Functional Programming

La programación funcional es un paradigma de programación que trata el cálculo como la evaluación de funciones matemáticas y evita cambiar el estado y los datos mutables. Es un estilo de programación declarativo que se enfoca en expresar qué se debe hacer en lugar de cómo se debe hacer. En la programación funcional, las funciones son ciudadanos de primera clase, lo que significa que **pueden ser tratadas como cualquier otro tipo de dato, como enteros o cadenas.**

La programación funcional enfatiza el uso de funciones puras, que son funciones que no tienen efectos secundarios y siempre producen el mismo resultado para el mismo input. Las funciones puras son deterministas y dependen únicamente de sus parámetros de entrada para calcular el resultado, haciéndolas predecibles y fáciles de razonar.

Algunos conceptos clave en la programación funcional incluyen:

1. **Datos Inmutables**: En la programación funcional, los datos son típicamente inmutables, lo que significa que una vez que se crean, no pueden ser modificados. En cambio, nuevos datos son creados aplicando transformaciones a los datos existentes.

2. **Funciones de Orden Superior**: Los lenguajes de programación funcional soportan funciones de orden superior, que son funciones que pueden tomar otras funciones como argumentos o devolver funciones como resultados. Esto permite un código más abstracto y reutilizable.

3. **Recursión**: La recursión es a menudo utilizada en la programación funcional para resolver problemas dividiéndolos en sub-problemas más pequeños y similares. La recursión reemplaza los bucles en la programación imperativa.

4. **Funciones de Primera Clase**: En los lenguajes de programación funcional, las funciones son ciudadanos de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como argumentos y devueltas como valores.

5. **Map, Filter y Reduce**: Los lenguajes de programación funcional comúnmente usan funciones como `map`, `filter` y `reduce` para procesar colecciones de datos de manera funcional. Estas funciones abstraen los detalles de la iteración y el estado mutable.

Lenguajes de programación funcional, como Haskell, Lisp y Erlang, están diseñados con principios funcionales en mente. Sin embargo, muchos lenguajes de programación mainstream, incluidos Python, JavaScript y Ruby, también soportan la programación funcional en diversos grados. Los desarrolladores pueden aplicar conceptos y técnicas de programación funcional en estos lenguajes para escribir código más conciso, predecible y mantenible.

En este cuaderno, exploraremos conceptos de programación funcional y cómo pueden ser aplicados en la práctica usando Python, un lenguaje que combina paradigmas de programación funcional e imperativa.

```python
# Ejemplo de función pura en Python:
def suma(a, b):
    return a + b

# La función siempre devolverá el mismo resultado para los mismos
inputs
resultado = suma(5, 3)  # resultado = 8

# Ejemplo de función de orden superior:
def aplicar_funcion(func, valor):
    return func(valor)

# Usando una función lambda
resultado = aplicar_funcion(lambda x: x * 2, 10)  # resultado = 20

# Ejemplo de uso de map, filter y reduce:
from functools import reduce

lista = [1, 2, 3, 4, 5]

# map: duplicar cada elemento de la lista
duplicados = list(map(lambda x: x * 2, lista))

# filter: obtener números pares
pares = list(filter(lambda x: x % 2 == 0, lista))

# reduce: sumar todos los elementos
suma_total = reduce(lambda x, y: x * y, lista)

print(f"Duplicados: {duplicados}, Pares: {pares}, Suma Total:
{suma_total}")
```

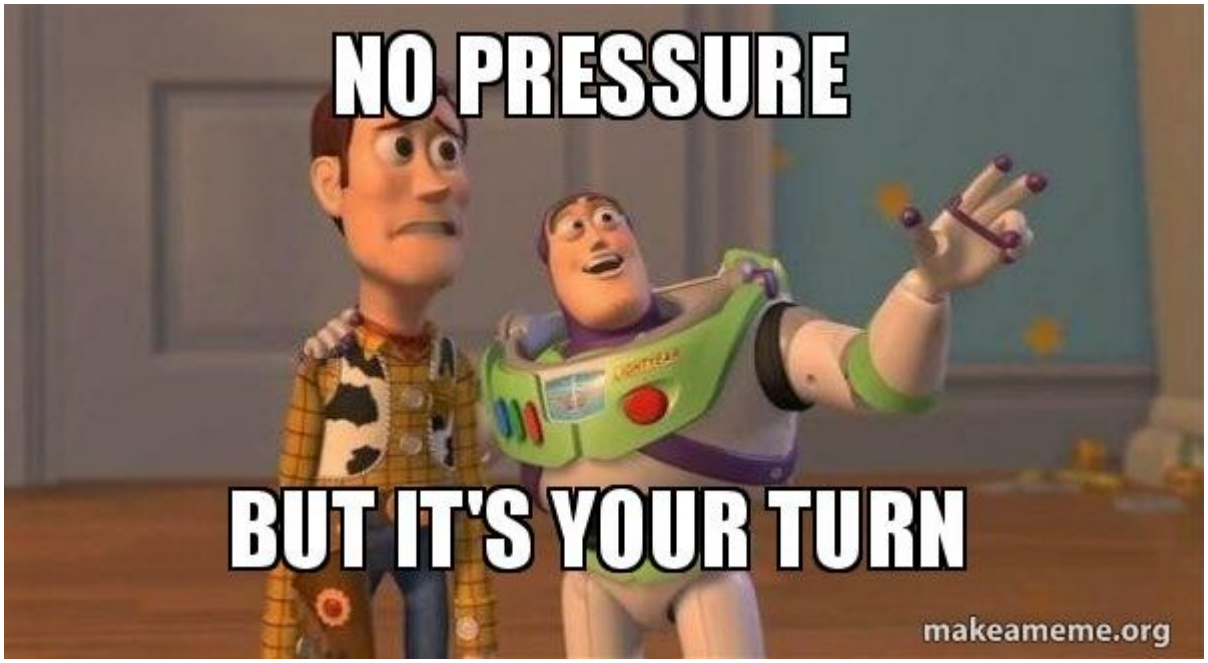
```python
# Salida:
# Duplicados: [2, 4, 6, 8, 10], Pares: [2, 4], Suma Total: 15

from functools import reduce

lista = [1, 2, 3, 4, 5]

# map: duplicar cada elemento de la lista
duplicados = list(map(lambda x: x * 2, lista))

# filter: obtener números pares
pares = list(filter(lambda x: x % 2 == 0, lista))

# reduce: sumar todos los elementos
suma_total = reduce(lambda x, y: x + y, lista)

print(f"Duplicados: {duplicados}, Pares: {pares}, Suma Total:
{suma_total}")
```

```
Duplicados: [2, 4, 6, 8, 10], Pares: [2, 4], Suma Total: 15
```

Tus contribuciones

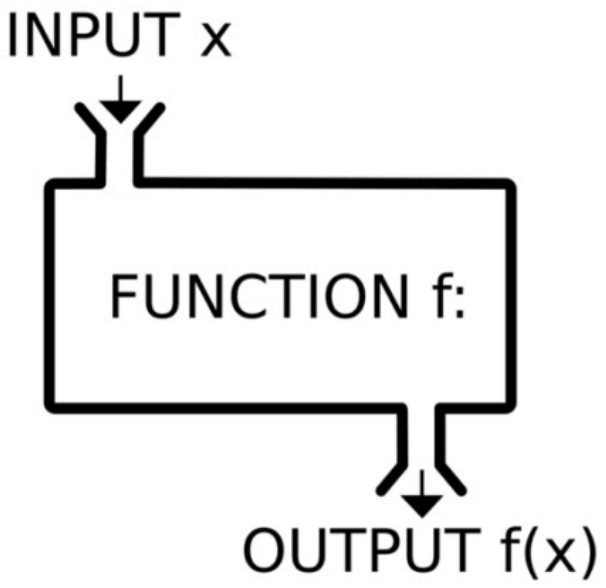
¿Qué piensas sobre las funciones en la programación funcional? ¿Tienes experiencias, ejemplos o preguntas relacionadas con sus propiedades y utilidad? Siéntete libre de contribuir con tus ideas y participar en discusiones con otros aprendices mientras profundizamos más en el mundo de la programación funcional. Tus contribuciones son valiosas y pueden mejorar nuestro entendimiento colectivo de este paradigma de programación.

Una función es/hace: ...



Utilidad de las Funciones: Las funciones son increíblemente útiles en varios escenarios de programación, y ofrecen varias ventajas:

- **Modularidad**: Las funciones encapsulan comportamientos lógicas específicas, lo que facilita organizar y mantener el código. El código modular es más reutilizable y fácil de probar.
- **Abstracción**: Las funciones permiten abstraer operaciones complejas detrás de una interfaz simple. Esta abstracción simplifica el consumo del código y promueve una clara separación de preocupaciones.
- **Reusabilidad**: Las funciones bien diseñadas pueden reutilizarse en diferentes partes de tu base de código o incluso en diferentes proyectos. Esta reutilización reduce el código duplicado y promueve la consistencia.
- **Pruebas**: Las funciones facilitan la escritura de pruebas unitarias. Con funciones puras, puedes predecir la salida para una entrada dada, simplificando el proceso de prueba.
- **Legibilidad**: Las funciones con nombres claros y responsabilidades bien definidas mejoran la legibilidad del código y facilitan que otros desarrolladores (y tu yo futuro) entiendan el código.



¿Qué es la programación funcional?: paradigmas

En el mundo de la programación, un paradigma es un enfoque o estilo fundamental para estructurar el código y resolver problemas. Diferentes lenguajes de programación a menudo se asocian con paradigmas específicos que guían cómo los desarrolladores escriben y organizan su

código. **Python**, como un lenguaje versátil y flexible, a menudo se refiere como un **lenguaje de programación multiparadigma**. Pero, ¿qué significa exactamente esto?

Cuando decimos que Python es un lenguaje de programación multiparadigma, queremos decir que acoge y soporta múltiples estilos de programación o paradigmas. Estos paradigmas son como diferentes lentes a través de los cuales los desarrolladores pueden ver y abordar problemas. Python es conocido por su filosofía, que enfatiza la legibilidad del código y la simplicidad, y acomoda varios paradigmas para ofrecer a los desarrolladores la libertad de elegir el enfoque más adecuado para una tarea dada.

