UTN.BA UNIVERSIDAD TECNOLÓGICA NACIONAL FACULTAD REGIONAL BUENOS AIRES | Centro de e-Learning Secretaría de Cultura y Extensión Universitaria

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Diplomatura en Python:

Curso de Python nivel avanzado.

UTN.BA | Centro de e-Learning
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Secretaría de Cultura y Extensión Universitaria

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 2:

Nuevas estructuras

Unidad 4:

Metaclases

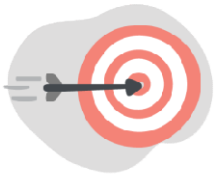
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Presentación

Cuando estudiamos decoradores, vimos que estos, nos permiten interceptar funciones y clases y extender sus funcionalidades, de forma similar, las metaclases nos permiten interceptar la creación de clases permitiéndonos a su vez un grado de control muy alto en el diseño de la arquitectura de nuestras aplicaciones y facilitándonos la comprensión de cómo funciona en el fondo el modelo de clases de python.

El desarrollo con metaclases se hace más común con el paso del tiempo, por lo que comprender cuando y como se utilizan es una herramienta muy útil en la actualidad.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Objetivos

Que los participantes logren…

Comprender la estructura de una metaclase.

Aprender en qué momento se ejecutan.

Conocer la relación que guardan con la clase type.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bloques temáticos

1.- Metaclases.

2.- En python3, las metaclases son subclases de type.

3.- Metaclases y herencia.

4.- Atributos en Superclases y Metaclases.

5.- Herencia de metaclases.

6.- Métodos de metaclases.

7.- Sobrecarga de operadores en metaclases.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

1.- Metaclases.

Las metaclases son otra forma de ejecutar código de forma automática, en cierto sentido las podemos considerar como una forma de extender el modelo de inserción de código de los decoradores. Nosotros hemos utilizado los decoradores de funciones y de clases para interceptar y agregar funcionalidad a las funciones e instancias de clases, de forma similar, las metaclases nos permiten interceptar y aumentar la lógica durante la creación de las clases. La principal diferencia entre los decoradores y las metaclases es su lugar en el tiempo de aplicación sobre las clases.

Este es un tema muy avanzado, el cual nos permite aprender cómo funciona realmente python y el cual debemos conocer de forma de poder aplicar a patrones avanzados de desarrollo que pueden ser difíciles de implementar de otra forma.

Al declarar una metaclase, le pedimos a Python que enrute la creación del objeto de clase a otra clase que proporcionamos, de forma de poder insertar algún código que se ejecute automáticamente al final de una declaración de clase.

Si lo pensamos un poco podríamos hacer esto con el conocimiento que tenemos hasta el momento (aunque la metaclases nos brinda más control como veremos más adelante), supongamos que tenemos una clase llamada "**MiClase**" la cual queremos extender con una funcionalidad para la cual la clase no fue creada, podríamos extender la clase creando una función que cumpla la rutina que necesitamos de la siguiente manera:

metaclase1.py

```
1  def extenter_clase(self, arg):
2      print(arg)
3
4  class MiClase1: pass
5  MiClase1. extenter_clase = extenter_clase
6
7  objeto = MiClase1()
8  objeto.extenter_clase("Este método se ha agregado a la clase")
```

Retorna:

```
Este método se ha agregado a la clase
```

Los objetos de la clase pueden como se muestra en el ejemplo anterior utilizar la función como si esta fuera un método de la clase, con lo cual hemos logrado extender la funcionalidad de la clase.

