CS50x Iran

Object-Oriented Programming

Week note

پاییز و زمستان ۱۴۰۴

1 فهرست

- در برنامه نویسی، الگوهای مختلفی وجود دارند. همانطور که زبان‌های برنامه‌نویسی دیگر را یاد می‌گیرید، شروع به تشخیص این الگوها می‌کنید.
- تا به حال، با روش مرحله به مرحله (step by step) کار کرده‌اید.
- برنامه‌نویسی شیءگرا (OOP) یک راه‌حل قابل توجه برای مشکلات مربوط به برنامه‌نویسی است.
- برای شروع، `code student.py` را در پنجره ترمینال تایپ کنید و کد زیر را اجرا کنید:

student.py

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

- توجه کنید که این برنامه، الگوی مرحله به مرحله را دنبال می‌کند. مانند آنچه که در بخش‌های قبلی این دوره دیدید.

- با تکیه بر کارهایی که در هفته های گذشته انجام دادیم، می‌توانیم توابعی را برای جدا کردن بخش‌های مختلف این برنامه ایجاد کنیم.

student.py

```
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()
```

- توجه کنید که چطور توابع `get_name` و `get_house` برخی نیازهای تابع `main`ما را به صورت جدا از هم برآورده می‌کنند. علاوه بر این، توجه کنید که خطوط پایانی کد بالا به مفسر می‌گویند که تابع `main` را اجرا کند.

- می‌توانیم برنامه خود را با ذخیره مشخصات دانشجویان، به صورت یک tuple نیز ساده تر کنیم. Tupleها یک دنباله از مقادیر هستند. بر خلاف لیست‌ها، Tupleها قابل تغییر نیستند (که اصطلاحاً به آنها immutable گفته می‌شود. به لیست‌ها که قابل تغییر هستند mutable می‌گویند). در حقیقت با این روش دو مقدار را برمی‌گردانیم.

student.py

```
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()
```

- توجه کنید که تابع `get_student` مقادیر را به صورت `name, house` برمی‌گرداند.

- به صورت زیر، با قرار دادن متغیر ها درون پرانتز باز و بسته که یک tuple می‌سازیم. می‌توانیم کد خود را به گونه‌ای تغییر دهیم که قادر به برگرداندن هر دو آیتم `house` و `name` به یک متغیر به نام `student` باشیم.

student.py

```
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()
```

- توجه کنید که `(name, house)` به وضوح به هر کسی که کد ما را می‌خواند می‌گوید که ما دو مقدار را در یک (tuple) برمی‌گردانیم. علاوه بر این، توجه کنید که ما می‌توانیم با استفاده از `student[0]` یا `student[0]` به مقادیر درون tuple دسترسی داشته باشیم.

- Tupleها غیرقابل تغییر هستند، به این معنی که ما نمی‌توانیم مقادیر آن را تغییر دهیم. تغییرناپذیری راهی است که می‌توانیم با استفاده از آن محتاطانه برنامه بنویسیم. به این معنی که از کد خود در برابر تغییرات ناخواسته مراقبت کنیم:

student.py

```
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()
```

- توجه کنید که اجرای این کد منجر به خطا می‌شود. چرا که Tupleها غیرقابل تغییر هستند، و نمی‌توانیم مقدار `student[1]` را دوباره تعیین کنیم یا تغییر دهیم.

- اگر می‌خواستیم انعطاف پذیری بیشتری در برنامه خود به خرج دهیم، می‌توانستیم از یک لیست به صورت زیر استفاده کنیم:

student.py

```
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()
```

- توجه کنید که Listها قابل تغییر هستند. این بدان معناست که یک برنامه نویس می‌تواند ترتیب house و name را تغییر دهد. ممکن است در برخی موارد بخواهید با قبول کاهش امنیت کد خود، از این ویژگی برای فراهم آوردن انعطاف پذیری بیشتر استفاده کنید. با این حال، اگر ترتیب این مقادیر قابل تغییر باشد، برنامه‌نویسانی که با شما همکاری می‌کنند ممکن است در آینده دچار اشکالاتی شوند و یا مشکلاتی را به وجود بیاورند.

