



**UTN.BA**  
UNIVERSIDAD TECNOLÓGICA NACIONAL  
FACULTAD REGIONAL BUENOS AIRES

**Centro de  
e-Learning**

**UNIDAD DIDÁCTICA II**

# **DIPLOMATURA EN PYTHON**

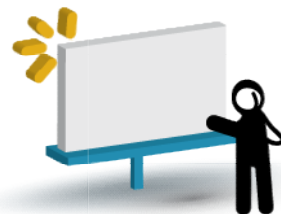
**Centro de e-Learning SCEU UTN - BA.**

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

**[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)**

## **Módulo V – Nivel Avanzado**

### **Unidad II – Delegación – Sobrecarga de operadores y nuevo estilo.**



## Presentación:

En el transcurso de esta unidad trabajaremos con la sobrecarga de operadores, es decir que crearemos clases que posean un comportamiento personalizado para los operadores que vienen por defecto en el núcleo de la distribución de python que nos hemos descargado. También nos comenzaremos a introducir en la arquitectura del nuevo estilo de programación que se establece a partir de la rama 3.x de python.



## Objetivos:

### Que los participantes:

Comiencen a trabajar con sobrecarga de operadores.

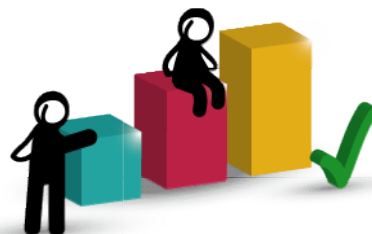
Se introduzcan en temas avanzados e la POO en python 3.

Analicen las diferencias entre el sistema de herencia de clases de python 2 y python 3.



## Bloques temáticos:

- 1.- Delegación.
- 2.- Namespaces.
- 3.- Sobrecarga de operadores.
- 4.- Introducción a los métodos especiales.
- 5.- Estructura de herencia en Python.



## Consignas para el aprendizaje colaborativo

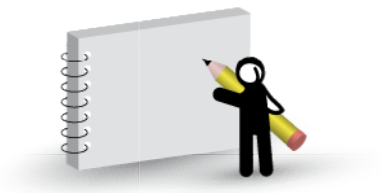
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC\*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



## Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.



## 1.- Delegación

Una técnica muy utilizada en la POO es la de delegación de métodos en la cual se le permite a cada clase hija que establezca la lógica según la cual se implementa una determinada acción. Supongamos que estamos creando una clase para describir personas, y que uno de los métodos de la clase personas es “comerArroz()”, en principio como no sabemos cuántas formas pueden existir en el mundo de comer el arroz, podríamos delegar en la clase hija la ejecución de una rutina (método de instancia) que sea llamado cuando invocamos el método de la clase padre “comerArroz” y que sea creada cuando tengamos la información necesaria de cómo se come el arroz en una cultura determinada. Veamos el ejemplo en acción:

### delegacion.py

```
1 class Personas:
2
3     def comerArroz(self):
4         self.accion()
5
6 class Chinos(Personas):
7     def accion(self):
8         print('Los chinos comen arroz con palillos')
9
10 x = Chinos()
11 x.comerArroz()
```

Retorna:

Los chinos comen arroz con palillos

La acción puede simplemente contener un pass, o un print con algún mensaje hasta que terminemos de implementar la lógica.





## 2.- Namespaces

Cuando trabajamos con módulos y paquetes, vimos que python le asigna un namespaces al módulo cuando lo recuperamos con "import" de forma de evitar colisiones. También vimos que los namespaces son guardados dentro de diccionarios, por lo que podemos acceder a los namespaces de módulo mediante el uso de `__dict__`.

Al crear una clase python trabaja de forma análoga ya que cada instancia de clase es considerada por python como un nuevo namespaces.

Los atributos de una clase son claves de diccionarios en Python, al crear una instancia se crea un diccionario que puede guardar información, e ir variando con el tiempo. Veamos esto con un ejemplo en el cual tenemos una clase "ClaseHija()" que hereda de la clase "ClasePadre" y en ambas clases se encuentra definido el método "Método1()" y el atributo 1. Al crear una instancia y aplicarle `__dict__()` nos retorna un diccionario vacío ya que no hemos pasado ningún parámetro en la clase durante la instancia. Como podemos ver la instancia es considerada como un diccionario.