En el caso de que tuviéramos muchas clases podríamos generalizar el ejemplo anterior creando una función administradora que nos permita pasar como parámetro el nombre de la clase a la cual le queremos agregar la nueva funcionalidad, de la siguiente manera:

metaclase2.py

```
1   def extenter_clase(self, arg):
2       print(arg)
3
4   def nueva_funcionalidad(Clase):
5       Clase. extenter_clase = extenter_clase
6
7   class MiClase1: pass
8   nueva_funcionalidad(MiClase1)
9
10  objeto = MiClase1()
11  objeto. extenter_clase ("Este método se ha agregado a la clase")
12
```

Retorna:

```
Este método se ha agregado a la clase
```

Una metaclase también nos permite agregar código al final de la declaración de una clase, al declarar una metaclase, le pedimos a Python que enrute la creación del objeto de clase a otra clase que proporcionamos (la metaclase):

Para comprender las metaclases, primero debemos comprender un poco más sobre el modelo de "**type**" de Python y lo que sucede al final de una declaración de clase. Como veremos aquí, los dos están íntimamente relacionados.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

En python 3 las instancias son objetos de una clase, y las clases son objetos de la clase type. Podemos ver esto rápidamente al ejecutar el siguiente código y consultar de que tipo son las instancias de clases y las clases:

metaclase3.py

```
1    class Auto():
2        color = "azul"
3
4    objeto = Auto()
5    print(type(objeto))
6    print(type(Auto))
7    print(type(Auto.__class__ ))
```

Retorna:

```
<class '__main__.Auto'>
<class 'type'>
<class 'type'>
```

2.- En python3, las metaclases son subclases de type

Las metaclases son subclases de la clase type, el uso de metaclases, nos permite implementar clases de clases personalizadas. Con todo detalle, todo se puede explicar de la siguiente forma:

• **type** es una clase que genera clases definidas por el usuario.

• Las metaclases son subclases de la clase **type**.

• Los objetos de clase son instancias de la clase **type**, o una subclase de la misma.

• Los objetos de instancia se generan a partir de una clase.

En otras palabras, para controlar la forma en que se crean las clases y aumentar su comportamiento, todo lo que debemos hacer es especificar que se cree una clase definida por el usuario a partir de una **metaclase** también definida por el usuario en lugar de la clase **type** normal.

Nota: Tenemos que tener en cuenta que esta relación no es exactamente igual a la herencia normal. Las clases definidas por el usuario también pueden tener superclases de las cuales ellas y sus instancias heredan los atributos como lo realizamos hasta ahora.

Nota: Las superclases de herencia se enumeran entre paréntesis en la declaración de la clase y se muestran en una tupla a partir de aplicarle a la clase con notación de punto el operador1 __bases__. Sin embargo, la relación con la clase **type** a partir de la cual se crea una clase y de la cual es una instancia posee una relación diferente.

Nota: La herencia busca diccionarios de espacios de nombres de instancia y clase, pero las clases también pueden adquirir un comportamiento de su tipo que no está expuesto a la búsqueda de herencia normal.

Ya hemos aprendido que cuando Python alcanza una declaración de clase, ejecuta su bloque de código anidado para crear sus atributos; todos los nombres asignados en el nivel superior del bloque de código anidado generan atributos en el objeto de clase resultante. Estos nombres suelen ser funciones de método creadas por definiciones anidadas, pero también pueden ser atributos arbitrarios asignados para crear datos de clase compartidos por todas las instancias. En términos técnicos, Python sigue un protocolo estándar para que esto ocurra: al final de una declaración de clase, y después de ejecutar todo su código anidado en un diccionario de espacio de nombres correspondiente al ámbito local de la clase, Python llama al objeto **"type"** para crear el objeto de clase como:

```
clase = type(nombreClase, superclases, diccionarioDeAtributos)
```

Debido a que la metaclase es una subclase de type, la llamada a __call__ de la clase **type**, delega las llamadas para crear e inicializar el nuevo objeto de clase a la metaclase, si se define versiones personalizadas de estos métodos:

```
type.__new__(typeclass, nombreClase, superclases, diccionarioDeAtributos)
type.__init__(class, nombreClase, superclases, diccionarioDeAtributos)
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Si la metaclase define sus propias versiones de __new__ o __init__, serán invocadas a su vez durante esta llamada por el método __call__ de la clase de tipo heredada, para crear e inicializar la nueva clase. El efecto neto es ejecutar automáticamente los métodos que proporciona la metaclase, como parte del proceso de construcción de la clase.

El método **__new__** crea y devuelve el nuevo objeto de clase, y luego el método **__init__** inicializa el objeto recién creado. Estas son las referencias que las subclases (metaclaes) de **type** generalmente usan para personalizar las clases.

Veamos un par de ejemplo para visualizar lo dicho hasta aquí, supongamos que partimos de la siguiente clase:

metaclase4.py

