Object-oriented programming Python
Classes

The focal point of **Object Oriented Programming (OOP)** are **objects**, which are created using **classes**.
The **class** describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition.
You can use the same class as a blueprint for creating multiple different objects.

Classes are created using the keyword **class** and an indented block, which contains class **methods** (which are functions).
Below is an example of a simple class and its objects.

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

This code defines a class named **Cat**, which has two attributes: **color** and **legs**.
Then the class is used to create 3 separate objects of that class.

__init__

The **__init__** method is the most important method in a class.
This is called when an instance (object) of the class is created, using the class name as a function.

All methods must have **self** as their first parameter, although it isn't explicitly passed, Python adds the **self** argument to the list for you; you do not need to include it when you call the methods. Within a method definition, **self** refers to the instance calling the method.

Instances of a class have **attributes**, which are pieces of data associated with them.
In this example, **Cat** instances have attributes **color** and **legs**. These can be accessed by putting a **dot**, and the attribute name after an instance.
In an **__init__** method, **self.attribute** can therefore be used to set the initial value of an instance's attributes.

Example:

```
class Cat:
   def __init__(self, color, legs):
      self.color = color
      self.legs = legs

felix = Cat("ginger", 4)
print(felix.color)
```

ginger

In the example above, the **__init__** method takes two arguments and assigns them to the object's attributes.
The **__init__** method is called the class **constructor**.

Methods

Classes can have other **methods** defined to add functionality to them.
Remember, that all methods must have **self** as their first parameter.
These methods are accessed using the same **dot** syntax as attributes.

Example:

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

   def bark(self):
      print("Woof!")

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Fido
Woof!

Class attributes are shared by all instances of the class.

Problem

You are making a video game! The given code declares a **Player** class, with its attributes and an **intro**() method. Complete the code to take the name and level from user input, create a Player object with the corresponding values and call the **intro**() method of that object.

Sample Input

Tony
12

Sample Output

Tony (Level 12)

Use the dot syntax to call the **intro**() method for the declared object.

```
class Player:
   def __init__(self, name, level):
      self.name = name
      self.level = level

   def intro(self):
      print(self.name + " (Level " + self.level + ")")

name=input()
level=input()

p=Player(name,level)
p.intro()
```

Input
abc X
Output
abc (Level X)

Inheritance

Inheritance provides a way to share functionality between classes.
Imagine several classes, **Cat**, **Dog**, **Rabbit** and so on. Although they may differ in some ways (only **Dog** might have the method **bark**), they are likely to be similar in others (all having the attributes **color** and **name**).
This similarity can be expressed by making them all inherit from a **superclass Animal**, which contains the shared functionality.
To inherit a class from another class, put the superclass name in parentheses after the class name.

Example:

```
class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
    def purr(self):
        print("Purr...")

class Dog(Animal):
    def bark(self):
        print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

brown
Woof!
A class that inherits from another class is called a **subclass**.
A class that is inherited from is called a **superclass**.
If a class inherits from another with the same attributes or methods, it overrides them.

```
class Wolf:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print("Grr...")

class Dog(Wolf):
    def bark(self):
        print("Woof")

husky = Dog("Max", "grey")
husky.bark()
```

Try it Yourself

Woof
In the example above, **Wolf** is the superclass, **Dog** is the subclass.
The function **super** is a useful inheritance-related function that refers to the parent class. It can be used to find the method with a certain name in an object's superclass.

Example:

```
class A:
    def spam(self):
        print(1)
```

```
class B(A):
   def spam(self):
      print(2)
      super().spam()

B().spam()
```
2
1
super().spam() calls the **spam** method of the superclass.
Problem

You are making a drawing application, which has a **Shape** base class.
The given code defines a Rectangle class, creates a Rectangle object and calls its **area()** and **perimeter()** methods.

Do the following to complete the program:
1. Inherit the **Rectangle** class from **Shape**.
2. Define the **perimeter**() method in the Rectangle class, printing the perimeter of the rectangle.

The perimeter is equal to **2*(width+height)**

