Ignacio Méndez Pérez
2023

Resumen Programación Avanzada

Objetos:

- Métodos y atributos privados:
  Métodos y atributos de objetos que comiencen con "__" serán privados de la clase, y solo podrán ser accedidos desde la misma clase. Si trato de crear esa clase llamada "`objeto`", y luego llamar al atributo/método con "`objeto.__atributo`" dará error.

- "*Properties*":
  Las properties en las clases de Python son métodos especiales que se utilizan para definir atributos que se comportan como variables, pero tienen lógica adicional para la lectura y escritura de los mismos. Permiten un control más fino sobre el acceso y la manipulación de los atributos de una clase.

  Las properties se definen utilizando el decorador `@property` antes del método que actuará como *getter* (método de obtención) y, opcionalmente, los decoradores `@<nombre_atributo>.setter` y `@<nombre_atributo>.deleter` para definir los métodos que actuarán como setter (método de asignación) y *deleter* (método de eliminación), respectivamente.

```python
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.__edad = edad

    @property
    def nombre(self):
        return self.__nombre

    @nombre.setter
    def nombre(self, nuevo_nombre):
        self.__nombre = nuevo_nombre

    @property
    def edad(self):
        return self.__edad

    @edad.setter
    def edad(self, nueva_edad):
        if nueva_edad ≥ 0:
            self.__edad = nueva_edad
        else:
            raise ValueError("La edad no puede ser negativa.")

    def __metodo_privado(self):
        print("Este es un método privado.")

    def otro_metodo(self):
        self.__metodo_privado()
        print(f"Nombre: {self.__nombre}, Edad: {self.__edad}")


persona = Persona("Juan", 25)

print(persona.nombre)  # Imprime "Juan"
persona.nombre = "Pedro"
print(persona.nombre)  # Imprime "Pedro"

print(persona.edad)  # Imprime 25
persona.edad = 30
print(persona.edad)  # Imprime 30

# Imprime "Este es un método privado." y "Nombre: Pedro, Edad: 30"
persona.otro_metodo()
```

- Herencia:

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def comer(self):
        print(f"{self.nombre} está comiendo.")


class Mascota:
    def __init__(self, nombre, dueño):
        self.nombre = nombre
        self.dueño = dueño

    def mostrar_detalles(self):
        print(f"{self.nombre} es la mascota de {self.dueño}.")


class Perro(Animal, Mascota):
    def __init__(self, nombre, dueño, raza):
        super().__init__(nombre)  # Llamada al constructor de la clase base Animal
        Mascota.__init__(self, nombre, dueño)  # Llamada al constructor de la clase base Mascota
        self.raza = raza

    def ladrar(self):
        print("¡Guau guau!")


perro = Perro("Fido", "Juan", "Labrador")
perro.comer()  # Llamada al método comer() de la clase base Animal
perro.mostrar_detalles()  # Llamada al método mostrar_detalles() de la clase base Mascota
perro.ladrar()  # Llamada al método ladrar() de la clase derivada Perro
```

Ignacio Méndez Pérez
2023

Listas, tuplas y "*named tuples*"

Estructura	Inmutable	Hasheable	Comentarios
Lista	❌	❌	Permiten agregar, eliminar, modificar elementos.
Tupla	✅	✅	Sirven como llaves de diccionarios y para retornar múltiples valores.
Named Tuple	✅	✅	Se puede acceder a cada posición mediante un nombre.

Resumen

Estructura	Insertar	Búsqueda por índice	Búsqueda por llave	Búsqueda por valor
Lista	✅	✅✅✅	❌	✅
Tupla	❌	✅	❌	✅
Stack	✅	❌	❌	❌
Cola (*deque*)	✅✅✅	✅	❌	✅
Diccionario	✅✅✅	❌	✅✅✅	✅
Set	✅✅✅	❌	✅✅✅	✅

Yield

En Python, la palabra clave `yield` se utiliza en una función para crear un generador. Un generador es un tipo especial de iterable que se puede recorrer de forma secuencial, pero a diferencia de una lista o una tupla, **no se almacenan todos los valores en memoria de una vez. En cambio, los valores se generan bajo demanda, conservando el estado de la función**.

Cuando se encuentra la instrucción `yield` dentro de una función, la función se convierte en un **generador**. Al llamar al generador, no se ejecuta la función en su totalidad de una vez, sino que se devuelve un objeto generador. Luego, el **generador se puede recorrer utilizando un bucle `for`**, por ejemplo, para obtener cada valor generado individualmente.