- همچنین، در این برنامه، می‌توان از یک دیکشنری نیز استفاده کرد. به یاد دارید که دیکشنری ها شامل جفت کلید-مقدار (key - value) هستند.

student.py

```
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()
```

- توجه کنید که در این حالت، دو جفت کلید-مقدار برگردانده می‌شود. یک مزیت این روش این است که ما می‌توانیم با استفاده از کلید ها مقادیر را در این دیکشنری تغییر دهیم، یا مقادیر جدید وارد کنیم.

- کد ما هنوز هم می‌تواند بهتر شود. توجه کنید که یک متغیر غیر ضروری در کد وجود دارد. ما می‌تونیم `student = {}` را حذف کنیم چون نیازی به ساخت یک دیکشنری خالی نداریم.

student.py

```
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()
```

- توجه کنید که می‌توانیم در یک خط با استفاده از از آکولاد باز و بسته {} جلوی `return` یک دیکشنری ساخته و آن را برگردانیم.

- ما می‌توانیم مورد خاص خود مثلا نام Padma را نیز در نسخه نوشته شده با دیکشنری کدمان اضافه کنیم:

student.py

```
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()
```

- توجه کنید که چطور مانند نسخه قبلی این کد، می‌توانیم از نام‌های کلیدی برای دسترسی به مقادیر دیکشنری `student` استفاده کنیم.

- در برنامه نویسی شئ‌گرا، کلاس ها یک روش برای ایجاد دیتا تایپ (نوع داده) شخصی با نام دلخواه می‌باشند.
- یک کلاس مانند قالب برای یک نوع از داده است، که ما می‌توانیم در آن دیتا تایپ خود را بسازیم و به آن اسم بدهیم.
- ما می‌توانیم کد خود را به صورت زیر تغییر دهیم تا کلاس خود را با نام Student بسازیم:

student.py

```
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()
```

- توجه کنید که طبق قاعده نام گذاری کلاس ها، Student با حرف بزرگ شروع شده است. همچنین توجه کنید که (...) بدین معناست که در ادامه به این قسمت برمی‌گردیم تا کد را کامل کنیم. دقت کنید که در تابع get_student، ما می‌توانیم یک متغیر student از کلاس Student با استفاده از سینتکس student = Student() بسازیم. همچنین توجه کنید که ما از "علامت نقطه" برای دسترسی به ویژگی های این متغیر student از کلاس Student استفاده می‌کنیم.

- هر زمان که یک کلاس را ایجاد می‌کنید و از آن قالب برای ساخت چیزی استفاده می‌کنید، شما یک "شی (object)" یا یک "نمونه" را ایجاد می‌کنید. در مورد کد ما، `student` یک شی (object یا instance) است.
- علاوه بر این، ما می‌توانیم یک سری نیازهای اولیه از ویژگی های پایه ای را که درون یک شی از کلاس `Student` مورد انتظار است، در آن قرار دهیم. برای این کار می‌توانیم کد خود را به صورت زیر تغییر دهیم:

student.py

```
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()
```

- توجه کنید که در داخل کلاس `Student`، ما ویژگی های این کلاس را تعریف کرده ایم. ما می‌توانیم یک تابع داخل `class Student` ایجاد کنیم که به آن method می‌گویند، و رفتار یک شئ از کلاس `Student` را تعیین می‌کند. درون این تابع، کلاس ما `name` و `house` را که به آن فرستاده شده است می‌گیرد و این متغیرها را به این شئ اختصاص می‌دهد. علاوه بر این، توجه کنید که دستور سازنده شئ یعنی `student = Student(name, house)`، تابع `__init__` را در کلاس `Student` فراخوانی می‌کند و یک `student` را ایجاد می‌کند. `self` به شئ ای که تازه ایجاد شده است اشاره می‌کند. می‌توانیم کد خود را به صورت زیر ساده تر کنیم. همچنین توجه کنید که با عبارت `return Student(name, house)` نسخه قبلی کد خود را که دستور سازنده شئ در یک خط جدا اجرا می‌شد، ساده تر کرده ایم.

student.py

```
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()
```

student.py

```python
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()
```

