Lecture 8 第 8 讲

Object-Oriented Programming

面向对象编程

- There are different paradigms of programming. As you learn other languages, you will start recognizing patterns like these.
  编程有不同的范例。随着你学习其他语言，你会开始识别这样的模式。
- Up until this point, you have worked procedurally step-by-step.
  到目前为止，您一直在逐步按程序进行工作。
- Object-oriented programming (OOP) is a compelling solution to programming-related problems.
  面向对象编程 （OOP） 是解决编程相关问题的一个引人注目的解决方案。
- To begin, type `code student.py` in the terminal window and code as follows:
  首先，在终端窗口中键入 `code student.py` 并按如下方式编写代码：

```
name = input("Name: ")
house = input("House: ")
print(f"{name} from {house}")
```

Notice that this program follows a procedural, step-by-step paradigm: Much like you have seen in prior parts of this course.
请注意，此程序遵循程序性的分步范式：与您在本课程的前几部分中看到的非常相似。

- Drawing on our work from previous weeks, we can create functions to abstract away parts of this program.
  借鉴我们前几周的工作，我们可以创建函数来抽象出该程序的部分内容。

```
def main():
    name = get_name()
    house = get_house()
    print(f"{name} from {house}")


def get_name():
    return input("Name: ")


def get_house():
    return input("House: ")


if __name__ == "__main__":
    main()
```

Notice how `get_name` and `get_house` abstract away some of the needs of our `main` function. Further, notice how the final lines of the code above tell the compiler to run the `main` function.
请注意 `get_name` 和 `get_house` 是如何抽象出 `main` 函数的一些需求。此外，请注意上面代码的最后几行如何告诉编译器运行 `main` 函数。

- We can further simplify our program by storing the student as a `tuple`. A `tuple` is a sequences of values. Unlike a `list`, a `tuple` can't be modified. In spirit, we are returning two values.
  我们可以通过将 student 存储为 `Tuples` 来进一步简化我们的程序。元组 是一系列值。与 列表 不同，元组 不能修改。在精神上，我们返回两个值。

```
def main():
    name, house = get_student()
    print(f"{name} from {house}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return name, house


if __name__ == "__main__":
    main()
```

Notice how `get_student` returns `name, house`.
请注意 `get_student` 是如何返回 `name, house` 的。

- Packing that `tuple`, such that we are able to return both items to a variable called `student`, we can modify our code as follows.
  打包该 元组，以便我们能够将这两个项目都返回给一个名为 `student` 的变量，我们可以按如下方式修改我们的代码。

```
def main():
    student = get_student()
    print(f"{student[0]} from {student[1]}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return (name, house)
```

```
if __name__ == "__main__":
    main()
```

Notice that `(name, house)` explicitly tells anyone reading our code that we are returning two values within one. Further, notice how we can index into `tuple`s using `student[0]` or `student[1]`.

请注意，`(name, house)` 明确地告诉任何阅读我们代码的人，我们将在一个中返回两个值。此外，请注意如何使用 `student[0]` 或 `student[1]` 索引到元组 s。

- `tuple`s are immutable, meaning we cannot change those values. Immutability is a way by which we can program defensively.

`Tuples` 是不可变的，这意味着我们无法更改这些值。不变性是我们进行防御性编程的一种方式。

```
def main():
    student = get_student()
    if student[0] == "Padma":
        student[1] = "Ravenclaw"
    print(f"{student[0]} from {student[1]}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return name, house


if __name__ == "__main__":
    main()
```

Notice that this code produces an error. Since `tuple`s are immutable, we're not able to reassign the value of `student[1]`.

请注意，此代码会生成错误。由于 `Tuples` 是不可变的，因此我们无法重新分配 `student[1]` 的值。

- If we wanted to provide our fellow programmers flexibility, we could utilize a `list` as follows.

如果我们想为我们的程序员同事提供灵活性，我们可以使用如下 `列表`。

```
def main():
    student = get_student()
    if student[0] == "Padma":
        student[1] = "Ravenclaw"
    print(f"{student[0]} from {student[1]}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return [name, house]


if __name__ == "__main__":
    main()
```

Note the lists are mutable. That is, the order of `house` and `name` can be switched by a programmer. You might decide to utilize this in some cases where you want to provide more flexibility at the cost of the security of your code. After all, if the order of those values is changeable, programmers that work with you could make mistakes down the road.

