title: Presentacion Python 2019
Author: Claudia Banchoff, Viviana Harari
description: clase 9
keywords: POO, iteradores, generadores
css: estilo.css

Seminario de Lenguajes - Python

Cursada 2019

Temario

- Programación Orientada a Objetos en Python (Parte II)
- Iteradores
- Generadores

Repaso por Python plus: excepciones

- ¿A qué llamamos una excepción?
- ¿Qué acción se toma después de levantada y manejada una excepción?: ¿se continúa con la ejecución de la unidad que lo provocó o se termina?
- ¿Cómo se alcanza una excepción?
- ¿Qué sucede cuando no se encuentra un manejador para una excepción levantada?

Repaso por Python plus: excepciones

- ¿Qué imprime el siguiente código?

```
dic = {1:'uno', 4:'cuatro', 5:'cinco'}
try:
        for x in range(1,6):
                print (dic[x])
except (KeyError):
        dic[x] = 'nuevo'
print (dic)
```

- ¿Qué podemos decir del bloque try except?

Repaso por Python plus: excepciones

- Y en este caso, ¿qué imprime el siguiente código?

```
dic = {1:'uno', 4:'cuatro', 5:'cinco'}
for x in range(1,6):
        try:
                print (dic[x])
        except (KeyError):
                dic[x] = 'nuevo'
print (dic)
```

Repaso por Python plus: excepciones

- ¿Qué imprime el siguiente código?

```
dic = {1:'Juan', 2:'Ana', 5:'Helena'}

try:
        for x in range(1,6):
                print (dic[x])
except (KeyError):
        dic[x] = 'Agregado'
else:
        print ('Se recorrió el diccionario y NO se agregaron valores')
print ("El diccionario al final del proceso es: ", dic)
```

- ¿Qué podemos decir de la cláusula else?

Repaso por Python plus: excepciones

- ¿Qué imprime el siguiente código?

```
def dividir (x,y):
        try:
                return x / y

        except ZeroDivisionError:
                print ('¡Division por CERO!')
                return 'Indefinido'
        finally:
                print('Ha FINALIZADO la ejecucion del modulo')
```

```
x = int(input('Ingresa el dividendo: '))
y = int(input('Ingresa el divisor: '))
result = dividir(x,y)
print ('La division es', result)
```

- ¿Qué podemos decir de la cláusula finally?

Repaso por Python plus: excepciones

- ¿Qué imprime el siguiente código?

```
dic = {1:'Juan',4:'Pedro',5:'Helena'}
y = 9
try:
        print ('Vamos a entrar a otro bloque TRY')

        try:
                for x in range(1,6):
                        print (dic[z])
        except (KeyError):
                dic[x] = 'Agregado'

        y = y + 1
        print ('El valor de y es:' + str(y))
except(NameError):
        print ('OJO! Se esta usando una variable que no existe')

print ('Se sigue con las siguientes sentencias del programa')
print ("El diccionario al final del proceso es: ", dic)
```

Repaso por Python plus: POO

- ¿Cómo definimos una clase?
- ¿A qué llamamos clase base y clase derivada?
- ¿Qué tipo de herencia provee Python?
- ¿__init__()?
- ¿A cuáles de las siguientes propiedades las considereamos privadas?

```
class A:
        def __init__(self, x, y, z):
                self.varX = x
                self._varY = y
                self.__varZ = z
```

Repaso por Python plus: POO

- ¿Qué imprime el siguiente código?

```
class A:
        def __init__(self, x, y, z):
                self.varX = x
                self._varY = y
                self.__varZ = z

        def demo(self):
                return("x: {} -- y:{} --- z:{}".format(self.varX, self._varY, self.__var

class B(A):
        def __init__(self):
                super().__init__("x", "y", "z")
        def demo(self):
                return(super().demo())
                #return("x: {} -- y:{} --- z:{}".format(self.varX, self._varY, self.__va

objB = B()
print(objB.demo())
```

Repaso por Python plus: POO

- ¿Qué imprime el siguiente código?

```
class A:
        def __init__(self, x, y, z):
                self.varX = x
                self._varY = y
                self.__varZ = z

        def demo(self):
                return("x: {} -- y:{} --- z:{}".format(self.varX, self._varY, self.__var

class B(A):
        def __init__(self):
                super().__init__("x", "y", "z")
                self.__varZ = "Z de B"

        def demo(self):
                return(super().demo())
                #return("x: {} -- y:{} --- z:{}".format(self.varX, self._varY, self.__va

objB = B()
print(objB.demo())
```

Retomamos POO

getters y setters

- ¿A qué nos referimos?

```
class Jugador ():
        "Define la entidad que representa a un jugador en el juego"
        def __init__(self, nom="Tony", nic="Ironman", clave="123", e_mail=""):
                self.__nombre = nom
                self.__nick = nic
                self.__contraseña = clave
                self.__mail = e_mail
                self.__vidas = 0
                self.__puntaje = 0
        def get_nombre(self):
                return self.__nombre
        def set_nombre(self, nuevo_nombre):
                self.__nombre = nuevo_nombre

        def incrementar_vidas(self, cant_vidas):
                self.__vidas += cant_vidas
        ....

jugador1 = Jugador()
jugador2 = Jugador("Bruce", "Batman", "4321", "batimail@gmail.com")

print(jugador1.get_nombre())
```

