Lectura

Funciones y estructuras de datos

Fundamentos de Python

Funciones y estructuras de datos

Python es un lenguaje de programación que destaca por su amplia gama de herramientas, las cuales, resultan fundamentales para organizar, almacenar y manipular información de manera eficiente, así como para estructurar el código de manera clara y reutilizable. Estas herramientas se conocen como estructuras de datos y funciones, y hacen de Python un lenguaje de programación poderoso para el desarrollo de software.

Tipos y estructuras de datos

Las estructuras de datos son la base para organizar y almacenar datos de manera eficiente, esto quiere decir que permite guardar diferentes datos juntos y facilita el cómo acceder a ellos, modificarlos o recorrerlos. Estas estructuras se clasifican en mutables e inmutables:

Estructuras de datos mutables

Es aquella cuyo contenido se puede cambiar después de haber sido creada. Es decir, puedes **modificar**, **agregar** o **eliminar** elementos mediante un índice, mismo que nos ayuda a ubicar y acceder al número de un elemento específico, sin tener que crear otra estructura nueva. Algunos ejemplos de datos mutables son:

- **Listas *(arrays):*** Son secuencias ordenadas de elementos como int, float, boolean, strings, entre otras, que se pueden manejar al mismo tiempo.

- **Conjuntos (*set*):** Colecciones no ordenadas de elementos únicos. No permiten duplicados y no tienen un orden específico, por lo que no se puede acceder a sus elementos mediante índices.

- **Diccionarios (*dict*):** Son colecciones de pares clave – valor, por ejemplo: "edad(clave):24(valor)". Cada clave debe ser única y se utiliza para acceder a su valor correspondiente. Los diccionarios no mantienen un orden específico de los elementos.

Estructuras de datos inmutables

Es aquella cuyo contenido no se puede modificar después de haber sido creada, es decir, no puedes agregar, eliminar o cambiar sus elementos; si necesitas modificar, debes crear una nueva estructura. Un ejemplo de dato inmutable es:

- **Tuplas *(tuple)*:** Secuencias ordenadas de elementos que no pueden ser modificadas después de su creación. Son similares a las listas, pero no permiten la modificación de sus elementos.

Array unidimensional y bidimensional

Los *arrays* o listas unidimensional y bidimensional hacen referencia a estructuras que almacenan datos en una o dos dimensiones, es decir, si es **unidimensional** solo necesita una posición (fila) para encontrar un elemento, y si es **bidimensional** necesita de dos posiciones (fila y columna) para encontrar dicho elemento, cada uno se distingue por distintas funciones, como que un array unidimensional es una secuencia lineal, mientras que un array bidimensional es una tabla con filas y columnas. A continuación, se describen cada una de estas estructuras:

Array unidimensional:

- Es un grupo de elementos lineales, como una lista de números o cadenas de texto.
- En Python, se puede representar con una lista simple, por ejemplo: mi_lista = [1, 2, 3, 4, 5].
- Cada elemento se accede mediante su posición (índice), comenzando desde 0, por ejemplo: mi_lista[0] accede al primer elemento (1).

Ejemplo de arrays unidimensional

```
# Creación de un array unidimensional (lista)
notas_estudiantes = [85, 92, 78, 90, 88]

print("Array Unidimensional - Notas de Estudiantes:")
print(notas_estudiantes)
# Recorrido del array con for
print("\nRecorriendo el array:")
for i, nota in enumerate(notas_estudiantes):
    print(f"Estudiante {i+1}: {nota}")
)
```

Array bidimensional:

- Es un grupo de *arrays* unidimensionales, organizados en filas y columnas.
- Se puede representar en Python con una matriz, por ejemplo: matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]].
- Cada elemento se accede mediante dos índices: matriz[0][0] accede al elemento en la primera fila y columna (1).

Ejemplo de arrays bidimensional

```
# Creación de una matriz simple 3x3 (3 filas y 3 columnas)
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Matriz completa:")
for fila in matriz:
    print(fila)
```

Se puede acceder y modificar listas usando índices o métodos como *append*, *insert*, *remove*, entre otros. Para **acceder** a un elemento específico, se usa la notación de corchetes ("[]") con el índice deseado, que empieza desde cero para el primer elemento. Para **modificar** un elemento, se asigna un nuevo valor al índice correspondiente. Además, se pueden realizar operaciones como la unión de dos o varias listas usando "+" y la replicación con "*".

