Learn to <u>think Differently</u>

Rishabh IO
linkedin/rishabhio

What comes to your mind when you think of the following terms ?

Planet

Animal

Food

Vehicle

Building

And some more terms

Furniture

Machine

Music

Book

Sport

Plant

Toy

Color

Profession

Human

Emotion

Software

How can you represent such ideas with Python ?

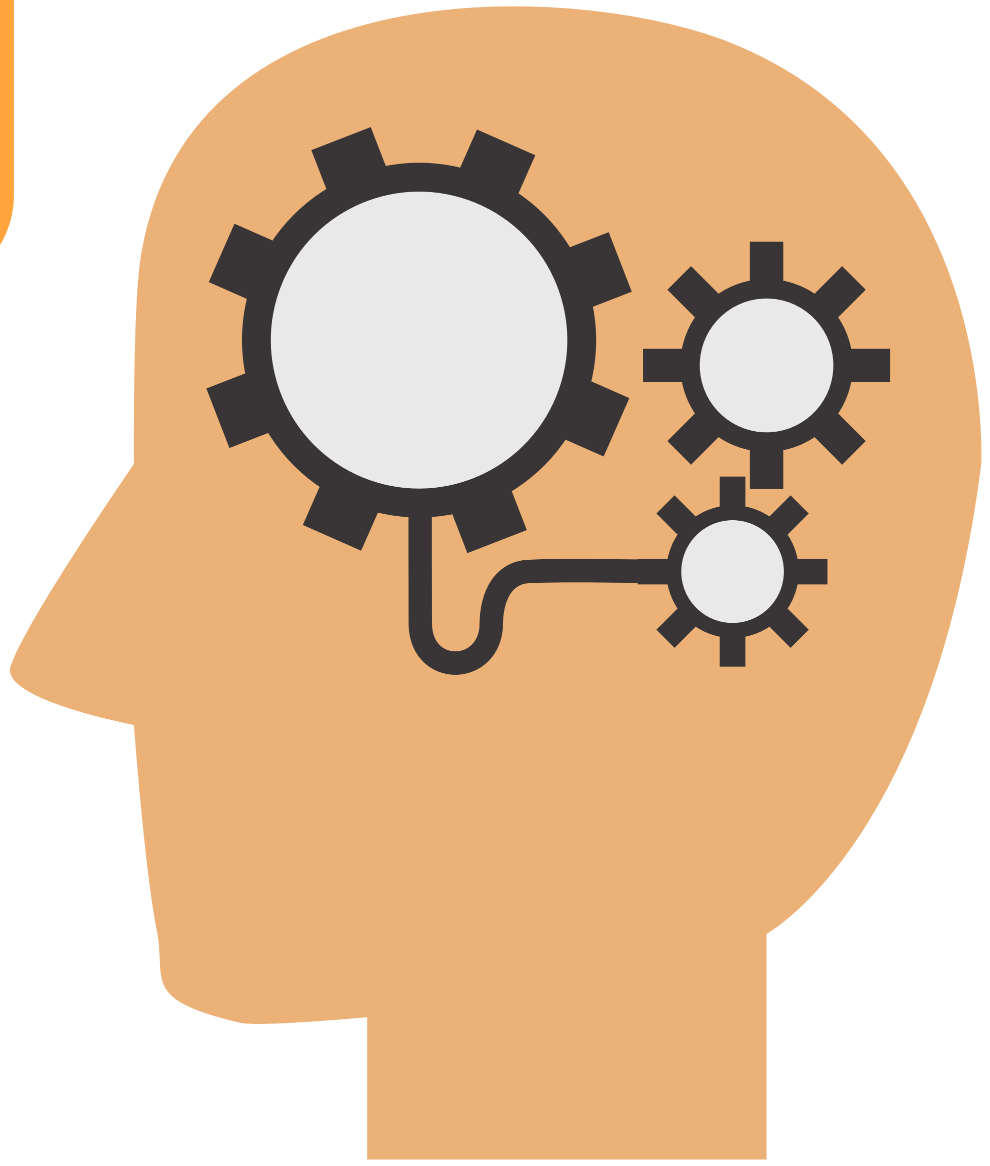
List ?

function ?

Dictionary ?

Tuple ?

set ?



Let's try to represent a Planet

Earth
12,742 km (approx)
5.97 x 10^24 kg (approx)

Earth is one of the concrete entities / objects exhibiting attributes and behaviour defined by the Planet blueprint or class

Blue Print

Planet ?

#Attributes

- Name
- Diameter
- Mass
- Gravity
- Revolution Period
- Rotation Period
- Surface Temperature
- Atmosphere
- Composition
- Magnetic Field
- Moons
- Ring System
- Distance from Sun
- Discovery Date

- Orbit around sun
- Rotate around axis
- Exert gravity on moon
- Exert gravity on objects
- Migrate season

#Behaviors

How about representing a Car

Mercedes
XYZ-class
Blue etc. . .

