Documentación de ejercicios básicos

Variables y operaciones básicas

Objetivo: Declarar variables y realizar operaciones aritméticas.

Código:

```
x = 5
 y = 3
 print("Suma:", x + y)
 print("Producto:", x * y)
```

Resultado esperado:

```
Suma: 8
 Producto: 15
```

Condicionales (if/else)

Objetivo: Evaluar condiciones y mostrar mensajes según el valor.

Código:

```
num = 7
 if num % 2 == 0:
        print("Es par")
 else:
        print("Es impar")
```

Resultado esperado:

```
Es impar
```

Bucles (for)

Objetivo: Iterar sobre un rango de números.

Código:

```
for i in range(1, 6):
    print(i)
```

Resultado esperado:

```
1
2
3
4
5
```

Listas y operaciones

Objetivo: Crear listas y acceder a sus elementos.

Código:

```
frutas = ["manzana", "banana", "cereza"]
print(frutas[0])
print(len(frutas))
```

Resultado esperado:

```
manzana
3
```

Funciones

Objetivo: Definir funciones y llamarlas con argumentos.

Código:

```
def saludar(nombre):
    return f"Hola, {nombre}"

print(saludar("Ana"))
```

Resultado esperado:

Hola, Ana

Ejercicios Avanzados

List Comprehensions (con condición)

Objetivo: Generar la lista de cuadrados solo para números pares.

Código:

```
nums = list(range(10))
 squares_even = [n**2 for n in nums if n % 2 == 0]
 print(squares_even)
```

Resultado esperado:

[0, 4, 16, 36, 64]

Dict Comprehension

Objetivo: Crear un diccionario {n: n2} para n del 1 al 5.**

Código:

```
d = {n: n**2 for n in range(1, 6)}
 print(d)
```

Resultado esperado:

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Generadores (yield)

Objetivo: Crear un generador de números Fibonacci hasta un límite.

Código:

```
def fib(limit):
    a, b = 0, 1
    while a <= limit:
    yield a
    a, b = b, a + b

print(list(fib(20)))
```

Resultado esperado:

```
[0, 1, 1, 2, 3, 5, 8, 13]
```

Decoradores (función simple)

Objetivo: Medir cuántas veces se llama una función.

Código:

```
def counter(fn):
    calls = {"n": 0}
    def wrapped(*args, **kwargs):
    calls["n"] += 1
    result = fn(*args, **kwargs)
    print(f"llamada #{calls['n']}")
    return result
    return wrapped

@counter
def greet(name):
    return f"Hola, {name}"

print(greet("Ana"))
print(greet("Juan"))
```

Resultado esperado:

```
llamada #1
Hola, Ana
llamada #2
Hola, Juan
```

Clases: __init__, métodos y __str__

Objetivo: Definir una clase con constructor y representación legible.

Código:

```
class Person:
    def __init__(self, name, age):
    self.name = name
    self.age = age
    def __str__(self):
    return f"{self.name} ({self.age})"

p = Person("Lucía", 30)
print(p)
```

Resultado esperado:

```
Lucía (30)
```

Excepciones personalizadas

Objetivo: Definir y lanzar una excepción propia.

Código:

```
class InvalidEmail(Exception):
        def __str__(self):
        return "Email inválido"

 def check_email(e):
        if "@" not in e:
        raise InvalidEmail()

 try:
    check_email("usuario#dominio.com")
 except InvalidEmail as ex:
        print(ex)
```

Resultado esperado:

```
Email inválido
```

Casos de Prueba – Python (Regex y Excepciones)

Expresiones Regulares (Regex)

Coincidencia sin distinguir mayúsculas/minúsculas

Objetivo: Verificar cómo se logra la coincidencia de un patrón sin importar si las letras están en mayúsculas o minúsculas.

Código probado (correcto):

```
import re
pattern = r"hello"
re_obj = re.compile(pattern, re.IGNORECASE)
if re_obj.match("Hello"):
    print("Found")
```

Código probado (incorrecto):

```
import re
pattern = r"/hello/i"
re_obj = re.compile(pattern)
if re_obj.search("Hello"):
    print("Found")
```

Resultado esperado: La primera opción funciona, la segunda NO es válida en Python (sintaxis de JavaScript).

División por múltiples delimitadores

Objetivo: Separar un texto usando más de un delimitador a la vez (coma y punto y coma).

Código probado (correcto):

```
import re
string = "Some text; 123, Some text, 123"
pattern = r"[,;]"
print(re.split(pattern, string))
```

Resultado esperado: ['Some text', ' 123', ' Some text', ' 123']

Capturar la coincidencia más corta posible (lazy matching)

Objetivo: Usar un patrón 'lazy' (no codicioso) para obtener coincidencias cortas.

Código probado (correcto):

```
import re
string = "Some text 'a', Some text 'b'"
pattern = r"'(.*?)'"
result = re.findall(pattern, string)
print(result)
```

Resultado esperado: ['a', 'b']

Uso de banderas re.MULTILINE y re.DOTALL

Objetivo: Verificar cómo afectan estas banderas la coincidencia en strings multilínea.

Código probado:

```
string = '''multiline
  string
'''
pattern = r"mul.+ing"
re_obj = re.compile(pattern, re.DOTALL)
print(re_obj.findall(string))
```

Resultado esperado: ['multiline\n string']

Manejo de Excepciones

Captura de excepciones específicas

Objetivo: Capturar correctamente un error específico (IndexError).

Código probado (correcto):

```
my_list = [1, 2, 3]
try:
    my_list[5] = 0
except IndexError:
    print("my_list[5] not found")
```

Resultado esperado: Imprime 'my_list[5] not found'

Un mismo handler para múltiples excepciones

Código probado (correcto):

```
try:
   my_func()
except (ValueError, IndexError):
   print("Exception caught")
```

Resultado esperado: Captura ValueError o IndexError.

Acciones que siempre se ejecutan (finally)

Código probado:

```
try:
   my_func()
except ValueError:
   print("ValueError")
finally:
   print("Finished")
```

Resultado esperado: Siempre se ejecuta 'Finished', ocurra o no excepción.

Relanzar una excepción capturada

Código probado (correcto):

```
try:
   my_func()
except IndexError:
   raise
```

Resultado esperado: Relanza la excepción IndexError.

Excepción personalizada

Código probado:

```
class MyException(Exception):
   def __str__(self):
      return "Custom exception"

try:
   raise MyException()
```

```
except MyException as e:
    print(e)
```

Resultado esperado: Imprime 'Custom exception'