برنامه شئ‌گرا شما را تشویق می‌کند تا تمام عملکردهای یک کلاس (مانند functionها) را در تعریف کلاس محصور کنید. اگر مشکلی پیش بیاید چه می‌شود؟ اگر کسی بخواهد چیزی تصادفی تایپ کند چه می‌شود؟ اگر کسی بخواهد دانش آموزی بدون نام ایجاد کند چه می‌شود؟ کد خود را به صورت زیر تغییر دهید. توجه کنید که اکنون چگونه بررسی می‌کنیم که نامی ارائه شده باشد و خانه (در این مثال، گروه های مدرسه هاگوارتز) مناسبی تعیین شده است. ما می‌توانیم استثناهای خود را ایجاد کنیم که برنامه نویس را از یک خطای احتمالی ایجاد شده توسط کاربر به کمک `raise` آگاه کند. در این موارد، ما `ValueError` را با یک پیام خطای خاص `raise` می‌کنیم.

student.py

```python
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()
```

پایتون به شما اجازه می‌دهد تا یک تابع خاص ایجاد کنید که توسط آن می‌توانید ویژگی های یک شئ را چاپ کنید. کد خود را به صورت زیر تغییر دهید. توجه داشته باشید که چگونه `__str__(self)` این امکان را می‌دهد که به وسیله آن متنی که حاوی اطلاعات دانش آموز است بازگردانده شود. بنابراین، اکنون می‌توانید به‌عنوان برنامه‌نویس، یک شئ، ویژگی‌های آن، یا تقریباً هر چیزی را که در رابطه با آن شئ می‌خواهید چاپ کنید.

- `__str__` یک متد داخلی (built-in) است و این یعنی به صورت پیش فرض در کلاسی که شما ایجاد کرده‌اید وجود دارد. ولی علاوه بر متدهای داخلی ما می‌توانیم متدهای خودمان را برای یک کلاس نیز ایجاد کنیم! کد خود را به صورت صفحه‌ی بعد تغییر دهید.
- توجه کنید که چگونه متد `charm` خود را تعریف می‌کنیم. برخلاف دیکشنری ها، کلاس ها می‌توانند توابعی داخلی (function) به نام متد (method) داشته باشند (دقت داشته باشید که به functionهایی که داخل یک کلاس نوشته شوند method گفته میشود). در این مورد، متد `charm` را به صورتی که برای مقادیر خاص ایموجی خاصی برگرداند، تعریف می‌کنیم. علاوه بر این، توجه داشته باشید که پایتون این توانایی را دارد که از ایموجی ها به طور مستقیم در کد ما استفاده کند.

student.py

```python
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()
```

- قبل از اینکه جلوتر برویم، ابتدا اجازه دهید تکه کد مرتبط به چک کردن `patronus` را از برنامه خود را حذف کنیم.
- کد خود را به صورت زیر تغییر دهید.
- دقت داشته باشید که حالا ما فقط دو متد `__str__` و `__init__` که هر دو متد داخلی هستند را در این کلاس خواهیم داشت.

student.py

```
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()
```

student.py

```python
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()
```

properties را می‌توان برای افزایش ایمنی کد استفاده کرد. در زبان پایتون ما می‌توانیم با استفاده از decoratorها به پیاده‌سازی properties بپردازیم، دقت داشته باشید که decoratorها با علامت @ شروع می‌شوند.

کد خود را به صورت رو به رو تغییر دهید:

توجه کنید که کدی که در صفحه‌ی قبل به آن اشاره شده، یک متد ساده است که به عنوان مثال قرار داده شده تا مفهوم property را شرح دهد.
دقت کنید که به چه صورتی @property را بالای تابعی به نام house نوشته‌ایم. با این کار، houseرا به عنوان یکproperty و خاصیت در کلاسِ خود تعریف کرده‌ایم. ما با تعریف house به عنوان یک property، قابلیت ویرایش و بازیابی ویژگیهای کلاسِ مثلا _house را بصورت کنترل شده، بدست آورده‌ایم. با تعریف house به عنوان property، می‌توانیم تابعی به نام setter را با استفاده از @house.setter تعریف کنیم و زمانیکه ویژگی house ویرایش و بازیابی شد، به‌صورت خودکار فراخوانی می‌شود. دستور student.house = "Gryffindor" مثالی برای تنظیم ویژگی house است و setter که در پشت این خط دستوری تعریف شده، علاوه بر تنظیم ویژگی house، مقدار آن را نیز بررسی می‌کند و در صورت نامعتبر بودن، خطایی مانند ValueError برمیگرداند. یعنی در صورتی که مقدار ورودی برای house از لیست خانه‌های هری‌پاتر ساخته نشده باشد، ما یک ValueError دریافت خواهیم کرد. در غیر اینصورت، مقدار house را برای به روزرسانی _house بکار می‌بریم.
حالا چرا از _house به جای house استفاده کرده‌ایم؟ در صفحه‌ی بعد به این نکته می‌پردازیم.

