# Giới thiệu về lập trình với Python của CS50

OpenCourseWare

Quyên tặng (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

## Bài giảng 8



## Lập trình hướng đối tượng

- Có nhiều mô hình lập trình khác nhau. Khi bạn học các ngôn ngữ khác, bạn sẽ bắt đầu nhận ra các mẫu như thế này.
- Cho đến thời điểm này, bạn đã làm việc theo từng bước một.
- Lập trình hướng đối tượng (OOP) là một giải pháp hấp dẫn cho các vấn đề liên quan đến lập trình.
- Để bắt đầu, hãy nhập `code student.py` vào cửa sổ terminal và mã như sau:

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

Lưu ý rằng chương trình này tuân theo mô hình quy trình từng bước: Giống như bạn đã thấy trong các phần trước của khóa học này.

- Dựa trên công việc của chúng tôi từ những tuần trước, chúng tôi có thể tạo các hàm để trừu tượng hóa các phần của chương trình này.

```
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()
```

Lưu ý cách `get_name` trừu `get_house` tượng hóa một số nhu cầu của `main` chức năng của chúng tôi. Hơn nữa, hãy chú ý cách các dòng cuối cùng của đoạn mã trên báo cho trình biên dịch chạy `main` hàm.

- Chúng ta có thể đơn giản hóa hơn nữa chương trình của mình bằng cách lưu trữ học sinh dưới dạng tệp `tuple`. A `tuple` là một dãy các giá trị. Không giống như a `list`, a `tuple` không thể sửa đổi được. Về mặt tinh thần, chúng tôi đang trả về hai giá trị.

```
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()
```

Lưu ý cách `get_student` trả về `name, house`.

- Đóng gói `tuple` sao cho chúng ta có thể trả lại cả hai mục cho một biến được gọi là `student`, chúng ta có thể sửa đổi mã của mình như sau.

```
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()
```

Lưu ý rằng điều đó `(name, house)` cho bất kỳ ai đọc mã của chúng tôi biết rõ ràng rằng chúng tôi đang trả về hai giá trị trong một. Hơn nữa, hãy chú ý cách chúng ta có thể lập chỉ mục vào `tuple`s bằng cách sử dụng `student[0]` hoặc `student[1]`.

- `tuple`s là bất biến, nghĩa là chúng ta không thể thay đổi những giá trị đó. Tính bất biến là một cách mà chúng ta có thể lập trình một cách phòng thủ.

```
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()
```

Lưu ý rằng mã này tạo ra lỗi. Vì `tuple`s là bất biến nên chúng ta không thể gán lại giá trị của `student[1]`.

- Nếu chúng tôi muốn mang lại sự linh hoạt cho các lập trình viên đồng nghiệp của mình, chúng tôi có thể sử dụng `list` như sau.

```
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()
```

Lưu ý các danh sách có thể thay đổi. Nghĩa là, thứ tự `house` và `name` có thể được chuyển đổi bởi một lập trình viên. Bạn có thể quyết định sử dụng điều này trong một số trường hợp mà bạn muốn mang lại sự linh hoạt hơn nhưng phải trả giá bằng tính bảo mật của mã của mình. Xét cho cùng, nếu thứ tự của các giá trị đó có thể thay đổi, các lập trình viên làm việc với bạn có thể mắc lỗi trong tương lai.

- Một từ điển cũng có thể được sử dụng trong việc thực hiện này. Hãy nhớ lại rằng từ điển cung cấp một cặp khóa-giá trị.

```
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()
```

Lưu ý trong trường hợp này, hai cặp khóa-giá trị được trả về. Ưu điểm của phương pháp này là chúng ta có thể lập chỉ mục vào từ điển này bằng cách sử dụng các khóa.

- Tuy nhiên, mã của chúng tôi có thể được cải thiện hơn nữa. Lưu ý rằng có một biến không cần thiết. Chúng tôi có thể xóa `student = {}` vì không cần tạo từ điển trống.

```
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()
```

Lưu ý rằng chúng ta có thể sử dụng `{}` dấu ngoặc nhọn trong `return` câu lệnh để tạo từ điển và trả về tất cả trong cùng một dòng.

