# Lập trình hướng đối tượng

- Lập trình hướng đối tượng (Object Oriented Programming - OOP) là một phương pháp tổ chức chương trình bằng cách đóng gói các thuộc tính (attribute) và hành vi (behavior) liên quan vào các đối tượng (object) riêng lẻ.
  - Ví dụ: một đối tượng người (Person) với các thuộc tính (attribute) như tên (name), tuổi (age), địa chỉ (address) và các hành vi (behavior) như đi bộ (walking), nói chuyện (talking), thở (breathing) và chạy (running).
- Một mô hình lập trình phổ biến khác là lập trình thủ tục (procedural programming), tổ chức chương trình giống như một công thức, trong đó nó cung cấp một tập hợp các bước, dưới dạng các hàm và khối lệnh theo thứ tự để hoàn thành một nhiệm vụ.

# Định nghĩa một lớp

- Các kiểu dữ liệu cơ sở như number, string, và list được thiết kế để mô tả các thông tin đơn giản.
- Giả sử ta cần lưu trữ một số thông tin phức tạp hơn như một nhân viên, gồm tên, tuổi, chức vụ và năm bắt đầu làm việc, ta có thể dùng danh sách:

In [1]:
```
nam = ["Nam Nguyen", 30, "Developer", 2018]
hai = ["Hai Tran", 35, "Senior Developer", 2012]
minh = ["Minh Nguyen", "Technical Lead", 2009]
```

- Tuy nhiên, cách này có thể khiến mã chương trình khó quản lý hơn.

# Lớp (class) và thể hiện (instance)

- Các lớp (class) được sử dụng để tạo cấu trúc (kiểu) dữ liệu do người dùng định nghĩa.
- Các lớp cũng định nghĩa các hàm được gọi là các phương thức (method), xác định các hành vi và hành động mà một đối tượng được tạo ra từ lớp có thể thực hiện với dữ liệu của nó.
- Một lớp là một bản thiết kế để xác định một thứ gì đó như thế nào. Nó không thực sự chứa bất kỳ dữ liệu nào.
- Một thể hiện (instance) là một đối tượng (object) được xây dựng từ một lớp và chứa dữ liệu thực.

# Định nghĩa một lớp

- Tất cả các `định nghĩa lớp` đều bắt đầu bằng từ khóa `class`, theo sau là `tên lớp` và `dấu hai chấm`.
- Bất kỳ mã nào được `thụt vào bên dưới định nghĩa lớp` được coi là một phần của `phần thân của lớp`.
- Một lớp đơn giản nhất trong Python (chỉ 3 từ và 2 dòng)

In [ ]:
```
class Dog:
    pass
```

- Từ khóa `pass` thường được sử dụng như một trình giữ chỗ cho biết mã cuối cùng sẽ đi đến đâu. Nó cho phép bạn chạy mã trên mà không gặp lỗi.

# Định nghĩa một lớp

- Ta sẽ thêm các thuộc tính `tên (name)` và `tuổi (age)` cho lớp `Dog`.
- Các thuộc tính này phải được xác định trong phương thức `__init__()`.
- Mỗi khi một đối tượng `Dog` mới được tạo, phương thức `__init__()` sẽ thiết lập trạng thái ban đầu của đối tượng bằng cách gán các giá trị thuộc tính của đối tượng.
  - `__init__()` là một `constructor`.
- Tham số đầu tiên của `__init__()` sẽ luôn là một biến được gọi `self`.
- Khi một `class instance` mới được tạo, nó được truyền một cách tự động cho tham số `self` trong `__init__()` để các thuộc tính mới có thể được định nghĩa trên đối tượng.

# Định nghĩa một lớp

- Định nghĩa lớp `Dog` với phương thức `__init__()` để tạo thuộc tính `name` và `age`:
- Phần thân của `` có 2 lệnh:
  - `self.name = name`: tạo một thuộc tính `name` và gán cho nó giá trị của tham số `name`.
  - `self.age = age`: tạo một thuộc tính `age` và gán cho nó giá trị của tham số `age`.

In [ ]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

# Instance attribute và class attribute

- Các thuộc tính được tạo trong phương thức `__init__()` được gọi là các `instance attribute` .
- Giá trị của `instance attribute` là cụ thể cho một `instance` cụ thể của lớp.
- Tất cả các `Dog object` đều có `age` và `name` , nhưng các giá trị cho thuộc tính `name` và `age` sẽ khác nhau tùy thuộc vào `Dog instance` .
- `Class attribute` là các thuộc tính có cùng giá trị cho tất cả các `instance` của lớp.
- `Class attribute` được định nghĩa bên ngoài phương thức `__init__()` .

