第 3 章

当对象相似时[1]

在编程的世界中，重复的代码被认为是邪恶的。我们不应该在不同的地方写多份相同或相似的代码。[2]我们可能在一处修改了一个 Bug，但忘记修改另一处 Bug，这会给我们带来无穷无尽的麻烦。

有很多合并功能相似的代码或对象的方法。在本章中，我们将讨论最著名的面向对象原则：继承。正如在第 1 章中所讨论的，继承让我们能够创建两个或多个类之间的“是一个”关系，将共有的逻辑抽象到超类并在每个子类中控制具体的细节。在本章中，我们将会讨论如下的 Python 语法和原则：

- 基本的继承。
- 从内置类型继承。
- 多重继承。
- 多态与鸭子类型。

本章的案例学习将会沿用上一章的示例。我们将使用继承和抽象的概念来设计 KNN 算法中的通用代码。

我们首先研究如何通过继承提炼共同特征，这样可以避免复制粘贴代码。

---

① 麦叔注：父类和超类具有同样的含义，基于作者所了解的行业习惯，本章优先使用父类。

② 麦叔注：我们同样需要避免把看起来相似但本质不同的代码勉强地写在一起，这同样会带来很多麻烦。最好的办法是，通过良好的设计，把真正相同的逻辑放在一起，又能够通过多态或模板方法等设计模式灵活而安全地处理差异性。

3.1 基本继承

严格来说，我们创建的所有类都使用了继承关系。所有的 Python 类都是名为 object 的特殊内置类的子类。object 类提供了类的基本数据和行为（它提供的所有方法都是双下画线开头__的特殊方法，只在内部使用），从而允许 Python 以同样的方式对待所有对象。

如果我们不明确地从其他类继承，那么我们的类将默认继承自 object。当然，我们也可以声明我们的类继承自 object，使用下面的语法：

```
class MySubClass(object):
    pass
```

这就是继承关系！严格来说，这个示例与第 2 章的第一个示例没有区别。因为如果没有明确提供其他**超类**，那么 Python 3 中的所有类都会默认继承自 object。超类，或者是父类，是指被继承的类，子类是继承自超类的类。在这个示例中，超类是 object，子类是 MySubClass。通常称子类源自父类或子类扩展自父类。

可能你已经从这个示例中搞清楚，继承关系只需要在类定义的基本语法上添加少量额外语法就可以：只要将父类的名称放进类名后及冒号前的括号内即可，这样就可以告诉 Python 新类应该继承自给定的父类。

在实践中应该如何应用继承关系？最简单和最明显的用法就是为已存在的类添加功能。让我们从一个简单的联系人管理器开始，这个管理器可以追踪多个人的名字和 E-mail 地址。Contact 类用一个类变量维护所有联系人的全局列表，并为每个联系人初始化姓名和地址：

```
class Contact:
    all_contacts: List["Contact"] = []

    def __init__(self, name: str, email: str) -> None:
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

```
    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}"
            f")"
        )
```

这个示例向我们介绍了**类变量**：all_contacts 列表，由于它是类定义的一部分，因此被这个类的所有实例所共享。这意味着只有一个 Contact.all_contacts 列表。我们也可以通过 Contact.all_contacts 访问，也可以在 Contact 对象中通过 self.all_contacts 访问，如果对象中找不到相应的变量（通过 self），就会从类中去寻找，从而都会指向同一个列表。

使用 self 访问变量时，有一个要注意的地方。使用 self 可以读取类变量的值。但如果你对 self.all_contacts =来**设定**变量的值，你实际上会创建一个只与那个对象相关的**新的**实例变量。原来的类变量将不会改变，仍然可以通过 Contact.all_contacts 访问。

通过下面的示例，可以看出类变量 Contact.all_contacts 记录了所有的联系人：