```
class Shape:
   def __init__(self, w, h):
      self.width = w
      self.height = h

   def area(self):
      print(self.width*self.height)

class Rectangle(Shape):
   def perimeter(self):
      print(2*(self.width+self.height))


w = int(input())
h = int(input())

r = Rectangle(w, h)
r.area()
r.perimeter()
```
Input
12 42
Output
504 108

Magic Methods

Magic methods are special methods which have **double underscores** at the beginning and end of their names.
They are also known as **dunders**.
So far, the only one we have encountered is **__init__**, but there are several others.
They are used to create functionality that can't be represented as a normal method.

One common use of them is **operator overloading**.
This means defining operators for custom classes that allow operators such as + and * to be used on them.
An example magic method is **__add__** for +.
```
class Vector2D:
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

first = Vector2D(5, 7)
second = Vector2D(3, 9)
result = first + second
print(result.x)
print(result.y)
```

```
8
16
```

The **__add__** method allows for the definition of a custom behavior for the + operator in our class.
As you can see, it adds the corresponding attributes of the objects and returns a new object, containing the result.
Once it's defined, we can add two objects of the class together.

More magic methods for common operators:
__sub__ for -
__mul__ for *
__truediv__ for /
__floordiv__ for //
__mod__ for %
__pow__ for **
__and__ for &
__xor__ for ^
__or__ for |

The expression **x + y** is translated into **x.__add__(y)**.
However, if x hasn't implemented __add__, and x and y are of different types, then **y.__radd__(x)** is called.
There are equivalent **r** methods for all magic methods just mentioned.

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __truediv__(self, other):
        line = "=" * len(other.cont)
        return "\n".join([self.cont, line, other.cont])

spam = SpecialString("spam")
hello = SpecialString("Hello world!")
print(spam / hello)
```

```
spam
============
Hello world!
```

In the example above, we defined the **division** operation for our class **SpecialString**.

Python also provides magic methods for comparisons.
__lt__ for <
__le__ for <=
__eq__ for ==
__ne__ for !=
__gt__ for >
__ge__ for >=

If **__ne__** is not implemented, it returns the opposite of **__eq__**.
There are no other relationships between the other operators.

Example:

```
class SpecialString:
   def __init__(self, cont):
      self.cont = cont

   def __gt__(self, other):
      for index in range(len(other.cont)+1):
         result = other.cont[:index] + ">" + self.cont
         result += ">" + other.cont[index:]
         print(result)

spam = SpecialString("spam")
eggs = SpecialString("eggs")
spam > eggs
```

>spam>eggs
e>spam>ggs
eg>spam>gs
egg>spam>s
eggs>spam>
As you can see, you can dThere are several magic methods for making classes act like containers.
__len__ for len()
__getitem__ for indexing
__setitem__ for assigning to indexed values
__delitem__ for deleting indexed values
__iter__ for iteration over objects (e.g., in for loops)
__contains__ for in

There are many other magic methods that we won't cover here, such as **__call__** for calling objects as functions, and **__int__**, **__str__**, and the like, for converting objects to built-in types.

Example:

```
import random
class VagueList:
   def __init__(self, cont):
      self.cont = cont

   def __getitem__(self, index):
      return self.cont[index + random.randint(-1, 1)]

   def __len__(self):
      return random.randint(0, len(self.cont)*2)

vague_list = VagueList(["A", "B", "C", "D", "E"])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

Try it Yourself
define any custom behavior for the overloaded operators.
1
9
D
D

We have overridden the len() function for the class VagueList to return a random number.
The indexing function also returns a random item in a range from the list, based on the expression.

Problem

We are improving our drawing application.
Our application needs to support adding and comparing two **Shape** objects.
Add the corresponding methods to enable addition + and comparison using the greater than > operator for the Shape class.

The addition should return a new object with the sum of the widths and heights of the operands, while the comparison should return the result of comparing the areas of the objects.
The given code creates two **Shape** objects from user input, outputs the area() of their addition and compares them.

