14 Objetos

Administrar programas más grandes

Al comienzo de este libro, presentamos cuatro patrones de programación básicos que utilizamos para construir programas:

- Código secuencial
- Código condicional (si las declaraciones)
- Código iterativo (bucles)
- Almacenar y reutilizar (funciones)

Después exploramos variables simples y estructuras de datos de recopilación como listas, tuplas y diccionarios.

A medida que construimos programas, diseñamos estructuras de datos y escribimos código para manipular esas estructuras de datos. Hay muchas formas de escribir programas y, a estas alturas, es probable que hayas escrito algunos programas que "no son tan elegantes" y otros que son "más elegantes". A pesar de que tus programas pueden ser pequeños, estás empezando a ver cómo escribir código tiene un poco de "arte" y "estética".

A medida que los programas alcanzan millones de líneas, cada vez es más importante escribir código que sea fácil de entender. Si estás trabajando en un programa de un millón de líneas, nunca podrás tener todo el programa en tu mente al mismo tiempo. Por lo tanto, necesitamos formas de dividir el programa en múltiples partes más pequeñas para resolver un problema, corregir un error o agregar una nueva función que tenemos menos que ver.

En cierto modo, la programación orientada a objetos es una forma de organizar tu código para que puedas hacer zoom en 500 líneas del código y entenderlo mientras ignoras las otras 999,500 líneas de código por el momento.

Comenzando

Al igual que muchos aspectos de la programación, es necesario aprender los conceptos de programación orientada a objetos antes de poder usarlos de manera efectiva. Así que enfoca este

capítulo como una manera de aprender algunos términos y conceptos y trabaja con algunos ejemplos simples para sentar las bases de un aprendizaje futuro. En el resto del libro usaremos objetos en muchos de los programas, pero no construiremos nuestros propios objetos nuevos en los programas.

El resultado clave de este capítulo es tener una comprensión básica de cómo se construyen los objetos y cómo funcionan y, lo que es más importante, cómo hacemos uso de las capacidades de los objetos que nos proporcionan las libreraís de Python y el propio Python.

Usando objetos

Resulta que hemos estado utilizando objetos a lo largo de este curso. Python nos proporciona muchos objetos incorporados. Aquí hay un código simple donde las primeras líneas deben ser muy simples y naturales para ti.

https://trinket.io/embed/python3/27adad9e85

Pero en lugar de centrarse en lo que logran estas líneas, echemos un vistazo a lo que realmente está sucediendo desde el punto de vista de la programación orientada a objetos. No te preocupes si los siguientes párrafos no tienen ningún sentido la primera vez que los lees porque todavía no hemos definido todos estos términos.

La primera línea es **construyendo** un objeto del tipo **lista**, la segunda y la tercera línea están llamando al método `append()`, la cuarta línea está llamando al método `sort ()`, y la quinta línea está recuperando el artículo en la posición 0.

La sexta línea está llamando al método `__getitem__()` en la lista `stuff` con un parámetro de cero.

```
print (stuff.__getitem__(0))
```

La séptima línea es una forma aún más detallada de recuperar el artículo 0 en la lista.

```
print (list.__getitem__(stuff,0))
```

En este código, lamamos al método `__getitem__` en la clase `list` y pasamos la lista (`stuff`) y el elemento que queremos recuperar de la lista como parámetros.

Las últimas tres líneas del programa son completamente equivalentes, pero es más conveniente simplemente usar la sintaxis del corchete para buscar un elemento en una posición particular en una lista.

Podemos observar las capacidades de un objeto mirando la salida de la función `dir()`:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

La definición precisa de `dir()` es que enumera los **métodos** y **atributos** de un objeto Python.

El resto de este capítulo definirá todos los términos anteriores, así que asegúrate de regresar después de terminar el capítulo y volver a leer los párrafos anteriores para verificar su comprensión.

Comenzando con programas

Un programa en su forma más básica toma algo de entrada, realiza algún procesamiento y produce algo de salida. Nuestro programa de conversión entre distintos sistemas de numeración de plantas, muestra un programa muy corto pero completo en tres pasos.

https://trinket.io/embed/python3/d3c8ba77d5

Si pensamos un poco más en este programa, existe el "mundo exterior" y el programa. Los puntos de entrada y salida son donde el programa interactúa con el mundo exterior. Dentro del programa,

tenemos código y datos para realizar la tarea que el programa está diseñado para resolver.

