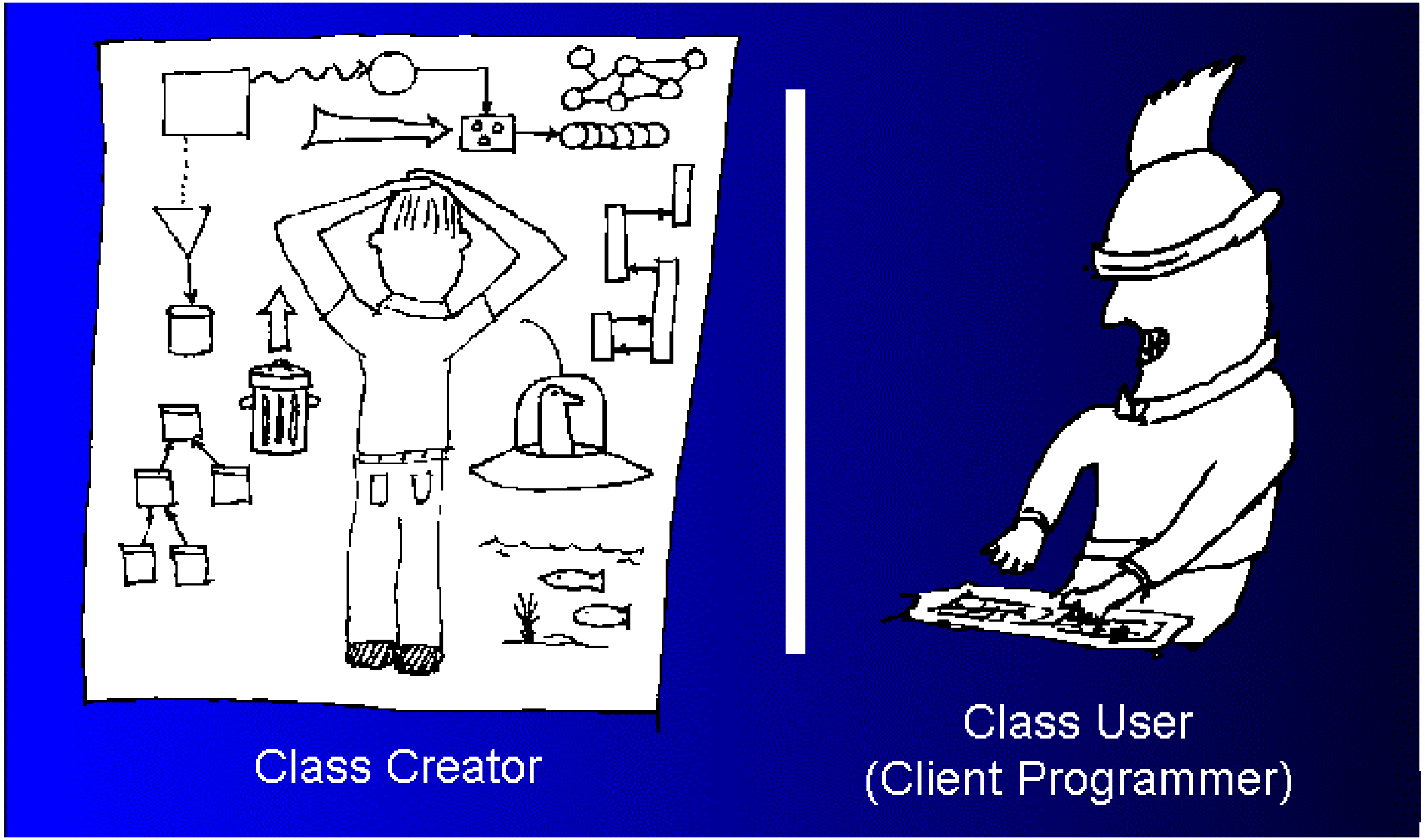
Module 7

class & OBJECTS(1)

Instructor: Jonny C.H Yu

In Python, everything is an object and it is time to create our OWN objects.