Los Tres Paradigmas Predominantes: Python soporta principalmente tres paradigmas predominantes, cada uno ofreciendo una forma distinta de organizar y estructurar el código:

1. **Programación Imperativa**: La programación imperativa es un paradigma tradicional y comúnmente utilizado. Se centra en describir la secuencia de pasos o instrucciones para alcanzar un objetivo específico. En la programación imperativa, escribes código que declara explícitamente cómo realizar acciones, manipular datos y controlar el flujo del programa. Python permite a los desarrolladores escribir código imperativo cuando es necesario.

```
# Ejemplo de Programación Imperativa

# Supongamos que queremos calcular la suma de los primeros 5 enteros
positivos (1 + 2 + 3 + 4 + 5) usando programación imperativa.

# Inicializar una variable para almacenar la suma
sum_of_numbers = 0

# Usar un bucle para iterar a través de los números
for i in range(1, 6):
    # Añadir el número actual a la suma
    sum_of_numbers += i

# Imprimir el resultado
print("La suma de los primeros 5 enteros positivos es:",
sum_of_numbers)
```

En este ejemplo:

- Comenzamos inicializando una variable `sum_of_numbers` para almacenar la suma.
- Utilizamos un bucle for para iterar a través de los números del 1 al 5. Dentro del bucle:
  - Añadimos el número actual `i` a `sum_of_numbers` usando el operador `+=`. Esta es la parte imperativa donde especificamos explícitamente los pasos para actualizar la suma.
- Finalmente, imprimimos el resultado.

1. **Programación Orientada a Objetos (POO)**: Python sobresale en la programación orientada a objetos. En este paradigma, el código se organiza alrededor de objetos, que

son instancias de clases. Los objetos encapsulan datos y comportamientos, facilitando la modelación de entidades del mundo real y sus interacciones. Python proporciona un soporte robusto para crear y trabajar con clases y objetos.

```
# Definir una clase llamada "Persona"
class Persona:
    # Método constructor para inicializar las propiedades del objeto
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

- Comenzamos definiendo una clase llamada "Persona". Piensa en una clase como un plano para crear objetos. En este caso, estamos creando un plano para representar personas.
- Dentro de la clase, tenemos un método especial llamado `__init__`. Este método es un constructor, lo que significa que se llama automáticamente cuando se crea un nuevo objeto de la clase.
- El método `__init__` toma tres parámetros: `self` (una referencia al objeto que se está creando), `nombre`, y `edad`. Inicializa dos propiedades del objeto: `nombre` y `edad`.

```
    # Método para mostrar información sobre la persona
    def mostrar_info(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")
```

- A continuación, definimos un método llamado `mostrar_info` dentro de la clase. Este método se utiliza para mostrar información sobre la persona.
- El parámetro `self` es una referencia al propio objeto. Nos permite acceder a las propiedades del objeto (en este caso, `nombre` y `edad`) dentro del método.
- Dentro del método `mostrar_info`, usamos la función `print` para mostrar el nombre y la edad de la persona.

```
# Crear dos instancias (objetos) de la clase Persona
persona1 = Persona("Alice", 30)
persona2 = Persona("Bob", 25)
```

- Ahora, creamos dos instancias (o objetos) de la clase `Persona`: `persona1` y `persona2`. Estos objetos representan a personas individuales.
- Proporcionamos valores para las propiedades `nombre` y `edad` al crear cada objeto. Por ejemplo, `persona1` representa a una persona llamada "Alice" que tiene 30 años.

```
# Llamar al método mostrar_info() para cada persona
persona1.mostrar_info()
persona2.mostrar_info()
```

- Finalmente, llamamos al método `mostrar_info` para cada objeto persona. Este método muestra su nombre y edad.

```
# Definir una clase llamada "Persona"
class Persona:
    # Método constructor para inicializar las propiedades del objeto
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    # Método para mostrar información sobre la persona
    def mostrar_info(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")

# Crear dos instancias (objetos) de la clase Persona
persona1 = Persona("Alice", 30)
persona2 = Persona("Bob", 25)

# Llamar al método mostrar_info() para cada persona
persona1.mostrar_info()
persona2.mostrar_info()

class Persona:
    # Método constructor para inicializar las propiedades del objeto
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    # Método para mostrar información sobre la persona
    def mostrar_info(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")

# Crear dos instancias (objetos) de la clase Persona
persona1 = Persona("Alice", 30)
persona2 = Persona("Bob", 25)

# Llamar al método mostrar_info() para cada persona
persona1.mostrar_info()
persona2.mostrar_info()

Nombre: Alice, Edad: 30
Nombre: Bob, Edad: 25
```

1. **Programación Funcional (PF)**: La programación funcional es un paradigma que trata el cálculo como la evaluación de funciones matemáticas y evita cambiar el estado y los datos mutables. Fomenta la inmutabilidad, funciones puras y el uso de funciones de

orden superior. Aunque Python no es un lenguaje puramente funcional como Haskell, ofrece características de programación funcional y puede ser utilizado en un estilo funcional para escribir código limpio y conciso.

Características Clave de la Programación Funcional

- **Inmutabilidad**: En la programación funcional, se prefiere el uso de datos inmutables, lo que significa que una vez que se crea un dato, no puede ser modificado. Esto ayuda a prevenir efectos secundarios no deseados y hace que el código sea más predecible.
- **Funciones Puras**: Las funciones puras son aquellas que no tienen efectos secundarios y siempre producen el mismo resultado dado el mismo conjunto de entradas. Esto significa que no modifican ningún estado externo ni dependen de él, lo que las hace fáciles de razonar y probar.
- **Funciones de Orden Superior**: Este concepto se refiere a funciones que pueden recibir otras funciones como argumentos o devolver funciones como resultado. Esto permite un enfoque más abstracto y modular en la programación.
- **Recursión**: En lugar de utilizar bucles, la programación funcional a menudo utiliza recursión para realizar tareas repetitivas. La recursión implica que una función se llame a sí misma para resolver problemas, dividiéndolos en subproblemas más pequeños.

Ejemplo de Programación Funcional en Python

A continuación, se presenta un ejemplo simple que ilustra varios conceptos de programación funcional:

```
# Definir una función pura que suma dos números
def sumar(x, y):
    return x + y

# Definir una función que utiliza una función de orden superior
def aplicar_funcion(funcion, lista):
    return [funcion(x) for x in lista]

# Crear una lista de números
numeros = [1, 2, 3, 4, 5]

# Aplicar la función 'sumar' a cada elemento de la lista
resultado = aplicar_funcion(lambda x: sumar(x, 10), numeros)

# Imprimir el resultado
print("Resultados después de aplicar la función:", resultado)
```

En este ejemplo:

- **Función Pura**: La función `sumar` toma dos argumentos y devuelve su suma. Es una función pura porque siempre devolverá el mismo resultado para los mismos inputs.

- **Funciones de Orden Superior**: La función `aplicar_funcion` toma otra función como argumento y aplica esa función a cada elemento de una lista, devolviendo una nueva lista con los resultados.
- **Uso de Lambda**: Se utiliza una función lambda para pasar `sumar` como argumento a `aplicar_funcion`, demostrando cómo las funciones pueden ser utilizadas de manera flexible en Python.

Ventajas de la Programación Funcional

- **Código Más Limpio y Conciso**: La programación funcional tiende a producir código más limpio y menos propenso a errores, ya que evita el uso de estados mutables y efectos secundarios.
- **Facilidad de Pruebas**: Las funciones puras son más fáciles de probar, ya que no dependen del estado externo.
- **Mejor Concurrente**: La inmutabilidad y la ausencia de efectos secundarios facilitan la escritura de código que se puede ejecutar en paralelo, lo que es útil en aplicaciones concurrentes.

La programación funcional es un paradigma poderoso que promueve un enfoque diferente para escribir código. Aunque Python no es un lenguaje puramente funcional, sus características funcionales permiten a los desarrolladores adoptar este estilo y beneficiarse de sus ventajas en términos de claridad, modularidad y mantenimiento del código.

Funciones Integradas

Python proporciona un conjunto de funciones y tipos integrados que están disponibles para su uso. Estas funciones abarcan una amplia gama de tareas y operaciones comunes. Puedes encontrar una lista completa de estas funciones en la documentación de Python, organizada en orden alfabético.

Ejemplo: Usando la función `print()`

Una de las funciones integradas más fundamentales en Python es `print()`. Te permite mostrar salida en la consola. Así es como se usa:

```
print("Hello!")
```

Entiende que estas funciones integradas son una parte integral de Python y están implementadas en C, lo que significa que su código fuente no es directamente accesible o inspeccionable desde Python mismo.

Aunque no podemos ver el código para funciones integradas como `print()` usando el módulo `inspect` de Python, todavía podemos usarlas de manera efectiva para realizar una amplia gama de tareas en nuestros programas de Python. La función `print()` se ve así

```
static PyObject *
builtin_print(PyObject *self, PyObject *args, PyObject *kwds)
```

```
{
    static char *kwlist[] = {"sep", "end", "file", 0};
    static PyObject *dummy_args = NULL;
    static PyObject *unicode_newline = NULL, *unicode_space = NULL;
    static PyObject *str_newline = NULL, *str_space = NULL;
    PyObject *newline, *space;
    PyObject *sep = NULL, *end = NULL, *file = NULL;
    int i, err, use_unicode = 0;

    if (dummy_args == NULL) {
        if (!(dummy_args = PyTuple_New(0)))
            return NULL;
    }
    if (str_newline == NULL) {
        str_newline = PyString_FromString("\n");
        if (str_newline == NULL)
            return NULL;
        str_space = PyString_FromString(" ");
        if (str_space == NULL) {
            Py_CLEAR(str_newline);
            return NULL;
        }
#ifdef Py_USING_UNICODE
        unicode_newline = PyUnicode_FromString("\n");
        if (unicode_newline == NULL) {
            Py_CLEAR(str_newline);
            Py_CLEAR(str_space);
            return NULL;
        }
        unicode_space = PyUnicode_FromString(" ");
        if (unicode_space == NULL) {
            Py_CLEAR(str_newline);
            Py_CLEAR(str_space);
            Py_CLEAR(unicode_space);
            return NULL;
        }
#endif
    }
    if (!PyArg_ParseTupleAndKeywords(dummy_args, kwds, "|OOO:print",
                                     kwlist, &sep, &end, &file))
        return NULL;
    if (file == NULL || file == Py_None) {
        file = PySys_GetObject("stdout");
        /* sys.stdout may be None when FILE* stdout isn't connected */
        if (file == Py_None)
            Py_RETURN_NONE;
    }
    if (sep == Py_None) {
        sep = NULL;
```

```
    }
    else if (sep) {
        if (PyUnicode_Check(sep)) {
            use_unicode = 1;
        }
        else if (!PyString_Check(sep)) {
            PyErr_Format(PyExc_TypeError,
                         "sep must be None, str or unicode, not
%.200s",
                         sep->ob_type->tp_name);
            return NULL;
        }
    }
    if (end == Py_None)
        end = NULL;
    else if (end) {
        if (PyUnicode_Check(end)) {
            use_unicode = 1;
        }
        else if (!PyString_Check(end)) {
            PyErr_Format(PyExc_TypeError,
                         "end must be None, str or unicode, not
%.200s",
                         end->ob_type->tp_name);
            return NULL;
        }
    }

    if (!use_unicode) {
        for (i = 0; i < PyTuple_Size(args); i++) {
            if (PyUnicode_Check(PyTuple_GET_ITEM(args, i))) {
                use_unicode = 1;
                break;
            }
        }
    }
    if (use_unicode) {
        newline = unicode_newline;
        space = unicode_space;
    }
    else {
        newline = str_newline;
        space = str_space;
    }

    for (i = 0; i < PyTuple_Size(args); i++) {
        if (i > 0) {
            if (sep == NULL)
                err = PyFile_WriteObject(space, file,
```

```
                                        Py_PRINT_RAW);
            else
                err = PyFile_WriteObject(sep, file,
                                        Py_PRINT_RAW);
            if (err)
                return NULL;
        }
        err = PyFile_WriteObject(PyTuple_GetItem(args, i), file,
                                 Py_PRINT_RAW);
        if (err)
            return NULL;
    }

    if (end == NULL)
        err = PyFile_WriteObject(newline, file, Py_PRINT_RAW);
    else
        err = PyFile_WriteObject(end, file, Py_PRINT_RAW);
    if (err)
        return NULL;

    Py_RETURN_NONE;
}
```

Todo el código anterior para decir hola! ...