```
1   def contador(maximo):
2       contador = 0
3       while contador ≤ maximo:
4           yield contador
5           contador += 1
6
7   generador = contador(5)
8
9   for valor in generador:
10      print(valor)
```

```
0
1
2
3
4
5
```

En el ejemplo, la función `contador` es un generador que produce una secuencia de números del 0 al valor máximo especificado. En lugar de retornar una lista completa de números, utiliza `yield` para generar y devolver cada número de forma individual en cada iteración del bucle `while`. Esto permite que el generador conserve su estado y continúe generando números bajo demanda.

Al llamar a contador(5), se devuelve un objeto generador que se asigna a la variable `generador`. Luego, al recorrer el generador en un bucle `for`, se invoca la función `contador` **hasta que alcanza la instrucción `yield`, que devuelve el siguiente valor generado**. Esto se repite en cada iteración del bucle `for` hasta que se agotan todos los valores generados.

Enumerate

```python
frutas = ['manzana', 'banana', 'cereza']

for indice, fruta in enumerate(frutas):
    print(f"Índice: {indice}, Fruta: {fruta}")
```

Zip

```python
nombres = ['Juan', 'María', 'Pedro']
edades = [25, 30, 35]

for nombre, edad in zip(nombres, edades):
    print(f"Nombre: {nombre}, Edad: {edad}")
```

Reversed

```python
numeros = [1, 2, 3, 4, 5]

for numero in reversed(numeros):
    print(numero)
```

Sorted

```python
numeros = [5, 3, 1, 4, 2]

numeros_ordenados = sorted(numeros)

print(numeros_ordenados)
```

Una `namedtuple` en Python es una subclase de `tuple` que permite asignar nombres a los elementos individuales dentro de la tupla. Proporciona una forma más legible y semántica de acceder a los elementos en lugar de utilizar índices numéricos.

La función `namedtuple` del módulo `collections` se utiliza para crear una clase de tupla nombrada. Se le pasa un nombre para la clase como primer argumento y una secuencia de nombres de campos como segundo argumento.

Cuando se crea una instancia de una tupla nombrada, se puede acceder a los elementos mediante sus nombres, como si fueran atributos de la instancia. Esto hace que el código sea más legible y autodocumentado.

```python
from collections import namedtuple

# Definir la estructura de la tupla nombrada
Persona = namedtuple("Persona", ["nombre", "edad", "ciudad"])

# Crear una instancia de la tupla nombrada
persona = Persona(nombre="Juan", edad=30, ciudad="Madrid")

# Acceder a los elementos de la tupla nombrada
print(persona.nombre)   # Imprime "Juan"
print(persona.edad)     # Imprime 30
print(persona.ciudad)   # Imprime "Madrid"

# También se puede acceder mediante índices
print(persona[0])       # Imprime "Juan"
print(persona[1])       # Imprime 30
print(persona[2])       # Imprime "Madrid"

# Desempaquetar los valores de la tupla nombrada
nombre, edad, ciudad = persona
print(nombre, edad, ciudad)  # Imprime "Juan 30 Madrid"
```

Ignacio Méndez Pérez
2023

Lambda