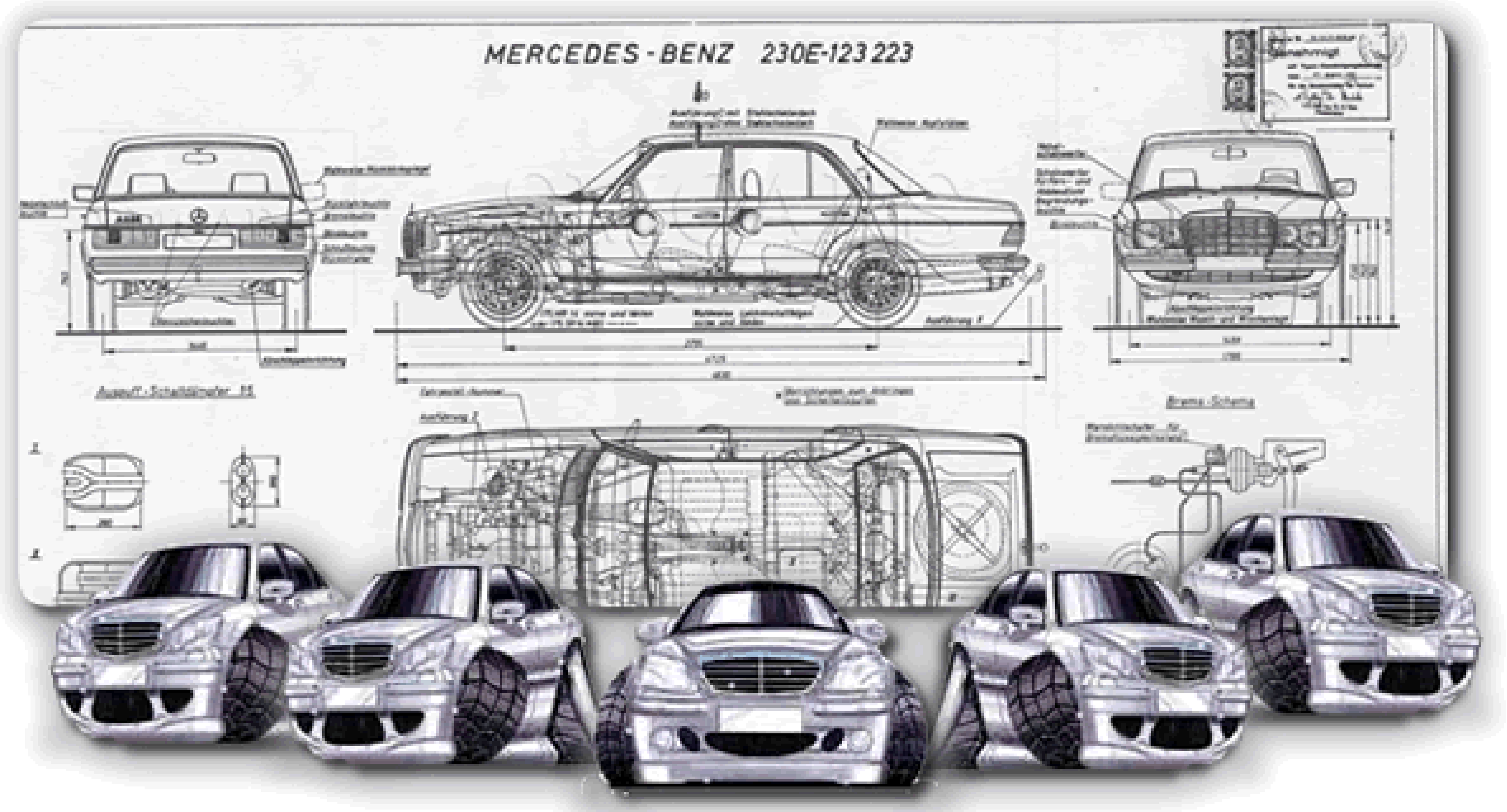
In Python, everything is an object and it is time to create our OWN objects.

```
class list(object):
    """
    list() -> new empty list
    list(iterable) -> new list initializ
    """
    def append(self, p_object): # real s
        """ L.append(object) -> None -- a
        pass

    def clear(self): # real signature un
        """ L.clear() -> None -- remove a
        pass

    def copy(self): # real signature unk
        """ L.copy() -> list -- a shallow
        return []

    def count(self, value): # real signa
```

```
lst = [1, 2, 3, 4]
lst.append(5)
print(f'The list contains: {lst}')
```