```
print("hola!")
```

Las funciones integradas en Python son herramientas esenciales que permiten a los desarrolladores realizar tareas comunes de manera rápida y eficiente. Comprender cómo funcionan estas funciones, incluso a un nivel superficial, puede ayudar a los desarrolladores a utilizarlas de manera más efectiva en sus proyectos. Siempre es recomendable revisar la documentación oficial para obtener información más detallada y ejemplos de uso.

Ejercicio: Exploración de Paradigmas

Contexto: Python es un lenguaje de programación versátil que admite múltiples paradigmas de programación: imperativo, orientado a objetos y funcional. Cada paradigma ofrece un enfoque diferente para resolver problemas. En este ejercicio, explorarás estos paradigmas trabajando en una tarea común: procesar una lista de números.

Tarea: Se te ha dado una lista de números: `[1, 2, 3, 4, 5]`.

1. **Paradigma Imperativo (Programación Procedural):** Escribe un script de Python utilizando programación imperativa para calcular la suma de todos los números en la lista e imprimir el resultado.

2. **Paradigma Orientado a Objetos:** Crea una clase de Python llamada `ProcesadorDeNumeros`. Esta clase debe tener un método llamado `calcular_suma`

que calcule la suma de los números en la lista y devuelva el resultado. Instancia la clase, llama al método e imprime el resultado.

3. **Paradigma Funcional:** Escribe una función de Python llamada `calcular_suma_funcional` utilizando un enfoque de programación funcional. Esta función debe tomar la lista de números como argumento y devolver la suma. Utiliza la función `reduce` del módulo `functools` para lograr esto. Llama a la función e imprime el resultado.

Consejos:

- Para el paradigma funcional, necesitarás importar el módulo `functools` y utilizar la función `reduce`.
- Puedes crear una variable de instancia en el paradigma orientado a objetos para almacenar la lista de números.

Resultado Esperado: Deberías tener tres implementaciones separadas, cada una imprimiendo el mismo resultado: la suma de los números en la lista `[1, 2, 3, 4, 5]`.

---

1. Implementación del Paradigma Imperativo

```
Definir la lista de números
numbers = [1, 2, 3, 4, 5]
sum_result = 0

# Calcular la suma utilizando un bucle
for number in numbers:
    sum_result += number

# Imprimir el resultado
print("Imperative Paradigm Result:", sum_result)
```

---

2. Implementación del Paradigma Orientado a Objetos

```
Definir la clase ProcesadorDeNumeros
class ProcesadorDeNumeros:
    # Inicializador de la clase
    def __init__(self, numeros):
        self.numeros = numeros

    # Método para calcular la suma
    def calcular_suma(self):
        return sum(self.numeros)

# Instanciar la clase y llamar al método
number_processor = ProcesadorDeNumeros([1, 2, 3, 4, 5])
```

```
result_object_oriented = number_processor.calcular_suma()
print("Object-Oriented Paradigm Result:", result_object_oriented)
```

---

3. Implementación del Paradigma Funcional

```
from functools import reduce

# Definir la función para calcular la suma de manera funcional
def calcular_suma_funcional(numeros):
    return reduce(lambda x, y: x + y, numeros)

# Llamar a la función
numbers = [1, 2, 3, 4, 5]
result_functional = calcular_suma_funcional(numbers)
print("Functional Paradigm Result:", result_functional)

# Definir la lista de números
numbers = [1, 2, 3, 4, 5]
sum_result = 0

# Calcular la suma utilizando un bucle
for number in numbers:
    sum_result += number

# Imprimir el resultado
print("Imperative Paradigm Result:", sum_result)
```

```
Imperative Paradigm Result: 15
```

```
# Definir la clase ProcesadorDeNumeros
class ProcesadorDeNumeros:
    # Inicializador de la clase
    def __init__(self, numeros):

        self.numeros = numeros

    # Método para calcular la suma
    def calcular_suma(self):
        return sum(self.numeros)

# Instanciar la clase y llamar al método
number_processor = ProcesadorDeNumeros([1, 2, 3, 4, 5])
result_object_oriented = number_processor.calcular_suma()
print("Object-Oriented Paradigm Result:", result_object_oriented)
```

```
Object-Oriented Paradigm Result: 15
```

```
from functools import reduce
```

```python
# Definir la función para calcular la suma de manera funcional
def calcular_suma_funcional(numeros):
    return reduce(lambda x, y: x + y, numeros)

# Llamar a la función
numbers = [1, 2, 3, 4, 5]
result_functional = calcular_suma_funcional(numbers)
print("Functional Paradigm Result:", result_functional)

Functional Paradigm Result: 15
```

FUNCIONES DEFINIDAS POR EL USUARIO

En Python, puedes crear tus propias funciones para encapsular un conjunto específico de instrucciones y reutilizarlas en tu código. Estas funciones se denominan funciones definidas por el usuario. Desglosemos los componentes clave de estas funciones:

- **`def`**: La palabra clave `def` se utiliza para definir una nueva función. Esta se sigue del nombre de la función, paréntesis y dos puntos. Por ejemplo, `def mi_funcion():` define una función llamada `mi_funcion`.
- **`nombre`**: El nombre de la función es elegido por el programador y debe ser descriptivo sobre su propósito. En el ejemplo anterior, el nombre de la función es `mi_funcion`.
- **`params`**: Dentro de los paréntesis, puedes especificar parámetros (también conocidos como argumentos) que la función aceptará. Los parámetros actúan como marcadores de posición para los valores que se pasan a la función al llamarla. Por ejemplo, `def saludar(nombre):` define una función que acepta un parámetro llamado `nombre`.
- **`args`**: Los argumentos son los valores reales que se proporcionan al llamar a una función. Por ejemplo, al llamar a `saludar("Alice")`, `"Alice"` es el argumento pasado al parámetro `nombre`.
- **`return`**: La instrucción `return` se utiliza para especificar lo que debe devolver la función. Las funciones pueden realizar cálculos o procesos y luego devolver un resultado. Por ejemplo, `def sumar(x, y): return x + y` define una función que suma dos números y devuelve el resultado.
- **`docstrings`**: Los docstrings (cadenas de documentación) proporcionan una descripción de lo que hace la función. Están encerrados en comillas triples (simples o dobles) y se colocan inmediatamente después de la definición de la función. Los docstrings ayudan a otros programadores (y a ti mismo) a comprender el propósito y uso de la función. Por ejemplo:

```python
    def saludar(nombre: str) -> str:
        """
```

```
        Esta función saluda a la persona pasada como parámetro.
        """
        return f"Hola, {nombre}!"
```

- **`source code`**: El código fuente de la función contiene las instrucciones que la función ejecuta. Este está indentado debajo de la definición de la función y se ejecuta cuando se llama.

Sintaxis para DEFINIR una función

Para definir una función en Python, utiliza la siguiente sintaxis:

```
def nombre_funcion(parametros_entrada):
    # El colon y la indentación indican que estamos definiendo una función.
    # Dentro de la función, se realiza alguna acción o se lleva a cabo una serie de operaciones.
    realizar_una_accion
    return() # La instrucción return especifica el valor que se devolverá de la función.
```

En esta sintaxis:

- `def` es la palabra clave utilizada para definir una función.
- `nombre_funcion` es el nombre que le das a tu función. Debe seguir las convenciones de nombrado de Python y ser descriptivo de su propósito.
- `parametros_entrada` son los parámetros (entradas) que la función espera. Están encerrados en paréntesis y separados por comas si hay múltiples parámetros.
- El colon `:` y la indentación (espacio en blanco) se utilizan para definir el bloque de código de la función. Todo lo indentado bajo la definición de la función es parte de su cuerpo.
- Dentro de la función, escribes el código para realizar acciones o cálculos específicos.
- La instrucción `return` especifica qué valor debe producir la función como salida, finalizando su ejecución y enviando el resultado de vuelta al llamador.

Sintaxis para LLAMAR a una función

Para llamar (invocar) una función en Python, utiliza la siguiente sintaxis:

`nombre_de_la_funcion(argumento1, argumento2, ...)`

- `nombre_de_la_funcion` es el nombre de la función que deseas llamar.
- `argumento1, argumento2, ...` son los valores o expresiones que se pasan como argumentos a la función. Estos son los inputs que la función utilizará.

Si deseas guardar el resultado devuelto por la función, puedes asignarlo a una variable de esta manera:

`resultado = nombre_de_la_funcion(argumento1, argumento2, ...)`

En este caso, `resultado` contendrá el valor devuelto por la función y podrás usarlo en tu código según sea necesario.

def

La declaración `def` sirve como el punto de partida para crear tus funciones personalizadas. Especifica el nombre de la función, los parámetros de entrada que espera y el bloque de código que constituye el cuerpo de la función. Las funciones definidas con `def` pueden ser llamadas múltiples veces a lo largo de tu programa, haciendo tu código más modular, legible y eficiente.

- Función simple

```
def addition(a, b):
    result = a * b
    # print(f"Hello! The result is {result}")
    return result
```

Llamando a la función

```
addition(10, 2)
```

- Una función, limpia y legible

```
def suma(a: int, b: int) -> int:
    """
    Esta función realiza la suma de dos números.

    Argumentos:
        a (int): El primer número.
        b (int): El segundo número.