In [ ]:
```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

# Khởi tạo một đối tượng

In [2]:
```
class Dog:
    pass
```

In [3]:
```
Dog()
```

Out[3]:
```
<__main__.Dog at 0x1956c60db80>
```

- Ta có một `Dog object` mới tại địa chỉ `0x1956c60db80` .

In [4]:
```
Dog()
```

Out[4]:
```
<__main__.Dog at 0x1956c60d6d0>
```

- Ta có một `Dog object` thứ hai tại địa chỉ `0x1956c60d6d0` .

# Khởi tạo một đối tượng

In [5]:
```
a = Dog()
b = Dog()
a == b
```

Out[5]:
```
False
```

- Mặc dù `a` và `b` đều là các thể hiện của lớp `Dog`, nhưng chúng đại diện cho hai đối tượng riêng biệt trong bộ nhớ.

# Class và instance attribute

- Tạo một lớp `Dog` mới với một `class attribute` là `species` và hai `instance attribute` là `name` và `age`:

In [6]:
```
class Dog:
    species = "Canis familiaris"    # Class attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- Để khởi tạo lớp `Dog` mới, ta cần cung cấp `name` và `age`, nếu không Python sẽ phát sinh một `TypeError`.

In [7]:
```
Dog()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-2dced99f65a6> in <module>
----> 1 Dog()

TypeError: __init__() missing 2 required positional arguments: 'name' and
'age'
```

# Class và instance attribute

- Để truyền các đối số cho tham số `name` và `age`, đặt các giá trị vào dấu ngoặc đơn sau tên lớp:

In [9]:
```
bingo = Dog("Bingo", 9)
```

- Phương phức `__init__()` có 3 tham số `self`, `name` và `age` nhưng ta chỉ cần truyền vào hai đối số.
- Khi ta khởi tạo một `Dog object`, Python sẽ tạo một `instance` mới và truyền nó cho tham số `self` của `__init__()`, vì vậy ta chỉ cần truyền vào các đối số cho tham số `name` và `age`.
- Sau khi tạo các `Dog instance`, ta có thể truy cập các `instance attribute` của chúng bằng cách sử dụng `dấu chấm (dot)`:

```
In [10]: bingo.name
```

```
Out[10]: 'Bingo'
```

```
In [11]: bingo.age
```

```
Out[11]: 9
```

# Class và instance attribute

- Ta có thể truy cập các `class attribute` theo cùng một cách:

```
In [12]: bingo.species
```

```
Out[12]: 'Canis familiaris'
```

- Giá trị của chúng cũng có thể được thay đổi:

```
In [13]: bingo.age = 10
         bingo.age
```

```
Out[13]: 10
```

```
In [14]: kino.species = "Bulldog"
         kino.species
```

```
Out[14]: 'Bulldog'
```

# Instance method

- `Instance method` là các hàm (function) được định nghĩa bên trong một lớp và chỉ được gọi từ một instance của lớp đó.
- Ta định nghĩa 2 `instance method` :
  - `description()` : trả về chuỗi thể hiện `name` và `age` của một `Dog object`
  - `speak()` : nhận thêm một tham số `sound` và trả về một chuỗi chứa `name` và `sound` của một `Dog object`

# Instance method

In [15]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

In [16]:
```
bingo = Dog("Bingo", 9)
bingo.description()
```

Out[16]:
```
'Bingo is 9 years old'
```

In [17]:
```
bingo.speak("Woof Woof")
```

Out[17]:
```
'Bingo says Woof Woof'
```

# Pythonic code

- Phương thức `description()` ở trên không phải là `cách của Python (Pythonic)` để trả về một chuỗi chứa thông tin hữu ích về một thể hiện của lớp.
- Khi ta tạo một đối tượng `list` , ta có thể sử dụng hàm `print()` để hiển thị một chuỗi giống như danh sách:

In [18]:
```
names = ["Bingo", "Kino", "Sam"]
print(names)
```

```
['Bingo', 'Kino', 'Sam']
```

- Nếu ta dùng `print()` với `Dog object` , ta thu được thông báo không hữu ích lắm.

In [19]:
```
print(bingo)
```

```
<__main__.Dog object at 0x000001956D7BEE20>
```

# Phương thức `__str__()`

- Ta có thể thay đổi những gì được in bằng cách xác định một phương thức đặc biệt là `__str__()` .