---

### namespaces.py

```
1 class ClasePadre:
2     atributo1 = "rojo"
3     def Metodo1(self):
4         pass
5
6 class ClaseHija(ClasePadre):
7
8     atributo1 = "azul"
9     def Metodo1(self):
10         pass
11
12 X = ClaseHija()
13 print(X.__dict__)           #Diccionario vacío correspondiente al namespace
14 print(X.__class__)
15 print(ClaseHija.__bases__)
16 print(ClasePadre.__bases__)
17 print(list(ClaseHija.__dict__.keys()))
18 print(list(ClasePadre.__dict__.keys()))
```

---



Retorna:

```
{  
<class '__main__.ClaseHija'  
(<class '__main__.ClasePadre'>,)  
(<class 'object'>,)  
['__module__', 'atributo1', 'Metodo1', '__doc__']  
['__module__', 'atributo1', 'Metodo1', '__dict__', '__weakref__', '__doc__']
```

Cómo podemos ver la instancia es considerada como un diccionario y mediante `__dict__`, `__class__`, `__bases__` y `__dict__.keys` hemos recuperado información del diccionario. También podríamos utilizar `__dict__.values`.

### 3.- Sobrecarga de operadores.

El término sobrecarga de operadores se refiere a interceptar las operaciones incorporadas en los métodos de una clase. Python invoca automáticamente sus métodos cuando aparecen instancias de la clase en las operaciones integradas (built-in) en su núcleo, y el valor de retorno de su método se convierte en el resultado de la operación correspondiente.

Las clases pueden interceptar a los operadores de Python, si recordamos de unidades anteriores cuando definimos el método especial `__str__()` dentro de una clase lográbamos retornar un valor específico al imprimir la instancia de una clase, por ejemplo si definimos la clase “MiClase” como se muestra a continuación, creamos una instancia de la misma y la imprimimos, obtenemos datos sobre el objeto y su ubicación en memoria:

---

#### operadores1.py

```
1 class MiClase():  
2  
3     def __init__(self, nombre):  
4         self.nombre = nombre  
5  
6 objeto1 = MiClase('Juan')  
7 print (objeto1)
```

Retorna:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148  
[www.sceu.frba.utn.edu.ar/e-learning](http://www.sceu.frba.utn.edu.ar/e-learning)



```
< __main__.MiClase object at 0x03A4E830>
```

Mientras que si agregamos el método `__str__()`, al imprimir el objeto obtenemos el valor del atributo de instancia.

#### operadores2.py

```
1 class MiClase():
2
3     def __init__(self, nombre):
4         self.nombre = nombre
5
6     def __str__(self):
7         return self.nombre
8
9 objeto1 = MiClase('Juan')
10 print (objeto1)
```

Retorna:

```
Juan
```

Los métodos nombrados con subrayados dobles (`__X__`) son métodos reservados en python y son utilizados para interceptar operaciones específicas.

Dichos métodos se llaman automáticamente cuando aparecen instancias en operaciones integradas. Por ejemplo, si un objeto de instancia hereda un método `__add__`, ese método se llama cada vez que el objeto aparece en una expresión `+`. El valor de retorno del método se convierte en el resultado de la expresión correspondiente.

Las clases pueden anular la mayoría de las operaciones de tipo integradas. Hay docenas de nombres de métodos de sobrecarga de operadores especiales para interceptar e implementar casi todas las operaciones disponibles para los tipos incorporados. Esto incluye expresiones, pero también operaciones básicas como impresión y creación de objetos. No hay valores predeterminados para los métodos de sobrecarga del operador, y no se requiere ninguno. Si una clase no define o hereda un método de sobrecarga de operadores, solo significa que la operación correspondiente no es compatible con las instancias de la clase. Si no hay `__add__`, por ejemplo, las expresiones `+` generan excepciones.