بیایید مجددا راجب این موضوع صحبت کنیم که چرا بجای `student._house` از `student.house` برای آپدیت کردن این مقدار استفاده کرده ایم؟ `house` یک property از کلاس ما است، و دلیل حضور آن این است که در خارج از کلاس اجازه اینکه مقدار یک object یا شئ تغییر کند، به سادگی داده نشود و بتوان یکسری validation یا هر نوع نظارت بر روی مقدار جدیدی که می‌خواهیم جایگزین مقدار اولیه کنیم داشته باشیم. در کد بالا `_house` مقدار اصلی متغییر ما است، برای مثال اگر شئ ای از کلاس `Student` داشته باشیم که مقدار اولیه `house` در آن Slytherin باشد، در واقع `_house` حاوی همان مقدار slytherin است. حال زمانی که ما متدی به نام `house` داشته باشیم که به کمک `@property` و `@house.setter` برای متغییر `_house` خود ساخته ایم، در واقع یک getter که برای زمانی که می‌خوایم مقدار را داشته باشیم و آن را آپدیت نکنیم و یک `setter` برای زمانی که می‌خوایم مقدار متغییر را آپدیت کنیم خواهیم داشت، به کمک این دو (getter, setter) ما می‌توانیم اجازه دسترسی مستقیم به مقدار `_house` را ندهیم و نظارت بیشتری برای دیدن و تغییر دادن این متغییر داشته باشیم. در نتیجه زمانی که این دو را در کد خود پیاده سازی کنیم، هر زمان نیاز به گرفتن مقدار `_house` داشتیم به جای `student._house` از `student.house` و هر زمان که نیاز به آپدیت کردن این مقدار داشتیم به جای `student._house = “Ravenclaw”` باید از `student._house = “Ravenclaw”` استفاده کنیم تا به ترتیب متدهای getter و setter ما call و اجرا شوند.

student.py

```
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

    # 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()
```

- کد خود را به صورت روبه‌رو تغییر دهید:
- توجه کنید که دقیقاً مانند کد قبلی، چگونه یک گیرنده (getter) و تنظیم کننده (setter) برای name ارائه می‌کنیم.

- می‌توانید در مستندات methodهای پایتون بیشتر بیاموزید.

- با اینکه در بخش‌های قبلی دوره به طور صریح اشاره نشده است، اما تاکنون در طول این دوره از کلاس‌ها و اشیاء استفاده کرده‌ایم.
- اگر به مستندات (داکیومنت) int رجوع کنید خواهید دید که int یک کلاس به همراه یک constructor است (متد __init__ که به صورت اتوماتیک در زمان ساخت یک شئ اجرا می‌شود). این کلاس یک قالب برای ایجاد اشیا (آبجکت های) از نوع int است. شما می‌توانید با رجوع به مستندات پایتون در مورد int اطلاعات بیشتری را کسب کنید.
- رشته ها (string) نیز یک کلاس هستند. اگر تا به حال از `str.lower()` استفاده کرده باشید در واقع شما در حال استفاده از متدی که داخل کلاس `str` است، بوده‌اید. با مراجعه به مستندات پایتون در مورد str میتوانید اطلاعات کامل تری بدست بیاورید.
- لیست هم یک نوع کلاس است. با نگاه کردن به مستندات برای لیست، می‌توانید متدهایی را که در آن موجود هست، مانند `list.append()` مشاهده کنید. می‌توانید با رجوع به مستندات لیست پایتون بیشتر بیاموزید.
- dict نیز یک کلاس در پایتون است. می‌توانید از مستندات dict در پایتون اطلاعات بیشتری بدست بیاورید.