```
1   class Material():
2       pass
3
4   class Auto(Material):
5       color = "azul"
6       def retornar_color(self, color):
7           return self.color
8
9   objeto = Auto()
10  print(type(objeto))
11  print(type(Auto))
12  print(type(Auto.__class__ ))
```

Al ejecutar este código nos retorna:

```
<class '__main__.Auto'>
<class 'type'>
<class 'type'>
```

En este ejemplo que ya comprendemos, podemos ver que Python internamente ejecuta un bloque de código de forma de crear dos atributos de clase (**color** y **retornar_color**) y luego llama al objeto type de forma de generar el siguiente objeto de clase al final de la declaración de la clase.

```
Auto = type('Auto', (Material,), {'color': 'azul', 'retornar_color':retornar_color, '__module__':'__main__'} )
```

Podemos expresar la clase anterior de una forma que nos permita comprender mejor lo que está pasando, de la siguiente manera:

metaclase5.py

```
1  Auto = type('Auto', (), {'color': "azul", 'retornar_color': (lambda x: x.color)})
2  objeto = Auto()
3  print(type(objeto))
4  print(type(Auto))
5  print(type(Auto.__class__ ))
```

El resultado de los dos ejemplos es el mismo cómo podemos ver:

```
<class '__main__.Auto'>
<class 'type'>
<class 'type'>
```

Como podemos ver, ambas formas son análogas, aunque en la segunda opción visualizamos más explícitamente el uso de **type**.

En definitiva, para crear una metaclase personalizada para la clase, debemos declararla de forma que intercepte la llamada de creación de instancia normal en una clase definida por el usuario.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

3.- Metaclases y herencia.

La **metaclase** se llama luego de las clases heredadas, ya que una clase puede de forma simultánea heredar de otra u otras clases y tener una **metaclase** asignada.

Veamos un ejemplo en el cual la clase posee tanto una superclase como una metaclase:

metaclase6.py

```
1   class MiMetaclase(type):
2       def __new__(meta, nombre_de_clase, superclase, diccionario_de_clase):
3           print('En new de metaclase:', meta, nombre_de_clase, superclase,
    diccionario_de_clase, sep='\n...')
4           return type.__new__(meta, nombre_de_clase, superclase,
    diccionario_de_clase)
5
6   class MiSuperclase:
7       pass
8
9   class MiClase1(MiSuperclase, metaclass=MiMetaclase):
10      atributo1 = 1
11      def metodo1(self, arg):
12          return self.atributo1 + 3*arg
13
14  print('Creando una instancia')
15  X = MiClase1()
16  print('atributo1:', X.atributo1, X.metodo1(7))
```

El resultado de los dos ejemplos es el mismo cómo podemos ver:

```
En new de metaclase:
...<class '__main__.MiMetaclase'>
...MiClase1
...(<class '__main__.MiSuperclase'>,)
...{'__module__': '__main__', '__qualname__': 'MiClase1', 'atributo1': 1, 'metodo1':
<function MiClase1.metodo1 at 0x01B287C8>}
Creando una instancia
atributo1: 1 22
```

Como podemos __new__ recibe tres parámetros:

- **nombre_de_clase** = El nombre de la clase de la clase que estoy pasando.
- **superclase** = Es una tupla con el listado de las superclases de la clase a la cual le estamos aplicando la metaclase.
- **diccionario_de_clase** = es un diccionario con los atributos de la clase que estamos pasando.

Además de __new__ contamos con __init__, el cual también es invocado por el operador __call__ del objeto **type**. En general:

- __new__ crea y devuelve el objeto de clase.
- __init__ inicializa la clase ya creada que se pasa como un argumento.

Las metaclases pueden usar una o ambas referencias de forma de administrar la clase en el momento de la creación:

metaclase7.py