## 4.- Introducción a los métodos especiales.

En resumen, una clase puede implementar ciertas operaciones que son invocadas por una sintaxis especial (como operaciones aritméticas o subíndices y segmentación) definiendo métodos con nombres especiales. Este es el enfoque de Python para la sobrecarga de operadores, lo que permite a las clases definir su propio comportamiento con respecto a los operadores.

### \_\_getitem\_\_ ():

Si hemos agregado este método a nuestra clase, el mismo es llamado automáticamente cuando a una instancia se le asigna un índice entre corchetes. El método `__getitem__` toma como primer parámetro a la instancia y como segundo parámetro el índice que estamos pasando.

#### getitem.py

```
1 class Indexador:
2     def __getitem__(self, indice):
3         return indice ** 0.5
4
5 X = Indexador()
6 print(X[64])
```

Retorna:

8.0

Además podemos al igual que como en su momento realizamos con las listas y los strings seleccionar un rango de elementos utilizando la misma notación que ya hemos visto al inicio del curso.

#### getitem2.py

```
1 class Indexador:
2     texto = "El día está lindo"
3     def __getitem__(self, indice):
4         return self.texto[indice]
5
```



```
6 X = Indexador()
7 print(X[3:8])
```

Retorna:

```
día e
```

### \_\_setitem\_\_():

Es llamado automáticamente cuando una instancia aparece en una expresión con índice en donde se asigna valor. El primer parámetro que toma es la instancia, el segundo el índice y el tercero el valor asignado.

#### getitem\_setitem.py

```
1 class Indexador:
2     lista = ['M','a','n','z','a','n','a']
3     def __getitem__(self, indice):
4         return self.lista[indice]
5
6     def __setitem__(self, indice, valor):
7         self.lista[indice] = valor
8
9 X = Indexador()
10 X[6] = 'o'
11 print(X.lista)
```

Retorna:

```
['M', 'a', 'n', 'z', 'a', 'n', 'o']
```

**Nota:** Recordar que no podemos asignarle un valor a una posición de un elemento inmutable como los strings ya que nos retornaría error.

### \_\_iter\_\_(), \_\_next\_\_():

En el caso en que nuestra clase realice una iteración, en lugar de `__getitem__()` es preferible utilizar `__iter__()`, este método puede ser utilizado junto con `__next__()` para ir recorriendo la iteración de a pasos. Para ver su funcionamiento, crearemos una clase que tome los valores de una lista, la cual debe contener valores positivos, los ordene y nos



retorne la raíz cuadrada de los valores ingresados. Dentro del método `__iter__()` se ha implementado un algoritmo de ordenamiento de vectores.

Hemos creado una clase adicional para que contenga el método `__next__()`, esta clase es instanciada desde `__iter__()` pasándole como parámetro la lista ya ordenada. Dentro del método `__next__()` hemos puesto una estructura de control "if" de forma de el índice con el cual recorremos la lista se encuentre acotado por la cantidad de elementos.

#### `iter_next.py`

```
1 class RaizCuadrada:
2     def __init__(self, A):
3
4         self.A = A
5         self.n = len(self.A)
6     def __iter__(self):
7         #bucle for para ordenar
8         for i in range(self.n-1):
9             for j in range(self.n-1):
10                if(self.A[j]>self.A[j+1]):
11                    aux=self.A[j]
12                    self.A[j]=self.A[j+1]
13                    self.A[j+1]=aux
14        return RecorrerIteracion(self.A)
15
16
17 class RecorrerIteracion:
18     def __init__(self, A):
19         self.A = A
20         self.longitud = len(self.A)-1
21         self.i = -1
22
23     def __next__(self):
24
25         if self.i == self.longitud :
26
27             raise StopIteration
28         self.i += 1
29         return self.A[self.i] ** 0.5
30
31 A=[81, 16, 64, 9]
32 x = RaizCuadrada(A)
33 p = iter(x)
```



```
34 print(next(p), next(p),next(p))
35 for i in RaizCuadrada(A):
36     print(i, end=' ')
```