请注意，列表是可变的。也就是说，`house` 和 `name` 的顺序可以由程序员切换。在某些情况下，您可能希望以牺牲代码的安全性为代价来提供更大的灵活性，从而决定使用它。毕竟，如果这些值的顺序是可变的，那么与您合作的程序员将来可能会犯错误。

- A dictionary could also be utilized in this implementation. Recall that dictionaries provide a key-value pair.

在此实现中也可以使用字典。回想一下，字典提供了一个键值对。

```
def main():
    student = get_student()
    print(f"{student['name']} from {student['house']}")


def get_student():
    student = {}
    student["name"] = input("Name: ")
    student["house"] = input("House: ")
    return student


if __name__ == "__main__":
    main()
```

Notice in this case, two key-value pairs are returned. An advantage of this approach is that we can index into this dictionary using the keys.

请注意，在这种情况下，将返回两个键值对。这种方法的一个优点是，我们可以使用 keys 索引到这个字典中。

- Still, our code can be further improved. Notice that there is an unneeded variable. We can remove `student = {}` because we don't need to create an empty dictionary.

尽管如此，我们的代码仍可进一步改进。请注意，有一个不需要的变量。我们可以删除 `student = {}`，因为我们不需要创建一个空字典。

```
def main():
    student = get_student()
    print(f"{student['name']} from {student['house']}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return {"name": name, "house": house}


if __name__ == "__main__":
    main()
```

Notice we can utilize `{}` braces in the `return` statement to create the dictionary and return it all in the same line.

请注意，我们可以在 `return` 语句中使用 `{}` 大括号来创建字典并在同一行中返回它。

- We can provide our special case with Padma in our dictionary version of our code.

我们可以在代码的字典版本中提供带有 Padma 的特殊情况。

```
def main():
    student = get_student()
    if student["name"] == "Padma":
        student["house"] = "Ravenclaw"
    print(f"{student['name']} from {student['house']}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return {"name": name, "house": house}
```

```
if __name__ == "__main__":
    main()
```

Notice how, similar in spirit to our previous iterations of this code, we can utilize the key names to index into our student dictionary.

请注意，在精神上与我们之前的代码迭代类似，我们如何利用键名称来索引我们的学生字典。

Classes 类

- Classes are a way by which, in object-oriented programming, we can create our own type of data and give them names.

  在面向对象编程中，类是一种我们可以创建自己的数据类型并为其命名的方法。

- A class is like a mold for a type of data – where we can invent our own data type and give them a name.

  类就像一种数据类型的模具——我们可以在其中发明自己的数据类型并给它们命名。

- We can modify our code as follows to implement our own class called `Student`:

  我们可以按如下方式修改我们的代码来实现我们自己的名为 `Student` 的类：

```
class Student:
    ...


def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    student = Student()
    student.name = input("Name: ")
    student.house = input("House: ")
    return student


if __name__ == "__main__":
    main()
```

Notice by convention that `Student` is capitalized. Further, notice the `...` simply means that we will later return to finish that portion of our code. Further, notice that in `get_student`, we can create a `student` of class `Student` using the syntax `student = Student()`. Further, notice that we utilize "dot notation" to access attributes of this variable `student` of class `Student`.

按照惯例，请注意 `Student` 是大写的。此外，请注意 `...` 只是意味着我们稍后将返回以完成该部分代码。此外，请注意，在 `get_student` 中，我们可以使用语法 `student = Student()` 创建类 `Student` 的 `student`。此外，请注意，我们使用"点表示法"来访问类 `Student` 的这个变量 `student` 的属性。

- Any time you create a class and you utilize that blueprint to create something, you create what is called an "object" or an "instance". In the case of our code, `student` is an object.