- Acceso:
  - Unidimensional: lista[**índice**]
  - Bidimensional: matriz[**fila**][**columna**]
- Modificación:
  - Cambiar valor: lista[**índice**] = nuevo_valor
  - Cambiar fila completa: matriz[**fila**][**columna**] = nueva_lista
- Métodos útiles:
  - *append*(**valor**): Añade elemento al final
  - *insert*(**indice**, **valor**): Añade elemento en posición específica
  - *remove*(**valor**): Elimina por valor

Ejemplo para acceder y cambiar listas.

```
# Creación de lista unidimensional
frutas = ["manzana", "banana", "cereza", "durazno"]
print("Lista original:", frutas)

# Acceso por índice
print("\nAccediendo a elementos:")
print("Primera fruta:", frutas[0])  # manzana
print("Última fruta:", frutas[-1])  # durazno

# Modificación por índice
frutas[1] = "pera"
print("\nDespués de modificar el segundo elemento:", frutas)

# Métodos básicos
frutas.append("uva")  # Agrega al final
print("\nDespués de append('uva'):", frutas)

frutas.insert(2, "kiwi")  # Inserta en posición 2
print("Después de insert(2, 'kiwi'):", frutas)

frutas.remove("cereza")  # Elimina por valor
print("Después de remove('cereza'):", frutas)

# Operaciones con listas
verduras = ["zanahoria", "espinaca"]
lista_compra = frutas + verduras  # Concatenación
print("\nLista de compras (concatenación):", lista_compra)

lista_repetida = verduras * 2  # Repetición
print("Lista repetida:", lista_repetida)
```

Conjuntos o *set*

Un *set* es un grupo **desordenado** y **sin elementos duplicados**. Es muy útil cuando necesitas guardar elementos únicos y realizar operaciones matemáticas como uniones, intersecciones, diferencias, etc. Los *set* son mutables, lo que significa que se pueden agregar o eliminar elementos después de su creación, pero los elementos individuales deben ser

inmutables (por ejemplo, números, cadenas, tuplas). Algunas de sus características principales son:

- **Unicidad:** Un *set* solo contiene elementos únicos. Si se intenta agregar un elemento que ya existe, no se duplicará.
- **Desorden:** Los elementos no tienen un orden definido, por lo que no se puede acceder a ellos por su posición.
- **Mutable:** Se pueden agregar o eliminar elementos después de crear el set.
- **Inmutabilidad de los elementos:** Los elementos individuales deben ser inmutables. No se pueden incluir listas, diccionarios o set dentro de un set.
- **Operaciones de conjuntos:** Los set soportan operaciones matemáticas de conjuntos como unión, intersección, diferencia, etc.

Ejemplo de set

```
# Creación de set usando constructor
numeros_primos = set([2, 3, 5, 7, 11, 13])  # Desde lista
vocales = set(('a', 'e', 'i', 'o', 'u'))  # Desde tupla

print("\nSet creado con set():")
print(numeros_primos)
print(type(numeros_primos))

# Set desde cadena (elimina duplicados)
letras = set("programacion")
print("\nSet desde string:", letras)
```

Diccionario o *dict*().

Es una estructura de datos que almacena pares clave - valor. Es muy útil cuando necesitas acceder a datos por una clave identificadora en lugar de un índice. Dentro de sus características principales podemos encontrar:

- **Colección desordenada:** Los elementos en un diccionario no tienen un orden específico.
- **Pares clave-valor:** Cada elemento se compone de una clave única y un valor asociado.
- **Indexación por clave:** Se accede a los valores a través de sus claves correspondientes, no por su posición.
- **Mutables:** Los diccionarios pueden ser modificados después de su creación, permitiendo agregar, eliminar o actualizar elementos.
- **Inmutabilidad de las claves:** Las claves deben ser de tipos inmutables, como cadenas, números o tuplas, mientras que los valores pueden ser de cualquier tipo.

Ejemplo de dict()

```
# Creación de diccionario (clave-valor)
estudiante = {
    "nombre": "Ana",
    "edad": 22,
    "carrera": "Ingeniería",
    "materias": ["Cálculo", "Programación"]
}
print("\n Ejemplo de Diccionario:")
print("Datos completos:", estudiante)
print("Nombre del estudiante:", estudiante["nombre"])

# Modificación
estudiante["edad"] = 23
estudiante["semestre"] = 5  # Añadir nueva clave

print("\nDespués de modificaciones:")
for clave, valor in estudiante.items():
    print(f"{clave}: {valor}")
```

Tuplas o tuple

Una tupla es una estructura de datos muy parecida a una lista, pero inmutable, lo que significa que no se puede modificar una vez creada. Se define utilizando paréntesis () y los elementos se separan por comas. A diferencia de las listas, las tuplas son ideales para representar datos que no deben cambiar y, al igual que un *set*, las tuplas cuentan con características principales como:

- **Ordenadas**: Los elementos de una tupla mantienen el orden en que se ingresaron, y se puede acceder a ellos a través de su índice.
- **Inmutables**: Una vez creada, no se pueden agregar, eliminar o modificar elementos dentro de una tupla.

