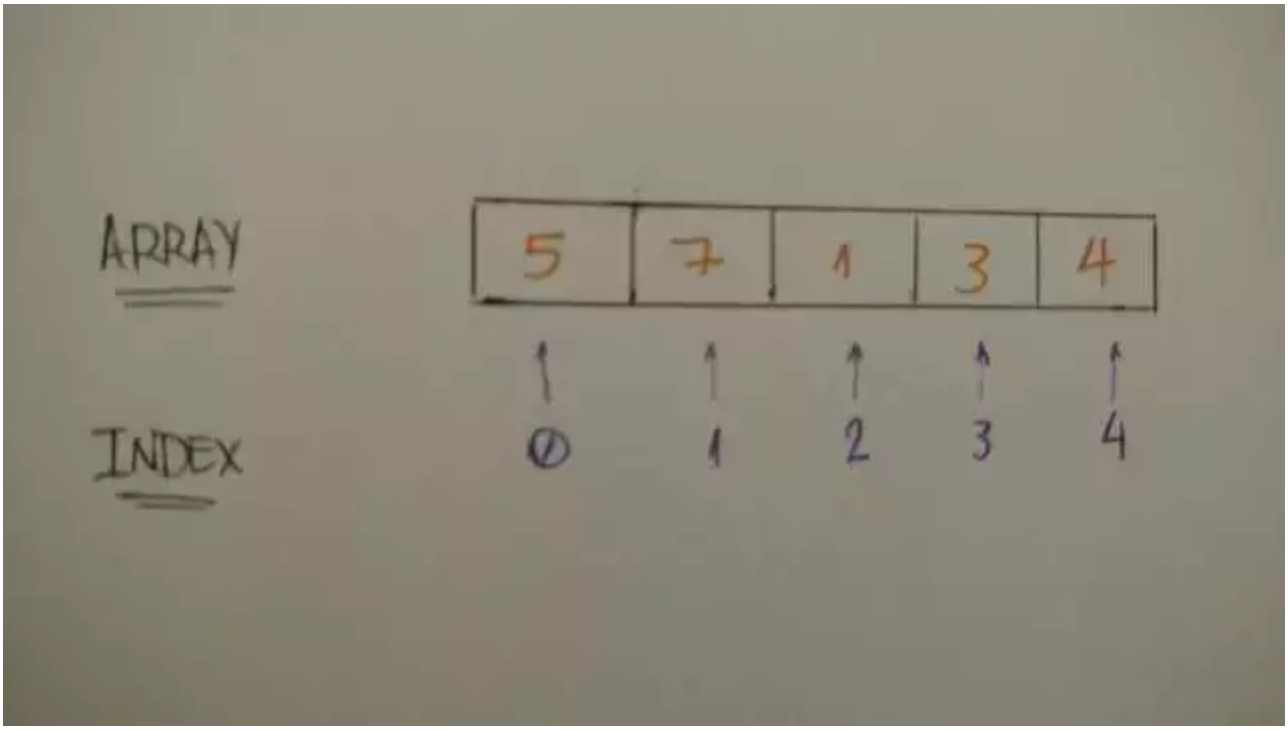
```
my_integers = [1, 2, 3, 4, 5]
```

这真的很简单。我们创建了一个叫做 my_integer 的数组并且把数据存到了里面。

也许你会问：“我要怎样获取数组里的值？”

问得好。列表有一个叫做索引的概念。第一个元素的下表是索引0（0）。第二个的索引是1，以此类推，你应该明白的。

为了使它更加简洁，我们可以用它的索引代表数组元素。我画了出来：



用 Python 的语法，也很好去理解：

```
my_integers = [5, 7, 1, 3, 4]

print(my_integers[0]) # 5print(my_integers[1]) # 7print(my_integers[4]) # 4
```

假如你不想存整数。你只想去存一些字符串，像你亲戚名字的列表。我的看起来是类似这样的：

```
relatives_names = [ "Toshiaki", "Juliana", "Yuji", "Bruno", "Kaio"]

print(relatives_names[4]) # Kaio
```