  每当你创建一个类并利用该蓝图创建某些东西时，你都会创建所谓的"对象"或"实例"。在我们的代码中，`student` 是一个对象。

- Further, we can lay some groundwork for the attributes that are expected inside an object whose class is `Student`. We can modify our code as follows:

  此外，我们可以为 class 为 `Student` 的对象中预期的 attributes 奠定一些基础。我们可以按如下方式修改我们的代码：

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house


def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    student = Student(name, house)
    return student


if __name__ == "__main__":
    main()
```

Notice that within `Student`, we standardize the attributes of this class. We can create a function within `class Student`, called a "method", that determines the behavior of an object of class `Student`. Within this function, it takes the `name` and `house` passed to it and assigns these variables to this object. Further, notice how the constructor `student = Student(name, house)` calls this function within the `Student` class and creates a `student`. `self` refers to the current object that was just created.

请注意，在 `Student` 中，我们标准化了这个类的属性。我们可以在 `Student` 类中创建一个函数，称为"方法"，它决定 `Student` 类对象的行为。在这个函数中，它接受传递给它的 `name` 和 `house`，并将这些变量分配给这个对象。此外，请注意构造函数如何在 `Student` 类中 `student = Student(name, house)` 调用此函数并创建 `student`。`self` 引用刚刚创建的当前对象。

- We can simplify our code as follows:

  我们可以按如下方式简化我们的代码：

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house


def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)


if __name__ == "__main__":
    main()
```

Notice how `return Student(name, house)` simplifies the previous iteration of our code where the constructor statement was run on its own line.

请注意 `return Student(name, house)` 如何简化我们代码的上一个迭代，其中 constructor 语句在其自己的行上运行。

- You can learn more in Python's documentation of classes.

  您可以在 Python 的类文档中了解更多信息。

`raise` 举

- Object-oriented program encourages you to encapusulate all the functionality of a class within the class definition. What if something goes wrong? What if someone tries to type in something random? What if someone tries to create a student without a name? Modify your code as follows:

  面向对象的程序鼓励您将类的所有功能封装在类定义中。如果出现问题怎么办？如果有人试图随机输入内容怎么办？如果有人尝试创建没有姓名的学生，该怎么办？按如下方式修改代码：

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Missing name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house


def main():
    student = get_student()
    print(f"{student.name} from {student.house}")


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)


if __name__ == "__main__":
    main()
```

  Notice how we check now that a name is provided and a proper house is designated. It turns out we can create our own exceptions that alerts the programmer to a potential error created by the user called `raise`. In the case above, we raise `ValueError` with a specific error message.

  请注意我们现在如何检查是否提供了名称并指定了合适的房屋。事实证明，我们可以创建自己的异常，以提醒程序员用户创建的潜在错误 `raise` 。在上面的例子中，我们引发 `ValueError` 并带有特定的错误消息。

- It just so happens that Python allows you to create a specific function by which you can print the attributes of an object. Modify your code as follows:

  碰巧的是，Python 允许您创建一个特定的函数，通过该函数可以打印对象的属性。按如下方式修改代码：

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"


def main():
    student = get_student()
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patronus: ")
    return Student(name, house, patronus)


if __name__ == "__main__":
    main()
```

  Notice how `def __str__(self)` provides a means by which a student is returned when called. Therefore, you can now, as the programmer, print an object, its attributes, or almost anything you desire related to that object.

  请注意 `def __str__(self)` 如何提供一种在调用时返回 student 的方法。因此，作为程序员，您现在可以打印对象、其属性或与该对象相关的几乎任何所需内容。

- `__str__` is a built-in method that comes with Python classes. It just so happens that we can create our own methods for a class as well! Modify your code as follows:

  `__str__` 是 Python 类附带的内置方法。碰巧我们也可以为 class 创建自己的方法！按如下方式修改代码：

```
class Student:
    def __init__(self, name, house, patronus=None):
        if not name:
            raise ValueError("Missing name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        if patronus and patronus not in ["Stag", "Otter", "Jack Russell terrier"]:
            raise ValueError("Invalid patronus")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"

    def charm(self):
        match self.patronus:
            case "Stag":
                return "🦌"
            case "Otter":
                return "🦦"
            case "Jack Russell terrier":
                return "🐶"
            case _:
                return "🪄"


def main():
    student = get_student()
    print("Expecto Patronum!")
    print(student.charm())


def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus = input("Patronus: ") or None
    return Student(name, house, patronus)


if __name__ == "__main__":
    main()
```

Notice how we define our own method `charm`. Unlike dictionaries, classes can have built-in functions called methods. In this case, we define our `charm` method where specific cases have specific results. Further, notice that Python has the ability to utilize emojis directly in our code.