Retorna:

```
3.0 4.0 8.0
3.0 4.0 8.0 9.0
```

Como podemos ver no solo podemos recorrer una instancia mediante un bucle for, sino que mediante next() podemos avanzar de a un paso en la iteración. Estos métodos también pueden ser aplicados a objetos iterables sin la necesidad de definirlos dentro de una clase como se muestra a continuación:

---

#### iter\_nextlista.py

```
1 A=[81, 16, 64, 9]
2
3 iterador = iter(A)
4 try:
5     while True:
6         print(iterador.__next__())
7
8 except StopIteration:
9     print("Hemos llegado al final de la lista.")
```

Retorna:

```
81
16
64
9
Hemos llegado al final de la lista.
```



### iter () yield:

En lugar de guardar la información en la instancia de la clase, podemos utilizar yield para guardar la información en una variable local y obtener el mismo efecto que al implementar `__next__()`.

#### iteryield.py

```
1 class RaizCuadrada:
2     def __init__(self, A):
3
4         self.A = A
5         self.n = len(self.A)
6         self.longitud = len(self.A)
7
8
9     def __iter__(self):
10        #bucle for pra ordenar
11        for i in range(self.n-1):
12            for j in range(self.n-1):
13                if(self.A[j]>self.A[j+1]):
14                    aux=self.A[j]
15                    self.A[j]=self.A[j+1]
16                    self.A[j+1]=aux
17
18        for valor in range(0 , self.longitud):
19            yield self.A[valor] ** 0.5
20
21 A=[81, 16, 64, 9]
22 x = RaizCuadrada(A)
23 p = iter(x)
24 print(next(p), next(p),next(p))
25 for i in RaizCuadrada(A):
26     print(i, end=' ')
```

Retorna:

```
3.0 4.0 8.0
3.0 4.0 8.0 9.0
```





## 5.- Estructura de herencia en Python.

Con el cambio de versión de python 2.x a python3 se ha introducido una modificación sustancial en el sistema de herencias de clases. Como hemos visto en este curso python utiliza un sistema de herencia de diamante el cual es conocido como “Nuevo Estilo” o “New Style”.

### Caso 1 Cambios en la clase.

En el nuevo estilo, para poder utilizar los métodos del built-in como: `__getitem__`, `__str__`, `__add__`, etc sobre una instancia debo declararla dentro de la clase como un método, esto no hacía falta en la versión 2.x en donde se podía utilizar directamente el método del built-in.

### Caso 2 Cambios en la clase (type y class)

En el nuevo estilo se podría decir que la distinción entre type y class ha cambiado grandemente o se ha desvanecido del todo.

#### Las clases son types

Los objetos del tipo type generan clases como sus instancias, y las clases generan instancias de sí mismas. Ambas son consideradas types, ya que generan instancias. De hecho no existe diferencia entre los tipos de objetos integrados en el núcleo (built-in types) como las listas o strings, con las clases definidas por los usuarios. Por este motivo podemos crear subclases de los built-in types. Una subclase de built-in type como list califica como una clase del nuevo estilo y se convierte en un nuevo type.

#### Los types son clases.

La generación de nuevas clases del tipo type son metaclasses. El usuario puede definir subclases de type e instancias que a su vez son clases. Las metaclasses son a la vez clases y type.

La diferencia principal entre las clases antiguas y las de nuevo estilo consiste en que a la hora de crear una nueva clase anteriormente no se definía realmente un nuevo tipo, sino



que todos los objetos creados a partir de clases, fueran estas las clases que fueran, eran de tipo instance.

En Python 2.x las instancias de las clases clásicas son del tipo "instance" pero los tipos de objetos del built-in son más específicas.

Comencemos por ver un ejemplo en la generación de un objeto de clase.