- برای درک بهتر اینکه چگونه تا با حال با کلاس‌ها در ارتباط بودید، به ترمینال خود بروید و `code type.py` را تایپ کنید و سپس کد زیر را بنویسید. توجه کنید که وقتی این کد را اجرا می‌کنیم به ما نشان می‌دهد که کلاس عدد `50` برابر int است.

```
type.py
print(type(50))
```

- می‌توانیم همان حالت را برای یک string نیز تست کنیم. توجه داشته باشید که خروجی این کد نشان می‌دهد که این متن یک شی از کلاس str است.

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

- همچنین می‌توانیم این حالت را به صورت زیر در list امتحان کنیم. توجه داشته باشید که خروجی این کد نشان می‌دهد که `[]` یک شی از کلاس list است.

```
type.py
print(type([]))
```

- همچنین می‌توانیم کد قبلی را با استفاده از توابع داخلی پایتون یعنی list به صورت زیر اعمال کنیم . خروجی این کد هم مانند خروجی کد قبلی نشان می‌دهد که `list()` برای ما یک شی از کلاس list درست می‌کند.

type.py
```
print(type(list()))
```

- به روش مشابه می‌توانیم همین تست را برای دیکشنری داشته باشیم، روش اول:

type.py
```
print(type({}))
```

- روش دوم:

type.py
```
print(type(dict()))
```

- توجه داشته باشید که خروجی هر دو کد بالا به ما نشان میدهد که چگونه { } یک شی از dict است و ()dict چگونه یک شی از dict برای ما میسازد.

- گاهی اوقات، می‌خواهیم قابلیت‌های جدیدی را به خود کلاس اضافه کنیم و متد های دلخواه خودمان را برای آن پیاده سازی کنیم.
- `@classmethod` یک تابع است که می‌توانیم برای اضافه کردن عملکرد ها و قابلیت های خاص به کل کلاس از آن استفاده کنیم.
- در پایین یک مثال از بدون استفاده از `classmethod` را می‌توانید ببینید. در پنجره ترمینال خود تایپ کنید `code hat.py` و کد زیر را بنویسید. توجه کنید که وقتی نام دانش‌آموز را به sorting hat می‌دهیم، به صورت رندوم و تصادفی به ما گفته می‌شود که کدام خانه به دانش‌آموز تعلق دارد. توجه داشته باشید که `hat = Hat()` یک شی به نام `hat` از کلاس `Hat` ایجاد می‌کند. با اجرای `hat.sort("Harry")`، نام دانش آموز را به متد `sort` که یکی از متد های کلاس `Hat` است می‌دهیم و این متد به صورت رندوم یکی از گروه ها را برای ما به خروجی می‌آورد.

hat.py

```
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")
```

- گاهی اوقات، ممکن است بخواهیم تابع `sort` را بدون اینکه شئ ای از `Hat` ایجاد کنیم، اجرا کنیم. می‌توانیم کد خود را به شکل زیر تغییر دهیم:

hat.py

```
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")
```

- توجه کنید که متد `__init__` حذف شده است زیرا نیازی به ایجاد شئ `hat` در هیچ جای کد خود نداریم. به همین دلیل به `self` نیز دیگر نیازی نداریم، و حذف می‌شود. تابع `self` را به عنوان `@classmethod` تعیین و `self` را با `cls` جایگزین می‌کنیم. در نهایت، توجه کنید که `Hat` طبق قرداد و سنت به صورت کپیتال (حرف اول بزرگ) نوشته شده است، چون `Hat` نام کلاس ماست.

- به فایل students.py برمی‌گردیم، و می‌توانیم کد خود را به شکل زیر تغییر دهیم تا برخی قابلیت‌های از دست رفته مربوط به `@classmethod` را به آن اضافه کنیم . توجه کنید که `get_student` حذف و یک `@classmethod` با نام `get` ایجاد شده است. این متد اکنون می‌تواند بدون ایجاد یک `student` صدا زده شود.

students.py

```
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()
```

- افزون بر `@classmethod`، که با متدهای نمونه (instance methodها) متفاوت هستند، انواع دیگری از متد ها نیز وجود دارد.

