



PYTHON

Conceptos avanzados

1

CONCEPTOS AVANZADOS

List comprehension
(Comprensión de listas)

CONCEPTOS AVANZADOS

- **List comprehension:**
 - Permite crear listas mediante código muy compacto.
 - Alternativa a bucle *for* y a función *map*.
 - Con condición *if*, alternativa a función *filter*.
 - Beneficios y usos:
 - Código compacto.
 - Permite realizar transformaciones.
 - Permite realizar filtros.

CONCEPTOS AVANZADOS

- List comprehension:

- Sintaxis:

- Básica:

- *nombre_lista = [expresión for miembro in iterable]*

- Condicionada:

- *nombre_lista = [expresión for miembro in iterable (if condición)]*

- Condicionada con asignación de valor por defecto:

- *nombre_lista = [expresión if condicion else valor_alternativo for miembro in iterable (if condición)]*

CONCEPTOS AVANZADOS

- List comprehension:
 - Aplicadas a la creación de conjuntos (set):
 - Elimina duplicados.
 - No respeta el orden de creación.
 - Sustituyendo corchetes por llaves:
 - *nombre_conjunto = { expresión for miembro in iterable }*
 - Aplicadas a la creación de diccionarios:
 - Debe incluir la clave.
 - Sustituyendo corchetes por llaves:
 - *nombre_diccionario = { clave: expresión for miembro in iterable }*

CONCEPTOS AVANZADOS

■ List comprehension:

```
>>> lista = [i for i in range(10)]
```

```
>>> lista
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> dias
```

```
('Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes')
```

```
>>> lista = [dia for dia in dias]
```

```
>>> lista
```

```
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes']
```

CONCEPTOS AVANZADOS

- **List comprehension:**

```
>>> def doble(numero):  
...     return numero*2  
...  
>>> lista = [doble(n) for n in range(5)]  
>>> lista  
[0, 2, 4, 6, 8]
```

CONCEPTOS AVANZADOS

- **List comprehension:**

```
>>> lista = [x for x in range(100) if x%2==0]
>>> lista
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40,
42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70,
72, 74, 76, 78, 80,
82, 84, 86, 88, 90, 92, 94, 96, 98]
```


CONCEPTOS AVANZADOS

- **List comprehension:**

```
>>> lista = [x if x%2==0 else "IMPAR" for x in range(100)]
>>> lista
[0, 'IMPAR', 2, 'IMPAR', 4, 'IMPAR', 6, 'IMPAR', 8, 'IMPAR',
10, 'IMPAR', 12, 'IMPAR', 14, 'IMPAR', 16, 'IMPAR', 18,
'IMPAR', 20, 'IMPAR', 22, 'IMPAR', 24, 'IMPAR', 26, 'IMPAR',
28, 'IMPAR', 30, 'IMPAR', 32, 'IMPAR', 34, 'IMPAR', 36, ...]
```

CONCEPTOS AVANZADOS

- **List comprehension:**

```
>>> conjunto = {x if x%2==0 else "IMPAR" for x in range(100)}  
>>> conjunto  
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,  
32, 'IMPAR', 34, 36,  
38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,  
68, 70, 72, 74, 76,  
78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98}
```

CONCEPTOS AVANZADOS

- List comprehension:
 - Utilizando operador walrus :=
 - Permite tratar una asignación como una expresión:
 - `print(lenguaje="Python")` → Generar un error
 - `print(lenguaje:= "Python")` → Escribe Python
 - Permite realizar una asignación del resultado de una operación dentro de un condicional.
 - `if ((x=x*2) == 20):` → Sin walrus → Error
 - `if ((x:=x*2) == 20):` → Con walrus → Ok

CONCEPTOS AVANZADOS

- List comprehension:
 - Utilizando operador walrus :=
 - Ejemplo de uso del operador con una función:

```
def invertir_texto(texto):  
    return texto[::-1]  
print(texto_al_reves:=invertir_texto("Python"))
```