```python
# Uso de lambda para definir una
# función anónima que suma dos números
suma = lambda x, y: x + y

resultado = suma(3, 5)
print(resultado)  # Imprime 8
```

Map

```python
# Uso de map para aplicar una
# función a una lista de números
numeros = [1, 2, 3, 4, 5]

doble = list(map(lambda x: x * 2, numeros))

print(doble)  # Imprime [2, 4, 6, 8, 10]
```

Filter

```python
# Uso de filter para filtrar números pares de una lista
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pares = list(filter(lambda x: x % 2 == 0, numeros))

print(pares)  # Imprime [2, 4, 6, 8, 10]
```

Reduce

```python
from functools import reduce

# Uso de reduce para calcular el producto de una lista de números
numeros = [1, 2, 3, 4, 5]

producto = reduce(lambda x, y: x * y, numeros)

print(producto)  # Imprime 120
```

*args

```python
def mi_funcion(*args):
    for arg in args:
        print(arg)

mi_funcion(1, 2, 3, 4, 5)
```

**kwargs

```python
def mi_funcion(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

mi_funcion(nombre="Juan", edad=30, ciudad="Madrid")
```

Decorador

En Python, los decoradores son una característica que permite modificar el comportamiento de una función o clase **sin cambiar su implementación interna**. Los decoradores son **funciones que envuelven a otra función** o clase y a**gregan funcionalidad adicional** antes o después de su ejecución, o incluso pueden modificar la propia función o clase.

Los decoradores se aplican utilizando la sintaxis del símbolo @, seguido del nombre del decorador antes de la definición de la función o clase que se va a decorar. Esto permite aplicar el decorador de forma clara y concisa.

```python
def decorador(funcion):
    def envoltura():
        print("Antes de ejecutar la función.")
        funcion()
        print("Después de ejecutar la función.")
    return envoltura

@decorador
def funcion_decorada():
    print("¡Hola, soy la función decorada!")

funcion_decorada()
```

Output:

```
Antes de ejecutar la función.
¡Hola, soy la función decorada!
Después de ejecutar la función.
```

Ignacio Méndez Pérez
2023

Threads

En Python, los *threads* (hilos) permiten ejecutar múltiples tareas de forma concurrente dentro de un mismo programa. Los *threads* son unidades de ejecución independientes que comparten el mismo espacio de memoria, lo que permite una ejecución más eficiente y paralela de tareas.

```python
import threading
import time

class MiThread(threading.Thread):
    def run(self):
        print("Thread iniciado")
        time.sleep(2)
        print("Thread finalizado")

# Crear instancia del thread
thread = MiThread()

# Iniciar ejecución del thread
thread.start()

# Realizar otras tareas en el thread principal
print("Tarea en el thread principal")

# Esperar a que el thread finalice
thread.join()

print("Programa finalizado")
```

```
Thread iniciado
Tarea en el thread principal
Thread finalizado
Programa finalizado
```

```python
import threading
import time

class MiThread(threading.Thread):
    def run(self):
        print("Thread iniciado")
        time.sleep(2)
        print("Thread finalizado")

# Crear instancia del thread
thread = MiThread()

# Iniciar ejecución del thread
thread.start()

# Realizar otras tareas en el thread principal
print("Tarea en el thread principal")

# No es necesario esperar a que el hilo finalice

print("Programa finalizado")
```

```
Thread iniciado
Tarea en el thread principal
Programa finalizado
Thread finalizado
```

```python
import threading
import time

class MiThread(threading.Thread):
    def run(self):
        print("Thread iniciado")
        time.sleep(2)
        print("Thread finalizado")

# Crear instancia del thread
thread = MiThread()
# Indicamos que el thread finalizará
# cuando el programa principal finalice
thread.daemon = True

# Iniciar ejecución del thread
thread.start()

# Realizar otras tareas en el thread principal
print("Tarea en el thread principal")

# No es necesario esperar a que el hilo finalice

print("Programa finalizado")
```

```
Thread iniciado
Tarea en el thread principal
Programa finalizado
```

Lock:

El método `lock()` se utiliza para crear un objeto de bloqueo que se puede utilizar para coordinar el acceso exclusivo a un recurso compartido entre múltiples *threads*. Un objeto de bloqueo permite que solo un hilo a la vez adquiera el bloqueo y acceda al recurso compartido.

En este ejemplo, se utiliza `lock()` para crear un objeto de bloqueo llamado `lock`. Luego, se definen dos funciones (`incrementar()` y `decrementar()`) que modifican la variable `recurso_compartido` de forma concurrente. Dentro de cada función, se utiliza el bloque `with lock` para adquirir el bloqueo y garantizar que solo un *thread* a la vez pueda acceder y modificar `recurso_compartido`. Al final, se imprime el valor final de `recurso_compartido`.

