Tarea de Investigación

Programación Orientada a Objetos

Integrantes:

Juan Pablo Munizaga

Nicolas Ambas

Sobrecarga de operadores en Python y manejo de listas

Uso de sobrecarga de operadores para aumentar la legibilidad del código y manejo de listas en Python

La sobrecarga de operadores en Python (Overloading). Se ilustra cómo se pueden realizar diferentes operaciones utilizando la sobrecarga de operadores a través de un ejemplo práctico.

Además, se hace una descripción detallada de las listas en Python, uno de los tipos de datos más versátiles y utilizados en este lenguaje de programación. Se abordan las diferentes formas de crear listas, sus propiedades y cómo acceder y modificar sus elementos.

Palabras claves:

Legibilidad: facilidad para leer y comprender el código.

Clase: plantilla que define un objeto, la cual contiene atributos (variables) y métodos (funciones).

Objeto: entidad que tiene un estado y un comportamiento definidos por una clase.

Método: función que pertenece a una clase.

Vector: objeto matemático que tiene magnitud y dirección.

Escalar: cantidad que tiene solo magnitud, en contraposición a un vector.

Tabla: lista que muestra las funciones nativas que se pueden sobrecargar en Python.

Listas: estructuras de datos que permiten almacenar un conjunto arbitrario de datos. Son ordenadas, pueden ser formadas por tipos arbitrarios, se pueden anidar y son mutables y dinámicas.

Mutable: capacidad de un objeto para cambiar su estado después de ser creado.

Dinámico: capacidad de un objeto para cambiar su tamaño después de ser creado.

Iterable: objeto que se puede recorrer o atravesar.

Índice: número que identifica la posición de un elemento dentro de una lista.

Acceder: obtener el valor de un elemento de una lista.

Anidar: colocar una lista dentro de otra.

Sobrecargas de operadores en Python

La sobrecarga de operadores (Overloading) es una herramienta que nos ofrece alguno de los lenguajes de programación orientados a objetos en este caso (Python), esto nos permite poder crear código con una legibilidad mayor ya que usamos los operadores nativos del lenguaje para hacer comparaciones y operaciones entre objetos que nosotros hemos creado además poder escribir menos código para desarrollar un problema y de forma muy sencilla ya que para sobrecargar un operador lo hacemos de forma análoga a la que normalmente se crea un método de una clase.

NOTA: *Todo en Python es un objeto y esto nos permite utilizar la sobrecarga de operadores en innumerables problemas a resolver esto hace que sea muy empleado y útil de aprender.*

Operaciones con Sobrecarga de Operadores:

```
class Vector(object):
    def __init__ (self, x, y):
        self.x = x
        self.y = y

    def __add__ (self, v):
        # Sumar dos Vectores
        return Vector (self.x + v.x, self.y + v.y)

    def __sub__ (self, v):
        # Restar dos Vectores
        return Vector (self.x - v.x, self.y - v.y)

    def __mul__ (self, s):
        # Multiplicar un Vectores por un escalar
        return Vector (self.x * s, self.y * s)
```

Dado que en Python hay varias funciones nativas también se pueden sobrecargar y operar con instancias de objetos, a continuación, se muestra una tabla con las funciones que se pueden sobrecargar.

Function	Method	Expression
Casting to `int`	`__int__(self)`	`int(a1)`
Absolute function	`__abs__(self)`	`abs(a1)`
Casting to `str`	`__str__(self)`	`str(a1)`
Casting to `unicode`	`__unicode__(self)`	`unicode(a1)` (Python 2 only)
String representation	`__repr__(self)`	`repr(a1)`
Casting to `bool`	`__nonzero__(self)`	`bool(a1)`
String formatting	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hashing	`__hash__(self)`	`hash(a1)`
Length	`__len__(self)`	`len(a1)`
Reversed	`__reversed__(self)`	`reversed(a1)`
Floor	`__floor__(self)`	`math.floor(a1)`
Ceiling	`__ceil__(self)`	`math.ceil(a1)`

Listas en Python

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo.
Son mutables y dinámicas, lo cual es la principal diferencia con los sets y las tuplas (arreglos).

Crear listas Python

Las listas en Python son uno de los tipos o estructuras de datos más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea

```
lista = [1, 2, 3, 4]
```

También se puede crear usando list y pasando un objeto iterable.

```
lista = list("1234")
```

Una lista sea crea con [] separando sus elementos con comas. Una gran ventaja es que pueden almacenar tipos de datos distintos.

```
lista = [1, "Hola", 3.67, [1, 2, 3]]
```

Algunas propiedades de las listas:

- Son ordenadas, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos arbitrarios
- Pueden ser indexadas con [i].
- Se pueden anidar, es decir, meter una dentro de la otra.
- Son mutables, ya que sus elementos pueden ser modificados.
- Son dinámicas, ya que se pueden añadir o eliminar elementos.
- Acceder y modificar listas

Si tenemos una lista a con 3 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde 0 a n-1 siendo n el tamaño de la lista.

```
a = [90, "Python", 3.87]
print(a[0]) #90
print(a[1]) #Python
print(a[2]) #3.87
```

Se puede también acceder al último elemento usando el índice [-1].

```
a = [90, "Python", 3.87]
print(a[-1]) #3.87
```

De la misma manera, al igual que [-1] es el último elemento, podemos acceder a [-2] que será el penúltimo.

```
print(a[-2]) #Python
```