注意我们如何定义自己的 method `charm`。与字典不同，类可以具有称为方法的内置函数。在这种情况下，我们定义了 `charm` 方法，其中特定情况具有特定结果。此外，请注意，Python 能够直接在我们的代码中使用表情符号。

- Before moving forward, let us remove our patronus code. Modify your code as follows:

在继续之前，让我们删除我们的 patronus 代码。按如下方式修改代码：

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Invalid name")
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"


def main():
    student = get_student()
    student.house = "Number Four, Privet Drive"
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)


if __name__ == "__main__":
    main()
```

Notice how we have only two methods: `__init__` and `__str__`.

请注意，我们只有两种方法：`__init__` 和 `__str__`。

Decorators 装饰

- Properties can be utilized to harden our code. In Python, we define properties using function "decorators", which begin with `@`. Modify your code as follows:

属性可以用来强化我们的代码。在 Python 中，我们使用函数“decorators”定义属性，函数以 `@` 开头。按如下方式修改代码：

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Invalid name")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    # Getter for house
    @property
    def house(self):
        return self._house

    # Setter for house
    @house.setter
    def house(self, house):
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self._house = house


def main():
    student = get_student()
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)


if __name__ == "__main__":
    main()
```

Notice how we've written `@property` above a function called `house`. Doing so defines `house` as a property of our class. With `house` as a property, we gain the ability to define how some attribute of our class, `_house`, should be set and retrieved. Indeed, we can now define a function called a "setter", via `@house.setter`, which will be called whenever the house property is set—for example, with `student.house = "Gryffindor"`. Here, we've made our setter validate values of `house` for us. Notice how we raise a `ValueError` if the value of `house` is not any of the Harry Potter houses, otherwise, we'll use `house` to update the value of `_house`. Why `_house` and not `house`? `house` is a property of our class, with functions via which a user attempts to set our class attribute. `_house` is that class attribute itself. The leading underscore, `_`, indicates to users they need not (and indeed, shouldn't!) modify this value directly. `_house` should *only* be set through the `house` setter. Notice how the `house` property simply returns that value of `_house`, our class attribute that has presumably been validated using our `house` setter. When a user calls `student.house`, they're getting the value of `_house` through our `house` "getter".

请注意我们是如何在名为 `house` 的函数上面编写 `@property` 的。这样做将 `house` 定义为我们类的属性。通过将 `house` 作为属性，我们能够定义如何设置和检索类的某些属性 `_house`。事实上，我们现在可以通过 `@house.setter` 定义一个名为“setter”的函数，每当设置 house 属性时都会调用该函数——例如，使用 `student.house = "Gryffindor"`。在这里，我们让 setter 为我们验证了 `house` 的值。请注意，如果 `house` 的值不是任何哈利波特学院，我们将如何引发 `ValueError`，否则，我们将使用 `house` 来更新 `_house` 的值。为什么 `_house` 而不是 `house`？`house` 是我们类的一个属性，用户可以通过这些函数来尝试设置我们的 class 属性。`_house` class 属性本身。前导下划线 `_` 向用户表示他们不需要（实际上也不应该）直接修改此值。`_house` *只能通过* 房屋 设置器进行设置。请注意 `house` 属性是如何简单地返回 `_house` 的值，这是我们的 class 属性，它可能已经使用我们的 `house` setter 进行了验证。当用户调用 `student.house` 时，他们通过我们的 `house` “getter”获取 `_house` 的值。

- In addition to the name of the house, we can protect the name of our student as well. Modify your code as follows:

除了学院的名字，我们还可以保护我们学生的名字。按如下方式修改代码：

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    # Getter for name
    @property
    def name(self):
        return self._name
```