它的原理跟存整数一样，很友好。

我们只学习了列表的索引是如何工作的，我还需要告诉你如何向列表的数据结构中添加一个元素（向列表中添加一个项目）。

最常用的向列表中添加新数据的方法是拼接。我们来看一下它是如何使用的：

```
bookshelf = []

bookshelf.append("The Effective Engineer")

bookshelf.append("The 4 Hour Work Week")
```

```
print(bookshelf[0]) # The Effective Engineerprint(bookshelf[1]) # The 4 Hour Work W
```

拼接超级简单，你仅需要把一个元素（比如“有效的机器”）作为拼接参数。

好了，关于列表的知识这些就够了，让我们来看一下其它的数据结构。

字典：Key-Value 数据结构

现在我们知道 List 是有索引的整型数字集合。但如果我们不像使用整型数字作为索引呢？我们可以用其他的一些数据结构，比如数字、字符串或者其他类型的索引。

让我们学习下字典这种数据结构。字典是一个键值对的集合。字典差不多长这样：

```
dictionary_example = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Key 是指向 value 的索引。我们如何访问字典中的 value 呢？你应该猜到了，那就是使用 key 。我们试一下：

```
dictionary_tk = {
  "name": "Leandro",
  "nickname": "Tk",
  "nationality": "Brazilian"
}


print("My name is %s" %(dictionary_tk["name"])) # My name is Leandro
print("But you can call me %s" %(dictionary_tk["nickname"])) # But you can call me Tk
print("And by the way I'm %s" %(dictionary_tk["nationality"])) # And by the way I'm Brazilian
```

我们有个 key(age)value(24），使用字符串作为 key 整型作为 value 。

我创建了一个关于我的字典，其中包含我的名字、昵称和国籍。这些属性是字典中的 key 。

就像我们学过的使用索引访问 list 一样，我们同样使用索引（在字典中 key 就是索引）来访问存储在字典中的 value 。

正如我们使用 list 那样，让我们学习下如何向字典中添加元素。字典中主要是指向 value 的 key 。当我们添加元素的时候同样如此：

```
dictionary_tk = {
  "name": "Leandro",
  "nickname": "Tk",
  "nationality": "Brazilian",
  "age": 24
}


print("My name is %s" %(dictionary_tk["name"])) # My name is Leandro
print("But you can call me %s" %(dictionary_tk["nickname"])) # But you can call me Tk
print("And by the way I'm %i and %s" %(dictionary_tk["age"], dictionary_tk["nationality"])) # And by the way I'm Brazilian
```

我们只需要将一个字典中的一个 key 指向一个 value 。没什么难的，对吧？

迭代：通过数据结构进行循环

跟我们在 Python 基础中学习的一样，List 迭代十分简单。我们 Python 开发者通常使用 For 循环。我们试试看：

```
bookshelf = [
  "The Effective Engineer",
  "The 4 hours work week",
  "Zero to One",
  "Lean Startup",
  "Hooked"
]
for book in bookshelf:
    print(book)
```

对于在书架上的每本书，我们打印（可以做任何操作）到控制台上。超级简单和直观吧。这就是 Python 的美妙之处。

对于哈希数据结构，我们同样可以使用 for 循环，不过我们需要使用 key 来进行。

```
dictionary = { "some_key": "some_value" }

for key in dictionary:

    print("%s --> %s" %(key, dictionary[key])) # some_key --> some_value
```

上面是如何在字典中使用 For 循环的例子。对于字典中的每个 key ，我们打印出 key 和 key 所对应的 value 。

另一种方式是使用 iteritems 方法。

```
dictionary = { "some_key": "some_value" }

for key, value in dictionary.items():

    print("%s --> %s" %(key, value))# some_key --> some_value
```

我们命名两个参数为 key 和 value ，但是这不是必要的。我们可以随意命名。我们看下：

```
dictionary_tk = {

  "name": "Leandro",

  "nickname": "Tk",

  "nationality": "Brazilian",

  "age": 24

}


for attribute, value in dictionary_tk.items():

    print("My %s is %s" %(attribute, value))