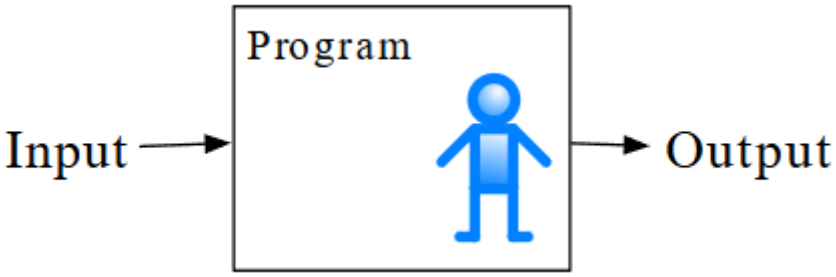

Imagen - Un programa

Cuando estamos "en" el programa, tenemos algunas interacciones definidas con el mundo "externo", pero esas interacciones están bien definidas y generalmente no son algo en lo que nos enfocamos. Mientras estamos programando, nos preocupamos solo por los detalles "dentro del programa".

Una forma de pensar acerca de la programación orientada a objetos es que estamos separando nuestro programa en múltiples "zonas". Cada "zona" contiene algunos códigos y datos (como un programa) y tiene interacciones bien definidas con el mundo exterior y las otras zonas dentro del programa.

Si miramos hacia atrás en la aplicación de extracción de enlaces donde usamos la biblioteca BeautifulSoup, podemos ver un programa que se construye conectando diferentes objetos para realizar una tarea:

https://trinket.io/embed/python3/9bb573e676

Leemos la URL en una cadena y luego la pasamos a `urllib` para recuperar los datos de la web. La biblioteca `urllib` usa la biblioteca `socket` para hacer la conexión de red real para recuperar los datos. Tomamos la cadena que recibimos de `urllib` y la entregamos a BeautifulSoup para que la analice. BeautifulSoup utiliza otro objeto llamado `html.parser`[1] y devuelve un objeto. Llamamos al método `tags()` en el objeto devuelto y luego obtenemos un diccionario de objetos `etiquetas`, hacemos un bucle a través de las etiquetas y llamamos al método `get()` para que cada etiqueta imprima el atributo 'href'.

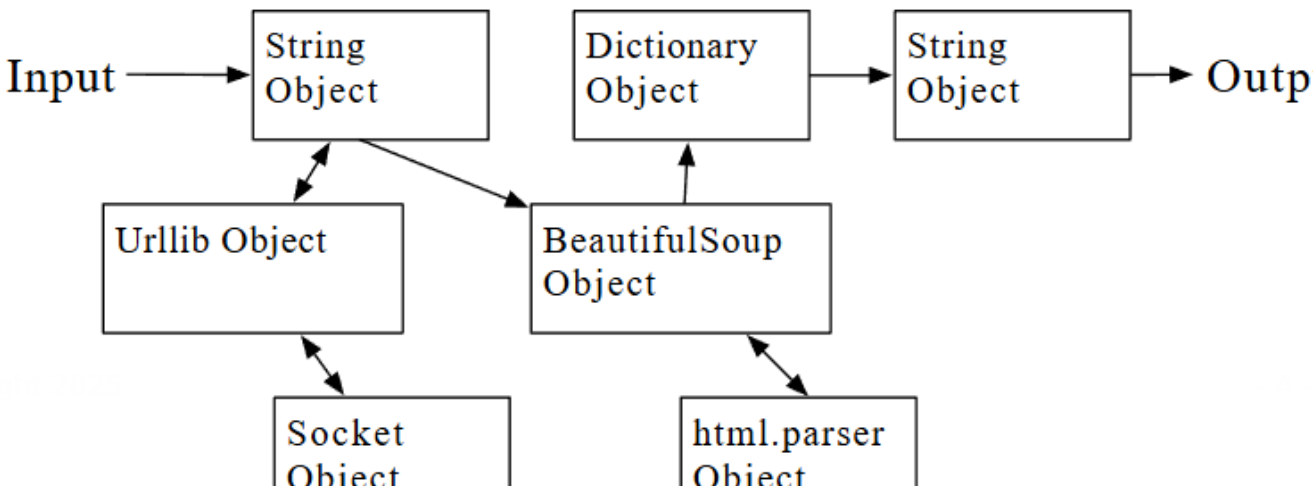

Imagen - Un programa como red de objetos

Podemos dibujar una imagen de este programa y cómo los objetos trabajan juntos.

