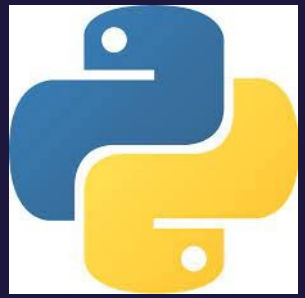
LENGUAJES y HERRAMIENTA PARA CIENCIAS DE DATOS I

POO en Python IV
(Iteradores, Generadores, Decoradores)



UCO ONLINE

Iteradores

- Definir clases que soportan la iteración:
  - __iter__()
  - __next__()

```python
class Invertir:
    def __init__(self, cadena):
        self.cadena = cadena
        self.index = len(cadena)
    def __iter__(self):
        return(self)
    def __next__(self):
        if self.index == 0:
            raise(StopIteration)
        self.index = self.index - 1
        return(self.cadena[self.index])
```

```python
# Declara iterable y recorre caracteres

cadena_invertida = Invertir('Iterable')
iter(cadena_invertida)

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

Generadores

- Forma sencilla y potente de crear iteradores
  - Función especial
    - Devuelve valores con yield
    - Crea automáticamente los métodos __iter__ y __next__
    - Guarda la variables locales y el estado de ejecución entre llamadas
    - Lanza automáticamente *StopIteration* al terminar

Generadores

Recorre una secuencia al revés

```
>>> def invertida(datos):
...     for index in range(len(datos)-1, -1, -1):
...             yield datos[index]
...
>>> for char in invertida('cadena'):
...     print(char)
...
a
n
e
d
a
c
>>>
```

Genera 10 números a partir de un inicio

```
>>> def gen_diez_numeros(inicio):
...     fin = inicio + 10
...     while inicio < fin:
...             inicio += 1
...             yield inicio
...
>>> for numero in gen_diez_numeros(23):
...     print(numero)
...
24
25
26
27
28
29
30
31
32
33
```

Decoradores

- Función que modifica el comportamiento de una función ya existente sin modificar el código

```
def nombre_decorador (funcion)
   def wrapper(*args, **kwargs):
      #Sentencias
      return funcion(*args, **kwargs)
   return wrapper
```

Decoradores

- Utilizarla
  - @decorador encima de la función a decorar
  - funcion = decorador(funcion)

```python
def debug(funcion):
    def wrapper(*args, **kwargs):
        print("Probando la funcion")
        return funcion(*args, **kwargs)

    return wrapper

def suma(a,b):
    return a + b
```

```python
suma = debug(suma)

print(suma(3,4))
```

```python
@debug
def suma(a,b):
    return a + b

print(suma(3,4))
```

Decoradores

- Pueden recibir argumentos
  - https://recursospython.com/guias-y-manuales/decoradores/