```
1   class MiMetaclase(type):
2       def __new__(meta, nombre_de_clase, superclase, diccionario_de_clase):
3           print('En __new__ de metaclase:', meta, nombre_de_clase, superclase,
    diccionario_de_clase, sep='\n...')
4           return type.__new__(meta, nombre_de_clase, superclase,
    diccionario_de_clase)
5
6       def __init__(Clase, nombre_de_clase, superclase, diccionario_de_clase):
7           print('En __init__ de metaclase:', nombre_de_clase, superclase,
    diccionario_de_clase, sep='\n...')
8           print('...init objetos de clase:', list(Clase.__dict__.keys()))
9
10  class MiSuperclase:
11      pass
12
13  class MiClase1(MiSuperclase, metaclass=MiMetaclase):
14      atributo1 = 1
15      def metodo1(self, arg):
16          return self.atributo1 + 3*arg
17
18  print('Creando una instancia')
19  X = MiClase1()
20  print('atributo1:', X.atributo1, X.metodo1(7))
```

El resultado de los dos ejemplos es el mismo cómo podemos ver:

```
En __new__ de metaclase:
...<class '__main__.MiMetaclase'>
...MiClase1
...(<class '__main__.MiSuperclase'>,)
...{'__module__': '__main__', '__qualname__': 'MiClase1', 'atributo1': 1, 'metodo1':
<function MiClase1.metodo1 at 0x039A8858>}
En __init__ de metaclase:
...MiClase1
...(<class '__main__.MiSuperclase'>,)
...{'__module__': '__main__', '__qualname__': 'MiClase1', 'atributo1': 1, 'metodo1':
<function MiClase1.metodo1 at 0x039A8858>}
...init objetos de clase: ['__module__', 'atributo1', 'metodo1', '__doc__']
Creando una instancia
atributo1: 1 22
```

En este caso, el método de inicialización de clase **__init__** se ejecuta después del método de construcción de **clase __new__**, pero ambos se ejecutan al final de la instrucción de clase antes de que se realicen las instancias. Por otro lado un __init__ en "**MiClase1"** se ejecutaría en el momento de la creación de la instancia, y no se verá afectado ni ejecutado por el __init__ de la metaclase:

Algunas notas importantes.

- Debido a que las metaclases se especifican de manera similar a las superclases de herencia, pueden ser un poco confusas a primera vista. Algunos puntos clave deberían ayudar a resumir y aclarar el modelo:

- Las metaclases suelen redefinir las clases de tipo __new__ y __init__ para personalizar la creación y la inicialización de la clase, y aunque es menos común, también pueden redefinir __call__.

- Las declaraciones de metaclase especifican una relación de instancia, que no es lo mismo que hemos llamado herencia hasta ahora. Dado que las clases son instancias de metaclases, el comportamiento definido en una metaclase se aplica a la clase, pero no a las instancias posteriores de la clase.

- Las instancias obtienen el comportamiento de sus clases y superclases, pero no de las metaclases.

- Técnicamente, la herencia de atributos para instancias normales generalmente busca solo los diccionarios __dict__ de la instancia, su clase y todas sus superclases; Las metaclases no se incluyen en la búsqueda de herencia para instancias normales.

- Técnicamente, las clases adquieren atributos de metaclase a través del enlace __class__ de la clase, al igual que las instancias normales adquieren nombres de su clase, pero la herencia a través de la búsqueda __dict__ se intenta primero.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

4.- Atributos en Superclases y Metaclases

Cuando trabajamos con metaclases, tenemos que tener en cuenta que las instancias de nuestras clases no tienen acceso a los atributos declarados en las metaclases, como lo tienen a las declaradas en las superclases. Además:

- Las instancias de clases tienen acceso a los atributos de la clase a través del operador __dict__
- El objeto de clase tiene acceso a los atributos de la metaclase a través del operador __class__

Un ejemplo simple nos permite evidenciar esto:

metaclase8.py