- ممکن است شما بخواهید از `@classmethod` استفاده کنید یا حداقل درمورد آن بدانید. اگرچه در این دوره به طور واضح و جامع پوشش داده نشده است، اما شما می‌توانید خودتان به دنبال آن بروید و درمورد متد های استاتیک و تفاوت های آن با متدهای کلاس بیشتر اطلاعات کسب کنید.

- وراثت، احتمالاً مهمترین و قدرتمند ترین ویژگی برنامه نویسی شئ‌گرا است.
- وراثت به این معنی است که شما می‌توانید یک کلاس را ایجاد کنید که متد ها، متغیر ها، و ویژگی ها را از کلاس دیگری به «ارث» ببرد.
- در ترمینال، `code wizard.py` را اجرا کنید و سپس کد زیر را اجرا کنید:

wizard.py

```
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")
```

- توجه کنید که یک کلاس به نام `Wizard` و یک کلاس به نام `Student` وجود دارد. علاوه بر این، توجه داشته باشید که یک کلاس به نام `Professor` نیز وجود دارد. هم دانش آموزان و هم استاد ها دارای نام هستند. همچنین، هم دانش آموزان و هم استادها جادوگر هستند. بنابراین، هر دو کلاس `Student` و `Professor` ویژگی‌های کلاس `Wizard` را به ارث می‌برند. کلاس `Student` که کلاس «فرزند» نامیده می‌شود، می‌تواند از طریق خط `super().__init__(name)` (که متد `init` از `Wizard` را اجرا می‌کند)، از کلاس "والد" یا "`super`" که در اینجا همان کلاس `Wizard` است ارث بری کند. در نهایت، توجه کنید که خطوط آخر این کد، یک جادوگر به نام Albus، یک دانش‌آموز به نام Harry و غیره را ایجاد می‌کنند.

- می‌توانید در اسناد پایتون درباره Exceptionها بیشتر در این مورد مطالعه کنید.
- حال که با مفهوم ارث بری آشنا شدیم، قابل ذکر است که تا الان با استفاده از استثناها یا همان Exceptionها در حال استفاده از ویژگی های ارث بری بودیم.
- استثناها به صورت سلسله مراتبی طراحی شده‌اند که شامل کلاس‌های فرزند، پدر و پدربزرگ می‌شوند. در زیر می‌توانید طرح شماتیک آن را ببینید:

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

- برخی اپراتورها مانند + و - می‌توانند اضافه تعریف "overloaded" شوند تا بتوانند قابلیت‌هایی فراتر از حساب و کتاب های ساده داشته باشند.
- در پنجره ترمینال خود، `code vault.py` را تایپ کنید. سپس، کد زیر را بنویسید:

wizard.py

```
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)
```

- توجه کنید که متد `__str__` یک رشته قالب‌بندی شده (Formatted String) را برمی‌گرداند. علاوه بر این، توجه کنید که متد `__add__` باعث می‌شود بتوانیم مقادیر دو کیف پول را به هم اضافه کنیم. `self` آن چیزی است که در سمت چپ عملگر + قرار دارد. other آنچه در سمت راست عملگر + قرار دارد است.

- برای کسب اطلاعات بیشتر در مورد اضافه تعریف اپراتورها (operator Overloading)، می‌توانید به مستندات پایتون مراجعه کنید.

در این هفته مباحث زیر را مورد بررسی قرار دادیم:

- Object-oriented programming ✓
- Classes ✓
- Raise ✓
- Class Methods ✓
- Static Methods ✓
- Inheritance ✓
- Operator Overloading ✓

حالا شما توانایی این را دارید که در پایتون از دستورات شرطی استفاده کنید و متناسب با نتیجه دریافتی، فرایند موردنظر خود را اجرا کنید.

در این هفته مباحث زیر را مورد بررسی قرار دادیم:

- Object-oriented programming ✓
- Classes ✓
- `raise` ✓
- Class Methods ✓
- Static Methods ✓
- Inheritance ✓
- Operator Overloading ✓

حالا یک توانایی به لیست رو به افزایش توانایی های شما در زبان پایتون اضافه شد. این رشد ادامه داره...

CS50x Iran

Harvard's Computer Science 50x Iran