CONCEPTOS AVANZADOS

- List comprehension:
 - Aplicado a list comprehension permite crear listas evaluando resultados de aplicación de operaciones.
 - Utilizando operador walrus :=
 - Sintaxis:
 - *nombre_lista = [expresión for miembro in iterable if (variable := expresión) condición]*

```
>>> def longitud(parametro):  
...     return(len(parametro))  
...  
>>> entrada = ["P1", "Palabra", "Texto", "Idioma"]  
>>> lista = [valor for n in entrada if (valor := longitud(n)) > 5]  
>>> lista  
[7, 6]
```

CONCEPTOS AVANZADOS

Funciones lambda

CONCEPTOS AVANZADOS

- Una función o expresión lambda es una función potencialmente anónima (sin nombre y de un solo uso).
- Alternativa a funciones convencionales con notación más compacta.
- Pueden tener muchos argumentos pero una única expresión.
- No admite *type hints*.
- Sintaxis:
 - `lambda parametro1, parametro2, ... : expresión()` → Sin nombre
 - `nombre = lambda parametro1, parametro2, ... : expresión` → Con nombre

```
def suma(a, b):  
    return a+b  
suma_lambda = lambda a, b : a + b  
print(suma(3,4))  
print(suma_lambda(3,4))  
print((lambda a, b:a+b)(3,4))
```

CONCEPTOS AVANZADOS

- Uso de función lambda anónima:
 - Conjuntamente con filter:

```
datos=[1,2,3,4,5]  
datos_filtrados = filter(lambda dato : dato>2 , datos)  
print(list(datos_filtrados))
```


CONCEPTOS AVANZADOS

Closures

CONCEPTOS AVANZADOS

- **Closures.**
 - Son funciones anidadas.
 - Permiten encapsular funcionalidad.
 - Las funciones internas tienen acceso a las variables de las externas.
 - Permite generar versiones de funciones configuradas para su posterior ejecución.

CONCEPTOS AVANZADOS

■ Closures. Sintaxis:

```
def función_externa():  
    declaración_variable1  
    def función_interna():  
        nonlocal declaración_variable2  
        declaración_variable3  
    función_interna()  
      
    return función_interna
```

CONCEPTOS AVANZADOS

- Closures. Ejemplo:
 - Ejecución inmediata de la función interior.

```
def saludar_exterior1(nombre):  
    mensaje = "Hola " + nombre  
    def saludar_interior1():  
        print(mensaje)  
    saludar_interior1()  
  
saludar_exterior1("Fernando")
```

CONCEPTOS AVANZADOS

- **Closures. Ejemplo:**

- Devolución de la función interior. Los valores de la función exterior se mantienen en memoria.

```
def saludar_exterior2(nombre):  
    mensaje = "Hola " + nombre  
    def saludar_interior2():  
        print(mensaje)  
    return saludar_interior2
```

```
saludar_exterior2("Fernando")()
```

CONCEPTOS AVANZADOS

- Closures. Ejemplo:

- Modificación de variables de la función interior en el ámbito interior: `nonlocal`

```
def saludar_exterior3(nombre):  
    mensaje = "Hola " + nombre  
    def saludar_interior3():  
        nonlocal mensaje #Sin esta declaración, el mensaje del ámbito externo  
no se modifica  
        mensaje = "Mensaje modificado"  
    saludar_interior3()  
    print(mensaje)  
  
saludar_exterior3("Fernando")
```

CONCEPTOS AVANZADOS

Generadores

CONCEPTOS AVANZADOS

- Un generador es una alternativa eficiente a estructuras de datos como listas y tuplas.
- El generador no crea la estructura completa en memoria.
- Los objetos proporcionados por el generador se generan bajo demanda.
- Son funciones que utiliza la sentencia *yield* como generador del retorno.
- Se utiliza la función como iterable.