Y si queremos modificar un elemento de la lista, basta con asignar con el operador = el nuevo valor.

```
a[2] = 1
print(a) #[90, 'Python', 1]
```

También podemos tener listas anidadas, es decir, una lista dentro de otra. Incluso podemos tener una lista dentro de otra lista y a su vez dentro de otra lista. Para acceder a sus elementos sólo tenemos que usar [] tantas veces como niveles de anidado tengamos.

```
x = [1, 2, 3, ['p', 'q', [5, 6, 7]]]
print(x[3][0])    #p
print(x[3][2][0]) #5
print(x[3][2][2]) #7
```

También es posible crear sublistas más pequeñas de una más grande. Para ello debemos de usar: entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el valor final que no está incluido. Por lo tanto [0:2] creará una lista con los elementos [0] y [1] de la original.

```
l = [1, 2, 3, 4, 5, 6]
print(l[0:2]) #[1, 2]
print(l[2:6]) #[3, 4, 5, 6]
```

Y de la misma manera podemos modificar múltiples valores de la lista a la vez usando:

```
l = [1, 2, 3, 4, 5, 6]
l[0:3] = [0, 0, 0]
print(l) #[0, 0, 0, 4, 5,
```

Hay ciertos operadores como el + que pueden ser usados sobre las listas.

```
l = [1, 2, 3]
l += [4, 5]
print(l) #[1, 2, 3, 4, 5]
```

Y una funcionalidad muy interesante es que se puede asignar una lista con n elementos a n variables.

```
l = [1, 2, 3]
x, y, z = l
print(x, y, z) #1 2 3
```

Iterar listas

En Python es muy fácil iterar una lista, mucho más que en otros lenguajes de programación.

```
lista = [5, 9, 10]
for l in lista:
    print(l)#5 #9 #10
```

Si necesitamos un índice acompañado con la lista, que tome valores desde 0 hasta n-1, se puede hacer de la siguiente manera.

```
lista = [5, 9, 10]
for index, l in
enumerate(lista):
    print(index, l)
#0 5
#1 9
#2 10
```

O si tenemos dos listas y las queremos iterar a la vez, también es posible hacerlo.

```
lista1 = [5, 9, 10]
lista2 = ["Jazz", "Rock",
"Djent"]
for l1, l2 in zip(lista1,
lista2):
    print(l1, l2)
#5 Jazz
#9 Rock
#10 Djent
```

Y por supuesto, también se pueden iterar las listas usando los índices como hemos visto al principio, y haciendo uso de len(), que nos devuelve la longitud de la lista.

```
lista1 = [5, 9, 10]
for i in range(0,
len(lista)):
    print(lista1[i])
#5
#9
#10
```

Métodos listas

El método append() añade un elemento al final de la lista.

```
l = [1, 2]
l.append(3)
print(l) #[1, 2, 3]
```

El método extend() permite añadir una lista a la lista inicial.

```
l = [1, 2]
l.extend([3, 4])
print(l) #[1, 2, 3, 4]
```

El método insert() añade un elemento en una posición o índice determinado.

```
l = [1, 3]
l.insert(1, 2)
print(l) #[1, 2, 3]
```

El método remove() recibe como argumento un objeto y lo borra de la lista.

```
l = [1, 2, 3]
l.remove(3)
print(l) #[1,2]
```

El método pop() elimina por defecto el último elemento de la lista, pero si se pasa como parámetro un índice permite borrar elementos diferentes al último.

```
l = [1, 2, 3]
l.pop()
print(l) #[1, 2]
```

El método reverse() inverte el órden de la lista.

```
l = [1, 2, 3]
l.reverse()
print(l) #[3, 2, 1]
```

El método sort() ordena los elementos de menos a mayor por defecto.

```
l = [3, 1, 2]
l.sort()
print(l) #[1, 2, 3]
```

Y también permite ordenar de mayor a menor si se pasa como parámetro reverse=True.

```
l = [3, 1, 2]
l.sort(reverse=True)
print(l) #[3, 2, 1]
```

El método index() recibe como parámetro un objeto y devuelve el índice de su primera aparición. Como hemos visto en otras ocasiones, el índice del primer elemento es el 0.

```
l = ["Periphery", "Intervals", "Monuments"]
print(l.index("Intervals"))
```

También permite introducir un parámetro opcional que representa el índice desde el que comenzar la búsqueda del objeto. Es como si ignorara todo lo que hay antes de ese índice para la búsqueda, en este caso el 4.

```
l = [1, 1, 1, 1, 2, 1, 4, 5]
print(l.index(1, 4)) #5
```

En resumen, el documento presenta dos temas importantes en Python: la sobrecarga de operadores y las listas. La sobrecarga de operadores es una herramienta que nos permite utilizar operadores nativos del lenguaje para hacer comparaciones y operaciones entre objetos que hemos creado, lo que facilita la legibilidad del código y la resolución de problemas. Por otro lado, las listas en Python son una estructura de datos versátil que nos permite almacenar cualquier tipo de datos y realizar operaciones como añadir o eliminar elementos y modificar valores. En general, ambos temas son fundamentales en Python y su conocimiento puede ser muy útil en la programación.

Bibliografía:

https://medium.com/@LuisMBaezCo/overloading-sobrecargar-operadores-en-python-5d7a75e2bfdf

https://ellibrodepython.com/listas-en-python