La clave aquí no es entender completamente cómo funciona este programa, sino ver cómo construimos una red de objetos interactivos y organizamos el movimiento de información entre los objetos para crear un programa. También es importante tener en cuenta que cuando miraste ese programa varios capítulos atrás, podías entender completamente lo que estaba sucediendo en el programa sin siquiera darte cuenta de que el programa estaba "orquestando el movimiento de datos entre objetos". Entonces solo eran líneas de código que hacían el trabajo.

Subdivisión de un problema: encapsulación

Una de las ventajas del enfoque orientado a objetos es que puede ocultar la complejidad. Por ejemplo, aunque necesitamos saber cómo usar el código `urllib` y BeautifulSoup, no necesitamos saber cómo funcionan internamente esas bibliotecas. Nos permite enfocarnos en la parte del problema que necesitamos para resolver e ignorar las otras partes del programa.

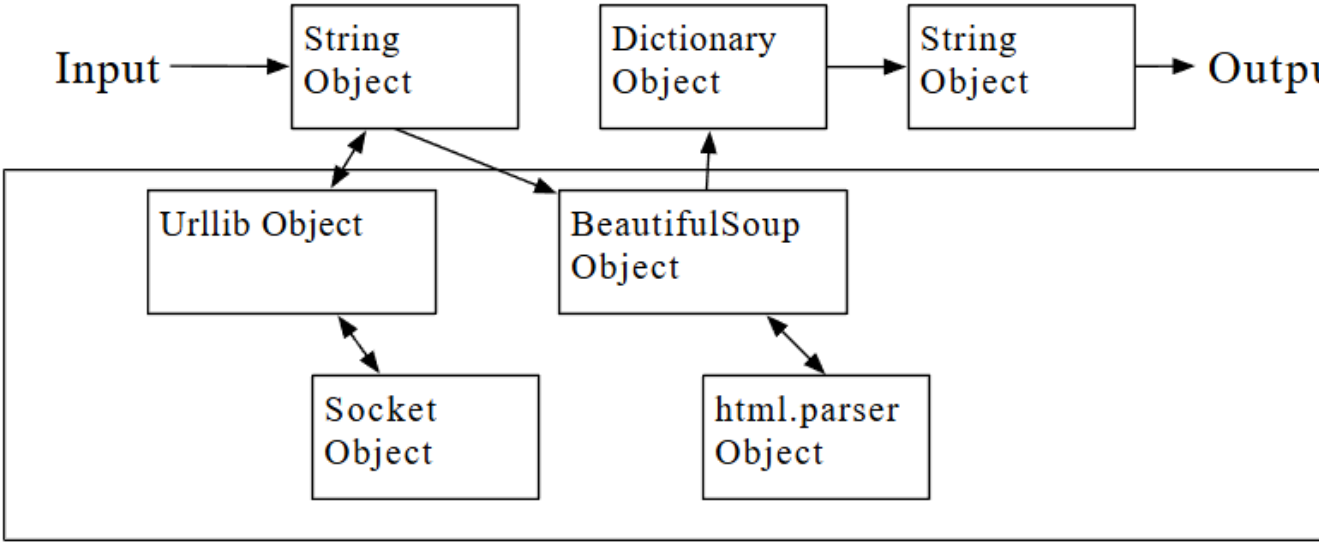

Imagen - Ignorar los detalles al usar un objeto

Esta capacidad de centrarse en una parte de un programa que nos importa e ignorar el resto del programa también es útil para los desarrolladores de los objetos. Por ejemplo, los programadores que desarrollan BeautifulSoup no necesitan saber ni preocuparse sobre cómo recuperamos nuestra página HTML, qué partes queremos leer o qué planeamos hacer con los datos que extraemos de la página web.