- Chúng tôi có thể cung cấp trường hợp đặc biệt của mình với Padma trong phiên bản từ điển mã của chúng tôi.

```
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()
```

Lưu ý rằng, tương tự như các lần lặp lại mã này trước đây, chúng ta có thể sử dụng các tên khóa để lập chỉ mục vào từ điển học sinh của mình.

## Các lớp ~~học~~ /Classes

- Lớp là một cách mà trong lập trình hướng đối tượng, chúng ta có thể tạo loại dữ liệu của riêng mình và đặt tên cho chúng.
- Lớp giống như khuôn cho một loại dữ liệu - nơi chúng ta có thể tạo ra loại dữ liệu của riêng mình và đặt tên cho chúng.
- Chúng ta có thể sửa đổi mã của mình như sau để triển khai lớp của riêng mình được gọi là `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()
```

Thông báo theo quy ước `Student` được viết hoa. Hơn nữa, hãy lưu ý `...` đơn giản có nghĩa là sau này chúng ta sẽ quay lại để hoàn thành phần mã đó. Hơn nữa, hãy lưu ý rằng trong `get_student`, chúng ta có thể tạo một `student` lớp `Student` bằng cú pháp `student = Student()`. Hơn nữa, hãy lưu ý rằng chúng tôi sử dụng "ký hiệu dấu chấm" để truy cập các thuộc tính của biến `student` lớp này `Student`.

- Bất cứ khi nào bạn tạo một lớp và sử dụng bản thiết kế đó để tạo ra thứ gì đó, bạn sẽ tạo ra thứ được gọi là "đối tượng" hoặc "phiên bản". Trong trường hợp mã của chúng tôi, `student` là một đối tượng.
- Hơn nữa, chúng ta có thể đặt một số nền tảng cho các thuộc tính được mong đợi bên trong một đối tượng có lớp `Student`. Chúng ta có thể sửa đổi mã của mình như sau:

```
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()
```

Lưu ý rằng bên trong `Student`, chúng tôi chuẩn hóa các thuộc tính của lớp này. Chúng ta có thể tạo một hàm bên trong `class Student`, được gọi là "phương thức", xác định hành vi của một đối tượng của lớp `Student`. Trong hàm này, nó lấy `name` và `house` truyền cho nó rồi gán các biến này cho đối tượng này. Hơn nữa, hãy chú ý cách hàm tạo `student = Student(name, house)` gọi hàm này trong `Student` lớp và tạo một tệp `student`. `self` đề cập đến đối tượng hiện tại vừa được tạo.

- Chúng ta có thể đơn giản hóa mã của mình như sau:

```
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()
```

Lưu ý cách `return Student(name, house)` đơn giản hóa việc lặp lại mã trước đó của chúng tôi trong đó câu lệnh khởi tạo được chạy trên dòng riêng của nó.

- Bạn có thể tìm hiểu thêm trong tài liệu về các lớp (https://docs.python.org/3/tutorial/classes.html) của Python .

## raise

- Chương trình hướng đối tượng khuyến khích bạn gói gọn tất cả chức năng của một lớp trong định nghĩa lớp. Lỡ có chuyện gì xảy ra thì sao? Điều gì sẽ xảy ra nếu ai đó cố gắng nhập nội dung nào đó ngẫu nhiên? Điều gì sẽ xảy ra nếu ai đó cố gắng tạo một học sinh không có tên? Sửa đổi mã của bạn như sau:

```
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()
```

Hãy chú ý cách chúng tôi kiểm tra xem tên đã được cung cấp và ngôi nhà thích hợp đã được chỉ định hay chưa. Hóa ra chúng ta có thể tạo các ngoại lệ của riêng mình để cảnh báo cho

người lập trình về một lỗi tiềm ẩn do người dùng tạo ra có tên là `raise`. Trong trường hợp trên, chúng tôi đưa ra `ValueError` một thông báo lỗi cụ thể.

- Tinh cờ là Python cho phép bạn tạo một hàm cụ thể để bạn có thể in các thuộc tính của một đối tượng. Sửa đổi mã của bạn như sau:

```
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()
```

Lưu ý cách `def __str__(self)` cung cấp phương tiện để trả về học sinh khi được gọi. Do đó, với tư cách là lập trình viên, bây giờ bạn có thể in một đối tượng, các thuộc tính của nó hoặc hầu hết mọi thứ bạn muốn liên quan đến đối tượng đó.

- `__str__` là một phương thức tích hợp đi kèm với các lớp Python. Tinh cờ là chúng ta cũng có thể tạo các phương thức của riêng mình cho một lớp! Sửa đổi mã của bạn như sau:

```
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()
```

Lưu ý cách chúng tôi xác định phương pháp riêng của chúng tôi `charm`. Không giống như từ điển, các lớp có thể có các hàm dựng sẵn được gọi là phương thức. Trong trường hợp này, chúng tôi xác định `charm` phương pháp của mình trong đó các trường hợp cụ thể có kết quả cụ thể. Hơn nữa, hãy lưu ý rằng Python có khả năng sử dụng biểu tượng cảm xúc trực tiếp trong mã của chúng tôi.

- Trước khi tiếp tục, chúng ta hãy xóa mã bảo trợ của mình. Sửa đổi mã của bạn như sau:

```
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()
```

Lưu ý rằng chúng tôi chỉ có hai phương thức: `__init__` và `__str__`.

## đồ trang trí /Decorators

- Các thuộc tính có thể được sử dụng để làm cứng mã của chúng tôi. Trong Python, chúng ta xác định các thuộc tính bằng cách sử dụng hàm "trang trí", bắt đầu bằng `@`. Sửa đổi mã của bạn như sau:

```
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()
```

Hãy chú ý cách chúng tôi viết ở `@property` trên một hàm có tên `house`. Làm như vậy được định nghĩa `house` là một thuộc tính của lớp chúng ta. Với `house` tư cách là một thuộc tính, chúng ta có khả năng xác định cách `_house` đặt và truy xuất một số thuộc tính của lớp của chúng ta, ,. Thật vậy, bây giờ chúng ta có thể định nghĩa một hàm được gọi là "setter", thông qua `@house.setter`, hàm này sẽ được gọi bất cứ khi nào thuộc tính house được đặt—ví dụ: với `student.house = "Gryffindor"`. Ở đây, chúng tôi đã xác thực các giá trị setter của `house` chúng tôi. Lưu ý cách chúng tôi tăng a `ValueError` nếu giá trị của `house` không phải là bất kỳ ngôi nhà nào trong Harry Potter, nếu không, chúng tôi sẽ sử dụng `house` để cập nhật giá trị của `_house`. Tại sao `_house` và không `house`? `house` là một thuộc tính của lớp của chúng tôi, với các chức năng mà qua đó người dùng cố gắng đặt thuộc tính lớp của chúng tôi. `_house` chính thuộc tính lớp đó là gì. Dấu gạch dưới ở đầu, `_`, cho người dùng biết rằng họ không cần (và thực tế là không nên!) Sửa đổi trực tiếp giá trị này. *chỉ* `_house` nên được thiết lập thông qua setter. Lưu ý cách thuộc tính chỉ trả về giá trị đó của thuộc tính lớp của chúng tôi có lẽ đã được xác thực bằng cách sử dụng trình thiết lập của chúng tôi. Khi người dùng gọi , họ sẽ nhận được giá trị thông qua "getter" của chúng tôi. `house` `house` `_house` `house` `student.house` `_house` `house`

- Ngoài tên nhà, chúng ta còn có thể bảo vệ tên học sinh của mình. Sửa đổi mã của bạn như sau:

```
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()
```

Lưu ý, giống như mã trước đó, chúng tôi cung cấp getter và setter cho tên.

- Bạn có thể tìm hiểu thêm trong tài liệu về các phương thức (https://docs.python.org/3/tutorial/classes.html) của Python .

## Kết nối với công việc trước đó trong khóa học này

- Mặc dù không được nêu rõ ràng trong các phần trước của khóa học này, nhưng bạn đã sử dụng các lớp và đối tượng trong suốt quá trình này.
- Nếu bạn tìm hiểu kỹ tài liệu về `int`, bạn sẽ thấy rằng đó là một lớp có hàm tạo. Đó là một kế hoạch chi tiết để tạo các đối tượng thuộc loại `int`. Bạn có thể tìm hiểu thêm trong tài liệu của Python về `int` (https://docs.python.org/3/library/functions.html#int).
- Chuỗi cũng là một lớp. Nếu bạn đã sử dụng `str.lower()`, bạn đang sử dụng một phương thức có trong lớp `str`. Bạn có thể tìm hiểu thêm trong tài liệu của Python về `str` (https://docs.python.org/3/library/stdtypes.html#str).
- `list` cũng là một lớp Nhìn vào tài liệu đó cho `list`, bạn có thể thấy các phương thức có trong đó, như `list.append()`. Bạn có thể tìm hiểu thêm trong tài liệu của Python về `list` (https://docs.python.org/3/library/stdtypes.html#list).
- `dict` cũng là một lớp trong Python. Bạn có thể tìm hiểu thêm trong tài liệu của Python về `dict` (https://docs.python.org/3/library/stdtypes.html#dict).
- Để xem bạn đã sử dụng các lớp như thế nào, hãy đi tới bảng điều khiển và nhập `code type.py` rồi viết mã như sau:

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

Lưu ý rằng bằng cách thực thi mã này, nó sẽ hiển thị rằng lớp `50` là `int`.

- Chúng ta cũng có thể áp dụng điều này `str` như sau:

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

Lưu ý cách thực thi mã này sẽ cho biết đây là lớp `str`.

- Chúng ta cũng có thể áp dụng điều này `list` như sau:

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

Lưu ý cách thực thi mã này sẽ cho biết đây là lớp `list`.

- Chúng ta cũng có thể áp dụng điều này cho `list` việc sử dụng tên của `list` lớp dựng sẵn của Python như sau:

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

Lưu ý cách thực thi mã này sẽ cho biết đây là lớp `list`.

- Chúng ta cũng có thể áp dụng điều này `dict` như sau:

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

Lưu ý cách thực thi mã này sẽ cho biết đây là lớp `dict`.

- Chúng ta cũng có thể áp dụng điều này cho `dict` việc sử dụng tên của `dict` lớp dựng sẵn của Python như sau:

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

Lưu ý cách thực thi mã này sẽ cho biết đây là lớp `dict`.

## Phương thức lớp

- Đôi khi, chúng tôi muốn thêm chức năng cho chính một lớp chứ không phải cho các phiên bản của lớp đó.
- `@classmethod` là một hàm mà chúng ta có thể sử dụng để thêm chức năng cho toàn bộ một lớp.
- Đây là một ví dụ về việc *không* sử dụng phương thức lớp. Trong cửa sổ terminal của bạn, gõ `code hat.py` và mã như sau:

```
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")
```

Lưu ý rằng khi chúng ta chuyển tên của học sinh vào chiếc mũ phân loại, nó sẽ cho chúng ta biết học sinh đó được phân vào nhà nào. Lưu ý rằng `hat = Hat()` khởi tạo một `hat`. Chức năng này `sort` luôn được xử lý bởi *thể hiện* của lớp `Hat`. Bằng cách thực thi `hat.sort("Harry")`, chúng tôi chuyển tên của học sinh vào `sort` phương thức của phiên bản cụ thể của `Hat`, mà chúng tôi đã gọi `hat`.

- Tuy nhiên, chúng ta có thể muốn chạy `sort` hàm mà không tạo một phiên bản cụ thể của chiếc mũ sắp xếp (xét cho cùng thì chỉ có một chiếc thôi!). Chúng ta có thể sửa đổi mã của mình như sau:

```
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")
```

Lưu ý cách `__init__` loại bỏ phương thức này vì chúng ta không cần khởi tạo một chiếc mũ ở bất kỳ đâu trong mã của mình. `self` do đó, không còn phù hợp và bị xóa. Chúng tôi chỉ định đây `sort` là `@classmethod`, thay thế `self` bằng `cls`. Cuối cùng, hãy chú ý cách `Hat` viết hoa theo quy ước ở gần cuối mã này, vì đây là tên lớp của chúng ta.

- Quay lại phần `students.py` chúng ta có thể sửa đổi mã của mình như sau, giải quyết một số cơ hội bị bỏ lỡ liên quan đến `@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()
```

Lưu ý rằng nó `get_student` bị xóa và một `@classmethod` cuộc gọi `get` được tạo. Phương pháp này bây giờ có thể được gọi mà không cần phải tạo một sinh viên trước.