```
1   class ControlMotor(type):
2       encendido = False
3
4   class Material():
5       material = 'plástico'
6
7   class Auto(Material, metaclass=ControlMotor):
8
9       marca = "renault"
10      def __init__(self, color):
11          self.color = color
12
13      def retornar_color(self,valor):
14          return self.color + str(valor)
15
16  objeto = Auto("azul")
17  print(objeto.material)         #Accedo a atributo de superclae
18  print(objeto.color)           #Accedo a atributo de instancia
19  print(Auto.__dict__)           #Accedo a diccionario de atributos de clase
20  print(Auto.__dict__['marca'])   #Accedo a clave 'marca' del diccionario de atributos
21  de clase
22  print(objeto.__dict__)         #Accedo a diccionario de atributos de instancia.
23  print(objeto.retornar_color(6))  #Acceso a método de instancia
```

```
24  print(Auto.__class__.encendido) #Accedo a atributo 'encendido' de metaclase
25  print(Auto.encendido) #Accedo a atributo 'encendido' de metaclase
26  try:
27      print(objeto.encendido)
28  except AttributeError:
29      print('La instancia no tiene acceso a este atributo')
30  print('---'*25)
```

```
plástico
azul
{'__module__': '__main__', 'marca': 'renault', '__init__': <function Auto.__init__ at
0x00A087C8>, 'retornarColor': <function Auto.retornarColor at 0x00A08858>, '__doc__':
None}
renault
{'color': 'azul'}
azul6
False
False
La instancia no tiene acceso a este atributo
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

5.- Herencia en metaclases

Podemos además de lo realizado hasta aquí realizar una herencia en metaclases, para visualizarlo creamos la clase "**Control**" de la cual hereda la metaclase de la clase **"Auto"** según se muestra a continuación.

metaclase9.py

```
1   class Control(type):
2       estado = 'inspección'
3
4   class ControlMotor(Control):
5       encendido = False
6
7   class Material():
8       material = 'plástico'
9
10  class Auto(Material, metaclass=ControlMotor):
11
12      marca = "renault"
13      def __init__(self, color):
14          self.color = color
15
16      def retornar_color(self,valor):
17          return self.color + str(valor)
18
19  objeto = Auto("azul")
20  print(Auto.__class__.estado)    #Accedo a atributo 'estado' de metaclase padre
21  print(Auto.__class__.encendido) #Accedo a atributo 'encendido' de metaclase
```

```
False
inspección
```

Podemos ver que es posible desde "Auto" acceder al atributo declarado en la metaclase padre.

6.- Métodos de metaclase

Lo comentado hasta aquí también se cumple con los métodos que declaremos en las metaclases, estos pueden ser accedidos desde la clase pero no desde la instancia de la clase como se muestra a continuación:

metaclase10.py

```
1   class ControlMotor(type):
2
3       def retornar_material(cls):
4           return "Retornar material"
5
6   class Auto(metaclass=ControlMotor): pass
7
8
9   print(Auto.retornar_material())
10  objeto = Auto()
11  try:
12      print(objeto.retornar_material())
13  except:
14      print('La instancia no tiene acceso a este método')
```

```
Retornar material
La instancia no tiene acceso a este método
```

7.- Sobrecarga de operadores en metaclases

De forma análoga ha como realizamos la sobrecarga de operadores para las instancias de una clase, podemos realizar una sobrecarga de operadores que sea aplicable a una instancia de clase, veamos un ejemplo:

metaclase11.py

```
1   class ControlMotor(type):
2
3       def __getitem__(cls, i):
4           return cls.color[i]*5
5
6   class Auto(metaclass=ControlMotor):
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
7         color = "Azul"
8         def __getitem__(self, indice):
9             return indice ** 0.5
10
11    print(Auto[1])
12
13    objeto = Auto()
14    print(objeto[64])
```

```
8.0
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bibliografía utilizada y sugerida

Libros y otros manuscritos:

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