# My name is Leandro

# My nickname is Tk

# My nationality is Brazilian

# My age is 24
```

可以看到我们使用了 attribute 作为字典中 key 的参数，这与使用 key 命名具有同样的效果。真是太棒了！

类&对象

一些理论：

对象是对现实世界实体的表示，如汽车、狗或自行车。 这些对象有两个共同的主要特征：数据和行为。

汽车有数据，如车轮的数量，车门的数量和座位的空间，并且它们可以表现出其行为：它们可以加速，停止，显示剩余多少燃料，以及许多其他的事情。

我们将数据看作是面向对象编程中的属性和行为。 又表示为：

数据→ 属性和行为 → 方法

而类是创建单个对象的蓝图。 在现实世界中，我们经常发现许多相同类型的对象。 比如说汽车。 所有的汽车都有相同的构造和模型（都有一个引擎，轮子，门等）。每辆车都是由同一套蓝图构造成的，并具有相同的组件。

Python 面向对象编程模式：ON

Python，作为一种面向对象编程语言，存在这样的概念：类和对象。

一个类是一个蓝图，是对象的模型。

那么，一个类是一个模型，或者是一种定义属性和行为的方法（正如我们在理论部分讨论的那样）。举例来说，一个车辆类有它自己的属性来定义这个对象是个什么样的车辆。一辆车的属性有轮子数量，能源类型，座位容量和最大时速这些。

考虑到这一点，让我们来看看 Python 的类的语法：

```
class Vehicle:
    pass
```

上边的代码，我们使用 class 语句来定义一个类。是不是很容易？

对象是一个类的实例化，我们可以通过类名来进行实例化。

```
car = Vehicle()
print(car) # <__main__.Vehicle instance at 0x7fb1de6c2638>
```

在这里，car 是类 Vehicle 的对象（或者实例化）。

记得车辆类有四个属性：轮子的数量，油箱类型，座位容量和最大时速。当我们新建一个车辆对象时要设置所有的属性。所以在这里，我们定义一个类在它初始化的时候接受参数：

```
class Vehicle:
        def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
```

```
        self.number_of_wheels = number_of_wheels

        self.type_of_tank = type_of_tank

        self.seating_capacity = seating_capacity

        self.maximum_velocity = maximum_velocity
```

这个 init 方法。我们称之为构造函数。因此当我们在创建一个车辆对象时，可以定义这些属性。想象一下，我们喜欢 Tesla Model S ，所以我们想创建一个这种类型的对象。它有四个轮子，使用电能源，五座并且最大时时速是250千米（155英里）。我们开始创建这样一个对象：

```
tesla_model_s = Vehicle(4, 'electric', 5, 250)
```

四轮+电能源+五座+最大时速250千米。

所有的属性已经设置了。但我们该如何访问这些属性值呢？我们给对象发送消息以向其请求该值。我们称之为方法。它是对象的行为。让我们实现它：

```
class Vehicle:

            def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):

        self.number_of_wheels = number_of_wheels

        self.type_of_tank = type_of_tank

        self.seating_capacity = seating_capacity

        self.maximum_velocity = maximum_velocity    def number_of_wheels(self):

        return self.number_of_wheels    def set_number_of_wheels(self, number):

        self.number_of_wheels = number
```

这是两个方法number_of_wheels和set_number_of_wheels的实现。我们将其称为 getter & setter。因为第一个函数是获取属性值，第二个函数是给属性设置新的值。

在 Python 中，我们可以使用@property (修饰符)来定义getters和setters。让我们看看实际代码：

```
class Vehicle:

            def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):

        self.number_of_wheels = number_of_wheels

        self.type_of_tank = type_of_tank

        self.seating_capacity = seating_capacity
```

```
        self.maximum_velocity = maximum_velocity    @property
    def number_of_wheels(self):
        return self.number_of_wheels    @number_of_wheels.setter
    def number_of_wheels(self, number):
        self.number_of_wheels = number
```

并且我们可以将这些方法作为属性使用：

```
tesla_model_s = Vehicle(4, 'electric', 5, 250)
print(tesla_model_s.number_of_wheels) # 4tesla_model_s.number_of_wheels = 2 # setting number of wheels to 2print(tesla_model_s.number_of_wheels) # 2
```

这和方法定义有轻微的不同。这里的方法是按照属性执行的。例如当我们设置新的轮胎数目时，我们并不将这两个看做参数，而是将数值2设置给number_of_wheels。这是编写python风格的getter和setter代码的一种方式。

但我们也可以将该方法用于其他事项，例如“make_noise”方法。让我们看看：

```
class Vehicle:
        def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.type_of_tank = type_of_tank
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity    def make_noise(self):
        print('VRUUUUUUUM')
```

当我们调用此方法时，它仅仅返回一个字符串“VRRRRUUUUM.”