## Phương thức tĩnh

- Hóa ra là ngoài `@classmethod` s, khác biệt với các phương thức cá thể, còn có các loại phương thức khác.
- Việc sử dụng `@staticmethod` có thể là điều bạn có thể muốn khám phá. Mặc dù không được đề cập rõ ràng trong khóa học này, nhưng bạn có thể truy cập và tìm hiểu thêm về các phương thức tĩnh cũng như sự khác biệt của chúng với các phương thức lớp.

## ~~Di sản~~ Inheritance/Kế thừa

- Kế thừa có lẽ là tính năng mạnh mẽ nhất của lập trình hướng đối tượng.
- Tinh cờ là bạn có thể tạo một lớp "kế thừa" các phương thức, biến và thuộc tính từ một lớp khác.
- Trong thiết bị đầu cuối, thực thi `code wizard.py`. Mã như sau:

```
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")
...
```

Lưu ý rằng có một lớp ở trên được gọi `Wizard` và một lớp được gọi là `Student`. Hơn nữa, hãy lưu ý rằng có một lớp được gọi là `Professor`. Cả sinh viên và giáo sư đều có tên. Ngoài ra, cả sinh viên và giáo sư đều là phù thủy. Vì vậy, cả hai `Student` đều `Professor` kế thừa những đặc điểm của `Wizard`. Trong lớp "con" `Student`, `Student` có thể kế thừa từ lớp "cha" hoặc "siêu" `Wizard` khi dòng `super().__init__(name)` chạy `init` phương thức của `Wizard`. Cuối cùng, hãy lưu ý rằng những dòng cuối cùng của mã này tạo ra một thuật sĩ tên là Albus, một học sinh tên là Harry, v.v.

## Kế thừa và ngoại lệ

- Mặc dù chúng tôi vừa giới thiệu tính kế thừa nhưng chúng tôi đã sử dụng tính năng kế thừa này trong suốt quá trình sử dụng các ngoại lệ.
- Điều ngẫu nhiên xảy ra là các trường hợp ngoại lệ xảy ra theo chế độ thừa kế, nơi có các tầng lớp con cái, cha mẹ và ông bà. Những điều này được minh họa dưới đây:

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

- Bạn có thể tìm hiểu thêm trong tài liệu về ngoại lệ (https://docs.python.org/3/library/exceptions.html) của Python .

/Nạp chồng

# Quá tải toán tử

"/nạp chồng"

- Một số toán tử như `+` và `-` có thể bị "quá tải" đến mức chúng có thể có nhiều khả năng hơn ngoài số học đơn giản.
- Trong cửa sổ terminal của bạn, gõ `code vault.py`. Sau đó, mã như sau:

```
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)
```

  Lưu ý cách `__str__` phương thức trả về một chuỗi được định dạng. Hơn nữa, hãy chú ý cách `__add__` phương pháp này cho phép cộng các giá trị của hai vault. `self` là những gì ở bên trái của `+` toán hạng. `other` đó là điều đúng đắn của `+`.
- Bạn có thể tìm hiểu thêm trong tài liệu về nạp chồng toán tử (https://docs.python.org/3/reference/datamodel.html#special-method-names) của Python .

# Tổng hợp

Bây giờ, bạn đã học được một cấp độ năng lực hoàn toàn mới thông qua lập trình hướng đối tượng.

- Lập trình hướng đối tượng
- Các lớp học
- `raise`

- Phương thức lớp
- Phương thức tĩnh
- Di sản
- Quá tải toán tử