```python
import threading

recurso_compartido = 0
lock = threading.Lock()

def incrementar():
    global recurso_compartido
    with lock:
        recurso_compartido += 1

def decrementar():
    global recurso_compartido
    with lock:
        recurso_compartido -= 1

hilo1 = threading.Thread(target=incrementar)
hilo2 = threading.Thread(target=decrementar)

hilo1.start()
hilo2.start()

hilo1.join()
hilo2.join()

print(recurso_compartido)
```

Ignacio Méndez Pérez
2023

Set:

El método `set()` se utiliza junto con un objeto `Condition` para señalar que una determinada condición se ha cumplido y desbloquear *threads* que están esperando a esa condición.

En este ejemplo, se utiliza `set()` junto con `notify()` para señalar que se ha producido un dato en la función `productor()`. Por otro lado, la función `consumidor()` utiliza `wait()` para esperar hasta que haya un dato disponible para consumir. Una vez que se señala que hay un dato, el *thread* consumidor continúa y consume el dato.

```python
import threading

condicion = threading.Condition()
recurso_compartido = []

def productor():
    with condicion:
        recurso_compartido.append('dato')
        condicion.notify()  # Señalar que se ha producido un dato

def consumidor():
    with condicion:
        while not recurso_compartido:
            condicion.wait()  # Esperar hasta que se produzca un dato
        dato = recurso_compartido.pop()
        print(f"Dato consumido: {dato}")

hilo_productor = threading.Thread(target=productor)
hilo_consumidor = threading.Thread(target=consumidor)

hilo_productor.start()
hilo_consumidor.start()

hilo_productor.join()
hilo_consumidor.join()
```

Wait:

El método `wait()` se utiliza junto con un objeto `Condition` para hacer que un *thread* espere hasta que se cumpla una cierta condición. El *thread* se suspende y libera el bloqueo hasta que otro *thread* llame a `notify()` o `notify_all()` en el mismo objeto `Condition` para señalar que se ha cumplido la condición y el *thread* puede continuar.

En este ejemplo, la función `consumidor()` utiliza `wait()` para esperar hasta que haya un dato disponible para consumir. Si la lista `recurso_compartido` está vacía, el *thread* consumidor se suspende llamando a `condicion.wait()`. Una vez que el *thread* productor añade un dato a la lista y llama a `condicion.notify()`, el *thread* consumidor se despierta y continúa su ejecución.

```python
import threading

condicion = threading.Condition()
recurso_compartido = []

def productor():
    with condicion:
        recurso_compartido.append('dato')
        condicion.notify()  # Señalar que se ha producido un dato

def consumidor():
    with condicion:
        while not recurso_compartido:
            condicion.wait()  # Esperar hasta que se produzca un dato
        dato = recurso_compartido.pop()
        print(f"Dato consumido: {dato}")

hilo_productor = threading.Thread(target=productor)
hilo_consumidor = threading.Thread(target=consumidor)

hilo_productor.start()
hilo_consumidor.start()

hilo_productor.join()
hilo_consumidor.join()
```

Error *handling*

```python
def dividir(a, b):
    try:
        resultado = a / b
        return resultado
    except ZeroDivisionError:
        print("Error: División por cero")
        raise

try:
    resultado = dividir(10, 0)
    print(f"El resultado de la división es: {resultado}")
except ZeroDivisionError:
    print("Error capturado en el bloque except")
```

```python
def multiply_numbers(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Ambos argumentos deben ser enteros")
    return a * b

# Ejemplos de uso
print(multiply_numbers(3, 4))  # Imprime: 12
print(multiply_numbers(2.5, 4))  # Genera TypeError
```

Ignacio Méndez Pérez
2023

Bytes y bytearray

```python
# Crear un objeto bytes
my_bytes = b"Hello, world!"