In [20]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

In [22]:
```
bingo = Dog("Bingo", 9)
print(bingo)
```

```
Bingo is 9 years old
```

# Dunder method

- Các phương thức `__init__()` và `__str__()` được gọi là các `dunder method` (**d**ouble **under**score).
- Có rất nhiều `dunder method` mà ta có thể sử dụng để tùy chỉnh các lớp trong Python. Dưới đây là một số operator dunder method.
  - `__add__` : addition(+)
  - `__sub__` : subtraction(-)
  - `__mul__` : multiplication(*)
  - `__truediv__` : division(/)
  - `__eq__` : equality (==)
  - `__lt__` : less than(<)
  - `__gt__` : greater than(>)
- Tham khảo thêm: https://www.tutorialsteacher.com/python/magic-methods-in-python (https://www.tutorialsteacher.com/python/magic-methods-in-python)

# Sự kế thừa (Inheritence)

- `Sự kế thừa (inheritence)` là quá trình một lớp tiếp nhận các `thuộc tính` và `phương thức` của lớp khác.
- Các lớp mới được hình thành được gọi là các `lớp con (child class)` , và các lớp mà các lớp con kế thừa từ chúng được gọi là các `lớp cha (parent class)` .
- Các `lớp con` có thể `ghi đè (override)` hoặc mở rộng (extend) các `thuộc tính` và `phương thức` của các `lớp cha` .
  - Các `lớp con` kế thừa `tất cả` các `thuộc tính` và `phương thức` của lớp cha nhưng cũng có thể chỉ định các `thuộc tính` và `phương thức` là `duy nhất cho chính chúng` .

# Parent Classes và Child Classes

In [23]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

# Parent Classes và Child Classes

In [24]:
```
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

In [25]:
```
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

# Parent Classes và Child Classes

In [26]:
```
miles.species
```

Out[26]:
```
'Canis familiaris'
```

In [27]:
```
buddy.name
```

Out[27]:
```
'Buddy'
```

In [28]:
```
print(jack)
```

```
Jack is 3 years old
```

In [29]:
```
jim.speak("Woof")
```

Out[29]:
```
'Jim says Woof'
```

# Phương thức `type()` và `isinstance()`

In [30]:
```
type(miles)
```

Out[30]:
```
__main__.JackRussellTerrier
```

In [31]:
```
isinstance(miles, Dog)
```

Out[31]:
```
True
```

In [32]:
```
isinstance(miles, Bulldog)
```

Out[32]:
```
False
```

# Mở rộng tính năng của lớp cha

- Vì các giống chó khác nhau có tiếng sủa hơi khác nhau, ta muốn cung cấp một giá trị mặc định cho đối số `sound` của các phương thức `speak()` tương ứng.
- Để làm điều này, ta cần `ghi đè (overide)` phương thức `speak()` trong định nghĩa của từng lớp con.

In [34]:
```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

In [35]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[35]:
```
'Miles says Arf'
```

In [36]:
```
miles.speak("Grrr")
```

Out[36]:
```
'Miles says Grrr'
```

# Phương thức `super()`

- Ta có thể truy xuất `parent class` từ trong một `method` của `child class` dùng `super()`.

In [37]:
```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)
```

In [38]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[38]:
```
'Miles says Arf'
```

- Khi ta gọi `super().speak(sound)` bên trong `JackRussellTerrier`, Python sẽ tìm phương thức `speak()` từ lớp cha (`Dog`) và gọi nó bằng biến `sound`.

# Tóm tắt

Ta đã tìm hiểu một số vấn đề sau:

- Định nghĩa một `class` (một bản thiết kế của một `object`)
- Khởi tạo một `object` từ một `class`
- Sử dụng các `attribute` và `method` để định nghĩa các `property` và `behavior` của một `object`
- Sử dụng `inheritance` để tạo các `child class` từ một `parent class`
- Tham chiếu đến một `method` của một `parent class` dùng phương thức `super()`
- Kiểm tra một `object` có kế thừa từ một `class` dùng phương thức `isinstance()`

# Public và Private

- Python không có khái niệm dữ liệu `private` hoặc `protected` giống như C++, Java, C#. Mọi thứ trong Python đều là `public`.
- Thay cho `private`, Python có một **ký hiệu** cho biến `non-public`.
  - Bất kỳ biến nào `bắt đầu bằng dấu gạch dưới (_)` đều được xác định là `non-public`.
  - **Đây chỉ là quy ước đặt tên và ta vẫn có thể truy cập trực tiếp vào biến.**

In [3]:
```
class Car:
    wheels = 0
    def __init__(self, color, model, year):
        self.color = color
        self.model = model
        self.year = year
        self._cupholders = 6
```

In [5]:
```
my_car = Car("yellow", "Beetle", "1969")
print(f"It has {my_car._cupholders} cupholders.")
```

```
It has 6 cupholders.
```

# Public và Private

- Ta có thể sử dụng `dấu gạch dưới kép` ( `__` ) phía trước một biến để che giấu một thuộc tính trong Python.
  - Khi Python nhìn thấy một biến bắt đầu bằng `__` , nó sẽ thay đổi nội bộ tên biến để gây khó khăn cho việc truy cập trực tiếp.

In [6]:
```
class Car:
    wheels = 0
    def __init__(self, color, model, year):
        self.color = color
        self.model = model
        self.year = year
        self.__cupholders = 6
```

In [8]:
```
my_car = Car("yellow", "Beetle", "1969")
print(f"It has {my_car.__cupholders} cupholders.")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-69dd0b0b1cde> in <module>
      1 my_car = Car("yellow", "Beetle", "1969")
----> 2 print(f"It has {my_car.__cupholders} cupholders.")