```
        return self._name

    # Setter for name
    @name.setter
    def name(self, name):
        if not name:
            raise ValueError("Invalid name")
        self._name = name

    @property
    def house(self):
        return self._house

    @house.setter
    def house(self, house):
        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
            raise ValueError("Invalid house")
        self._house = house


def main():
    student = get_student()
    print(student)


def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)


if __name__ == "__main__":
    main()
```

Notice how, much like the previous code, we provide a getter and setter for the name.

请注意，与前面的代码非常相似，我们为名称提供了 getter 和 setter。

- You can learn more in Python's documentation of methods.

  您可以在 Python 的方法文档中了解更多信息。

Connecting to Previous Work in this Course

连接到本课程中的先前作业

- While not explicitly stated in past portions of this course, you have been using classes and objects the whole way through.

  虽然在本课程的前面部分没有明确说明，但您在整个过程中一直在使用类和对象。

- If you dig into the documentation of `int`, you'll see that it is a class with a constructor. It's a blueprint for creating objects of type `int`. You can learn more in Python's documentation of `int`.

  如果你深入研究 `int` 的文档，你会发现它是一个带有构造函数的类。它是创建 `int` 类型的对象的蓝图。您可以在 Python 的 `int` 文档中了解更多信息。

- Strings too are also a class. If you have used `str.lower()`, you were using a method that came within the `str` class. You can learn more in Python's documentation of `str`.

  字符串也是一个类。如果您使用了 `str.lower()`，那么您使用的是 `str` 类中的方法。您可以在 Python 的 `str`.

- `list` is also a class. Looking at that documentation for `list`, you can see the methods that are contained therein, like `list.append()`. You can learn more in Python's documentation of `list`.

  `list` 也是一个类。查看 `list` 的文档，你可以看到其中包含的方法，比如 `list.append()`。您可以在 Python 的 `list` 文档中了解更多信息。

- `dict` is also a class within Python. You can learn more in Python's documentation of `dict`.

  `dict` 也是 Python 中的一个类。您可以在 Python 的 `dict` 文档中了解更多信息。

- To see how you have been using classes all along, go to your console and type `code type.py` and then code as follows:

  要查看您一直以来如何使用类，请转到控制台并键入 `code type.py`，然后按如下所示键入 code：

```
print(type(50))
```

  Notice how by executing this code, it will display that the class of `50` is `int`.

  请注意，通过执行此代码，它将显示 `50` 的类是 `int`。

- We can also apply this to `str` as follows:

  我们也可以将其应用于 `str`，如下所示：

```
print(type("hello, world"))
```

  Notice how executing this code will indicate this is of the class `str`.

  请注意，执行此代码将如何指示这是 `str` 类。

- We can also apply this to `list` as follows:

  我们也可以将其应用于 `list`，如下所示：

```
print(type([]))
```

  Notice how executing this code will indicate this is of the class `list`.

  请注意执行此代码将如何指示这是类 列表 的。

- We can also apply this to a `list` using the name of Python's built-in `list` class as follows:

  我们还可以使用 Python 的内置 `list` 类的名称将其应用于 列表，如下所示：

```
print(type(list()))
```

  Notice how executing this code will indicate this is of the class `list`.

  请注意执行此代码将如何指示这是类 列表 的。

- We can also apply this to `dict` as follows:

  我们也可以将其应用于 `dict`，如下所示：

```
print(type({}))
```

  Notice how executing this code will indicate this is of the class `dict`.

  请注意执行此代码将如何指示这是 `dict` 类的。

- We can also apply this to a `dict` using the name of Python's built in `dict` class as follows:

我们还可以使用 Python 的内置 `dict` 类的名称将其应用于 `dict`，如下所示：

```
print(type(dict()))
```

Notice how executing this code will indicate this is of the class `dict`.

请注意执行此代码将如何指示这是 `dict` 类的。

Class Methods 类方法

- Sometimes, we want to add functionality to a class itself, not to instances of that class.

  有时，我们想为类本身添加功能，而不是为该类的实例添加功能。

- `@classmethod` is a function that we can use to add functionality to a class as a whole.

  `@classmethod` 是一个函数，我们可以使用它来向整个类添加功能。

- Here's an example of *not* using a class method. In your terminal window, type `code hat.py` and code as follows:

  下面是一个*不使用* class 方法的示例。在终端窗口中，键入 `code hat.py` 和 code，如下所示：