```
>>> c_1 = Contact("Dusty", "dusty@example.com")
>>> c_2 = Contact("Steve", "steve@itmaybeahack.com")
>>> Contact.all_contacts
[Contact('Dusty', 'dusty@example.com'), Contact('Steve', 'steve@itmaybeahack.com')]
```

这个简单的类允许我们追踪每个联系人的一些数据，但是如果我们的某些联系人同时也是供货商，我们需要从他们那里下单，该怎么办？我们可以为 Contact 类添加一个 order()方法，但是这样将会造成可以给客户、家人、朋友等不是供应商的联系人下单，这是不合理的。更好的做法是，创建一个新的 Supplier 类，它继承自 Contact 类，但是拥有一个额外的 order()方法，这个方法接收一个尚未定义的 Order 对象作为参数：

```
class Supplier(Contact):
    def order(self, order: "Order") -> None:
```

```
        print(
            "If this were a real system we would send "
            f"'{order}' order to '{self.name}'"
        )
```

现在，用交互式 Python 来测试这个类，我们可以发现所有的 Contact（包括 Supplier）的__init__方法都接收 name 和 email 参数，但是只有 Supplier 实例有 order()方法：

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net

>>> from pprint import pprint
>>> pprint(c.all_contacts)
[Contact('Dusty', 'dusty@example.com'),
 Contact('Steve', 'steve@itmayb[illegible]'),
 Contact('Some Body', 'somebody@example.net'),
 Supplier('Sup Plier', 'supplier@example.net')]

>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to 'Sup
Plier'
```

所以，所有 Contact 能做的事情我们的 Supplier 类也可以做（包括把它自己加入 Contact.all_contacts 列表中），同时它也能做供货商需要处理的特殊事务。这就是继承之美。

注意，Contact.all_contacts 保存了所有 Contact 类的实例，以及它的子类 Supplier 的实例。但如果我们使用 self.all_contacts，那么将**不会**把所有对象都保存到 Contact 类中，因为 Supplier 的实例将会被保存在 Supplier.all_contacts 中。

3.1.1　扩展内置对象

继承的一个有趣的用法是给内置类添加新功能。在前面看到的 Contact 类中，我们将联系人添加到所有联系人列表中。如果想要根据名字搜索这个列表呢？我们可以为 Contact 类添加一个搜索方法，但是这个方法似乎应该属于列表本身。

下面的示例展示了如何通过继承内置类来实现这个功能，在这里我们继承 list 类。我们规定新定义的 list 子类只能存放 Contact 的实例，我们可以使用 list["Contact"]的写法把这一规定告诉 ***mypy*** 工具。为了使这个语法在 Python 3.9 中生效，我们需要从__future__包中引入 annotations 模块：

```
from __future__ import annotations
class ContactList(list["Contact"]):
    def search(self, name: str) -> list["Contact"]:

        matching_contacts: list["Contact"] = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name: str, email: str) -> None:
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}" f")"
        )
```

我们没有使用通用的 list 类作为实例变量，而是创建了一个新的 ContactList 类来继承内置的 list 数据类型，然后实例化这个子类并将其赋值给 all_contacts 列表。

我们可以用如下的方式测试这个新的搜索功能：

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@sloop.net")
>>> c3 = Contact("Jenna C", "cutty@sark.io")
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

我们有两种创建通用 list 对象的方法。使用类型提示，我们有了另一种不需要实际创建列表实例就可以声明列表变量的方法。

首先，用[]创建一个空列表，实际上这是用 `list()`创建空列表的简便方式，这两种语法完全一样：

```
>>> [] == list()
True
```

实际上，[]语法就是所谓的**语法糖**（syntax sugar）[①]，其在底层调用 `list()`构造方法。这种写法只需要写 2 个字符（[]），而不是 6 个字符（`list()`）。这里的 `list` 是指一种数据类型，是一个我们可以继承的类。

mypy 或类似的工具可以检查 `ContactList.search()`方法，以确保它创建了一个只包含 `Contact` 对象的 `list` 实例。请使用 0.8.2 或更新的版本，因为老版本的 ***mypy*** 不完全支持这些关于泛型的注解。

因为 `Contact` 类的定义在 `ContactList` 定义的下面，也就是说，在定义 `ContactList` 的时候还没有定义 `Contact` 类，所以我们在指定类型的时候要使用字符串代表未被定义的 `Contact` 类，使用这种写法：`list["Contact"]`。但通常我们会先定义被引用的类，然后再定义使用它的类。如果我们先定义 `Contact` 类，再定义 `ContactList` 类，就可以直接使用类名而不用写字符串，也就是这样：`list[Contact]`。