```
tesla_model_s = Vehicle(4, 'electric', 5, 250)
tesla_model_s.make_noise() # VRUUUUUUUM
```

封装: 隐藏信息

封装是一种限制直接访问对象数据和方法的机制。但与此同时，它使得在数据上操作更简单（对象的方法）。

> “封装可被用于隐藏数据成员和成员函数。按照这个定义，封装意味着 对象的内部表示一般在对象定义的外部视图中隐藏。” —Wikipedia

对象的所有内部表示都对外部隐藏了。只有对象本身可以与其内部数据交互。

首先，我们需要理解公开的、非公开的实例变量和方法的工作原理。

公共实例变量

对于 Python 类，我们可以在我们的构造函数方法中初始化一个公共实例变量。让我们看看这个：

在这个构造方法中:

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name
```

在这里，我们将 first_name 值作为参数应用于公共实例变量。

```
tk = Person('TK')
print(tk.first_name) # => TK
```

在类中:

```
class Person:
    first_name = 'TK'
```

在这里，我们不需要将 first_name 作为参数，所有的实例对象都有一个用 TK 初始化的类属性。

```
tk = Person()
print(tk.first_name) # => TK
```

太酷了，现在我们已经了解到，我们可以使用公共实例变量和类属性。关于公共部分的另一个有趣的事情是我们可以管理变量值。我的意思是什么呢？我们的对象可以管理它的变量值：Get 和 Set 变量值。

还是在 Person 类中，我们想为它的 first_name 变量设置另一个值：

```
tk = Person('TK')
tk.first_name = 'Kaio'print(tk.first_name) # => Kaio
```

这就可以了，我们只是为 first_name 实例变量设置另一个值（kaio），并更新了值。就这么简单。因为这是一个公共变量，我们是可以这么做的。

Non-public 实例变量

> 这里我们并没有使用术语“private”，因为在Python中所有属性都不是真的私有的（没有通常不必要的工作量）。— PEP 8

作为 public instance variable（公共实例变量），我们可以在构造方法或类内部定义 non-public instance variable （非公共实例变量）。语法上的区别是：对于 non-public instance variables （非公共实例变量），在变量名前使用下划线(_)。

> “除了从对象内部外无法被访问的‘Private’实例变量在Python中并不存在。然而，这里有一个多数Python代码都会遵守的惯例：使用下划线作为前缀的命名(例如 _spam)应该被认为是API的非公开部分（不管是函数、方法还是数据成员）” — Python软件基础

这里是示例代码:

```
class Person:
    def __init__(self, first_name, email):
        self.first_name = first_name
        self._email = email
```

你看到了email变量了吗？这就是我们如何定义非公共变量的方法：

```
tk = Person('TK', 'tk@mail.com')
print(tk._email) # tk@mail.com
```

我们可以访问并更新它。非公共变量仅仅是一个惯用法，并且应该被当做API的非公共部分。

所以我们使用一个在类定义内部的方法来实现该功能。让我们实现两个方法(email 和 update_email)以加深理解：

```
class Person:
    def __init__(self, first_name, email):
        self.first_name = first_name
        self._email = email    def update_email(self, new_email):
        self._email = new_email    def email(self):
        return self._email
```

现在我们可以使用这两个方法来更新及访问非公开变量了。示例如下：

```
tk = Person('TK', 'tk@mail.com')
```

```
print(tk.email()) # => tk@mail.comtk._email = 'new_tk@mail.com'print(tk.email()) # => tk@mail.comtk.update_email('new_tk@mail.com')

print(tk.email()) # => new_tk@mail.com
```