```
class Shape:
    def __init__(self, w, h):
        self.width = w
        self.height = h

    def area(self):
        return self.width * self.height

    def __add__(self, other):
        return Shape(self.width + other.width, self.height + other.height)

    def __gt__(self, other):
        return self.area() > other.area()

w1 = int(input())
h1 = int(input())
w2 = int(input())
h2 = int(input())

s1 = Shape(w1, h1)
s2 = Shape(w2, h2)
result = s1 + s2

print(result.area())
print(s1 > s2)
```

Input
2 5 1 3
Output
24
True

Data Hiding

A key part of object-oriented programming is **encapsulation**, which involves packaging of related variables and functions into a single easy-to-use object -- an instance of a class.
A related concept is **data hiding**, which states that implementation details of a class should be hidden, and a clean standard interface be presented for those who want to use the class.
In other programming languages, this is usually done with private methods and attributes, which block external access to certain methods and attributes in a class.

The Python philosophy is slightly different. It is often stated as **"we are all consenting adults here"**, meaning that

you shouldn't put arbitrary restrictions on accessing parts of a class. Hence there are no ways of enforcing that a method or attribute be strictly private.

However, there are ways to discourage people from accessing parts of a class, such as by denoting that it is an implementation detail, and should be used at their own risk.

```
class Queue:
   def __init__(self, contents):
      self._hiddenlist = list(contents)

   def push(self, value):
      self._hiddenlist.insert(0, value)

   def pop(self):
      return self._hiddenlist.pop(-1)

   def __repr__(self):
      return "Queue({})".format(self._hiddenlist)

queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
print(queue)
print(queue._hiddenlist)
```

```
Queue([1, 2, 3])
Queue([0, 1, 2, 3])
Queue([0, 1, 2])
[0, 1, 2]
```

In the code above, the attribute **_hiddenlist** is marked as private, but it can still be accessed in the outside code. The **__repr__** magic method is used for string representation of the instance.

Strongly private methods and attributes have a **double underscore** at the beginning of their names. This causes their names to be mangled, which means that they can't be accessed from outside the class.
The purpose of this isn't to ensure that they are kept private, but to avoid bugs if there are subclasses that have methods or attributes with the same names.
Name mangled methods can still be accessed externally, but by a different name. The method **__privatemethod** of class **Spam** could be accessed externally with **_Spam__privatemethod**.

```
class Spam:
   __egg = 7
   def print_egg(self):
      print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

```
7
7


Traceback (most recent call last):
  File "file0.py", line 9, in <module>
```

```
    print(s.__egg)
AttributeError: 'Spam' object has no attribute '__egg'
```

Basically, Python protects those members by internally changing the name to include the class name.

Problem
We are working on a game. Our **Player** class has **name** and private **_lives** attributes.
The **hit**() method should decrease the lives of the player by 1. In case the lives equal to 0, it should output "**Game Over**".
Complete the hit() method to make the program work as expected.
The code creates a **Player** object and calls its **hit(**) method multiple times.

```
class Player:
    def __init__(self, name, lives):
        self.name = name
        self._lives = lives

    def hit(self):
        self._lives-=1
        if self._lives==0:
            print("Game Over")


p = Player("Cyberpunk77", 3)
p.hit()
p.hit()
p.hit()
```

Class & Static Methods

Class Methods

Methods of objects we've looked at so far are called by an instance of a class, which is then passed to the **self** parameter of the method.
Class methods are different -- they are called by a class, which is passed to the **cls** parameter of the method.
A common use of these are factory methods, which instantiate an instance of a class, using different parameters than those usually passed to the class constructor.
Class methods are marked with a **classmethod decorator**.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    @classmethod
    def new_square(cls, side_length):
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

new_square is a class method and is called on the class, rather than on an instance of the class. It returns a new object of the class **cls**.
Technically, the parameters **self** and **cls** are just conventions; they could be changed to anything else. However, they are universally followed, so it is wise to stick to using them.

Static Methods

Static methods are similar to class methods, except they don't receive any additional arguments; they are identical to normal functions that belong to a class.
They are marked with the **staticmethod** decorator.