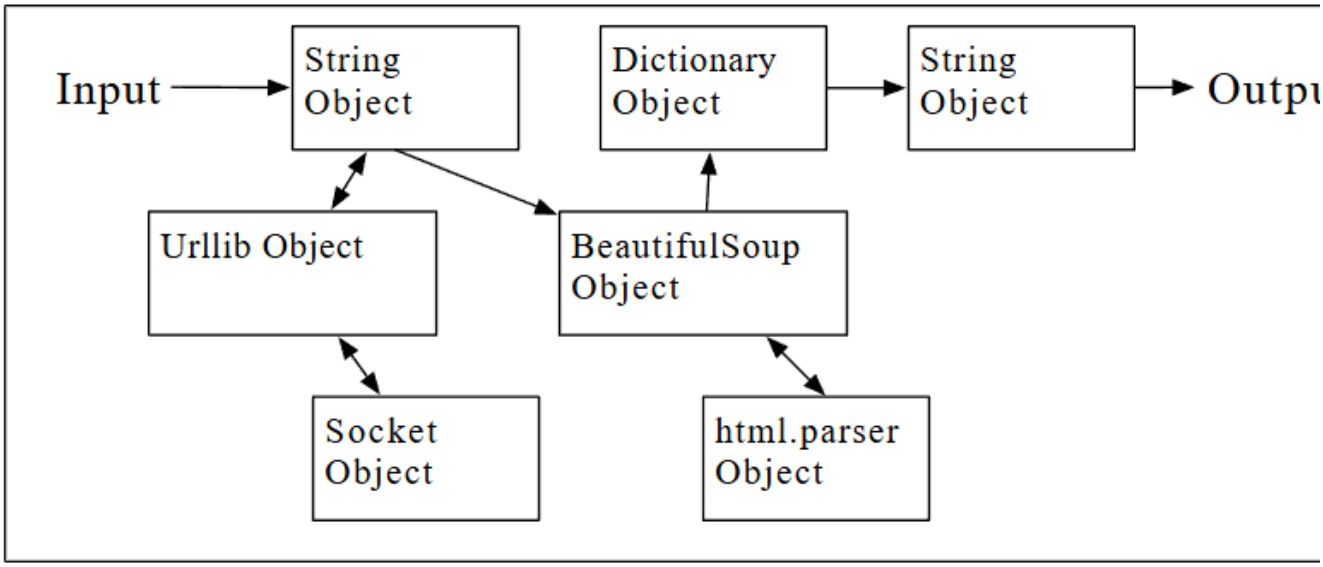

Imagen - Ignorar los detalles al construir un objeto

A esta idea de ignorar los detalles internos de los objetos que usamos también la llamamos "encapsulación". Esto significa que podemos saber cómo usar un objeto sin saber cómo cumple internamente lo que debemos hacer.

Nuestro primer objeto Python

En su forma más simple, un objeto es un código, con estructuras de datos, que es más pequeño que un programa completo. Definir una función nos permite almacenar un poco de código y darle un nombre y luego invocar ese código usando el nombre de la función.

Un objeto puede contener una serie de funciones (que denominamos "métodos"), así como los datos que utilizan esas funciones. A los elementos de datos que forman parte del objeto los llamamos "atributos".

Usamos la palabra clave `class` para definir los datos y el código que conformará cada uno de los objetos. La palabra clave class incluye el nombre de la clase y comienza un bloque de código con sangría donde incluimos los atributos (datos) y los métodos (funciones).

https://trinket.io/embed/python3/b88be451e5

Cada método se parece a una función, comenzando con la palabra clave `def` y consiste en un bloque de código con sangría. Este ejemplo tiene un atributo (x) y un método (party). Los métodos tienen un primer parámetro especial que nombramos por convención `self`.

Al igual que la palabra clave `def` no hace que se ejecute el código de función, la palabra clave `class` no crea un objeto. En cambio, la palabra clave `class` define una plantilla que indica qué datos y código estarán contenidos en cada objeto de tipo `PartyAnimal`. La clase es como un cortador de galletas y los objetos creados usando la clase son las propias galletas[2]. No eches azucar glass en el cortador de galletas, espolvoreando el glaseado en las galletas y puedes realizar un glaseado diferente en cada galleta.

Imagen - Una clase y dos objetos

Si continúa con el código de ejemplo, vemos la primera línea de código ejecutable:

```
an = PartyAnimal()
```

Aquí es donde le pedimos a Python que construya un **objeto** o "instancia de la clase llamada PartyAnimal". Parece una llamada de función a la clase en sí misma y Python construye el objeto con los métodos y datos correctos, devolviendo el objeto que luego se asigna a la variable `an`. En cierto modo, esto es bastante similar a la siguiente línea que hemos estado usando todo este tiempo:

```
counts = dict()
```

Aquí le estamos indicando a Python que construya un objeto usando la plantilla `dict` (ya presente en Python), devuelva la instancia del diccionario y la asigne a la variable `count`.