- 我们使用first_name TK和email tk@mail.com初始化了一个新对象
- 使用方法访问非公开变量email并输出它
- 尝试在类外部设置一个新的email
- 我们需要将非公开变量视为API的非公开部分
- 使用我们的实例方法来更新非公开变量
- 成功！我们使用辅助方法在类内部更新了它。

公共方法

对于公共方法，我们也可以在类中使用它们：

```
class Person:
    def __init__(self, first_name, age):
        self.first_name = first_name
        self._age = age    def show_age(self):
        return self._age
```

让我们来测试一下:

```
tk = Person('TK', 25)
print(tk.show_age()) # => 25
```

很好 - 我们在类中使用它没有任何问题。

非公共方法

但是用非公开的方法，我们无法做到这一点。 如果我们想实现相同的 Person 类，现在使用有下划线（ _ ）的 show_age 非公共方法。

```
class Person:
    def __init__(self, first_name, age):
        self.first_name = first_name
        self._age = age    def _show_age(self):
        return self._age
```

现在，我们将尝试用我们的对象来调用这个非公共的方法：

```
tk = Person('TK', 25)
print(tk._show_age()) # => 25
```

我们可以访问和更新它。 非公共的方法只是一个惯例，应该被视为API的非公开部分。

以下是我们如何使用它的一个例子：

```
class Person:
    def __init__(self, first_name, age):
        self.first_name = first_name
        self._age = age    def show_age(self):
        return self._get_age()    def _get_age(self):
        return self._age
tk = Person('TK', 25)
print(tk.show_age()) # => 25
```

这里有一个 _get_age 非公共方法和一个 show_age 公共方法。 show_age 可以被我们的对象（不在我们的类中）使用，而 _get_age 只用在我们的类定义里面使用（在 show_age 方法里面）。 但是同样的，这样的做法通常是惯例。

封装小结

通过封装，我们可以确保对象的内部表示是对外部隐藏的。

继承：行为和特征

某些物体有一些共同之处：它们的行为和特征。

例如，我继承了我父亲的一些特征和行为。我继承了他的眼睛和头发的特征，以及他的急躁和内向的行为。

在面向对象编程中，类可以继承另一个类的共同特征（数据）和行为（方法）。

我们来看另一个例子，并用 Python 实现它。

想象一下汽车。车轮数量，座位容量和最大速度都是一辆车的属性。我们可以说 ElectricCar 类从普通的 Car 类继承了这些相同的属性。

```
class Car:
    def __init__(self, number_of_wheels, seating_capacity, maximum_velocity):
```

```
        self.number_of_wheels = number_of_wheels
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity
```

我们 Car 类的实现:

```
my_car = Car(4, 5, 250)
print(my_car.number_of_wheels)
print(my_car.seating_capacity)
print(my_car.maximum_velocity)
```

一旦初始化，我们就可以使用所有创建的实例变量。太棒了。

在 Python 中，我们将父类作为子的参数来进行继承。 ElectricCar 类可以继承我们的 Car 类。

```
class ElectricCar(Car):
    def __init__(self, number_of_wheels, seating_capacity, maximum_velocity):
        Car.__init__(self, number_of_wheels, seating_capacity, maximum_velocity)
```

就这么简单。我们不需要实现任何其他方法，因为这个类已经完成了父类的继承（继承自 Car 类）。我们来证明一下：

```
my_electric_car = ElectricCar(4, 5, 250)
print(my_electric_car.number_of_wheels) # => 4
print(my_electric_car.seating_capacity) # => 5
print(my_electric_car.maximum_velocity) # => 250
```

干的漂亮。

就到这吧！

我们已经学到了很多关于 Python 的基础知识：

- Python 变量是如何工作的
- Python 条件语句是如何工作的
- Python 循环（ while 和 for ）是如何工作的
- 如何使用链表：Collection |Array

- 字典式键值集合
- 我们如何遍历这些数据结构
- 对象和类
- 作为对象数据的属性
- 作为对象行为的方法
- 使用 Python getter/setter 以及属性修饰器
- 封装：隐藏信息
- 继承：行为和特征

恭喜！ 你完成了这个关于 Python 的密集的知识点。