2.x	2.x con object	3.x
<code>class C(): pass</code>	<code>class C(): pass</code>	<code>class C(): pass</code>
<code>I = C()</code>	<code>I = C()</code>	<code>I = C()</code>
<code>print(type(I))</code>	<code>print(type(I))</code>	<code>print(type(I))</code>
<code>&lt;type 'instance'&gt;</code>	<code>&lt;class ' __main__ .C'&gt;</code>	<code>&lt;class ' __main__ .C'&gt;</code>
<code>print(I.__class__)</code>	<code>print(I.__class__)</code>	<code>print(I.__class__)</code>
<code>__main__ .C</code>	<code>&lt;class ' __main__ .C'&gt;</code>	<code>&lt;class ' __main__ .C'&gt;</code>
<code>print(type(C))</code>	<code>print(type(C))</code>	<code>print(type(C))</code>
<code>&lt;type 'classobj'&gt;</code>	<code>&lt;type 'type'&gt;</code>	<code>&lt; class 'type'&gt;</code>
<code>print(C.__class__)</code>	<code>print(C.__class__)</code>	<code>print(C.__class__)</code>
<code>AttributeError: class C has no attribute ' __class__ '</code>	<code>&lt;type 'type'&gt;</code>	<code>&lt; class 'type'&gt;</code>

En el Nuevo estilo, el tipo de una instancia, es el mismo que la instancia de la cual se origina, y el tipo de la clase es el mismo que el de los objetos del tipo built-in. En el nuevo estilo la clase posee un atributo del tipo `__class__` ya que la clase es una instancia del tipo `type`.

Lo anterior también se cumple para toda clase en python 3.x ya que todas las clases son consideradas automáticamente pertenecientes al nuevo estilo incluso si no poseen una superclase definida explícitamente.

Como podemos ver al ejecutar el código en python 3.x `class` es `type` y `type` es `class`

2.x	2.x con object	3.x
<code>print(type([1,2,3]), [1,2,3].__class__)</code>		
<code>&lt;type 'list'&gt;</code>	<code>&lt;type 'list'&gt;</code>	<code>&lt;class 'list'&gt;</code>
<code>print(type(list), list.__class__)</code>		
<code>&lt;type 'type'&gt;</code>	<code>&lt;type 'type'&gt;</code>	<code>&lt;class 'type'&gt;</code>



**Nota:** En el nuevo estilo, el type de una instancia da la clase de la cual ha sido creada, mientras que en el estilo clásico da como resultado 'Instance' con lo cual al comparar en el estilo clásico el type de dos instancias da lo mismo.

### Type es instancia de object y object de type

---

**isinstance(type, object)**

**True**

**isinstance(object, type)**

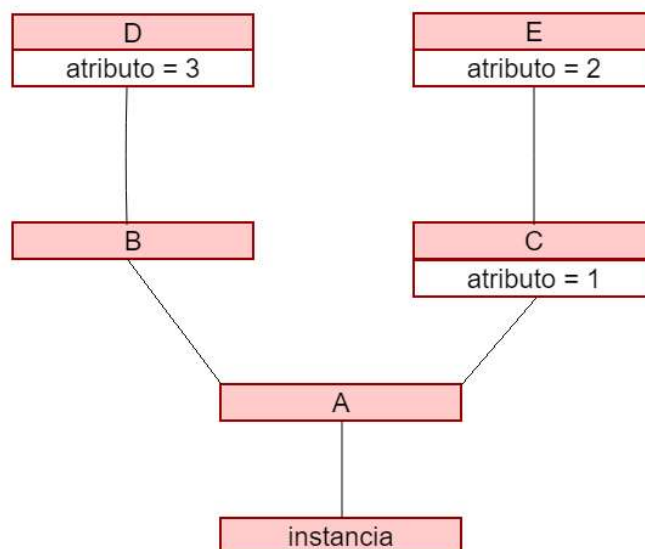
**True**

---

Esto implica que en el nuevo estilo, las clases heredan los atributos de object mientras que en el estilo clásico no.

## DFLR (Python 2.x) vs MRO (Estructura de Diamante – Python 3.x)

Podemos ver una comparación entre las estructuras de herencia de los dos estilos, tomemos referencia la siguiente estructura.