# Acceder a los elementos individuales
print(my_bytes[0])  # Imprime 72 (valor ASCII de 'H')
print(my_bytes[7])  # Imprime 119 (valor ASCII de 'w')

# Intentar modificar un elemento (genera un error)
my_bytes[0] = 65  # TypeError: 'bytes' object does not support item assignment
```

```python
# Crear un objeto bytearray
my_bytearray = bytearray(b"Hello, world!")

# Acceder a los elementos individuales
print(my_bytearray[0])  # Imprime 72 (valor ASCII de 'H')
print(my_bytearray[7])  # Imprime 119 (valor ASCII de 'w')

# Modificar un elemento
my_bytearray[0] = 65  # Modifica el primer byte a 65 (valor ASCII de 'A')
print(my_bytearray)  # Imprime bytearray(b'Aello, world!')
```

Lectura de archivos

- **"r"**: Modo de lectura (lectura por defecto). Abre el archivo en modo lectura, lo que te permite leer el contenido del archivo.
- **"w"**: Modo de escritura. Abre el archivo en modo escritura. Si el archivo existe, su contenido se sobrescribe. Si el archivo no existe, se crea uno nuevo.
- **"a"**: Modo de adjuntar (append). Abre el archivo en modo adjuntar. Si el archivo existe, el nuevo contenido se añade al final del archivo. Si el archivo no existe, se crea uno nuevo.
- **"rb"**: Modo de solo lectura en modo binario. Similar a "r", pero se usa para leer archivos binarios, como imágenes o archivos de audio.

```python
# Queremos leer los bytes de un archivo encriptado,
# y para recuperar el archivo original tenemos que
# armar grupos de 8 bytes e invertirlos
# Al leer el archivo como bytes no usamos encoding,
# estamos trabajando con los bytes directamente
with open("path_archivo", "rb") as archivo:
    # Leemos todos los bytes y los usamos como lista
    original = bytearray(archivo.read())
    # Hacemos un bytearray para la nueva versión
    modificado = bytearray()
    # Ahora podemos hacer el procesamiento
    for i in range(0, len(original), 8):
        segmento = original[i:i+8] # Agrupamos 8
        segmento = segmento[::-1] # Invertimos
        modificado.extend(segmento)
```

Pickle

La librería *pickle* en Python se utiliza para la serialización y deserialización de objetos Python. La serialización es el proceso de convertir un objeto en una secuencia de bytes, mientras que la deserialización es el proceso inverso, es decir, convertir una secuencia de bytes en un objeto.

```python
import pickle

tupla = ("a", 1, 3, "hola")
serializacion = pickle.dumps(tupla)

print(serializacion)
# b'\x80\x03X\x01\x00\x00\x00aq\x00K ...
print(type(serializacion))
# <class 'bytes'>
print(pickle.loads(serializacion))
# ('a', 1, 3, 'hola')
```

JSON

La librería json en Python se utiliza para trabajar con el formato de datos JSON (*JavaScript Object Notation*). JSON es un formato de intercambio de datos ampliamente utilizado debido a su simplicidad y legibilidad tanto para humanos como para máquinas. La librería json permite serializar objetos Python en formato JSON y deserializar datos JSON en objetos Python.

```python
import json

tupla = ("a", 1, 3, "hola")
serializacion = json.dumps(tupla)

print(serializacion)
# ['a', 1, 3, 'hola']
print(type(serializacion))
# <class 'str'>
print(json.loads(serializacion))
# ['a', 1, 3, 'hola']
```

Ignacio Méndez Pérez
2023

pickle

```python
import pickle

lista = [1, 2, 3, 7, 8, 3]

with open("mi_lista.bin", 'wb') as file:
    pickle.dump(lista, file)

with open("mi_lista.bin", 'rb') as file:
    lista_cargada = pickle.load(file)

print(f"¿Las listas son iguales? {lista == lista_cargada}")
print(f"¿Las listas son el mismo objeto? {lista is lista_cargada}")

> ¿Las listas son iguales? True
> ¿Las listas son el mismo objeto? False
```

JSON