Car is one of the concrete entities / objects exhibiting attributes and behaviour defined by the Car blueprint or class

Blue Print

Car ?

#Attributes

- Name
- Manufacturer
- Model
- Year
- Color
- Fuel Type
- Engine Size
- Number of Doors
- Seating Capacity
- Horsepower
- Wheel Size
- Interior Features
- Exterior Features

#Behaviors

- accelerate
- brake
- start_engine
- change_direction
- indicate_fuel
- lock

Lastly! Let's represent a Pen

Parker
XYZ-model
Black etc. . .

Parker is one of the concrete entities / objects exhibiting attributes and behaviour defined by the Pen blueprint or class

Blue Print

Pen ?

#Attributes

Brand
Model
Color
Material
Ink Color
Tip Type
Length
Diameter
Weight
Cap
Grip
Refillable

write
refill
grip

#Behaviors

Introduction to Class

In Python, a class is a blueprint or a template that allows you to define the structure and behavior of objects. (new data types)

Syntax of Class in Python

```
class <ClassName>:
    # attributes ( data )
    # behaviours ( methods )
```

Example of Class in Python

```
class Car:              →  New Datatype
    name = None
    color = None

car1 = Car()            →  New Object
car1.name = "Maruti"
car1. color = "White"
```

Constructor of Class in Python

```
class Car:
    def __init__(self, name, color):
        self.name = name
        self.color = color
        self.state = "OFF"
    def start(self):
        self.state = "ON"

maruti = Car("m-800", "blue")
maruti.start()
print(maruti.state)
```

Example: Person Class

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I'm {self.age}
years old.")


person1 = Person("John", 30)
print(person1.age)
person1.greet()
```

Example: Book Class

```
class Book:
    def __init__(self, title, author, year, genre):
        self.title = title
        self.author = author
        self.year = year
        self.genre = genre

    def get_summary(self):
        return f"{self.title} by {self.author}, published in
{self.year}. Genre: {self.genre}"

book1 = Book("To Kill a Mockingbird", "Harper Lee", 1960,
"Fiction")
summary = book1.get_summary()
```

Try creating a class for

Animal

Building

Vehicle

Food

Planet

Inheriting Blueprints

```
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def calculate_salary(self):
        print("Calculating salary...")
```

```
class Manager(Employee):
    def __init__(self, name, employee_id):
        super().__init__(name, employee_id)

    def approve_leave(self):
        print("Approving leave...")
```

Inheriting Blueprints

```python
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def calculate_salary(self):
        print("Calculating salary...")
```

```python
class Engineer(Employee):
    def __init__(self, name, employee_id):
        super().__init__(name, employee_id)

    def solve_problem(self):
        print("Solving problem...")
```

Inheriting Blueprints

```python
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def calculate_salary(self):
        print("Calculating salary...")
```

```python
class Engineer(Employee):
    def __init__(self, name, employee_id):
        super().__init__(name, employee_id)

    def solve_problem(self):
        print("Solving problem...")
```

Example: Furniture

```
class Furniture:
    def __init__(self, material):
        self.material = material

    def sit(self):
        print("Sitting...")
```

```
class Chair(Furniture):
    def __init__(self, material,
has_wheels):
        super().__init__(material)
        self.has_wheels =
has_wheels

    def adjust_height(self):
        print("Adjusting height...")
```

```
class Recliner(Chair):
    def __init__(self, material,
has_wheels, has_headrest):
        super().__init__(material,
has_wheels)
        self.has_headrest =
has_headrest

    def recline(self):
        print("Reclining...")
```

Key "Words"

__init__ : Special method used to initialize an object's attributes when creating an instance

self : Parameter referring to the instance of a class, used to access its attributes and methods

super(): Function to call a method from a parent class, enabling inheritance and extending functionality

Privacy by "Convention"

_private_name : Instance methods / attributes which have their name starting with single '_' are by convention considered private.

However, this is just a convention and Python doesn't strictly enforce any limitations on the access.

_ _dunders_ _

__dunder_method__ : These are often referred to as "magic" or "dunder" (double underscore) methods.

Reserved for special methods or attributes that have specific predefined functionality in Python.

Examples include __init__, __str__, __len__, etc

Lambda in Python

can only have one expression

```
lambda arguments: expression

add = lambda x, y: x + y
result = add(5, 3)
```

Any number of arguments

Lambda offers a convenient way to define functions which have only 1 expression

Map in Python

```
map(function, iterable)

numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared))
# Output: [1, 4, 9, 16, 25]
```

the map() function is a built-in function that allows you to apply a specific function to each element of an iterable (such as a list, tuple, or string) and returns a new iterator with the results.

Filter in Python

```
filter(function, iterable)

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0,
numbers)
print(list(even_numbers))
# Output: [2, 4, 6]
```

the filter() function is a built-in function that allows you to create a new iterator or list by filtering out elements from an iterable (such as a list, tuple, or string) based on a specific condition