CONCEPTOS AVANZADOS

■ Ejemplo:

```
def funcion_generacion():  
    for factura in range(NUMERO_ELEMENTOS):  
        yield Factura()  
  
for i in funcion_generacion():  
    pass
```

CONCEPTOS AVANZADOS

- Ejemplo de análisis de uso de memoria utilizando generadores (1/3):

```
import datetime
import sys
```

```
NUMERO_ELEMENTOS = 1_000_000
```

```
class Factura:
    def __init__(self) -> None:
        self.numero = 1000
        self.fecha = datetime.datetime(2020,2,28)
        self.nombre_cliente = "Sociedad Anónima, S.A."
        self.importe = 2_000.38
```

CONCEPTOS AVANZADOS

- Ejemplo de análisis de uso de memoria utilizando generadores (2/3): (**SIN GENERADOR**)

```
def get_facturas():  
    facturas=[]  
    for factura in range(NUMERO_ELEMENTOS):  
        facturas.append(Factura())  
    return facturas  
for factura in get_facturas():  
    pass
```

CONCEPTOS AVANZADOS

- Ejemplo de análisis de uso de memoria utilizando generadores (3/3): (**CON GENERADOR**)

```
def get_facturas_generador():  
    for factura in range(NUMERO_ELEMENTOS):  
        yield Factura()  
for i in get_facturas_generador():  
    pass
```

CONCEPTOS AVANZADOS

- Ejemplo de análisis de uso de memoria utilizando generadores (RESULTADO):

```
print(sys.getsizeof(get_facturas()))→8448728  
print(sys.getsizeof(Factura()))→48
```

CONCEPTOS AVANZADOS

Decoradores

CONCEPTOS AVANZADOS

- Un decorador es una función que permite “envolver” (*decorar*) a otras funciones → Utilizan *closures*.
- Recibe como parámetro una función y devuelve como retorno otra función:

```
def función_decorador(funcion):  
    def envoltorio():  
        #Acciones anteriores  
        funcion()  
        #Acciones posteriores  
    return envoltorio
```

- Se utiliza añadiendo una anotación `@función_decorador` inmediatamente antes de la función que se quiere “decorar”.

CONCEPTOS AVANZADOS

- Ejemplo:

```
#SIN PARÁMETROS
```

```
def asteriscos(funcion):  
    def wrapper():  
        print("*****")  
        funcion()  
        print("*****")  
    return wrapper
```

```
@asteriscos  
def saludador():  
    print("Hola")
```

```
saludador()
```


CONCEPTOS AVANZADOS

- Pueden admitir parámetros.

```
def función_decorador(funcion):
    def envoltorio(*args):
        #Acciones anteriores
        funcion()
        #Acciones posteriores
    return envoltorio
```

CONCEPTOS AVANZADOS

- Ejemplo:

```
#CON PARÁMETROS
```

```
def asteriscos_parametros(funcion):  
    def wrapper(*args):  
        print("*****")  
        print(args)  
        funcion(*args)  
        print("*****")  
    return wrapper
```

```
@asteriscos_parametros  
def saludador_parametros(parametro1, parametro2):  
    print(f"Hola {parametro1} y {parametro2}")
```

```
saludador_parametros("Python", "Java")
```

CONCEPTOS AVANZADOS

- Pueden generar retornos.

```
def función_decorador(funcion):
    def envoltorio(*args):
        #Acciones anteriores
        funcion()
        #Acciones posteriores
        return retorno
    return envoltorio
```

CONCEPTOS AVANZADOS

- Ejemplo:

```
#CON RETORNO
def asteriscos_retorno(funcion):
    def envoltorio(*args):
        print("*****")
        print("Sumando...")
        resultado = funcion(*args)
        print("*****")
        return resultado
    return envoltorio

@asteriscos_retorno
def sumar(s1, s2):
    return s1+s2

print("Resultado",sumar(3,4))
```

CONCEPTOS AVANZADOS

- Pueden lanzar excepciones.

```
def función_decorador(funcion):
    def envoltorio(*args):
        if condición:
            raise Exception
            #Acciones anteriores
            funcion()
            #Acciones posteriores
            return retorno
    return envoltorio
```

CONCEPTOS AVANZADOS

■ Ejemplo:

```
#CON GENERACIÓN DE EXCEPCIONES
def validador(funcion):
    def wrapper(*args):
        if (len(args)==0):
            raise Exception("Sin argumentos")
        resultado = funcion(*args)
        return resultado
    return wrapper

@validador
def calcular_doble(n1):
    return n1*2

try:
    print(calcular_doble())
except Exception as e:
    print(e)
```