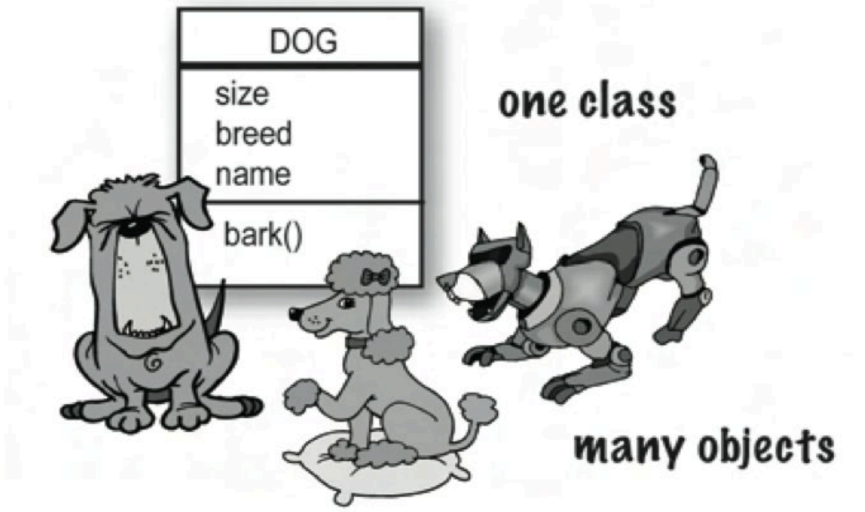
```
The list contains: [1, 2, 3, 4, 5]
```

We use the class mechanism to specify our own objects



Classes vs. Instances

- **Class: code that specifies the data attributes and methods of a particular type of object**
  - Similar to a blueprint of a house
- **Instance: an object created from a class**
  - Similar to a specific house built according to the blueprint
  - There can be many instances of one class



How to Define a Class

- All class definitions start with the class keyword followed by the name of class and a colon
  - `class Class_name:`
    - Class names often start with uppercase letter
    - Any code indented below the class definition is part of class body

```
class Dog:
    pass
```

- The body of the Dog class consists of a single statement: the pass keyword.
- pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Instantiate Objects

- **Creating a new object from a class is called instantiating an object.**

[1]You can instantiate a new Dog object by typing the name of the class followed by parentheses.

[2] You now have a new Dog object at 0x7fee58403d50. This funny-looking string of letters and numbers is a memory address that indicates where the Dog object is stored in your computer's memory.

[3] If you instantiate a second object, it will be stored at different memory address.

[4] When you compare a and b using the == operator, the result is False because they represent two distinct objects in memory.

```
[1]: class Dog:
         pass

[2]: Dog()

[2]: <__main__.Dog at 0x7fee58403d50>

[3]: Dog()

[3]: <__main__.Dog at 0x7fee702fc410>

[4]: a = Dog()
     b = Dog()
     a == b

[4]: False
```

In Python, everything is an object and it is time to create our OWN objects.

```python
class Dog:
    pass
```

```python
Dog()
```

```
<__main__.Dog at 0x7fee58403d50>
```

```python
Dog()
```

```
<__main__.Dog at 0x7fee702fc410>
```

```python
a = Dog()
b = Dog()
a == b
```

```
False
```

__init__() method

Define a Class with __init__ method

- The attributes of all objects must have are defined in a method called __init__()
- When a new object is created, __init__() sets the initial state of the object by assigning the values of the object's attributes.

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

In the body of __init__(), there are two statements using the self variable:

- **self.name = name** creates an attribute called name and assigns to it the value of the name parameter.
- **self.age = age** creates an attribute called age and assigns to it the value of the age parameter.
- You can give __init__() any number of parameters, but the first parameter will always be a variable called self.
- When a new class instance is created, the instance is automatically passed to the self parameter in __init__() so that new attributes can be defined on the object.

Instance Attributes and Class Attributes

- Attributes created in __init__() are called instance attributes or object attributes.
  - An instance attribute is specified to a particular object of a class.
  - For example, all Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog object.
- Class attributes are attributes that have the same value for all objects.
  - You can define a class attribute by assigning a value to a variable name outside of __init__().