    Devuelve:
        int: El resultado de la suma.
    """
    resultado = a + b
    return resultado
```

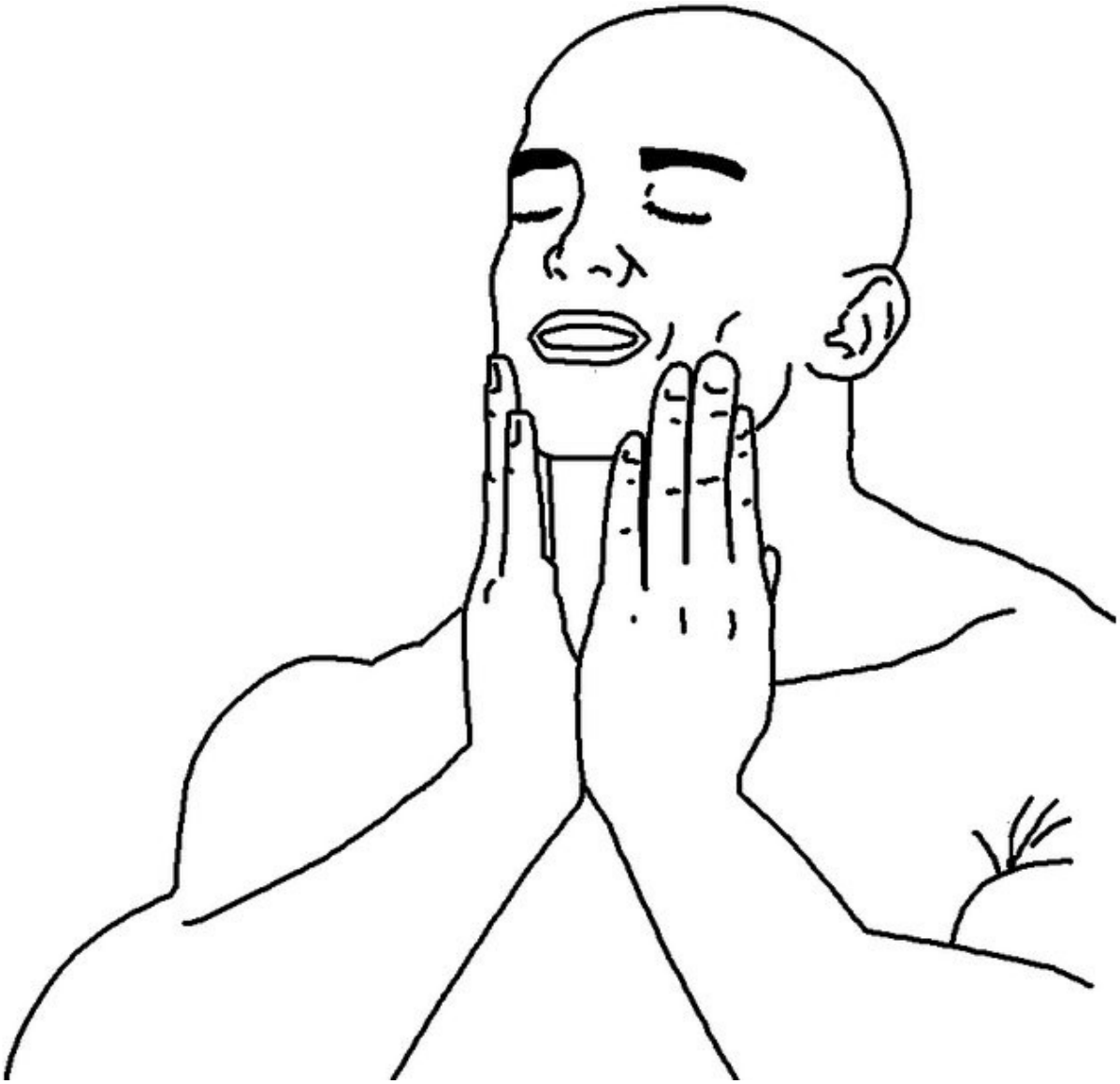
- Llamando a la función

```
resultado = suma(10, 2)
print(f"El resultado es {resultado}")
```

En esta versión revisada:

- Hemos añadido docstrings claros e informativos que describen qué hace la función, qué argumentos acepta y qué devuelve.
- Hemos utilizado anotaciones de tipo de datos (`a: int`, `b: int`, `-> int`) para indicar los tipos esperados para los parámetros de la función y los valores de retorno.

- Hemos proporcionado un nombre de función significativo (`suma`) que representa con precisión su propósito.
- Hemos eliminado el comentario de la instrucción `print` para mantener la función enfocada en su tarea principal de suma.



nombre

La declaración del nombre de una función en Python es un aspecto crucial de la legibilidad y comprensión del código. Elegir nombres descriptivos y significativos para tus funciones mejora la claridad de tu código, facilitando tanto para ti como para otros comprender su propósito y funcionalidad.

Ejemplo:

```
def adder():
    return
```

Sumador de ¿qué? Quizás, si somos más claros...

```
def number_adder():
    return
```

Ahora, parece ser más claro. Incluso podemos ser más descriptivos (¡ayuda a los demás!):

```
def sumador_de_numeros():
    """
    Esta función utiliza snake_case para su nombre, lo cual es una
convención en Python.
    El snake_case hace que los nombres de las funciones sean legibles
y fáciles de entender al separar las palabras con guiones bajos.
    """
    return
```

Por otro lado...

```
# No recomendado -> JavaScript
def CamelCase():  # CamelCase
    """
    En Python, no se recomienda usar CamelCase para los nombres de las
funciones.
    CamelCase se utiliza típicamente en lenguajes como JavaScript,
mientras que Python
    utiliza convencionalmente snake_case para los nombres de las
funciones para mantener
    la consistencia y legibilidad.
    """
    return
```

parámetros

Los parámetros de las funciones en Python se utilizan para pasar valores o datos a una función cuando se llama. Estos parámetros actúan como marcadores de posición que reciben los valores de entrada, permitiendo que las funciones trabajen con datos específicos sin codificarlos directamente. Las funciones de Python pueden aceptar cero o más parámetros, y cada parámetro puede tener un valor predeterminado o ser especificado como obligatorio. Los parámetros hacen que las funciones sean versátiles y reutilizables, permitiéndoles realizar diferentes acciones basadas en la entrada proporcionada.

En Python, los parámetros se definen dentro de los paréntesis () al declarar una función, y sus nombres sirven como variables dentro del bloque de código de la función. Exploremos cómo trabajar con parámetros de funciones en Python.

```
def greetings():
    return "Hello!"

greetings()
```

Ahora definimos dos parámetros, `(param_1, param_2)`.

```
def greetings(other_person, myself):
    return f"Hello {other_person}! My name is {myself}"
```

¡Así es como me gusta!

```
def saludos(otra_persona: str, yo_mismo: str) -> str:
    """
    Esta función saluda a otra persona y se presenta.

    Argumentos:
        otra_persona (str): El nombre de la otra persona.
        yo_mismo (str): El nombre de la persona que se presenta.

    Devuelve:
        str: Un mensaje de saludo.

    Ejemplo:
        >>> saludos("Alice", "Bob")
        'Hola Alice! Mi nombre es Bob'
    """
    return f"Hola {otra_persona}! Mi nombre es {yo_mismo}"
```

argumentos

En Python, los argumentos de las funciones son los valores o variables que pasamos a una función cuando la llamamos. Estos argumentos son esenciales para que la función realice su tarea. Python ofrece flexibilidad en cómo podemos trabajar con los argumentos de las funciones, permitiéndonos pasar valores de diversas maneras, incluidos argumentos posicionales, argumentos de palabra clave, valores predeterminados y más. Entender cómo trabajar con argumentos de funciones es crucial para crear funciones versátiles y reutilizables.

```
def greetings(other_person, myself):
    return f"Hello {other_person}! My name is {myself}"

greetings("Santi", "Clara")

greetings("Laura", "Albert", "Santi")
```

Recapitulación:

- **Parámetros**: Son marcadores de posición formales definidos en la declaración de la función. Sirven como nombres simbólicos para los valores que una función espera

recibir y utilizar dentro de su cuerpo. Los parámetros son como variables esperando ser llenadas con datos reales.

- **Argumentos**: Cuando llamamos a una función, proporcionamos datos reales o valores llamados argumentos. Estos argumentos se enumeran dentro de paréntesis después del nombre de la función durante la llamada a la función. Cuando se ejecuta la función, estos argumentos se asignan a los parámetros correspondientes dentro del cuerpo de la función.

Para ilustrar, piensa en los parámetros como variables "imaginarias" en la definición de la función: tienen nombres pero no valores hasta que proporcionas los argumentos. Por otro lado, los argumentos son los datos tangibles y reales que suministras al invocar la función. Llenan los marcadores de posición definidos por los parámetros.

Comprender la distinción entre parámetros y argumentos es fundamental al trabajar con funciones en la programación.

Parámetros por defecto

En Python, las funciones pueden tener parámetros con valores por defecto. Estos parámetros proporcionan una manera de definir un valor que se utilizará si el argumento correspondiente no se proporciona al llamar a la función. Esta característica permite que ciertos argumentos de la función sean opcionales, mejorando la flexibilidad y usabilidad de tus funciones. Los parámetros por defecto son especialmente útiles cuando deseas proporcionar valores predeterminados razonables o comunes, al mismo tiempo que permites a los usuarios personalizarlos según sea necesario.

Ejemplo de uso de parámetros por defecto

```
def greetings_default(myself, someone_else, language="en"):
    if language == "en":
        return f"Hello {someone_else}! I'm {myself}"
    else:
        return f"¡Hola {someone_else}, soy {myself}!"
```

Uso de la función

```
# '''python
# Llamadas a la función con diferentes argumentos
greetings_default("Alfons", "Santi", language="en")
greetings_default("Alfons", "Santi")
greetings_default("Alfons", "Santi", "es")
# '''
```

Otra función con parámetros por defecto

```
def calculation(num1, num2, operation="+"):
    # Sumar dos números
    if operation == "+":
```

```
        return num1 + num2
    # Multiplicar dos números
    elif operation == "*":
        return num1 * num2
    # Dividir dos números
    elif operation == "/":
        return num1 / num2
    else:
        return "Cuidado. Esto no está permitido"
```

Uso de la función con diferentes operaciones

```
# Llamadas a la función con diferentes operaciones
calculation(10, 4, operation="+")
calculation(10, 4)  # Usando valores predeterminados
calculation(10, 4, "*")
calculation(10, 4, "/")
calculation(10, 4, "**")  # Operación no permitida
calculation(10, 4, "//")  # Operación no permitida
```

Argumentos posicionales y argumentos de palabra clave

En las funciones de Python, puedes definir parámetros que aceptan valores ya sea a través de su posición en la llamada a la función (argumentos posicionales) o especificando los nombres de los parámetros explícitamente (argumentos de palabra clave). Esta flexibilidad es útil para crear funciones más versátiles y comprensibles.