### diamante-dflr.py

```
1 class D: atributo = 3
2 class B(D): pass
3 class E: atributo = 2
4 class C(E): atributo = 1
5 class A(B, C): pass
6 instancia = A()
7 print(instancia.atributo)
```

La comparativa de retorno de resultado nos muestra que:

	DFLR	MRO
Retorno.	3	3
Orden de búsqueda.	Instancia,A,B,D,C,E	Instancia,A,B,D,C,E,Object
Forma de encontrar la herencia.	'---'	print(A.__mro__)

## Nuevo estilo – extensiones.

El uso de `__slots__` permite restringir el la cantidad de atributos de instancia de una clase.

### slots

Para utilizar `__slots__` asignamos una secuencia de nombres strings a la variable `__slots__` al inicio de la declaración de la clase. Únicamente estos nombres pueden ser asignados como atributos de instancia. Los nombres deben ser asignados antes de ser referenciados.

El uso de slot permite ahorrar velocidad en la ejecución ya que python en lugar de crear un diccionario para guardar todos los posibles valores de atributos, limita el lugar de memoria reservado para los atributos a un valor por cada atributo registrado en slot.



### slot1.py

```
1 class Limites(object):
2     __slots__ = ['edad', 'sexo', 'trabajo', 'salario']
3     pass
4
5 x = Limites()
6 x.edad = 4
7 print(x.edad)
8
10 print('-----')
11 x.peso = 40
12 print(x.peso)
```

Retorna:

```
4
-----
Traceback (most recent call last):
  File "C:\Users\juanb\Documents\000-TRABAJOS-2018\000-MEDRANO-2019\004-
Python-Diplomatura\Modulo-5-y-6\Unidad-2a-Manipulación de propiedades\slot1.py", line
10, in <module>
    x.peso = 40
AttributeError: 'Limites' object has no attribute 'peso'
```

Dado que la información no se guarda en un diccionario, no puedo usar `__dict__` para recuperar los atributos, sin embargo aún los puedo recuperar y modificar usando `getattr()` y `setattr()`:

### slot2.py

```
1 class Limites(object):
2     __slot__ = ['edad', 'sexo', 'trabajo', 'salario']
3
4     def imprimir(self):
5         print(self.edad , 'ddd')
6
7 x = Limites()
8 x.edad = 4
10 print(x.edad)
11 print(x.imprimir())
12
13 setattr(x, 'sexo', 'masculino')
```



```
14 print('-----')
15 print(getattr(x, 'edad'))
16 print(getattr(x, 'sexo'))
17
18 print('-----')
19 x.peso = 40
```

Retorna:

```
4
4 ddd
None
-----
4
masculino
-----
```

Si necesitamos recuperar los valores en forma de diccionario o extender la cantidad de atributos, aún lo podemos realizar de forma explícita:

### slot3.py

```
1 class Limites:
2     __slots__ = ['edad', 'sexo', 'trabajo', 'salario', '__dict__']
3
4     def __init__(self):
5         self.d = 4
6
7 x = Limites()
8 print(x.d)
10 x.edad = 4
11 print(x.__dict__)
12 print(x.__slots__)
```

Retorna:

```
4
{'d': 4}
['edad', 'sexo', 'trabajo', 'salario', '__dict__']masculino
-----
```



## Static y métodos de clase.

A partir de python 2.2 es posible definir dos tipos de métodos dentro de una clase que pueden ser invocados sin que exista una instancia, estos son los métodos de clase y los estáticos. En algunas ocasiones necesitamos saber información sobre las clases pero no sobre las instancias, en estos casos es útil guardar esta info dentro de las clases (un ejemplo podría ser saber cuántas instancias se han creado de la clase).