La función property()

- Veamos un ejemplo simple...

```
class Demo:
    def __init__(self):
        self.__x = 0
    def getx(self):
        return self.__x
    def setx(self, value):
        self.__x = value
    def delx(self):
        del self.__x
    x = property(getx, setx, delx, "x es una propiedad")
```

```
obj = Demo()
obj.x = 10
print(obj.x)
del obj.x
```

- **property(fget=None, fset=None, fdel=None, doc=None)**
- Más info: https://docs.python.org/3/library/functions.html?highlight=property#property

---

@property

```
class Demo:
    def __init__(self):
        self.__x = 0

    @property
    def x(self):
        return self.__x

obj = Demo()
obj.x = 10 # Esto dará error
print(obj.x)
```

- @property es un **decorador**: una función que recibe una función como argumento y retorna otra función.
    - Más info: https://recursospython.com/guias-y-manuales/decoradores/
- ¿Cuál es la función que estamos aplicando? ¿Y cuál es la que pasamos como argumento?
- No podemos modificar la propiedad x. ¿Por qué?

El ejemplo completo

```
class Demo:
    def __init__(self):
        self.__x = 0

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, value):
        self.__x = value

    @x.deleter
    def x(self):
        del self.__x

obj = Demo()
obj.x = 10
print(obj.x)
del obj.x
```

Herencia y propiedades

- Observemos este código: ¿qué significa?

```
class Demo:
        def __init__(self):
                self.__x = 0

        @property
        def x(self):
                return self.__x

        @x.setter
        def x(self, value):
                self.__x = value

class Demo1(Demo):
        def __init__(self):
                super().__init__()

obj = Demo1()
obj.x = 10
print(obj.x)
```

Veamos la implementación de PySimpleGUI

- Analicemos PySimpleGUI
- Observemos alguna de las clases.
- Busquemos las propiedades definidas.

¿Qué observan en el siguiente código?

```
cadena = "Seminario de Python"
for caracter in cadena:
        print(caracter)

lista = ['esto', 'es', 'una', 'lista']
for palabra in lista:
        print(palabra)

superheroes = { 'Ironman' : 'Marvel', 'Batman' : 'DC'}
for clave, valor in superheroes.items():
        print("{} : {}". format(clave, valor))

for linea in open("pp.txt"):
        print(linea.rstrip())
```

---

class: destacado

Iteradores

- Todas son secuencias iterables. ¿Qué significa?
    - Todas pueden ser recorridas por la estructura: **for** var **in** secuencia.
    - Todas implementan un método especial denominado __**iter**__.
    - __iter__ devuelve un iterador capaz de recorrer la secuencia.

Un iterador es un objeto que permite recorrer uno a uno los elementos de una estructura de datos para poder operar con ellos.

Iteradores

- Un iterador tiene que implementar un método **next** que debe devolver los elementos, de a uno por vez, comenzando por el primero.
- Y al llegar al final de la estructura, debe levantar una excepción de tipo **StopIteration**.
- Los siguientes códigos son equivalentes:

```
lista = ['esto', 'es', 'una', 'lista']
for palabra in lista:
        print(palabra)

iterador = iter(lista)
while True:
        try:
                palabra = next(iterador) # o iterador.__next__()
        except StopIteration:
                 break
        print(palabra)
```

- La función **iter** retorna un objeto iterador.

Iteradores

- Veamos este ejemplo: ¿Qué creen que imprime?

```
class CadenaInvertida:
     def __init__(self, cadena):
         self.__cadena = cadena
         self.__posicion = len(cadena)

     def __iter__(self):
         return(self)

     def __next__(self):
         if self.__posicion == 0:
             raise(StopIteration)
         self.__posicion = self.__posicion - 1
         return(self.__cadena[self.__posicion])

cadena_invertida = CadenaInvertida('Seminario de Python')

for caracter in cadena_invertida:
     print(caracter, end=' ')
```

Analicemos estas funciones

```
def otro_fibonacci(n):
        a, b = 0, 1
        lista = [a]
        for i in range(n):
                lista.append(b)
                a, b = b, a + b
        return (lista)

def fibonacci(n):
        a, b = 0, 1
        for i in range(n):
                yield a
                a, b = b, a + b

print(list(fibonacci(10)))
print("-------")
print(otro_fibonacci(10))
```

- **¿yield?**

Generadores

- Permiten crear iteradores.
- Son funciones pero utilizan **yield** en vez de **return**.
- Se genera una excepción de tipo **StopIteration** si encuentra un **return** durante la ejecución de un generador.
- Más info: https://es.stackoverflow.com/questions/6048/cu%C3%A1l-es-el-funcionamiento-de-yield-en-python

---

Generadores

- Tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas.
- Una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración **yield** (que es donde terminó la función en la última llamada).

```
def gen_diez_numeros(inicio):
    fin = inicio + 10
    while inicio < fin:
        inicio+=1
        yield inicio, fin
```

```
for inicio, fin in gen_diez_numeros(23):
    print(inicio, fin)
```

¿Y esto?

```
def pares():
     index = 1
     while True:
         yield index*2
         index = index + 1


par = pares()
for i in range(10):
  print(next(par), end=' ')

lista_de_pares =[next(par) for i in range(10)]
print(lista_de_pares)
```

- ¿loop infinito?

¿Cómo seguimos?