```
def suma(num_1, algun_nombre=conunvalor):
```

Argumentos Posicionales

Los argumentos posicionales se pasan a una función basándose en su posición en la llamada a la función. El orden de los argumentos es fundamental; los valores se asignan a los parámetros en el mismo orden en que se definen.

Ejemplo:

```
def suma(num_1, num_2):
    return num_1 + num_2

resultado = suma(5, 3)  # num_1 recibe 5, num_2 recibe 3
print(f"El resultado de la suma es: {resultado}")  # Salida: El
resultado de la suma es: 8
```

En este ejemplo, `num_1` toma el valor `5`, y `num_2` toma el valor `3`. Si intentas cambiar el orden, como `suma(3, 5)`, obtendrás un resultado diferente (8 en lugar de 8, pero asignados a diferentes parámetros).

Argumentos de Palabra Clave

Los argumentos de palabra clave se pasan especificando los nombres de los parámetros junto con sus valores en la llamada a la función. Esto permite pasar argumentos en cualquier orden, lo que puede mejorar la claridad del código, especialmente si hay muchos parámetros.

Ejemplo:

```
def saludo(nombre, edad=30):
    return f"Hola, {nombre}! Tienes {edad} años."

mensaje = saludo(nombre="Alice", edad=25)
print(mensaje)  # Salida: Hola, Alice! Tienes 25 años.

mensaje2 = saludo(edad=40, nombre="Bob")  # Se puede cambiar el orden
print(mensaje2)  # Salida: Hola, Bob! Tienes 40 años.
```

En este ejemplo, se puede ver que `nombre` y `edad` se pueden especificar en cualquier orden gracias a los argumentos de palabra clave. Si no se proporciona el argumento `edad`, la función utilizará el valor predeterminado de `30`.

Ventajas de Usar Argumentos Posicionales y de Palabra Clave

- **Flexibilidad**: Los argumentos de palabra clave permiten a los usuarios llamar a funciones sin preocuparse por el orden de los parámetros.
- **Legibilidad**: El uso de argumentos de palabra clave puede hacer que el código sea más fácil de entender al proporcionar contexto sobre qué parámetros están siendo pasados.
- **Valores Predeterminados**: Los argumentos de palabra clave con valores predeterminados permiten a los desarrolladores proporcionar comportamientos útiles por defecto sin forzar a los usuarios a siempre especificar un argumento.

Consideraciones

- Al mezclar argumentos posicionales y de palabra clave, los argumentos posicionales deben ir siempre antes de los argumentos de palabra clave en la definición de la función.
- Si intentas pasar un argumento posicional después de haber usado un argumento de palabra clave, Python lanzará un error de sintaxis. Por ejemplo, la siguiente llamada a la función sería incorrecta:

```
saludo(25, nombre="Alice")  # Esto causará un error
```

En conclusión, comprender cómo funcionan los argumentos posicionales y de palabra clave es esencial para escribir funciones efectivas y legibles en Python.

`*args`

En Python, el parámetro especial `*args` en una función se utiliza para pasar opcionalmente un número variable de argumentos **posicionales**. Esto permite que la función acepte cualquier cantidad de argumentos, los cuales se recopilan en una tupla. Esto es especialmente útil cuando no sabemos de antemano cuántos argumentos se pasarán a la función.

Para ilustrar el uso de `*args`, vamos a crear una función que calcula el costo total de un carrito de compras. Esta función tomará los precios de los artículos como argumentos posicionales y aplicará un descuento utilizando un argumento de palabra clave.

Ejemplo práctico de `*args`:

Primero, echemos un vistazo a una función simple que suma tres números:

```
def addition(a, b, c):
    return a + b + c

print(addition(10, 4, 10))  # Salida: 24
```

En este caso, la función `addition` espera exactamente tres argumentos. Sin embargo, a menudo queremos funciones que sean más flexibles. Aquí es donde `*args` resulta útil.

Definiendo la Función con `*args`

Ahora vamos a definir la función `calcular_costo_total` utilizando `*args` para aceptar un número variable de precios de artículos:

```
def calcular_costo_total(descuento=0, *articulos):
    """
    Calcula el costo total de los artículos en un carrito de compras,
    considerando un descuento opcional.

    Argumentos:
        descuento (float): Un porcentaje de descuento (0-100) para
aplicar
        al costo total. Por defecto es 0.
        *articulos (float): Argumentos posicionales que representan
los
        precios de los artículos individuales.

    Devuelve:
        float: El costo total después de aplicar el descuento.

    Ejemplo:
        >>> calcular_costo_total(10, 20, 30, 40)  # Aplicando un 10%
        de descuento a los artículos
        108.0
    """
    costo_total = sum(articulos)  # Sumar todos los precios
    costo_con_descuento = costo_total * (1 - descuento / 100)  #
Aplicar el descuento
    return costo_con_descuento
```

En este ejemplo:

- Definimos una función `calcular_costo_total` que acepta un descuento opcional (un porcentaje) como argumento de palabra clave y cualquier número de precios de artículos como argumentos posicionales usando `*articulos`.
- Calculamos el costo total de los artículos sumando sus precios utilizando la función `sum()`.
- Aplicamos el descuento si se proporciona, y la función devuelve el costo final.

Ejemplos de Uso

Ahora, probemos nuestra función `calcular_costo_total` con diferentes conjuntos de precios y descuentos:

```
# Calcular el costo total de los artículos con un descuento del 10%
total1 = calcular_costo_total(10, 20, 30, 40)
print("Total Cost (10% Discount):", total1)  # Salida: 108.0

# Calcular el costo total de los artículos con un descuento del 20%
total2 = calcular_costo_total(20, 50, 75, 100, 500)
print("Total Cost (20% Discount):", total2)  # Salida: 592.0
```

Al llamar a `calcular_costo_total`, pasamos los precios de los artículos como argumentos posicionales, y el descuento como argumento de palabra clave. La función es capaz de manejar cualquier número de precios, lo que la hace muy versátil.

Resumen

- `*args` permite que las funciones acepten un número variable de argumentos posicionales.
- Se recopilan en una tupla dentro de la función, lo que permite su manipulación fácil.
- Esto es útil en situaciones donde se requiere flexibilidad en el número de argumentos, como en cálculos de precios, listados de nombres, entre otros.

**kwargs

En Python, `**kwargs` es un parámetro especial en una función que te permite pasar un número variable de argumentos de palabra clave. Estos argumentos se pasan como un diccionario, lo que te permite manejar argumentos nombrados de manera flexible dentro de tu función. Esto es especialmente útil cuando deseas proporcionar parámetros opcionales o manejar una variedad de argumentos de palabra clave en una sola función. Vamos a explorar cómo funciona `**kwargs` y cómo usarlo eficazmente.

Ejemplo 1: Usando `**kwargs` para Argumentos de Palabra Clave

A continuación, se muestra una función que muestra la información del usuario utilizando `**kwargs` para recibir argumentos adicionales:

```
from typing import Dict, Any

def mostrar_informacion_usuario(nombre: str, edad: int, **kwargs: Any)
-> str:
```

```
    """
    Muestra la información del usuario.

    Argumentos:
        nombre (str): El nombre del usuario.
        edad (int): La edad del usuario.
        **kwargs: Argumentos de palabra clave adicionales para la
información
        del usuario (por ejemplo, ciudad, país).

    Devuelve:
        str: Una cadena de información del usuario formateada.

    Ejemplo:
        >>> mostrar_informacion_usuario("Alice", 30, ciudad="Nueva
York", pais="EE.UU.")
        'Nombre: Alice, Edad: 30, Ciudad: Nueva York, País: EE.UU.'
    """
    info_usuario = {
        "Nombre": nombre,
        "Edad": edad,
        **kwargs,  # Agregando los argumentos de palabra clave
adicionales
    }
    cadena_info = ", ".join(f"{clave}: {valor}" for clave, valor in
info_usuario.items())
    return cadena_info
```

Uso del Ejemplo 1:

```
# Ejemplo de uso:
user_info = mostrar_informacion_usuario("Alice", 30, ciudad="Nueva
York", pais="EE.UU.")
print(user_info)  # Salida: Nombre: Alice, Edad: 30, Ciudad: Nueva
York, País: EE.UU.
```

Ejemplo 2: Usando `*args` y `**kwargs` Juntos

A continuación, crearemos una función que combina `*args` y `**kwargs` para realizar operaciones en números:

```
def procesar_datos(*args: int, **kwargs: Dict[str, Any]) -> int:
    """
    Procesa datos realizando operaciones en argumentos posicionales
    y utilizando argumentos de palabra clave.

    Argumentos:
        *args (int): Número variable de valores enteros para el
procesamiento.
```

```
        **kwargs: Argumentos de palabra clave para especificar la
operación
        ('add' o 'multiply').

    Devuelve:
        int: El resultado del procesamiento de datos.

    Ejemplo:
        >>> procesar_datos(2, 3, 4, operacion='add')
        9
        >>> procesar_datos(2, 3, 4, operacion='multiply')
        24
    """
    operacion = kwargs.get('operacion', 'add')  # Recuperar la
operación
    if operacion == 'add':
        resultado = sum(args)  # Sumar todos los argumentos
    elif operacion == 'multiply':
        resultado = 1
        for valor in args:
            resultado *= valor  # Multiplicar todos los argumentos
    else:
        raise ValueError("Operación especificada inválida.")

    return resultado
```

Uso del Ejemplo 2:

```
# Resultados de las operaciones
result1 = procesar_datos(2, 3, 4, operacion='add')
result2 = procesar_datos(2, 3, 4, operacion='multiply')
print(result1)  # Salida: 9
print(result2)  # Salida: 24
```

Orden de los Parámetros

En Python, `**kwargs` es un nombre comúnmente utilizado para el parámetro que recopila argumentos de palabra clave en una función. Aunque puedes usar un nombre diferente (por ejemplo, `**argumentos_personalizados`), es una convención ampliamente aceptada usar `**kwargs` para hacer tu código más legible y mantenible.

```
# Definición de una función con `**kwargs`
def ejemplo(arg1, arg2, *args, **kwargs):
    pass
```

Recuerda que `**kwargs` siempre se representa como un diccionario en Python. Es una sintaxis especial que te permite pasar un número variable de argumentos de palabra clave a una función, y esos argumentos de palabra clave se recopilan y almacenan como pares clave-valor en un diccionario dentro de la función.

```python
from typing import Dict, Any

def mostrar_informacion_usuario(nombre: str, edad: int, **kwargs: Any)
-> str:
    """
    Muestra la información del usuario.

    Argumentos:
        nombre (str): El nombre del usuario.
        edad (int): La edad del usuario.
        **kwargs: Argumentos de palabra clave adicionales para la
información
        del usuario (por ejemplo, ciudad, país).

    Devuelve:
        str: Una cadena de información del usuario formateada.

