Lecture 8

Object-Oriented Programming

- There are different paradigms of programming. As you learn other languages, you will start recognizing patterns like these.
- Up until this point, you have worked procedurally step-by-step.
- Object-oriented programming (OOP) is a compelling solution to programming-related problems.
- To begin, type `code student.py` in the terminal window and code as follows:

  ```
  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.
- Drawing on our work from previous weeks, we can create functions to abstract away parts of this program.

  ```
  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.

- 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.

```
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`.

- Packing that `tuple`, such that we are able to return both items to a variable called `student`, we can modify our code as follows.

```
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]`.

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

```
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]`.

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

```
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.

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

```
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.

- 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.

```
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.

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

```
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.

Classes

- Classes are a way by which, in object-oriented programming, we can create our own type of data and give them names.
- A class is like a mold for a type of data – where we can invent our own data type and give them a name.
- We can modify our code as follows to implement our own class called `Student`:

```
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`.
- 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.
- 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:

```
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.

- We can simplify our code as follows:

```
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.

- You can learn more in Python's documentation of classes (https://docs.python.org/3/tutorial/classes.html).

`raise`

- 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:

```
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.

- 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:

```
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.

- `__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:

```
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.

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

```
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__`.

Decorators

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

```
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".

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

```
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()
```

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

- You can learn more in Python's documentation of methods (https://docs.python.org/3/tutorial/classes.html).

Connecting to Previous Work in this Course

- While not explicitly stated in past portions of this course, you have been using classes and objects the whole way through.
- 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` (https://docs.python.org/3/library/functions.html#int).
- 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` (https://docs.python.org/3/library/stdtypes.html#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` (https://docs.python.org/3/library/stdtypes.html#list).
- `dict` is also a class within Python. You can learn more in Python's documentation of `dict` (https://docs.python.org/3/library/stdtypes.html#dict).
- To see how you have been using classes all along, go to your console and type `code type.py` and then code as follows:

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

  Notice how by executing this code, it will display that the class of `50` is `int`.
- We can also apply this to `str` as follows:

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

  Notice how executing this code will indicate this is of the class `str`.
- We can also apply this to `list` as follows:

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

  Notice how executing this code will indicate this is of the class `list`.
- We can also apply this to a `list` using the name of Python's built-in `list` class as follows:

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

  Notice how executing this code will indicate this is of the class `list`.
- We can also apply this to `dict` as follows:

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

  Notice how executing this code will indicate this is of the class `dict`.
- We can also apply this to a `dict` using the name of Python's built in `dict` class as follows:

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

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

Class Methods

- Sometimes, we want to add functionality to a class itself, not to instances of that class.
- `@classmethod` is a function that we can use to add functionality to a class as a whole.
- Here's an example of *not* using a class method. In your terminal window, type `code hat.py` and code as follows:

```
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`.

- 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:

```
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.

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

```
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.

Static Methods

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

Inheritance

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

- It just so happens that you can create a class that "inherits" methods, variables, and attributes from another class.
- In the terminal, execute `code wizard.py`. Code as follows:

```
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.

Inheritance and Exceptions

- While we have just introduced inheritance, we have been using this all along during our use of exceptions.
- It just so happens that exceptions come in a heirarchy, where there are children, parent, and grandparent classes. These are illustrated below:

```
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 (https://docs.python.org/3/library/exceptions.html).

Operator Overloading

- Some operators such as `+` and `-` can be "overloaded" such that they can have more abilities beyond simple arithmetic.
- In your terminal window, type `code vault.py`. Then, code as follows:

```
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} K

    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 `+`.

- You can learn more in Python's documentation of operator overloading (https://docs.python.org/3/reference/datamodel.html#special-method-names).

Summing Up

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

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