```python
import json

lista = [1, 2, 3, 7, 8, 3]

with open("mi_lista.bin", 'w') as file:
    json.dump(lista, file)

with open("mi_lista.bin", 'r') as file:
    lista_cargada = json.load(file)

print(f"¿Las listas son iguales? {lista == lista_cargada}")
print(f"¿Las listas son el mismo objeto? {lista is lista_cargada}")

> ¿Las listas son iguales? True
> ¿Las listas son el mismo objeto? False
```

RegEx

1. “`.`” : Coincide con cualquier carácter excepto una nueva línea.
2. “`^`” : Coincide con el inicio de una cadena.
3. “`$`” : Coincide con el final de una cadena.
4. “`*`” : Coincide con cero o más repeticiones del elemento anterior.
5. “`+`” : Coincide con una o más repeticiones del elemento anterior.
6. “`?`” : Coincide con cero o una repetición del elemento anterior.
7. “`{n}`”: Coincide exactamente con n repeticiones del elemento anterior.
8. “`{n,}`”: Coincide con al menos n repeticiones del elemento anterior.
9. “`{n, m}`”: Coincide con entre n y `m` repeticiones del elemento anterior.
10. “`[]`” : Define un conjunto de caracteres permitidos en esa posición.
11. “`|`” : Se utiliza para especificar alternativas. Coincide con cualquiera de las expresiones separadas por el operador `|`.
12. “`()`” : Agrupan expresiones y capturan los resultados.
13. “`\`” : Escapa caracteres especiales o indica secuencias especiales como `\d`, `\w`, etc.
14. “`\d`”: Coincide con cualquier dígito.
15. “`\D`”: Coincide con cualquier carácter que no sea un dígito.
16. “`\w`”: Coincide con cualquier carácter alfanumérico o guion bajo.
17. “`\W`”: Coincide con cualquier carácter que no sea alfanumérico o guion bajo.
18. “`\s`”: Coincide con cualquier espacio en blanco.
19. “`\S`”: Coincide con cualquier carácter que no sea un espacio en blanco.
20. “`(?i)`”: Realiza una coincidencia sin distinción entre mayúsculas y minúsculas.
21. “`(?m)`”: Permite que ^ y $ coincidan con el inicio y el final de cada línea en lugar de solo la cadena completa.

El patrón utilizado en este caso es "`IIC\d{4}`", que busca el texto "`IIC`" seguido de exactamente cuatro dígitos (`0-9`). El meta-carácter `\d` se utiliza para representar cualquier dígito.

```python
import re

texto = "Mi cursos favoritos son IIC2233 e IIC1105, y \
        los menosfavoritos son FIS1533 e IIC321."
for curso in re.findall("IIC\d{4}", texto):
    # Encuentra todas las coincidencias y retorna una lista
    print(f"Se nombró al curso {curso}")
```

```python
import re

texto = "Mis correos son correo1@example.com y correo2@example.com"
patron = r"\b[\w.-]+@[\w.-]+\.\w+\b"

resultado = re.findall(patron, texto)
print(resultado)  # Imprime ['correo1@example.com', 'correo2@example.com']
```

- "\b": Representa un límite de palabra, lo cual significa que se busca una coincidencia al inicio o final de una palabra. Ayuda a garantizar que no se capturen partes de palabras más largas.
- "[\w.-]+": Representa una o más ocurrencias de caracteres alfanuméricos (\w), puntos (.) o guiones (-). Esto corresponde a la parte del nombre de usuario de la dirección de correo electrónico.
- "@": Coincide literalmente con el carácter @, que separa el nombre de usuario del dominio en una dirección de correo electrónico.
- "[\w.-]+": Representa una o más ocurrencias de caracteres alfanuméricos (\w), puntos (.) o guiones (-). Esto corresponde a la parte del dominio de la dirección de correo electrónico.
- "\.": Coincide literalmente con el carácter punto (.). Se utiliza un \ para escapar el punto, ya que el punto por sí solo representa cualquier carácter en una expresión regular.
- "\w+": Representa una o más ocurrencias de caracteres alfanuméricos (\w). Esto corresponde a la parte de la extensión de la dirección de correo electrónico.
- "\b": Otro límite de palabra, para asegurar que la coincidencia termine al final de una palabra.

Networking

En el siguiente ejemplo, el servidor espera conexiones entrantes en la dirección IP `127.0.0.1` (`localhost`) y el puerto `1234`. Cuando el cliente se conecta, el servidor acepta la conexión y luego entra en un bucle donde recibe mensajes del cliente y envía respuestas. El cliente se conecta al servidor en la misma dirección y puerto, y entra en un bucle donde solicita al usuario que ingrese mensajes para enviar al servidor. Ambos ejemplos utilizan `client_socket.send()` para enviar datos y `client_socket.recv()` para recibir datos.

Ignacio Méndez Pérez
2023