    Ejemplo:
        >>> mostrar_informacion_usuario("Alice", 30, ciudad="Nueva
York", pais="EE.UU.")
        'Nombre: Alice, Edad: 30, Ciudad: Nueva York, País: EE.UU.'
    """
    info_usuario = {
        "Nombre": nombre,
        "Edad": edad,
        **kwargs,  # Agregando los argumentos de palabra clave
adicionales
    }
    cadena_info = ", ".join(f"{clave}: {valor}" for clave, valor in
info_usuario.items())
    return cadena_info

# Ejemplo de uso:
user_info = mostrar_informacion_usuario("Alice", 3, ciudad="Nueva
York", pais="EE.UU.")
print(user_info)  # Salida: Nombre: Alice, Edad: 30, Ciudad: Nueva
York, País: EE.UU.
```

```
Nombre: Alice, Edad: 3, ciudad: Nueva York, pais: EE.UU.
```

Ejercicio: Generador de Informes de Ventas

Escenario Empresarial: Imagina que trabajas para una empresa minorista que vende gadgets electrónicos. Tu tarea es crear un programa en Python que genere informes de ventas basados en varios parámetros. Necesitas definir una función que llame a varias funciones auxiliares para procesar y devolver un resumen detallado de los datos de ventas.

Instrucciones:

1. Define una función llamada `generate_sales_report` con los siguientes parámetros:
   - `store_name` (str): El nombre de la tienda minorista.
   - `date` (str): La fecha del informe de ventas.
   - `*sales_data` (tuplas): Un número variable de tuplas donde cada tupla representa la venta de un gadget electrónico y contiene los siguientes elementos:
     - Nombre del artículo (str)
     - Precio unitario (float)
     - Cantidad vendida (int)
2. Utiliza **anotaciones de tipo de datos** para especificar los tipos de datos esperados para cada parámetro.
3. Incluye un **docstring** que explique qué hace la función y describa cada parámetro.
4. Implementa la función `generate_sales_report` de manera que llame a las siguientes funciones auxiliares para calcular y devolver la siguiente información:
   - **`calculate_total_sales(sales_data: List[Tuple[str, float, int]], include_tax: bool) -> float`**: Calcula el total de ventas de todos los artículos y, opcionalmente, incluye un impuesto del 10%.
   - **`calculate_average_price(sales_data: List[Tuple[str, float, int]]) -> float`**: Calcula el precio unitario promedio de los artículos vendidos.
   - **`count_total_items(sales_data: List[Tuple[str, float, int]]) -> int`**: Cuenta el número total de artículos vendidos.
   - **`find_min_max_price_item(sales_data: List[Tuple[str, float, int]]) -> Tuple[Tuple[str, float], Tuple[str, float]]`**: Encuentra el artículo más caro y el más barato vendido, devolviendo sus nombres y precios.
   - **`generate_report(store_name: str, date: str, total_sales: float, average_price: float, total_items: int, min_item: Tuple[str, float], max_item: Tuple[str, float], currency: str) -> str`**: Genera el informe final de ventas en formato de texto.
5. Usa el parámetro `**kwargs` en `generate_sales_report` para permitir argumentos de palabra clave opcionales. Incluye los siguientes argumentos de palabra clave opcionales:
   - `include_tax` (bool, predeterminado False): Si es True, calcula la cantidad total de ventas incluyendo un impuesto sobre las ventas del 10%.
   - `currency` (str, predeterminado 'USD'): El símbolo de moneda a usar en el informe.
6. Crea un informe de ventas de ejemplo usando la función `generate_sales_report`. Incluye una mezcla de diferentes gadgets, cantidades y precios en los datos de ventas.

7. Imprime el informe de ventas, incluyendo toda la información calculada, en un formato amigable para el usuario.

8. Prueba la función con diferentes conjuntos de datos de ventas y argumentos de palabra clave opcionales para asegurarte de que funcione como se espera.

Ejemplo de Uso:

```
sales_report = generate_sales_report(
    "Gadget Store",
    "2023-09-15",
    ("Phone", 499.99, 10),
    ("Tablet", 299.99, 5),
    ("Laptop", 899.99, 3),
    ("Smart Watch", 199.99, 15),
    ("Headphones", 149.99, 20),
    ("Camera", 499.99, 2),
    ("Drone", 1599.99, 1),
    ("Speaker", 99.99, 7),
    include_tax=True,
    currency='EUR'
)
print(sales_report)
```

Salida Esperada:

```
Informe de Ventas de Gadget Store
Fecha: 2023-09-15
Total de Ventas: EUR20349.31
Precio Promedio: EUR293.64
Total de Artículos Vendidos: 63
Artículo Más Barato: Speaker a EUR99.99
Artículo Más Caro: Drone a EUR1599.99
```

return

- La instrucción `return` se utiliza para devolver algo desde una función.
- Una función puede tener más de una instrucción `return`, pero solo una de ellas se ejecuta cuando se llama a la función.
- Es importante señalar que, después de que una función devuelve algo, la ejecución de la función se detiene y el control se devuelve al llamador.

`función con return`:

En Python, las funciones a menudo incluyen la instrucción `return` para proporcionar un resultado o valor de vuelta al código que llamó a la función. Aquí tienes un ejemplo de una función simple con una instrucción `return`:

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

addition_return(2, 3)
```

En este ejemplo:

- La función `addition_return` toma dos argumentos, `a` y `b`.
- Utiliza la instrucción `return` para calcular la suma de `a` y `b`.
- Cuando llamas a esta función con `addition_return(2, 3)`, devuelve `5`, que es el resultado de sumar `2` y `3`.

`función sin return`: NoneType

En Python, no todas las funciones están diseñadas para devolver un valor. Algunas funciones tienen como objetivo realizar acciones u operaciones sin producir un resultado específico que necesite ser capturado. Estas funciones se denominan como que tienen un tipo de retorno `NoneType`, lo que se representa con la palabra clave `None`. Cuando una función no incluye una instrucción `return` o incluye una instrucción `return` sin una expresión, implícitamente devuelve `None`.

Exploremos funciones sin valor de retorno usando un ejemplo:

```
def addition_print(a, b):
    print(a + b)

addition_print(2, 3)
```

En este ejemplo, la función `addition_print` toma dos argumentos, `a` y `b`, y realiza la suma de estos valores. Sin embargo, en lugar de usar una instrucción de retorno para proporcionar un resultado, utiliza la función print para mostrar el resultado en la consola.

Cuando llamas a `addition_print(2, 3)`, realiza la suma `2 + 3`, que resulta en `5`. Sin embargo, no hay una instrucción de retorno explícita, por lo que la función implícitamente devuelve `None`. Por lo tanto, si fueras a asignar el resultado de `addition_print(2, 3)` a una variable, esa variable contendría el valor `None`.

Funciones como estas se usan comúnmente cuando quieres realizar alguna acción o efecto secundario, como imprimir salida, sin devolver un valor específico. **Pero, ¿cuáles son los tipos?**

```
type(addition_return(2, 3))
type(addition_print(2, 3))
```

`accediendo al valor del resultado de la función`: print & return

En Python, las funciones juegan un papel crucial en la realización de varias tareas y cálculos. Cuando llamas a una función, puede proporcionarte un resultado, el cual podrías querer acceder o usar en tu programa. Esta sección explora dos maneras comunes de acceder al valor del resultado de una función: usando declaraciones `print` y utilizando la instrucción `return`.

Profundicemos en estos dos enfoques para entender cómo nos permiten trabajar con los resultados producidos por las funciones. **Empecemos con return:**

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

addition(3, 5)

result = addition(3, 5)

result * 100

def greeting(name):
    return f"Hello! My name is {name}"

laura_says = greeting("Laura")
laura_says.upper()
```

Ahora, lo imprimimos:

```
def greeting(name):
    print(f"Hello! My name is {name}")

laura_says = greeting("Laura")

laura_says.upper()
```

`más de un return`

En Python, las funciones son herramientas versátiles que pueden realizar múltiples comprobaciones o tareas y proporcionar resultados basados en diversas condiciones. Esta sección explora el concepto de tener más de una instrucción return dentro de una función.

Considera la función de ejemplo `validador_de_contraseña`. Verifica dos condiciones: la longitud de una contraseña y la presencia de un carácter guión ("-"). Dependiendo de estas condiciones, la función puede devolver diferentes mensajes.

```
def password_validator(password):

    if len(password) > 8:
        print("long")

    if "-" in password:
        print("there's a dash")


password_validator("test-8characters")

def password_validator(password):
```

```
    if len(password) > 8:
        print("BEFORE RETURN")
        return "ok"
        print("AFTER RETURN")

    if "-" in password:
        print("BEFORE RETURN")
        return "ok2"
        print("AFTER RETURN")
```

password_validator ("test-8characters")

Docstrings: cadenas de documentación

En la programación, escribir código que sea claro, conciso y bien documentado es esencial. El código no es solo para las computadoras; también es para los humanos que lo leen, entienden y mantienen. La calidad de tu código puede impactar significativamente la productividad y colaboración de tu equipo.

La Importancia de la Legibilidad del Código

1. **Escrito Una Vez, Leído Muchas Veces**: El código típicamente se escribe una vez pero se lee y mantiene muchas veces a lo largo de su vida útil. Por lo tanto, es crucial hacer que tu código sea fácil de entender.

2. **Entendiendo Tu Trabajo**: Cuando escribes código, piensa en tus colegas y futuros desarrolladores que puedan necesitar trabajar con él. Un código claro y autoexplicativo les ayuda a entender tus intenciones y evita malentendidos.

3. **Documentación con Docstrings**: Una forma de mejorar la claridad del código es utilizando docstrings. Un docstring es un comentario de múltiples líneas que describe el propósito, parámetros y valores de retorno de una función, clase o módulo.

4. **Facilita el Mantenimiento**: Al documentar tu código, facilitas el mantenimiento y las actualizaciones. Esto es especialmente importante en proyectos de larga duración donde las personas pueden cambiar con el tiempo.

5. **Herramientas de Ayuda**: Muchas herramientas de documentación y entornos de desarrollo integrado (IDE) pueden generar automáticamente documentación basada en los docstrings, lo que aumenta la eficiencia en la creación de documentación.

Ejemplo: La Función Gato

En este ejemplo, hemos definido una función simple llamada `gato`. Esta función devuelve la cadena "miau". Para hacer este código más comprensible, hemos añadido un docstring que explica lo que hace la función `gato`.

```
def gato():
    """
    Esta función devuelve el sonido de un gato, que es 'miau.'

    Returns:
        str: El sonido que hace un gato.
    """
    return "miau"
```

En este ejemplo, hemos definido una función `gato` que devuelve "miau". Hemos declarado un docstring que explica lo que hace la función. Para obtener el docstring de una función, necesitamos mostrar el atributo `__doc__` (usar `print(gato.__doc__)`).

Para ver el docstring de la función `gato`, podemos usar:

```
print(gato.__doc__)
```

También podemos utilizar la función `help()` para obtener una descripción más detallada de la función, incluyendo su docstring:

```
help(gato)
```

Estándares de Código Limpio para Docstrings

A continuación, se presenta una plantilla para seguir al escribir docstrings:

```
"""
Describe el propósito y uso de la función.