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

- Use class attributes to define attributes that should have the same value for every object.
- Use instance attributes for attributes that vary from one object to another.

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

```
[2]: Dog()
```

```
---------------------------------------------------
TypeError                                       Trac
<ipython-input-2-2dced99f65a6> in <module>
----> 1 Dog()

TypeError: __init__() missing 2 required posit
```

```
[3]: buddy = Dog("Buddy", 9)
```

```
[4]: miles = Dog("Miles", 4)
```

```
[5]: buddy.name
[5]: 'Buddy'
```

```
[6]: miles.age
[6]: 4
```

```
[7]: Dog.species
[7]: 'Canis familiaris'
```

```
[8]: buddy.species
[8]: 'Canis familiaris'
```

Instantiate Objects

- [1] We now define a Dog class with __init__ method.
- [1] Instance attributes are initialized in the __init__ method while class attribute is assigned before the method.
- [2] Every Dog object MUST instantiate from __init__. To instantiate objects of this Dog class, you need to provide values for the name and age. If you don't, then Python raises a TypeError.
- [3], [4] You can create two new Dog objects.
- [5] – [6] You can access the object attributes via object as usual.
- [7] – [8] You can access the class attributes via class or object.

In Python, everything is an object and it is time to create our OWN objects.

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

L1_Dog.ipynb

```
[3]: buddy = Dog("Buddy", 9)

[4]: miles = Dog("Miles", 4)

[5]: buddy.name

[5]: 'Buddy'

[6]: miles.age

[6]: 4

[7]: Dog.species

[7]: 'Canis familiaris'

[8]: buddy.species

[8]: 'Canis familiaris'
```

Example: Point class

- In two dimensions, a point has two coordinates x and y.
- Define a Point class so we can instantiate a Point object with given values of x and y. If x and y are not provided, treat the point as the origin.
- Use the following client codes to test your class implementation.

```
p = Point(4, 2)
```

```
r = Point()
```

```
print(f'Point p: x = {p.x} and y = {p.y}')
```

```
Point p: x = 4 and y = 2
```

```
print(f'Point r: x = {r.x} and y = {r.y}')
```

```
Point r: x = 0 and y = 0
```

Example: Point class (Ans)

- In two dimensions, a point has two coordinates x and y.
- Define a Point class so we can instantiate a Point object with given values of x and y. If x and y are not provided, treat the point as the origin.
- Use the following client codes to test your class implementation.

[1]:
```
class Point:
    """ Point class represents and manipulates
    x,y coords. """

    def __init__(self, x=0, y=0):
        """ Create a new point at x, y """
        self.x = x
        self.y = y
```

[2]:
```
p = Point(4, 2)
```

[3]:
```
r = Point()
```

[4]:
```
print(f'Point p: x = {p.x} and y = {p.y}')
```

```
Point p: x = 4 and y = 2
```

[5]:
```
print(f'Point r: x = {r.x} and y = {r.y}')
```

```
Point r: x = 0 and y = 0
```

L1_Point_Init.ipynb

Instance methods

Instantiate Objects

- Instance methods are functions that are defined inside a class and can only be called from an object of that class.
- Just like __init__(), an instance method's first parameter is always self.

```
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}"
```

- description() returns a string displaying the name and age of the dog.
- speak() has one parameter called sound and returns a string containing the dog's name and the sound the dog makes.

Instantiate Objects

```
[1]: 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}"
```

```
[2]: miles = Dog("Miles", 4)
```

```
[3]: miles.description()
```

```
[3]: 'Miles is 4 years old'
```

```
[4]: miles.speak('Woof Woof')
```

```
[4]: 'Miles says Woof Woof'
```

```
[5]: miles.speak('Bow Wow')
```

```
[5]: 'Miles says Bow Wow'
```

L1_Dog.ipynb

Example: Point class

- Define a Point class so we can instantiate a Point object with given values of x and y. If x and y are not provided, treat the point as the origin.
- Add a distance method to return the distance of the point from the origin. Use the following client codes to test your class implementation.

```
[2]: p = Point(4, 2)
```

```
[3]: r = Point()
```

```
[4]: print(f'Point p: x = {p.x} and y = {p.y}')
```

```
Point p: x = 4 and y = 2
```

```
[5]: print(f'Distane of point p from the origin is {p.distance():.2f}')
```

```
Distane of point p from the origin is 4.47
```

```
[6]: print(f'Point r: x = {r.x} and y = {r.y}')
```

```
Point r: x = 0 and y = 0
```

```
[7]: print(f'Distane of point r from the origin is {r.distance():.2f}')
```

```
Distane of point r from the origin is 0.00
```

