Seminario # 16 Lenguajes dinámicos 1

1 Conceptos Generales

1.1 Clases en Python:

Las clases en Python proveen una forma de empaquetar datos y funcionalidades juntas. Al crear una nueva clase se crea un nuevo tipo de objeto permitiendo crear nuevas instancias de ese tipo. Las clases también participan en la naturaleza dinámica de Python (se crean en tiempo de ejecución y pueden modificarse luego de su creación).

```
class classname:
    <declaración 1>
    ...
    ...
    ...
    <declaración n>
```

1.2 Métodos en Python:

Para definir métodos de una clase en Python utilizamos la siguiente sintaxis:

```
class classname:
    def methodname(self, args):
        #implementacion del método
        pass
```

Utilizamos `self` como primer parámetro de los métodos porque al utilizar la notación `X.methodname()` donde `X` es una instancia de `classname` Python automáticamente le pasa como primer parámetro al método la instancia `X`. Para los métodos que no sean particulares de la instancia esto no es necesario y los invocamos de la forma `classname.method2()` (funciones estáticas en C#)

También podemos generar objetos funciones utilizando la notación *lambda*. Al utilizar `def` lo que hacemos es asignarle una función a un nombre. Al utilizar la notación *lambda* devolvemos la función permitiéndonos asignársela a un nombre o utilizarla como función anónima. Las funciones *lambda* se crean utilizando la siguiente sintaxis:

```
lambda param1, param2, ... , paramN : expresión
```

1.3 Sobrecarga de operadores:

- Es la acción de interceptar operaciones built-in e invocar métodos definidos en la clase para realizar estas. Python se encarga automáticamente de invocar a los métodos de una clase cuando instancias de esta aparecen en operaciones built-in. Esto permite que nuestras clases se asemejen más a los tipos built-in del lenguaje.
- Para implementar la sobrecarga de operadores en una clase nuestra debemos definir en ella métodos con nombres específicos.

Método	Invocados por:
`__init__`	Creación de un objeto: x=Class(args)
`__add__`	X+Y
`__or__`	X or Y
`__str__`	`print(X), str(X)`
`__getattr__`	`X.Undefined`
`__getattribute__`	`X.Defined`
`__getitem__`	`X[key]`
`__setitem__`	`X[key]=value`
`__iter__ , __next__`	`for i in X`

1.4 Expresiones Generadoras:

Estos conceptos son bastante parecidos en su sintaxis, la única diferencia es que el primero se escribe entre paréntesis y el segundo entre corchetes. Las expresiones generadoras devuelven un objeto generador que produce resultados por demanda.

```
Gen = (x**2 for x in range(1000))
```

En cada iteración del objeto `Gen` nos va a devolver el valor que devuelva el iterador `range(1000)` aplicándole la función `x**2`.

1.5 List Comprehension:

Las *list comprehension* coleccionan el resultado de aplicar una expresión a una secuencia de valores y los retornan en una nueva lista. Supongamos que queremos guardar el código **ASCII** de todos los caracteres de un string. Quizas lo primero que se nos ocurra sea algo como esto:

```
res = []
for x in 'listCopm':
    res.append(ord(x))
```

Pero utilizando una expresión de *list comprehension* podemos lograr el mismo resultado con algo así:

```
res = [ord(x) for x in 'listComp']
```

Las *list comprehension* son incluso más generales, permitiéndonos utilizar una condición luego del `for` para agregar una selección lógica de los elementos.

```
res = []
for x in range(5):
    if x % 2 == 0:
        res.append(x**2)
```

```
res = [x**2 for x in range(5) if x % 2 == 0]
```

Ambos códigos producen el mismo resultado, y no es difícil apreciar cuál de los dos es más conciso.

> Debemos tener en cuenta que las *list comprehension* pueden llegar a ser bastante engorrosas debido al anidamiento de ciclos for que podemos hacer en ellas. Sin embargo, en estos casos, existe una ganancia substancial en cuanto a rendimiento al usar éstas a pesar de su complejidad extra, ya que las *list comprehension* son el doble de rápidas que los ciclos `for` corrientes.

Ejercicio del Seminario:

2 Conjunto de Wirth

El conjunto wirth es un conjunto q tiene la siguiente definición:

- 1 pertenece al conjunto
- Si x pertenece entonces x*2+1 pertenece y* $x3+1$ pertenece En la funcion `__next__` vamos controlando la forma de ir retornado los elementos y cuando se llega a la cantidad requerida por la instancia de la clase entonces se lanza la excepción `StopIteration` q es la forma que tiene python de detener el proceso de iteración.

```
class WirthIter:
    def __init__(self,count:int):
```

```
        self.count = count
        self.i = 0
        self.current = 1
        self.even = []
        self.odd = []
```