Parameters:
    param_1 (data_type): Descripción del primer parámetro.
    param_2 (data_type): Descripción del segundo parámetro. (Si
aplica)
    ...
    param_n (data_type): Descripción del enésimo parámetro. (Si
aplica)

Arguments:
    arg_1 (data_type): Descripción del primer argumento. (Si aplica)
    arg_2 (data_type): Descripción del segundo argumento. (Si aplica)
    ...
    arg_n (data_type): Descripción del enésimo argumento. (Si aplica)

Keyword Arguments:
    kwarg_1 (data_type): Descripción del primer argumento de palabra
clave. (Si aplica)
    kwarg_2 (data_type): Descripción del segundo argumento de palabra
clave. (Si aplica)
    ...
```

```
    kwarg_n (data_type): Descripción del enésimo argumento de palabra
clave. (Si aplica)

Returns:
    return_type: Descripción del valor de retorno.

Examples:
    >>> nombre_funcion(arg_1, arg_2, kwarg_1=valor)
    Salida esperada o descripción del resultado.
    >>> otra_funcion(arg)
    Salida esperada o descripción del resultado.
"""
```

Código Fuente

En esta sección, exploraremos cómo acceder e imprimir el código fuente de una función usando el módulo `inspect` de Python. Aprenderemos cómo recuperar el código real que define una función. Esto puede ser útil para entender cómo está implementada una función o para propósitos de depuración.

El módulo `inspect` proporciona varias funciones que permiten obtener información sobre los objetos en Python, incluidas funciones, métodos, clases y módulos. Con `inspect.getsource()`, podemos obtener el código fuente de una función en forma de cadena.

Profundicemos en cómo puedes acceder y mostrar el código fuente de una función en Python.

```python
import inspect

def gato():
    """Esta función devuelve el sonido de un gato, que es 'miau.'"""
    return "miau"

print(inspect.getsource(gato))
```

La salida de este código sería:

```
def gato():
    """Esta función devuelve el sonido de un gato, que es 'miau.'"""
    return "miau"
```

Ámbito de las Funciones

En Python, el ámbito de una variable se refiere a la región de código donde esa variable puede ser accedida o modificada. Entender el ámbito de las variables es crucial cuando se trabaja con funciones, ya que determina qué variables puede ver y manipular una función.

Hay dos direcciones principales del ámbito de las variables en Python: "de exterior a interior" y "de interior a exterior". En la dirección "de exterior a interior", las variables definidas fuera de

una función son accesibles dentro de la función. En la dirección "de interior a exterior", las variables definidas dentro de una función no son accesibles fuera de esa función.

Exploraremos ambos escenarios y discutiremos cómo el ámbito de las variables afecta el comportamiento de las funciones.

Ejemplo: Suma

En el siguiente ejemplo, definimos una función `suma` que toma dos argumentos y devuelve su suma. También mostramos cómo se puede acceder a una variable global definida fuera de la función.

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

a = 10

resultado = suma(a, 10)
print(resultado)  # Salida: 20
print(a)          # Salida: 10
```

Ejemplo: Ámbito de Interior a Exterior

En este segundo ejemplo, definimos una variable dentro de la función, la cual no es accesible fuera de ella. Esto ilustra cómo las variables locales no pueden ser vistas fuera de su función.

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

resultado = suma(10, 10)
print(resultado)  # Salida: 20
# print(d)        # Esto causaría un error: NameError: name 'd' is not
defined
```

Variables Globales

También podemos usar variables globales en Python. Una variable global es aquella que se define fuera de todas las funciones y puede ser accedida por cualquier función en el mismo archivo. Sin embargo, si deseas modificarla dentro de una función, debes declararla como `global`.

```
a = 10

def modificar_global():
    global a
    a += 5
```

```
modificar_global()
print(a)  # Salida: 15
```

En este ejemplo, la función `modificar_global` utiliza la declaración `global` para modificar la variable `a` definida fuera de la función.

Extra: variables globales

Lambda

Las funciones lambda, también conocidas como funciones anónimas, son una forma concisa de definir funciones pequeñas y simples en Python. A diferencia de las funciones regulares definidas usando la palabra clave `def`, las funciones lambda son anónimas y a menudo se utilizan para operaciones cortas y únicas.

Sintaxis de una Función Lambda

Una función lambda tiene la siguiente sintaxis:

```
lambda <lista de parámetros>:<expresión de retorno>
```

En esta sintaxis, `<lista de parámetros>` representa la lista de parámetros que la función lambda toma, y `<expresión de retorno>` es la expresión que define lo que la función lambda devuelve. Las funciones lambda son particularmente útiles cuando necesitas pasar una función simple como argumento a otra función o usarlas en situaciones donde una definición completa de la función es innecesaria.

Ejemplo 1: Función Lambda que suma dos números

```
add = lambda x, y: x + y
result = add(3, 5)  # Resultado: 8
print(result)
```

Ejemplo 2: Función Lambda que calcula el cuadrado de un número

```
square = lambda x: x ** 2
result = square(4)  # Resultado: 16
print(result)
```

Ejemplo 3: Función Lambda para verificar si un número es par

```
is_even = lambda x: x % 2 == 0
result = is_even(6)  # Resultado: True
print(result)
```

Ejemplo 4: Función Lambda para extraer el último carácter de una cadena

```
get_last_char = lambda s: s[-1]
result = get_last_char("Hola")  # Resultado: "a"
print(result)
```

Estas funciones lambda son concisas y se pueden utilizar en situaciones en las que necesitas una función rápida y breve para realizar una tarea específica. Las funciones lambda se usan a menudo con funciones como `map()`, `filter()` y `sorted()` para definir un comportamiento personalizado para ordenar o procesar datos.

Desafío Lambda

Convierte estas funciones regulares en lambda.

Desafío 1:

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

Versión lambda:

```
add = lambda a, b: a + b
```

Desafío 2:

```
def square(x):
    return x ** 2
```

Versión lambda:

```
square = lambda x: x ** 2
```

Desafío 3:

```
def is_even(num):
    return num % 2 == 0
```

Versión lambda:

```
is_even = lambda num: num % 2 == 0
```

`lambda` como opción de algunas otras funciones integradas

Además de usar funciones `lambda` en expresiones independientes, también se pueden emplear como argumentos en llamadas a otras funciones. Esta capacidad permite un comportamiento

personalizado al trabajar con varias funciones integradas, como `sorted()`, `filter()`, `map()` y más.

Un caso de uso común es proporcionar una función `lambda` como argumento al parámetro clave en funciones de ordenación como `sorted()` y `list.sort()`. Esto te permite definir criterios de ordenación personalizados para tus datos.

Aquí tienes un ejemplo de la sintaxis general de uso de funciones `lambda` como claves de ordenación:

```
sorted_list = sorted(iterable, key=lambda item:
custom_sort_logic(item))
```

En esta sintaxis:

- `iterable` es la colección de elementos que deseas ordenar.
- `custom_sort_logic(item)` es la función lambda que define cómo los elementos deben ser comparados y ordenados basándose en alguna lógica personalizada.

Veamos un ejemplo para ilustrar este concepto más a fondo:

```
students = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 22},
    {"name": "Charlie", "age": 28},
]

# Ordenar los estudiantes basándose en sus edades usando una función
lambda
sorted_students = sorted(students, key=lambda student: student["age"])
print(sorted_students)
```

Salida esperada:

```
[{'name': 'Bob', 'age': 22}, {'name': 'Alice', 'age': 25}, {'name':
'Charlie', 'age': 28}]
```

En este ejemplo, usamos una función `lambda` como el argumento `key` en la función `sorted()` para ordenar la lista de estudiantes basándonos en sus edades. La función `lambda` especifica que queremos ordenar a los estudiantes por su clave **"age"**.

Esta flexibilidad de usar funciones `lambda` como argumentos te permite adaptar el comportamiento de las funciones integradas para satisfacer tus requisitos específicos, haciendo tu código más expresivo y adaptable.

Funciones como retorno de otra función

En Python, las funciones pueden tratarse como objetos de primera clase, lo que significa que pueden asignarse a variables, pasarse como argumentos y, lo que es más importante, devolverse desde otras funciones.

Esto abre muchas posibilidades para la programación funcional y permite crear patrones de diseño avanzados, como funciones especializadas, decoradores, y más.

Conceptos clave:

1. **Funciones como objetos de primera clase**: Puedes asignar funciones a variables, almacenarlas en estructuras de datos (como listas o diccionarios) o devolverlas desde otras funciones.

2. **Funciones de orden superior**: Son funciones que toman otras funciones como argumentos o devuelven funciones como resultado. Ejemplo típico: los decoradores.

3. **Clausuras (closures)**: Una clausura se refiere a una función interna que "recuerda" las variables de su función contenedora (aunque la función externa haya terminado su ejecución). Esto permite mantener el estado entre llamadas de funciones.

Ejemplo básico: Funciones mayor y menor

```
def greater_function(a, b):
    if a > b:
        return a
    elif b > a:
        return b

greater_function(3, 10)  # Resultado: 10

def lesser_function(x, y):
    if x > y:
        return y
    elif y > x:
        return x

lesser_function(3, 10)  # Resultado: 3
```

Función que devuelve otras funciones

Ahora veamos cómo una función puede devolver otras funciones. En este ejemplo, la función `comparison` devuelve `greater_function` o `lesser_function` dependiendo del argumento que le pasemos.

```
def comparison(type_of_comparison):  # "lesser" o "greater"
    if type_of_comparison == "greater":
        return greater_function
    elif type_of_comparison == "lesser":
        return lesser_function

# Usar la función "comparison" para devolver y ejecutar la función
adecuada
comparison("greater")(3, 10)  # Resultado: 10
comparison("lesser")(3, 10)  # Resultado: 3
```

Explicación del código:

1. Definimos dos funciones, `greater_function` y `lesser_function`, que comparan dos valores y devuelven el mayor o el menor, respectivamente.
2. La función `comparison` toma un argumento `type_of_comparison` (puede ser "greater" o "lesser") y devuelve la función correspondiente.
3. Cuando llamamos a `comparison("greater")`, obtenemos una referencia a `greater_function`, y luego la invocamos con `(3, 10)`, devolviendo el mayor número.
4. Similarmente, `comparison("lesser")` devuelve `lesser_function` y ejecuta la comparación.

Uso de lambda para simplificar la función devuelta

Las funciones lambda permiten definir funciones pequeñas y concisas. Podemos usar lambda para devolver una función más compacta en vez de una definida con `def`.

```
def comparison(type_of_comparison):  # "lesser" o "greater"
    if type_of_comparison == "greater":
        return greater_function
    elif type_of_comparison == "lesser":
        return lambda x, y: x if x < y else y  # Devuelve la menor de las dos

# Usar la función "comparison" con una lambda para "lesser"
comparison("lesser")(3, 10)  # Resultado: 3
```

Aplicaciones comunes:

Las funciones que devuelven otras funciones son especialmente útiles en los siguientes casos:

1. **Decoradores**: En Python, los decoradores son un tipo común de función que devuelve otras funciones. Son útiles para añadir comportamiento adicional a una función existente sin modificar su estructura interna.

2. **Funciones parametrizadas**: Las funciones que devuelven otras funciones permiten crear funciones parametrizadas. Por ejemplo, puedes definir una función externa que devuelva funciones con configuraciones específicas, adaptando el comportamiento de las funciones internas según el contexto.

3. **Closures y estado mantenido**: A veces es útil que una función mantenga cierto estado a través de varias ejecuciones. Las clausuras permiten que las funciones internas "recuerden" valores de su entorno incluso después de que la función externa haya terminado.

Ejemplo: Clausuras en acción

En este ejemplo, usamos una función para crear una calculadora de potencia que genera funciones especializadas para elevar números a un exponente específico.

