GOBIERNO DE
EL SALVADOR

SECRETARÍA DE
INNOVACIÓN

Agenda
Sesión 10/18

1. **Punto 1:** closures
2. **Punto 2:** decoradores
3. **Punto 3:** generadores

PROGRAMACIÓN CON PYTHON

CLOSURES

Closures

Primero recordemos…

```
def lista_alumnos(listado):

    def mostrar_lista():
        print("Hola")


listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
'''Estamos invocando la función lista
alumnos pero no la función mostrar_lista
por eso no obtenemos el resultado "Hola" '''
```

Closures

Primero recordemos…

```
def lista_alumnos(listado):

    def mostrar_lista():
        print("Hola")
    mostrar_lista() #ahora si llamamos a la función

listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
```

Closures

Importante

```
def lista_alumnos(listado):

        def mostrar_lista():
            print("Hola")
        mostrar_lista()
listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)

def otra_funcion():
        mostrar_lista() #no podemos llamar a esta función porque fue definida dentro de otra
otra_funcion()
```

Closures

Importante

```
def mostrar_lista():
        print("Hola")

def otra_funcion():
        mostrar_lista() #ahora si podemos llamar a esta función
otra_funcion()
```

Closures

Importante

```
def lista_alumnos(listado):
        for alumno in listado:
            print(alumno)

listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
#########
def lista_alumnos(listado):
        def listar():
            for alumno in listado:
                print(alumno)
        listar()
listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
```

Closures

Importante

```
def lista_alumnos(listado):
        def listar():
            for alumno in listado:
                print(alumno)
        listar()
        print(listado)
        #imprimimos todo el parámetro después de ejecutar listar
        #el parámetro impreso aparece antes de ser declarado
listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
```

Closures

Importante

```
#No podemos modificar variables locales en funciones anidadas
def lista_alumnos(listado):
        def listar():
                listado = ['Claudia','Juan','Iris']
                for alumno in listado: #variable local
                        print(alumno)
        listar()
        print(listado)
listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank'] #variable local
lista_alumnos(listado)
'''Observamos que la función listar() imprime el valor de su variable
local (listado) pero el parámetro (listado) de la función
lista_alumnos(listado) no ha cambiado.
Recordar que para Python las 2 variables listado son diferentes'''
```

Closures

Importante

```
'''
Si necesitamos cambiar las variables locales debemos hacer
uso de nonlocal
'''
def lista_alumnos(listado):
        def listar():
                nonlocal listado #de esta manera
                listado = ['Claudia','Juan','Iris']
                for alumno in listado:
                        print(alumno)
        listar()
        print(listado)
listado = ['Luis', 'Ana', 'Carlos', 'Laura', 'Rafael', 'Frank']
lista_alumnos(listado)
```

Closures

```
def closure(parametro):
    def funcion():
        return parametro + 1 # parametro es de la función closure()
    return funcion

variable = closure(parametro=2)
print (variable()) # Imprime 3
```

Un closure es una función (contenedora) que dentro de ella, definen otra función y la retorna al ser invocado.

Cabe destacar, que dichas funciones internas, tienen la capacidad de reconocer y “recordar” el valor de variables y parámetros definidos dentro del closure:

Closures

Se logra ver un poco más claro cuando utilizamos una función en la cual declaramos variables que las queremos utilizar como globales, y queremos evitar que las mismas se llenen de basura o interfieran en otro proceso y lo afecten, es decir el closure es una forma de encapsular las variables declaradas y elimina el riesgo de que sobre escriban a otras variables, es decir permite referenciar una variable que existe y pertenece a una función contenedora.

Closures

Closure = cierre/clausura

Se define cuando una función genera dinámicamente otra función.

```
#closure
def lista_alumnos(listado):
        listado = listado #local

        def listar():
                print(listado)
        return listar #retornamos la función listar

#asignamos la función lista_alumnos a una variable
nueva_variable = lista_alumnos("Luis, Ana, Carlos, Laura")
#llamamos a la nueva funcion creada en la variable
nueva_variable()
```

Closures

- **Cualquier función es un closure si en el momento de su creación guarda variables de un contexto padre de esa función.**
- **Siguen existiendo después de que su contexto padre ha sido ejecutado, como son las funciones internas devueltas desde una función externa.**
- **Y usan variables libres del contexto de la función externa.**
- **Es una función evaluada en un entorno contenido con una o más variables dependientes de otro entorno.**
- **Las variables dentro del closure mantienen su vida útil hasta que el closure muere.**
- **Una función dentro de otra función contenedora puede hacer referencia a las variables después de que la función contenedora ha terminado de ejecutarse.**

Closures

Otro ejemplo

```
def mi_closure(elparametro):
        def funcion_a_retornar():
            return elparametro + 1
        return funcion_a_retornar

variable_guarda_funcion=mi_closure(10)
print(variable_guarda_funcion())
```

Closures

Otro ejemplo

```
def mi_closure_validador(num1,num2):
        def funcion_validar():
                if num1>0 and num2>0:
                        return "Son mayores que cero"
                else:
                        return "No son mayores que cero"
        return funcion_validar

variable_guarda_funcion=mi_closure_validador(10,11)
print(variable_guarda_funcion())
```

PROGRAMACIÓN CON PYTHON

DECORADORES

Decoradores

A veces tendremos que modificar una función existente sin cambiar su estructura o líneas de código, esto para agregar mayor funcionabilidad a la función.

```
#Así se ve un decorador en Python
@decorador
def funcion():
        pass

#El decorador está siendo aplicado a la función
#Un decorador es una función que toma como argumento una función y retorna otra función.
```

Decoradores

Para crear decoradores debemos tener definidas por lo menos 3 funciones.