Al utilizar la forma sin generador la función `__iter__` devuelve el iterador a recorrer, si se retorna self, la misma clase será este iterador, de ahí que la forma de moverse lo decide la función `__next__`. En la función `__next__` vamos controlando la forma de ir retornado los elementos y cuando se llega a la cantidad requerida por la instancia de la clase entonces se lanza la excepción `StopIteration` que es la forma que usa python para detener el proceso de iteración. Para esto nos auxiliaremos de dos listas, un indice `self.i` y un `self.current` los cuales definimos en la clase `WirthIter`.

```
    #La forma de iterar sin Generador
    def __iter__(self):
        return self
    def __next__(self):
        if self.i == self.count:
            raise StopIteration
        self.i += 1
        old_current = self.current
        self.even.append(2 * self.current + 1)
        self.odd.append(3 * self.current + 1)
        if self.even[0] < self.odd[0]:
            self.current = self.even[0]
            self.even.__delitem__(0)
        else:
            self.current = self.odd[0]
            self.odd.__delitem__(0)
        return old_current
```

Las expresiones generadoras son similares a las *list comprehension* pero estas retornan un objeto que produce resultados en demanda, por lo que al usar el `yield` en la funcion `__iter__`, estamos generando un iterador sobre el cual se va trabajar en el momento que se tome una instancia de la clase `WirthIter` como iterador. Fíjese que en este caso no se implemento la funcion `__next__` porque estamos retornando el objeto a iterar y en este caso no es necesario redefinir la forma de iterar por la clase (ver líneas 34-47 del archivo .py adjunto).

```
    #La forma de iterar con Generador
        def __iter__(self):
```

```
        i=0
        while i < self.count:
            yield self.current
            i += 1
            self.even.append(2 * self.current + 1)
            self.odd.append(3 * self.current + 1)
            if self.even[0] < self.odd[0]:
                self.current = self.even[0]
                self.even.__delitem__(0)
            else :
                self.current = self.odd[0]
                self.odd.__delitem__(0)
```

3 ConcatIterable

ConcatIterable: Iterable conformado por los elementos de dos iterables puestos a continuación.

```
class ConcatIterable:
    def __init__(self,firstIter,secondIter):
        self.first = firstIter
        self.second = secondIter
```

Para resolver este ejercicio sin utilizar generadores se modifica la función `__next__` de forma tal que, cuando acabe con los elementos del primer iterable, continue con el segundo y lance la excepción `StopIteration` cuando no tenga mas objetos por los que iterar. Para ello se llama a la función `__iter__` de cada iterable en la definición de la clase y de esta manera podremos recorrer con `__next__` cada objeto iterable puesto que no sabemos cual es la forma de iterar en cada uno.

```
    #La forma de iterar sin Generador
    def __iter__(self):
        return self

    def __next__(self):
        try:
           value = self.first.__next__()
        except:
            try:
                value = self.second.__next__()
```

```
            except:
                raise StopIteration
```

En este caso nos auxiliamos del generador `yield` podemos recorrer ambos iteradores y formar un objeto iterable que contiene todos los elementos del primero y segundo iterable puestos uno a continuacion del otro. Existen muchas otras opciones como guardar los elementos en una lista mas grande o simplemente retornar `self` y modificar el comportamiento de la funcion `__next__` pero en este ejemplo sólo usando la función `__iter__` se soluciona el problema en muy pocas líneas, además solo con un `for` por cada iterable se pueden recorrer perfectamente sin acudir a sus funciones `__iter__` y `__next__`.

```
    #La forma de iterar con Generador
    def __iter__(self):
        for x in self.first:
            yield x
        for y in self.second:
            yield y
```

4 WhereIterable:

WhereIterable: Iterable conformado por aquellos elementos de un iterable que cumplen un determinado predicado.

```
class WhereIterable:
    def __init__(self,iter,predicado):
        self.iter = iter
        self.predicado = predicado
        self.current = 0
```

Para resolver este problema al trabajar sin generadores hemos de verificar que mientras no se cumpla el predicado continúe al próximo elemento. En caso que se terminen los elementos del iterable la función `__next__` lanza una excepción la cual es capturada y envía una `StopIteration`. Como pueden ver nos auxiliamos de la función `__next__` de la misma forma que en el ejercicio anterior.

```
    #La forma de iterar sin Generador
    def __iter__(self):
        return self

    def __next__(self):
```

```
        try:
            self.current = self.iter.__next__()
            try:
                if not self.predicado(self.current)
            except:
                ...
            while not self.predicado(self.current):
                self.current = self.iter.__next__()
            return self.current
        except:
            raise StopIteration
```

El generador yield va formando el iterable de los elementos que resulten `True` del predicado. Fíjese que al igual que en el ejercicio anterior se recorre con un `for` el iterable y tampoco es necesario redefinir la funcion `__next__` para la clase `WhereIterable`. El `try`, al igual que cuando vemos el ejemplo sin generador, sólo cuida que el predicado de la entrada retorne un valor `bool`.

```
    #La forma de iterar con Generador
    def __iter__(self):
        for x in self.iter:
            try:
                if self.predicado(x):
                    yield x
            except:
                ...
```

5 StopIteration

Como hemos visto durante el seminario el mecanismo que existe en Python para indicar el fin de iteración es lanzar una excepción `StopIteration`.

Aunque este mecanismo sigue la filosofía de Python: "Es mejor pedir perdón que pedir permiso", no creemos que sea la forma más adecuada de indicar el fin de iteración, debido al riesgo que lleva consigo manejar código mediante excepciones. Algunas variantes propuestas para cambiar esto son:

- Tener un valor especial que indique el final de la colección.
- Una función `End()` que indique si terminó o no la colección.