Example: Point class (Ans)

```
[1]: class Point:
         """ Point class represents and manipulates
         x,y coords. """

         def __init__(self, x=0, y=0):
             """ Create a new point at x, y """
             self.x = x
             self.y = y

         def distance(self):
             return (self.x**2 + self.y**2)**0.5
```

```
[2]: p = Point(4, 2)
```

```
[3]: r = Point()
```

```
[4]: print(f'Point p: x = {p.x} and y = {p.y}')
```

```
Point p: x = 4 and y = 2
```

```
[5]: print(f'Distane of point p from the origin is {p.distance():.2f}')
```

```
Distane of point p from the origin is 4.47
```

```
[6]: print(f'Point r: x = {r.x} and y = {r.y}')
```

```
Point r: x = 0 and y = 0
```

```
[7]: print(f'Distane of point r from the origin is {r.distance():.2f}')
```

```
Distane of point r from the origin is 0.00
```

L1_Point_Distance.ipynb

__str__() method

__str__(): motivation

- When you create a list object, you can use print() to display a string that looks like the list:

```
[1]: names = ['David', 'Homer', 'Jay', 'Mikasa']
```

```
[2]: print(names)
```

```
['David', 'Homer', 'Jay', 'Mikasa']
```

- Let us see what happens when you print the Dog object

```
[4]: miles = Dog("Miles", 4)
```

```
[5]: print(miles)
```

```
<__main__.Dog object at 0x7fb01012ab90>
```

- This is not very helpful and you can change what gets printed by defining a special instance method called __str__().

__str__() method

```
[1]: class Dog:
         species = "Canis familiaris"
         def __init__(self, name, age):
             self.name = name
             self.age = age
         # An instance method
         def speak(self, sound):
             return f"{self.name} says {sound}"
         # Replace description() with __str__()
         def __str__(self):
             return f"{self.name} is {self.age} years old"
```

```
[2]: miles = Dog("Miles", 4)
```

```
[3]: print(miles)
```

```
Miles is 4 years old
```

L1_Dog.ipynb

Example: Car class

- Create a Car class with two instance attributes: color, which stores the name of the car's color as a string, and mileage, which stores the number of miles on the car as an integer.
- Implement an instance method called drive(), which takes a number as an argument and adds that number to the mileage attribute.
- Implement __str__() method
- Use the following client codes to test your class implementation

```
[2]: c1 = Car("blue", 100)
     c2 = Car("red", 200)
     cars = [c1, c2]
```

```
[3]: for c in cars:
         print(c)
```

```
The blue car has 100 miles
The red car has 200 miles
```

```
[4]: c1.drive(500)
```

```
[5]: for c in cars:
         print(c)
```

```
The blue car has 600 miles
The red car has 200 miles
```

Example: Car class (Ans)

[1]:
```python
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def drive(self, miles):
        self.mileage = self.mileage + miles

    def __str__(self):
        return f'The {self.color} car has {self.mileage} miles'
```

[2]:
```python
c1 = Car("blue", 100)
c2 = Car("red", 200)
cars = [c1, c2]
```

[3]:
```python
for c in cars:
    print(c)
```

```
The blue car has 100 miles
The red car has 200 miles
```

[4]:
```python
c1.drive(500)
```

[5]:
```python
for c in cars:
    print(c)
```

```
The blue car has 600 miles
The red car has 200 miles
```

L1_Car.ipynb

Other Double UNDERscore (Dunder) or magic methods

Magic Methods

- Special methods which add "magic" to your class.

- Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.

- For example, when you add two numbers using the + operator, internally, the `__add__()` method will be called.