AttributeError: 'Car' object has no attribute '__cupholders'
```

# Public và Private

- Khi ta cố tình truy cập một biến bắt đầu bằng `__` (như `__cupholders` ở trên), Python phát sinh lỗi `AttributeError` (biến không tồn tại)
- Thực sự, khi Python nhìn thấy một thuộc tính bắt đầu bằng `__` , nó sẽ thay đổi thuộc tính bằng cách thêm `dấu gạch dưới` ( `_` ) vào trước tên ban đầu của thuộc tính, theo sau là tên lớp.
  - Nó thay đổi tên biến để gây khó khăn cho việc truy cập trực tiếp.
  - Tuy vậy, ta vẫn có thể truy cập trực tiếp bằng cách thay đổi tên theo quy tắc này

In [9]:
```
print(f"It has {my_car._Car__cupholders} cupholders")
```

```
It has 6 cupholders
```

# Access Control

- Trong các ngôn ngữ khác như C++, Java, C#, ta có thể truy cập các thuộc tính `private` bằng các phương thức `getter` và `setter`.
  - Các phương thức `getter` và `setter` là `public`
- Với Python, ta cũng có thể làm như vậy nhưng sẽ làm cho mã dài dòng (trái với triết lý của Python)
- Python cung cấp một cách khác đơn giản hơn để thực hiện việc này bằng cách dùng `decorator` syntax với `property`

# Property và decorator

- `Property` cho phép các hàm được khai báo trong các lớp Python tương tự như các phương thức `getter` và `setter` của C++/Java/C#, và cho phép ta `xóa` các thuộc tính.
- Để cung cấp quyền truy cập có kiểm soát, ta định nghĩa hàm `voltage()` để trả về giá trị cho biến `private` này.
  - Bằng cách sử dụng `@property` decoration, ta đánh dấu nó là một `getter` mà bất kỳ ai cũng có thể truy cập trực tiếp.
- Tương tự, ta `decorate` hàm `voltage()` với `@voltage.setter` để định nghĩa một hàm `setter`.
- Cuối cùng, ta `decorate` hàm `voltage()` với `@voltage.deleter` để cho phép xóa thuộc tính (`deleter`).

In [10]:
```
class Car:
    def __init__(self, color, model, year):
        self.color = color
        self.model = model
        self.year = year
        self._voltage = 12

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, volts):
        print("Warning: this can cause problems!")
        self._voltage = volts

    @voltage.deleter
    def voltage(self):
        print("Warning: the radio will stop working!")
        del self._voltage
```

# Property và decorator

In [11]:
```
my_car = Car("yellow", "beetle", 1969)
print(f"My car uses {my_car.voltage} volts")
```

```
My car uses 12 volts
```

In [12]:
```
my_car.voltage = 6
```

```
Warning: this can cause problems!
```

In [13]:
```
print(f"My car now uses {my_car.voltage} volts")
```

```
My car now uses 6 volts
```

In [14]:
```
del my_car.voltage
```

```
Warning: the radio will stop working!
```

# Property và decorator

- `@property` , `@.setter` , và `@.deleter` decoration cho phép ta có thể điều kiển truy xuất các thuộc tính mà không cần sử dụng các phương thức khác nhau.
  - Khi ta viết `my_car.voltage` , Python gọi hàm `voltage()` với `@property` decoration.
  - Khi ta gán `my_car.voltage = 6` , Python gọi hàm `voltage()` với `@voltage.setter` decoration.
  - Khi ta gọi `del my_car.voltage` , Python gọi hàm `voltage()` với `@voltage.deleter` decoration.

# Tài liệu tham khảo


- **Tài liệu tham khảo chính**

1. David Amos, ***Object-Oriented Programming (OOP) in Python 3***, July 06, 2020. https://realpython.com/python3-object-oriented-programming/ (https://realpython.com/python3-object-oriented-programming/) (Last accessed on April 13, 2021)

- **Tài liệu đọc thêm**

1. Isaac Rodriguez, ***Inheritance and Composition: A Python OOP Guide***. https://realpython.com/inheritance-composition-python/ (https://realpython.com/inheritance-composition-python/) (Last accessed on April 13, 2021)
2. Jon Fincher, ***Object-Oriented Programming in Python vs Java***. https://realpython.com/oop-in-python-vs-java/ (https://realpython.com/oop-in-python-vs-java/) (Last accessed on April 13, 2021)


In [ ]:
```
```