#funcion1, funcion2, funcion3

La funcion1 recibe como parámetro la funcion2 para dar como resultado la funcion3

```
def funcion1():
        pass #indica que por el momento la condicional, ciclo o función no ejecute nada.


def funcion2(): #función a decorar
        print("Decoraremos esta función 2")
```

```
#La funcion1 recibirá como parámetro la funcion2
def funcion1(parametro):
        #hacemos que la funcion1 retorne una nueva funcion3
        def funcion3():
                pass
        return funcion3

def funcion2():
        print("Decoraremos esta función 2")

#Estructura básica de un decorador
```

```
#La funcion1 recibirá como parámetro la funcion2
def funcion1(parametro):
        #hacemos que la funcion1 retorne una nueva funcion3
        def funcion3():
                print("Código antes de ejecutar la funcion")
                parametro() #ejecutamos la funcion
                print("Código después de ejecutar la funcion")
        return funcion3

def funcion2():
        print("Decoraremos esta función 2")
```

```
#La funcion1 recibirá como parámetro la funcion2
def funcion1(parametro):
        #hacemos que la funcion1 retorne una nueva funcion3
        def funcion3():
                print("Código antes de ejecutar la funcion")
                parametro() #ejecutamos la funcion
                print("Código después de ejecutar la funcion")
        return funcion3

@funcion1 #utilizamos @ para poder decorar una función
def funcion2():
        print("Decoraremos esta función 2")
#Una vez decorada la función la invocamos
funcion2()
```

Decoradores

Otro ejemplo

```
def suma():
        print(20+10)

def resta():
        print(30-12)

suma()
resta()
```

Decoradores

Otro ejemplo

```
def funcion_decoradora(funcion_parametro):
        def funcion_interna():
                print("A punto de realizar un calculo")
                funcion_parametro()
                print("Ya se realizó el calculo")
        return funcion_interna


def suma():
        print(20+10)
def resta():
        print(30-12)
suma()
resta()
```

Decoradores

Otro ejemplo

```
def funcion_decoradora(funcion_parametro):
        def funcion_interna():
                print("A punto de realizar un calculo")
                funcion_parametro()
                print("Ya se realizó el calculo")
        return funcion_interna


@funcion_decoradora
def suma():
        print(20+10)
suma()
def resta():
        print(30-12)
resta()
```

Decoradores

¿Para qué sirven?

Cuando tenemos X cantidad de funciones pero solo queremos agregar funcionabilidades diferentes a Y cantidad de funciones de éstas X. Evitamos ir una por una y repetir el mismo código en cada una.

Podemos crear funciones decoradores que abran conexiones a BD y otras funciones decoradora que cierren la BD.

PROGRAMACIÓN CON PYTHON

GENERADORES

Generadores

Son estructuras que extraen valores de una función y se almacenan en objetos iterables.

Estos objetos los podremos recorrer con un bucle.

Los valores se almacenan de uno en uno en el generador.

Cada vez que el generador almacena un valor, este permanece en un estado de “pausa” hasta que solicita el siguiente. Esta característica del generador se conoce como “suspensión de estado”.

Generadores

Una función tradicional nos devuelve los valores de golpe, un generador lo hace de uno en uno.

Ventajas:

- **Son más eficientes que las funciones tradicionales**
- **Útiles con listas de valores infinitas**
- **En determinados problemas será más útil que un generador devuelva los valores de uno en uno y no todos de una vez.**

Generadores

Sintaxis

```
def funcion_generador():
    yield funcion_generador #construye objeto iterable y los almacena de uno en uno.
```

Generadores

Ejemplo

```
def multiplica(finaliza):
        inicio=1
        while inicio<finaliza:
                yield inicio * 2
                inicio+=1

#muestra_tabla=multiplica(10) #guardando objeto iterable

#print(muestra_tabla)

#for i in muestra_tabla:
        #print(i)
```

Generadores

Ejemplo

```
def multiplica(finaliza):
        inicio=1
        while inicio<finaliza:
                yield inicio * 2
                inicio+=1

muestra_tabla=multiplica(10)

#print(muestra_tabla)

secuencia = 0
for i in muestra_tabla:
        secuencia+=1
        print(2,"*",secuencia,"=",i)
```

RESUMEN DE SESIÓN

GOBIERNO DE
EL SALVADOR

SECRETARÍA DE
INNOVACIÓN