Magic Methods for Comparison

Method	Description
__eq__(*self*, *other*)	*self* == *other*
__ne__(*self*, *other*)	*self* != *other*
__lt__(*self*, *other*)	*self* < *other*
__gt__(*self*, *other*)	*self* > *other*
__le__(*self*, *other*)	*self* <= *other*
__ge__(*self*, *other*)	*self* >= *other*

Magic Methods for Math

Method	Description
__add__(*self*, *other*)	*self* + *other*
__sub__(*self*, *other*)	*self* - *other*
__mul__(*self*, *other*)	*self* * *other*
__floordiv__(*self*, *other*)	*self* // *other*
__truediv__(*self*, *other*)	*self* / *other*
__mod__(*self*, *other*)	*self* % *other*
__pow__(*self*, *other*)	*self* ** *other*

Magic Methods for Comparison

```
[1]: class Word:
         def __init__(self, text):
             self.text = text
         def __eq__(self, word2):
             return self.text.lower() == word2.text.lower()
```

```
[2]: first = Word('ha')
```

```
[3]: second = Word('HA')
```

```
[4]: third = Word('Ha Ha')
```

```
[5]: first == second
```

```
[5]: True
```

```
[6]: first == third
```

```
[6]: False
```

- [1] We define a Word class with __init__ and __eq__ methods.
- [1] __eq__ now does customized == comparison by ignoring case.
- [2-6] The client codes to test our class implementation

Other Useful Magic Methods

Two Display Methods

- __str__ is tried first for the print operation and the str built-in function (the internal equivalent of which print runs). It generally should return a user-friendly display.

- __repr__ is used in all other contexts: for interactive echoes, the repr function, and nested appearances, as well as by print and str if no __str__ is present. It should generally return an as-code string that could be used to re-create the object, or a detailed display for developers.

Method	Description
`__str__`(*self*)	`str`(*self*)
`__repr__`(*self*)	`repr`(*self*)
`__len__`(*self*)	`len`(*self*)

Magic Methods for Display

```
[7]: class Word2:
         def __init__(self, text):
             self.text = text
         def __eq__(self, word2):
             return self.text.lower() == word2.text.lower()
         def __str__(self):
             return self.text
         def __repr__(self):
             return('Word("'+ self.text + '")')
```

```
[8]: first = Word2('Ha Ha Ha')
```

```
[9]: first
```

```
[9]: Word("Ha Ha Ha")
```

```
[10]: print(first)
```

```
Ha Ha Ha
```

- [1] We define a Word2 class with __init__, __eq__, __str__, and __repr__ methods.
- [9-10] The client codes that call __repr__ and __str__ methods.

L1_MagicMethods.ipynb

Example: Number Class Using Magic Methods

- Implement a Number class that supports the operators specified by the client codes.

```
[1]:

[2]: a = Number(20)

[3]: a

[3]: Number: 20

[4]: b = Number(10)

[5]: b

[5]: Number: 10

[6]: c = Number(5)

[7]: print(a + b)

30

[8]: print(a + b + c)

35
```

Example: Number Class Using Magic Methods (Ans)

- Implement a Number class that supports the operators specified by the client codes.

L1_Number.ipynb

```
[1]: class Number:
         def __init__(self, num):
             self._num = num
         def __add__(self, num):
             return Number(self._num + num._num)
         def __repr__(self):
             return f'Number: {self._num}'
         def __str__(self):
             return f'{self._num}'

[2]: a = Number(20)

[3]: a

[3]: Number: 20

[4]: b = Number(10)

[5]: b

[5]: Number: 10

[6]: c = Number(5)

[7]: print(a + b)

     30

[8]: print(a + b + c)

     35
```

Food for Thought: Managing Complexity

- **Programming is about managing complexity.**

- **There are two powerful mechanisms, to accomplish this: decomposition and abstraction.**

- **Decomposition creates structure.**
  - It allows us to break a program into parts that are self-contained.
  - **Functions** are the major facilitator.

- **Abstraction hides details.**
  - It allows us to use a piece of code as if it were a black box. For example, **lists**.
  - **Abstract data types** are the major facilitator.
  - **Class** allows us to create abstract data types of our own.

Summary

- This module covered:
  - The `class` keyword
  - The `self` parameter
  - `__init__` (constructor)
  - Instance and class attributes
  - Instance methods
  - `__str__`
  - Comparison magic methods, math magic methods, display magic methods

To be continued…