Cuando la clase PartyAnimal se usa para construir un objeto, la variable `an` se usa para apuntar a ese objeto. Usamos `an` para acceder al código y los datos de esa instancia particular de tipo PartyAnimal.

Cada objeto/instancia de Partyanimal contiene en su interior una variable `x` y un método/función llamado `party`. Llamamos a ese método `party` en esta línea:

```
an.party()
```

Cuando se llama al método `party`, el primer parámetro (al que llamamos por convención `self`) apunta a la instancia particular del objeto PartyAnimal que llama a `party` desde dentro. Dentro del método `party`, vemos la línea:

```
self.x = self.x + 1
```

Esta sintaxis que usa el operador 'punto' está diciendo 'la x dentro de uno mismo'. Entonces, cada vez que se llamas a `party()`, el valor interno de `x` se incrementa en 1 y el valor se imprime.

Para ayudar a entender la diferencia entre una función global y un método dentro de una clase / objeto, la siguiente línea es otra forma de llamar al método `party` dentro del objeto `an`:

```
PartyAnimal.party(an)
```

En esta variación, accedemos al código desde dentro de la **clase** y pasamos explícitamente el puntero del objeto `an` como primer parámetro (es decir, `self` dentro del método). Puedes pensar en `an.party()` como una abreviatura de la línea anterior.

Cuando el programa se ejecuta, produce el siguiente resultado:

```
So far 1
So far 2
So far 3
So far 4
```

El objeto se construye y el método `party` se llama cuatro veces, incrementando e imprimiendo el valor de `x` dentro del objeto `an`.

Clases como tipos

Como hemos visto, en Python, todas las variables tienen un tipo. Y podemos usar la función `dir` incorporada para examinar las capacidades de una variable. Podemos usar `type` y `dir` con las clases que creamos.

https://trinket.io/embed/python3/a502e6571d

Cuando este programa se ejecuta, produce la siguiente salida:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

Puedes ver que usando la palabra clave `class`, hemos creado un nuevo tipo. Desde la salida de `dir`, puede ver que tanto el atributo de entero `x` como el método `party` están disponibles en el objeto.

Object Lifecycle

En los ejemplos anteriores, estamos definiendo una clase (plantilla) y usando esa clase para crear una instancia de esa clase (objeto) y luego usando la instancia. Cuando el programa termina, todas las variables se descartan. Por lo general, no pensamos mucho en la creación y destrucción de variables, pero a menudo, a medida que nuestros objetos se vuelven más complejos, debemos tomar alguna acción dentro del objeto para configurar las cosas a medida que se está construyendo el objeto y posiblemente limpiar las cosas a medida que objeto está siendo descartado.

Si queremos que nuestro objeto sea consciente de estos momentos de construcción y destrucción, agregamos métodos especialmente nombrados a nuestro objeto:

https://trinket.io/embed/python3/75323d8f81

Cuando este programa se ejecuta, produce la siguiente salida:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

Como Python está construyendo nuestro objeto, llama a nuestro método `__init__` para darnos la oportunidad de configurar algunos valores predeterminados o iniciales para el objeto. Cuando Python encuentra la línea:

```
an = 42
```

... en realidad, "elimina nuestro objeto", reutilizando la variable `an` para almacenar el valor `42`. Justo en el momento en que nuestro objeto `an` se está 'destruyendo', se llama a nuestro código destructor (`__del__`). No podemos evitar que nuestra variable se destruya, pero podemos hacer cualquier limpieza necesaria justo antes de que nuestro objeto ya no exista.

Cuando se desarrollan objetos, es bastante común agregar un constructor a un objeto para que se establezca en los valores iniciales del objeto, pero es relativamente raro que se necesite un destructor para un objeto.

Muchas instancias

Hasta ahora, hemos estado definiendo una clase, creando un solo objeto, utilizando ese objeto y luego desechando el objeto. Pero el poder real en la orientación a objetos ocurre cuando hacemos muchos ejemplos de nuestra clase.

Cuando creamos varios objetos de nuestra clase, es posible que deseamos configurar diferentes valores iniciales para cada uno de los objetos. Podemos pasar datos a los constructores para dar a cada objeto un valor inicial diferente:

https://trinket.io/embed/python3/a1464da1b6

El constructor tiene un parámetro 'self' que apunta a la instancia del objeto y luego los parámetros adicionales que se pasan al constructor a medida que se construye el objeto:

```
s = PartyAnimal('Sally')
```

Dentro del constructor, la línea:

```
self.name = nam
```

Copia el parámetro que se pasa (`nam`) en el atributo `name` dentro de la instancia del objeto.