```
import random


class Hat:
    def __init__(self):
        self.houses = ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]

    def sort(self, name):
        print(name, "is in", random.choice(self.houses))


hat = Hat()
hat.sort("Harry")
```

Notice how when we pass the name of the student to the sorting hat, it will tell us what house is assigned to the student. Notice that `hat = Hat()` instantiates a `hat`. The `sort` functionality is always handled by the *instance* of the class `Hat`. By executing `hat.sort("Harry")`, we pass the name of the student to the `sort` method of the particular instance of `Hat`, which we've called `hat`.

请注意，当我们将学生的姓名传递给分院帽时，它会告诉我们分配给学生的学院。请注意，`hat = Hat()` 实例化一个 `hat`。排序功能始终由类 `Hat` 的*实例*处理。通过执行 `hat.sort("Harry")`，我们将学生的姓名传递给 `Hat` 的特定实例的 `sort` 方法，我们称之为 `hat`。

- We may want, though, to run the `sort` function without creating a particular instance of the sorting hat (there's only one, after all!). We can modify our code as follows:

  不过，我们可能希望在不创建 sorting hat 的特定实例的情况下运行 `sort` 函数（毕竟只有一个！我们可以按如下方式修改我们的代码：

```
import random


class Hat:

    houses = ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]

    @classmethod
    def sort(cls, name):
        print(name, "is in", random.choice(cls.houses))


Hat.sort("Harry")
```

Notice how the `__init__` method is removed because we don't need to instantiate a hat anywhere in our code. `self`, therefore, is no longer relevant and is removed. We specify this `sort` as a `@classmethod`, replacing `self` with `cls`. Finally, notice how `Hat` is capitalized by convention near the end of this code, because this is the name of our class.

请注意 `__init__` 方法是如何被删除的，因为我们不需要在代码中的任何位置实例化 hat。因此，`self` 不再相关并被删除。我们将此 排序 指定为 `@classmethod`，将 `self` 替换为 `cls`。最后，请注意 `Hat` 是如何按照约定在此代码末尾处大写的，因为这是我们类的名称。

- Returning back to `students.py` we can modify our code as follows, addressing some missed opportunities related to `@classmethod`s:

  回到 `students.py` 我们可以按如下方式修改我们的代码，解决一些与 `@classmethod` 相关的错失机会：

```
class Student:
    def __init__(self, name, house):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    @classmethod
    def get(cls):
        name = input("Name: ")
        house = input("House: ")
        return cls(name, house)


def main():
    student = Student.get()
    print(student)


if __name__ == "__main__":
    main()
```

Notice that `get_student` is removed and a `@classmethod` called `get` is created. This method can now be called without having to create a student first.

请注意，`get_student` 已删除，并创建了一个名为 `get` 的`@classmethod`。现在可以调用此方法，而无需先创建 student。

Static Methods 静态方法

- It turns out that besides `@classmethod`s, which are distinct from instance methods, there are other types of methods as well.

  事实证明，除了不同于实例方法的 `@classmethod` 之外，还有其他类型的方法。

- Using `@staticmethod` may be something you might wish to explore. While not covered explicitly in this course, you are welcome to go and learn more about static methods and their distinction from class methods.

  使用 `@staticmethod` 可能是您可能希望探索的事情。虽然本课程没有明确介绍，但欢迎你去学习更多关于静态方法的信息以及它们与类方法的区别。

Inheritance 遗产

- Inheritance is, perhaps, the most powerful feature of object-oriented programming.

  继承可能是面向对象编程最强大的功能。

- It just so happens that you can create a class that "inherits" methods, variables, and attributes from another class.

  碰巧的是，您可以创建一个从另一个类"继承"方法、变量和属性的类。

- In the terminal, execute `code wizard.py`. Code as follows:

  在终端中，执行 `code wizard.py` 。代码如下：