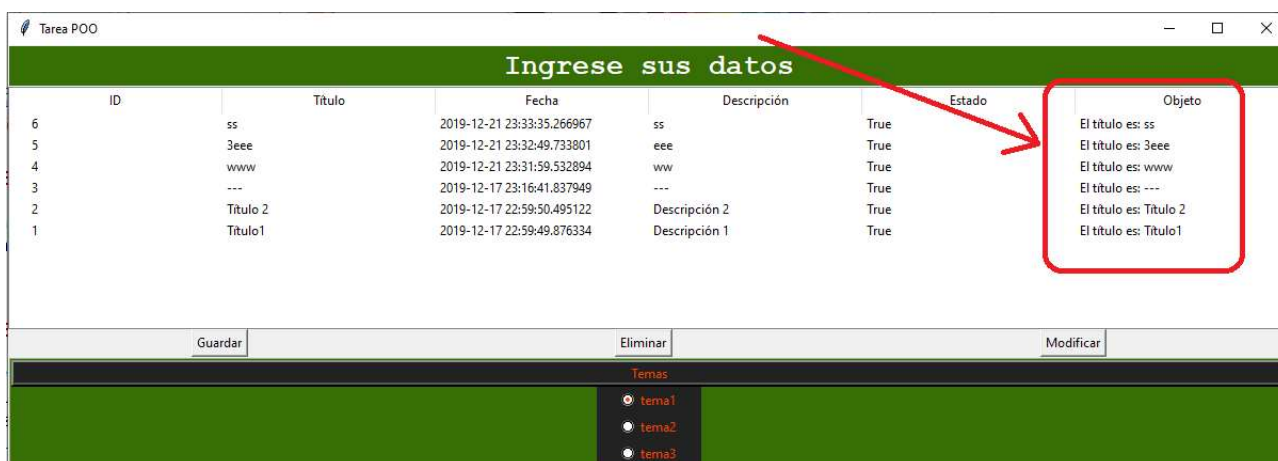
Python realiza esto con la idea de métodos estáticos, las cuales son funciones simples que no implementan el uso de self y están diseñados para trabajar con atributos de clase en lugar de atributos de instancia. Los métodos estáticos NUNCA reciben un argumento "self".

De forma menos habitual, python también implementa métodos de clase al cual se le pasa un objeto de clase, en lugar de uno de instancia y pueden ser invocados tanto por la clase o por una instancia.

## Tareas de la Unidad 2

### Tarea 1 - Obligatoria

Implemente el método `__str__()`, analizado en el tema “Sobrecarga de operadores”, dentro de la clase “Noticia”, la cual es mapeada por el ORM para agregar un texto personalizado en una nueva columna de la ventana principal “treeView” que aparezca cada vez que se hace referencia al objeto (esto en nuestro caso pasa en cada vuelta del bucle for)



ID	Titulo	Fecha	Descripción	Estado	Objeto
6	ss	2019-12-21 23:33:35.266967	ss	True	El título es: ss
5	3eee	2019-12-21 23:32:49.733801	eee	True	El título es: 3eee
4	www	2019-12-21 23:31:59.532894	ww	True	El título es: www
3	---	2019-12-17 23:16:41.837949	---	True	El título es: ---
2	Titulo 2	2019-12-17 22:59:50.495122	Descripción 2	True	El título es: Titulo 2
1	Titulo1	2019-12-17 22:59:49.876334	Descripción 1	True	El título es: Titulo1

Guardar Eliminar Modificar

Temas

☒ tema1 ☐ tema2 ☐ tema3

### Tarea 2 - Opcional

Realice un programa que permita calcular de forma aproximada cuanto más rápido es usar o no slots.

**Nota:** El valor puede variar entre sistemas operativos y entre una ejecución a otra dependiendo de los procesos que el sistema operativo este realizando.

La resolución de este problema estará a disposición de los alumnos una vez transcurrida una semana.

**Nota 2:** En Windows 10 he medido una diferencia de entre el 12% y el 19%. En la medición he utilizado el módulo “timeit”





## Bibliografía utilizada y sugerida

### Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

### Manual online

<https://docs.python.org/3.7/tutorial/>

<https://docs.python.org/3.7/library/index.html>

<https://docs.python.org/3/reference/datamodel.html>



## Lo que vimos

En esta unidad hemos trabajado sobre la delegación en clases, sobrecarga de operadores y algunos puntos importantes del nuevo estilo de programación.

---



## Lo que viene:

En la siguiente unidad comenzaremos a trabajar sobre la manipulación de atributos.