La salida del programa muestra que cada uno de los objetos (`s` y `j`) contienen sus propias copias independientes de `x` y `nam`:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

Herencia

Otra característica poderosa de la programación orientada a objetos es la capacidad de crear una nueva clase al extender una clase existente. Al extender una clase, llamamos a la clase original "clase principal" y a la nueva clase como "clase secundaria".

Para este ejemplo, moveremos nuestra clase `PartyAnimal` a su propio archivo:

https://trinket.io/embed/python3/859f1006a3

Luego, podemos 'importar' la clase `PartyAnimal` en un nuevo archivo y extenderlo de la siguiente manera:

https://trinket.io/embed/python3/980f08c259

Cuando estamos definiendo el objeto `CricketFan`, indicamos que estamos extendiendo la clase `PartyAnimal`. Esto significa que todas las variables (`x`) y métodos (`party`) de la clase `PartyAnimal` son heredadas por la clase `CricketFan`.

Puedes ver que dentro del método `six` en la clase `CricketFan`, podemos llamar al método `party` desde la clase `PartyAnimal`. Las variables y los métodos de la clase principal se **fusionan** en la clase secundaria.

A medida que se ejecuta el programa, podemos ver que `s` y `j` son instancias independientes de `PartyAnimal` y `CricketFan`. El objeto `j` tiene capacidades adicionales más allá del objeto `s`.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

En la salida de `dir` para el objeto `j` (instancia de la clase `CricketFan`) puedes ver que tiene los atributos y métodos de la clase principal, así como los atributos y métodos que se agregaron cuando la clase era ampliada para crear la clase `CricketFan`.

Resumen

Esta es una introducción muy rápida a la programación orientada a objetos que se centra principalmente en la terminología y la sintaxis de definir y utilizar objetos. Revisemos rápidamente el código que vimos al principio del capítulo. En este punto, debes comprender completamente lo que está sucediendo.

https://trinket.io/embed/python3/ee59ee4759

La primera línea construye un objeto `list`. Cuando Python crea el objeto `list`, llama al método **constructor** (denominado `__init__`) para configurar los atributos de datos internos que se utilizarán para almacenar los datos de la lista. Debido a la **encapsulación** no necesitamos saber, ni tenemos que preocuparnos por esto, ya que los atributos de datos internos están ordenados.

No estamos pasando ningún parámetro al **constructor** y cuando el constructor regresa, usamos la variable `stuff` para apuntar a la instancia devuelta de la clase `list`.

Las líneas segunda y tercera están llamando al método `append` con un parámetro para agregar un nuevo elemento al final de la lista actualizando los atributos dentro de `stuff`. Luego, en la cuarta línea, llamamos al método `sort` sin parámetros para ordenar los datos dentro del objeto `stuff`.

Luego imprimimos el primer elemento de la lista usando los corchetes, que son un atajo para llamar al método `__getitem__` dentro del objeto `stuff`. Y esto es equivalente a llamar al método `__getitem__` en la clase `list` pasando el objeto `stuff` como primer parámetro y la posición que estamos buscando como segundo parámetro.

Al final del programa, el objeto `stuff` se descarta, pero no antes de llamar al **destructor** (llamado `__del__`) para que el objeto pueda limpiar los cabos sueltos según sea necesario.

Esos son los fundamentos y la terminología de la programación orientada a objetos. Hay muchos detalles adicionales sobre cómo usar mejor los enfoques orientados a objetos al desarrollar aplicaciones y bibliotecas grandes que están fuera del alcance de este capítulo.[3]

> [1]. https://docs.python.org/3/library/html.parser.html ↩

> [2]. Cookie image copyright CC-BY https://www.flickr.com/photos/dinnerseries/23570475099 ↩

> [3]. Si tienes curiosidad por saber dónde está definida la clase de la lista, echa un vistazo (esperemos que la URL no cambie) https://github.com/python/cpython/blob/master/Objects/listobject.c - La lista de clases está escrita en un lenguaje llamado "C". Si echas un vistazo a ese código fuente y te resulta curioso, quizás quieras explorar algunos cursos de informática. ↩

Revision #1
Created 5 April 2025 12:07:54 by Javier Quintana
Updated 5 April 2025 12:11:17 by Javier Quintana