UTN.BA | Centro de e-Learning
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 1:

Nuevas estructuras

Unidad 1:

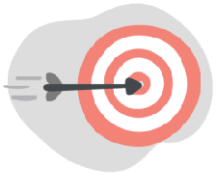
Delegación – Sobrecarga de operadores - Slots

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

En el trascurso 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.

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…

Comiencen a trabajar con sobrecarga de operadores.

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

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.- Delegación.

2.- Namespaces.

3.- Sobrecarga de operadores.

4.- Introducción a los métodos especiales.

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.- 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 comer_arroz(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.comer_arroz()
```

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.

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

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. Cómo 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()))
```

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

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 especifico 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:

```
<__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 credo 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   mi_lista=[81, 16, 64, 9]
2
3   iterador = iter(mi_lista)
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
```

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

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
------------------------
```

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