Example:

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == "pineapple":
            raise ValueError("No pineapples!")
        else:
            return True

ingredients = ["cheese", "onions", "spam"]
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
```

No output.
Static methods behave like plain functions, except for the fact that you can call them from an instance of the class.

Problem

The given code takes 2 numbers as input and calls the static **area**() method of the **Shape** class, to output the area of the shape, which is equal to the height multiplied by the width.
To make the code work, you need to define the Shape class, and the static area() method, which should return the multiplication of its two arguments.
Use the **@staticmethod** decorator to define a static method.

```
class Shape:
    @staticmethod
    def area(height, width):
        return height * width

w = int(input())
h = int(input())

print(Shape.area(w, h))
```

Input
4
5
Output
20

Properties

Properties provide a way of customizing access to instance attributes.
They are created by putting the **property** decorator above a method, which means when the instance attribute with the same name as the method is accessed, the method will be called instead.
One common use of a property is to make an attribute **read-only**.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
        self._pineapple_allowed = False

    @property
    def pineapple_allowed(self):
        return self._pineapple_allowed

    @pineapple_allowed.setter
    def pineapple_allowed(self, value):
        self._pineapple_allowed = value

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

```
False
True
```

Properties can also be set by defining **setter/getter** functions.
The **setter** function sets the corresponding property's value.
The **getter** gets the value.
To define a **setter**, you need to use a decorator of the same name as the property, followed by a dot and the **setter** keyword.
The same applies to defining **getter** functions.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
        self._pineapple_allowed = False

    @property
    def pineapple_allowed(self):
        return self._pineapple_allowed

    @pineapple_allowed.setter
    def pineapple_allowed(self, value):
        if value:
            password = input("Enter the password: ")
            if password == "Sw0rdf1sh!":
                self._pineapple_allowed = value
            else:
                raise ValueError("Alert! Intruder!")

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

Problem
Properties

We are improving our game and need to add an **isAlive** property, which returns **True** if the lives count is greater than 0.
Complete the code by adding the **isAlive** property.
The code uses a while loop to hit the **Player**, until its lives count becomes 0, using the **isAlive** property to make the condition.

```
class Player:
   def __init__(self, name, lives):
      self.name = name
      self._lives = lives

   def hit(self):
      self._lives -= 1

   @property
   def isAlive(self):
       if self._lives > 0:
          return True


p = Player("Cyberpunk77", int(input()))
i = 1
while True:
   p.hit()
   print("Hit # " + str(i))
   i += 1
   if not p.isAlive:
      print("Game Over")
      break
```

Input
7
Output
Hit # 1
Hit # 2
Hit # 3
Hit # 4
Hit # 5
Hit # 6
Hit # 7
Game Over

Problem

You are creating a shooting game!
The game has two types of enemies, **aliens** and **monsters**. You shoot the aliens using your **laser**, and monsters using your **gun**.
Each hit decreases the lives of the enemies by 1.
The given code declares a generic **Enemy** class, as well as the **Alien** and **Monster** classes, with their corresponding lives count.
It also defines the **hit**() method for the Enemy class.

You need to do the following to complete the program:
1. Inherit the **Alien** and **Monster** classes from the **Enemy** class.
2. Complete the while loop that continuously takes the weapon of choice from user input and call the corresponding

object's hit() method.

Sample Input
laser
laser
gun
exit

Sample Output
Alien has 4 lives
Alien has 3 lives
Monster has 2 lives

The while loop stops when the user enters '**exit**'.

```
class Enemy:
  name = ""
  lives = 0
  def __init__(self, name, lives):
    self.name = name
    self.lives = lives

  def hit(self):
    self.lives -= 1
    if self.lives <= 0:
       print(self.name + ' killed')
    else:
        print(self.name + ' has '+ str(self.lives) + ' lives')

m = Enemy('Monster',3)
a = Enemy('Alien',5)

while True:
    x = input()
    if x == 'exit':
      break
    elif x=='laser':
      a.hit()
    else:
      m.hit()
```

Input
gun
gun
gun
exit
Output
Monster has 2 lives
Monster has 1 lives
Monster killed