```python
import socket

# Configuración del servidor
host = '127.0.0.1'  # Dirección IP del servidor
port = 1234  # Puerto del servidor

# Crear un socket TCP/IP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Vincular el socket a la dirección y el puerto
server_socket.bind((host, port))

# Escuchar conexiones entrantes
server_socket.listen(1)

print('El servidor está en espera de conexiones ... ')

# Aceptar la conexión entrante
client_socket, client_address = server_socket.accept()
print('Conexión establecida desde:', client_address)

# Recibir datos del cliente y enviar una respuesta
while True:
    data = client_socket.recv(1024).decode()
    if not data:
        break  # Si no se reciben datos, se sale del bucle

    print('Mensaje recibido del cliente:', data)

    # Enviar una respuesta al cliente
    response = 'Respuesta del servidor'
    client_socket.send(response.encode())

# Cerrar la conexión con el cliente y el socket del servidor
client_socket.close()
server_socket.close()
```

```python
import socket

# Configuración del cliente
host = '127.0.0.1'  # Dirección IP del servidor
port = 1234  # Puerto del servidor

# Crear un socket TCP/IP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Conectar al servidor
client_socket.connect((host, port))

# Enviar datos al servidor y recibir una respuesta
while True:
    message = input('Ingrese un mensaje para el servidor: ')
    client_socket.send(message.encode())

    if message.lower() == 'bye':
        break  # Si el mensaje es 'bye', se sale del bucle

    # Recibir la respuesta del servidor
    response = client_socket.recv(1024).decode()
    print('Respuesta del servidor:', response)

# Cerrar la conexión con el servidor
client_socket.close()
```

Conexión inicial: Servidor

```python
class Servidor
    def __init__(self):
        self.host = "localhost" # Valor arbitrario
        self.port = 8080 # Valor arbitrario
        self.socket_servidor = \
            socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket_servidor.bind((self.host, self.port))
        self.socket_servidor.listen()
        print("Servidor iniciado.")
        self.socket_cliente = None
        Self.juego = None

    def esperar_conexion(self):
        print("Esperando cliente...")
        socket, _ = self.socket_servidor.accept()
        self.socket_cliente = socket
        print("¡Servidor conectado a cliente!")
        self.interactuar_con_cliente()
    ...
if __name__ == "__main__":
    servidor = Servidor()
    while True:
        try:
            servidor.esperar_conexion()
        except KeyboardInterrupt:
            print("\nServidor interrumpido")
            break
```

Crear socket hábil de escuchar conexiones

Acepta conexión y comienza a interactuar con ese cliente.

Servidor una vez creado, espera de a un cliente por siempre.

Conexión inicial: Cliente

```python
class Cliente:
    def __init__(self):
        self.host = "localhost" # Valor arbitrario
        self.port = 8080 # Valor arbitrario
        self.socket_cliente = \
            socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            self.socket_cliente.connect((self.host, self.port))
            print("Cliente conectado exitosamente al servidor.")
            self.interactuar_con_servidor()
        except ConnectionRefusedError:
            self.cerrar_conexion()
    ...
if __name__ == "__main__":
    Cliente()
```

Crear *socket*

Intenta conectarse al servidor, y si lo logra comienza a interactuar.