```python
def power_creator(exponent):
    def power(base):
        return base ** exponent
    return power

# Crear funciones especializadas para elevar al cuadrado y al cubo
square = power_creator(2)
cube = power_creator(3)

print(square(4))  # Resultado: 16
print(cube(3))    # Resultado: 27
```

Explicación:

- La función `power_creator` toma un argumento `exponent` y devuelve una función interna `power`, que eleva un número `base` a ese exponente.
- Las funciones `square` y `cube` son clausuras que "recuerdan" el exponente con el que fueron creadas, permitiendo que se utilicen para elevar números al cuadrado o al cubo, respectivamente.

Ventajas de las clausuras:

1. **Encapsulamiento**: Las clausuras encapsulan su entorno, lo que significa que pueden "recordar" los valores de las variables incluso después de que la función externa haya terminado.
2. **Estado persistente**: Pueden mantener el estado entre llamadas, lo que es útil para crear funciones que mantienen algún tipo de información a lo largo del tiempo.
3. **Personalización dinámica**: Permiten crear funciones personalizadas al vuelo, con comportamientos específicos, sin necesidad de crear funciones separadas para cada caso.

Ejemplo con funciones anidadas:

```python
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

my_func = outer_function("¡Hola!")
my_func()  # Resultado: ¡Hola!
```

Explicación:

- En este caso, `outer_function` devuelve `inner_function`, que recuerda el mensaje `msg` que se le pasó en el momento de su creación.
- Cuando ejecutamos `my_func()`, imprime el mensaje recordado, demostrando cómo la clausura captura el entorno local.

Las funciones que devuelven otras funciones y las clausuras son herramientas poderosas en Python que permiten un control flexible y dinámico sobre el comportamiento del código. Ya sea

para crear decoradores, manejar estado, o generar funciones especializadas, este patrón es una parte esencial de la programación funcional en Python.

Recursión

La recursión es una técnica donde una función se llama a sí misma como parte de su ejecución. Este concepto puede ser útil en muchos escenarios, como en problemas que pueden dividirse en subproblemas más pequeños. Algunos de los casos más comunes donde se utiliza la recursión incluyen la búsqueda de directorios, el cálculo de factoriales o números de Fibonacci, y la solución de problemas de tipo "divide y vencerás".

Sin embargo, la recursión debe usarse con precaución debido a los riesgos de generar llamadas infinitas que pueden terminar en un desbordamiento de pila (stack overflow). Por lo tanto, cada función recursiva debe tener un **caso base** que detenga la recursión.

Ejemplo básico de recursión: Factorial

El factorial de un número (n) (denotado como (n!)) es el producto de todos los enteros positivos menores o iguales a (n). Se puede definir de manera recursiva: el factorial de 0 es 1, y el factorial de (n) es (n \times \text{factorial}(n-1)).

```
def factorial(n):
    if n == 0:  # Caso base: el factorial de 0 es 1
        return 1
    else:
        return n * factorial(n - 1)  # Llamada recursiva

print(factorial(5))  # Resultado: 120
```

Explicación:

En este ejemplo, la función `factorial` se llama a sí misma hasta que el argumento (n) es 0, momento en el cual la recursión se detiene. Cuando (n) llega a 0, el caso base devuelve 1. En los casos recursivos, la función multiplica (n) por el resultado de (\text{factorial}(n-1)), acumulando el producto hasta alcanzar el caso base.

Diferencia entre recursión y bucles `while`

A menudo, los problemas que pueden resolverse mediante recursión también pueden resolverse con bucles, como el bucle `while` o el bucle `for`. El código recursivo y el de bucles iterativos pueden parecer similares, pero tienen diferencias importantes.

Por ejemplo, aquí hay dos versiones de un código que pide al usuario ingresar un número menor que 5:

Versión con bucle `while`

```
def user_input_while():
    value = int(input("Por favor, ingresa un valor menor que 5: "))
    while value > 5:
```

```
        value = int(input("Por favor, ingresa un valor menor que 5: "))
    return value

print(user_input_while())
```

Explicación:

En esta versión, se utiliza un bucle `while` que seguirá solicitando un número al usuario hasta que se introduzca un valor menor que 5. La lógica es sencilla y permite que el usuario continúe ingresando valores hasta cumplir con la condición deseada.

Versión recursiva

```
def user_input_recursive():
    value = int(input("Por favor, ingresa un valor menor que 5: "))
    if value > 5:
        return user_input_recursive()  # Llamada recursiva si la condición no se cumple
    return value

print(user_input_recursive())
```

Explicación:

En esta versión recursiva, la función se llama a sí misma si el valor ingresado es mayor que 5. Esto implica que la función "repite" el proceso de captura del valor hasta que se cumpla la condición, mostrando cómo la recursión puede sustituir un bucle, aunque en algunos casos puede ser menos eficiente en cuanto a memoria.

Comparación:

- **`while` loop**: La función iterativa usa un bucle que sigue ejecutándose hasta que se cumple una condición específica (en este caso, que `value` sea menor que 5).
- **Recursión**: La función recursiva se llama a sí misma si no se cumple la condición, lo que esencialmente repite el proceso de la misma manera que lo haría un bucle, pero usa una llamada a función en lugar de una iteración.

Comparación de métodos de resolución de problemas

Cuando se trata de resolver problemas, podemos utilizar bucles `for`, bucles `while`, o recursión. Cada uno tiene sus ventajas y desventajas, dependiendo de la situación.

- **Bucles `for`**: Útiles para iterar sobre secuencias de elementos. Son más legibles cuando el número de iteraciones es conocido de antemano.
- **Bucles `while`**: Son ideales para situaciones donde no se sabe cuántas iteraciones se necesitarán, ya que dependen de una condición.
- **Recursión**: Excelente para problemas que se pueden descomponer en subproblemas similares. Sin embargo, tiene un costo adicional en términos de uso de memoria debido a la pila de llamadas.

Ejemplo: Método de Newton-Raphson

El método de Newton-Raphson es un algoritmo iterativo utilizado para encontrar raíces de funciones. Dado un valor inicial (x_0), la fórmula del método es:

[x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}]

Este método requiere que la función sea diferenciable y que la derivada no sea cero en el punto que se evalúa.

Implementación del método de Newton-Raphson

1. Versión con `for`

```
def newton_raphson_for(f, df, x0, tolerance=1e-7,
max_iterations=1000):
    x = x0
    for _ in range(max_iterations):
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tolerance:
            return x_new
        x = x_new
    return None  # No se encontró la raíz

# Ejemplo de uso
def f(x): return x**2 - 2  # f(x) = x^2 - 2
def df(x): return 2*x       # f'(x) = 2x

print(newton_raphson_for(f, df, 1.0))  # Resultado aproximado de
sqrt(2)
```

Explicación:

En esta versión, el método utiliza un bucle `for` para realizar un número máximo de iteraciones. En cada iteración, se calcula un nuevo valor (x) utilizando la fórmula de Newton-Raphson. Si el cambio entre (x) y (x_{new}) es menor que la tolerancia especificada, se devuelve el nuevo valor como la raíz.

2. Versión con `while`

```
def newton_raphson_while(f, df, x0, tolerance=1e-7,
max_iterations=1000):
    x = x0
    iterations = 0
    while iterations < max_iterations:
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tolerance:
            return x_new
        x = x_new
        iterations += 1
```

```
    return None  # No se encontró la raíz

print(newton_raphson_while(f, df, 1.0))  # Resultado aproximado de sqrt(2)
```

Explicación:

Esta versión utiliza un bucle `while` y lleva un contador de iteraciones. El proceso es similar al anterior, pero continúa hasta que se alcance el número máximo de iteraciones o se encuentre una solución suficientemente precisa.

3. Versión recursiva

```
def newton_raphson_recursive(f, df, x, tolerance=1e-7):
    x_new = x - f(x) / df(x)
    if abs(x_new - x) < tolerance:
        return x_new
    return newton_raphson_recursive(f, df, x_new, tolerance)

print(newton_raphson_recursive(f, df, 1.0))  # Resultado aproximado de sqrt(2)
```

Explicación:

En la versión recursiva, la función se llama a sí misma hasta que se cumple la condición de tolerancia. Esto muestra cómo la recursión puede ser utilizada para implementar un algoritmo que normalmente es iterativo. Sin embargo, hay que tener cuidado con la profundidad de las llamadas y el uso de memoria.

Otras aplicaciones de la recursión

La recursión se utiliza en varios algoritmos y estructuras de datos, como:

- **Recorrido de árboles**: La recursión es ideal para realizar recorridos en estructuras de datos jerárquicas, como árboles binarios.
- **Backtracking**: Problemas como el Sudoku o el problema de las N reinas se resuelven utilizando recursión.
- **Algoritmos de búsqueda**: Como la búsqueda binaria, que divide el espacio de búsqueda en mitades recursivamente.

Peligros de la recursión: Exceso de recursión

La recursión, aunque poderosa, tiene un límite inherente: la pila de llamadas. Python tiene un límite en la profundidad de recursión, que evita desbordamientos de pila. Si una función recursiva se ejecuta demasiadas veces sin llegar a un caso base, se producirá un error de "RecursionError".

```
def infinite_recursion():
    return infinite_recursion()  # Esta función nunca alcanza un caso
```

```
base

infinite_recursion()  # Resultado: RecursionError: maximum recursion
depth exceeded
```

Este error ocurre cuando la recursión se adentra demasiado sin encontrar un caso base.

Alternativa a la recursión: Algoritmos iterativos

Muchas veces, los problemas que podrían resolverse con recursión también pueden abordarse con algoritmos iterativos (bucle `while`, bucles `for`, etc.). En casos donde la recursión no es necesaria o puede volverse ineficiente debido a la profundidad, es recomendable usar una alternativa iterativa.

```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5))  # Resultado: 120
```

Conclusión:

La recursión es una herramienta útil en Python que permite descomponer problemas complejos en soluciones más simples mediante la repetición de funciones. Sin embargo, debe usarse con cuidado y solo cuando sea la solución más adecuada, ya que puede generar desbordamientos de pila si no se maneja correctamente. Alternativas iterativas, como los bucles `while` y `for`, a menudo pueden ser más eficientes en términos de espacio y tiempo, y es importante elegir el enfoque adecuado según el problema específico que se esté abordando.