```
class Wizard:
    def __init__(self, name):
        if not name:
            raise ValueError("Missing name")
        self.name = name

    ...


class Student(Wizard):
    def __init__(self, name, house):
        super().__init__(name)
        self.house = house

    ...


class Professor(Wizard):
    def __init__(self, name, subject):
        super().__init__(name)
        self.subject = subject

    ...


wizard = Wizard("Albus")
student = Student("Harry", "Gryffindor")
professor = Professor("Severus", "Defense Against the Dark Arts")
...
```

Notice that there is a class above called `Wizard` and a class called `Student`. Further, notice that there is a class called `Professor`. Both students and professors have names. Also, both students and professors are wizards. Therefore, both `Student` and `Professor` inherit the characteristics of `Wizard`. Within the "child" class `Student`, `Student` can inherit from the "parent" or "super" class `Wizard` as the line `super().__init__(name)` runs the `init` method of `Wizard`. Finally, notice that the last lines of this code create a wizard called Albus, a student called Harry, and so on.

请注意，上面有一个名为 `Wizard` 的类和一个名为 `Student` 的类。此外，请注意，有一个名为 `Professor` 的类。学生和教授都有名字。此外，学生和教授都是巫师。因此， `Student` 和 `Professor` 都继承了 `Wizard` 的特性。在 "child" 类 `Student` 中， `Student` 可以从 "parent" 或 "super" 类 `Wizard` 继承，因为 `super()._init__(name)` 行运行 `Wizard` 的 `init` 方法。最后，请注意，此代码的最后几行创建了一个名为 Albus 的向导、一个名为 Harry 的学生，依此类推。

Inheritance and Exceptions

继承和异常

- While we have just introduced inheritance, we have been using this all along during our use of exceptions.

  虽然我们刚刚引入了继承，但我们在使用异常时一直在使用它。

- It just so happens that exceptions come in a heirarchy, where there are children, parent, and grandparent classes. These are illustrated below:

  碰巧的是，异常出现在 heirarch 中，其中有 children、parent 和 grandparent 类。这些示例如下所示：

```
BaseException
 +-- KeyboardInterrupt
 +-- Exception
      +-- ArithmeticError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- KeyError
      +-- NameError
      +-- SyntaxError
      |    +-- IndentationError
      +-- ValueError
 ...
```

- You can learn more in Python's documentation of exceptions.

  您可以在 Python 的异常文档中了解更多信息。

Operator Overloading 作员重载

- Some operators such as `+` and `-` can be "overloaded" such that they can have more abilities beyond simple arithmetic.

  某些运算符（如 `+` 和 `-` ）可以"重载"，以便它们可以具有超出简单算术的更多功能。

- In your terminal window, type `code vault.py`. Then, code as follows:

  在终端窗口中，键入 `code vault.py` 。然后，编写如下代码：

```
class Vault:
    def __init__(self, galleons=0, sickles=0, knuts=0):
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f"{self.galleons} Galleons, {self.sickles} Sickles, {self.knuts} Knuts"

    def __add__(self, other):
        galleons = self.galleons + other.galleons
        sickles = self.sickles + other.sickles
        knuts = self.knuts + other.knuts
        return Vault(galleons, sickles, knuts)


potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)
```

```
total = potter + weasley
print(total)
```

Notice how the `__str__` method returns a formatted string. Further, notice how the `__add__` method allows for the addition of the values of two vaults. `self` is what is on the left of the `+` operand. `other` is what is right of the `+`.

请注意 `__str__` 方法如何返回格式化字符串。此外，请注意 `__add__` 方法如何允许添加两个 vault 的值。`self` 是 `+` 作数左侧的内容。`other` 是 `+` 的右侧。

- You can learn more in Python's documentation of operator overloading.

  您可以在 Python 的运算符重载文档中了解更多信息。

Summing Up 总结

Now, you've learned a whole new level of capability through object-oriented programming.

现在，您已经通过面向对象的编程学习了全新的功能水平。

- Object-oriented programming

  面向对象的编程
- Classes 类
- `raise` 举
- Class Methods 类方法
- Static Methods 静态方法
- Inheritance 遗产
- Operator Overloading 作员重载