作为第二个示例，我们可以扩展 `dict` 类。它是一些键值对的集合。与列表相似，它可以用{}作为语法糖，更简单地构造字典。下面是一个字典的扩展类，它可以追踪

① 麦叔注：语法糖是指为了让代码写起来简单而创建的特殊写法，它的底层通常是另一种稍微复杂点的写法。

字典中最长的 key：

```
class LongNameDict(dict[str, int]):
    def longest_key(self) -> Optional[str]:
        """实际上和 max(self, key=len)功能相同，但是更直观"""
        longest = None
        for key in self:
            if longest is None or len(key) > len(longest):
                longest = key
        return longest
```

类型提示 dict[str, int]规定了这个类的 key 必须是 str 类型的，value 必须是 int 类型的。这样可以帮助 ***mypy*** 判定 longest_key()方法的合理性。因为 key 是字符串，所以可以在方法中使用 len 判定 key 的长度。最后的结果是一个 str 类型或者 None，所以方法的返回值被描述为 Optional[str]。（返回 None 合理吗？也许不合理，或许抛出 ValueError 异常更合理一点儿，不过这要等到第 4 章介绍）。

我们定义的类处理的是字符串和整数。也许字符串是用户名，整数是用户在网站上读过的文章数。除了核心用户名和阅读历史，我们也需要知道最长的名字有多长，这样就可以确定展示用户名和阅读历史的表格需要多宽。我们可以在交互式解释器中简单测试一下：

```
>>> articles_read = LongNameDict()
>>> articles_read['lucy'] = 42
>>> articles_read['c_c_phillips'] = 6
>>> articles_read['steve'] = 7
>>> articles_read.longest_key()
'c_c_phillips'
>>> max(articles_read, key=len)
'c_c_phillips'
```

如果我们需要一个更常规的字典，该怎么办呢？比如，字典中的值可能是字符串**或**整数。我们需要使用一个更宽泛的类型提示，可以这样写：dict[str, Union[str, int]]。使用 Union，我们指定的字典的值可能是字符串或整数。

大多数内置类型都可以用相似的方法扩展。这些内置类型可以分为几类，它们有各自的类型提示。

泛型集合 `set`、`list`、`dict`，使用形如 `set[something]`、`list[something]`和 `dict[key, value]`的类型提示指定集合中可以存放的具体类型（something），而不是存放什么都可以。为了使用这种泛型类型的注解，需要在代码第一行加上 `from __future__ import annotations`。

- 使用 `typing.NamedTuple` 可以定义新的不可变元组，并可以给元组中的元素命名。这会在第 7 章和第 8 章中涵盖。
- Python 具有与文件相关的 I/O 对象的类型提示。一种新的文件可以使用类型提示 `typing.TextIO` 或 `typing.BinaryIO` 来描述内置的文件操作。
- 通过扩展 `typing.Text` 可以创建新的字符串类型。在大多数情况下，内置的 `str` 类可以满足我们的所有需求。
- 新的数字类型通常衍生于 `numbers` 模块中的内置数字类型，因为它们提供了很多数字类型的基本功能。

我们将在本书中大量使用泛型集合。如前所述，我们将在后面的章节中讨论命名元组。内置类型的其他扩展对本书来说太高级了，因而不会涵盖。在下一节中，我们将更深入地研究继承的好处，以及如何在子类中选择性地利用超类的特性。

3.1.2 重写和 super

继承关系很适合向已存在的类中添加新的行为，但是如何修改某些行为？我们的 `contact` 类只接收 `name` 和 `email` 作为初始化参数。这对于大部分联系人足够了，但是如果想要为好朋友添加一个电话号码，该怎么办？

正如我们在第 2 章中看到的，我们可以在构造完联系人之后设定新的 `phone` 属性。如果想要让第 3 个变量可以在初始化过程中设定，则必须重写`__init__()`方法。重写意味着在子类中修改或替换超类原有的方法（用相同的名称）。重写不需要特殊的语法，子类中新创建的方法将会被优先调用，而不是用超类的方法，如下面的代码所示：

```
class Friend(Contact):
```