- **Heterogéneas**: Pueden contener elementos de diferentes tipos de datos (enteros, cadenas, etc.).
- **Eficiencia**: Las tuplas son más eficientes en términos de memoria y velocidad que las listas, especialmente cuando se trata de colecciones de datos que no necesitan ser modificadas.
- **Uso común**: Se utilizan para representar datos que naturalmente van juntos, como coordenadas (x, y), o para devolver múltiples valores de una función.

Ejemplo de tupla o tuple

```
# Creación de tupla usando constructor
coordenadas = tuple([10, 20, 30])  # Desde una lista
colores_rgb = tuple(("rojo", "verde", "azul"))  # Desde otra tupla

print("Tupla creada con tuple():")
print(coordenadas)
print(type(coordenadas))

# Conversión de cadena a tupla
letras = tuple("Python")
print("\nTupla desde string:", letras)
```

Casting

Un casting, también llamado conversión de tipos, es el proceso de **convertir** un valor de un tipo de dato a otro, como de cadena a entero, de entero a flotante, entre otros. Esta herramienta tiene diferentes usos, por ejemplo:

- Para poder realizar operaciones adecuadas: Algunas operaciones solo funcionan si los datos son del tipo correcto.
- Para cambiar el formato de salida: Convertir un número a cadena, para mostrarlo dentro de un mensaje.
- Para evitar errores por tipos incompatibles: El casting garantiza que todos los valores sean del tipo necesario para una función u operación específica.

Existen varios tipos comunes de estas conversiones, entre ellas están:

Ejemplo de Castings

```
De cadena a entero
edad = "25"
edad_entero = int(edad)
print(f"Edad como entero: {edad_entero} (tipo: {type(edad_entero)})")
# De flotante a entero (trunca decimales)
precio = 99.95
precio_entero = int(precio)
print(f"Precio truncado: {precio_entero}")  # 99
# De booleano a entero
activo = True
activo_entero = int(activo)
print(f"Booleano a entero: {activo_entero}")  # 1

# De cadena a flotante
peso = "65.5"
peso_flotante = float(peso)
print(f"\nPeso como flotante: {peso_flotante} (tipo:
{type(peso_flotante)})")
# De entero a flotante
cantidad = 100
cantidad_flotante = float(cantidad)
print(f"Entero a flotante: {cantidad_flotante}")  # 100.0
# De booleano a flotante
valido = False
valido_flotante = float(valido)
print(f"Booleano a flotante: {valido_flotante}")  # 0.0
# De entero a cadena
codigo = 404
codigo_cadena = str(codigo)
```

```
print(f"\nCódigo como cadena: '{codigo_cadena}' (tipo:
{type(codigo_cadena)})")
# De lista a cadena
numeros = [1, 2, 3]
numeros_cadena = str(numeros)
print(f"Lista a cadena: {numeros_cadena}")  # "[1, 2, 3]"
# De booleano a cadena
es_valido = True
es_valido_cadena = str(es_valido)
print(f"Booleano a cadena: {es_valido_cadena}")  # "True"

# De entero a booleano
print("\nConversiones a booleano:")
print("Cero:", bool(0))          # False
print("Uno:", bool(1))           # True
print("Cien:", bool(100))        # True
# De cadena a booleano
print("Cadena vacía:", bool(""))    # False
print("Texto:", bool("Hola"))       # True

# De lista a booleano
print("Lista vacía:", bool([]))     # False
print("Lista con datos:", bool([1, 2]))  # True

# A lista
cadena = "Python"
lista_caracteres = list(cadena)
print(f"\nCadena a lista: {lista_caracteres}")  # ['P', 'y', 't', 'h', 'o',
'n']

# A conjunto (elimina duplicados)
lista_repetida = [1, 2, 2, 3, 3, 3]
conjunto_unico = set(lista_repetida)
print(f"Lista a conjunto: {conjunto_unico}")  # {1, 2, 3}

# A tupla
rango = range(3)
tupla_rango = tuple(rango)
print(f"Range a tupla: {tupla_rango}")  # (0, 1, 2)
```

Funciones en Python

Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto. Las funciones tienen un doble objetivo:

- Dividir y organizar el código en partes más sencillas.
- Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Existen dos tipos de funciones según su origen:

Funciones nativas

El intérprete de Python ya cuenta con una serie de funciones y tipos incluidos en él, que están siempre disponibles. Están listados a continuación, en orden alfabético.

Funciones incorporadas

A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		_
	iter()		__import__()

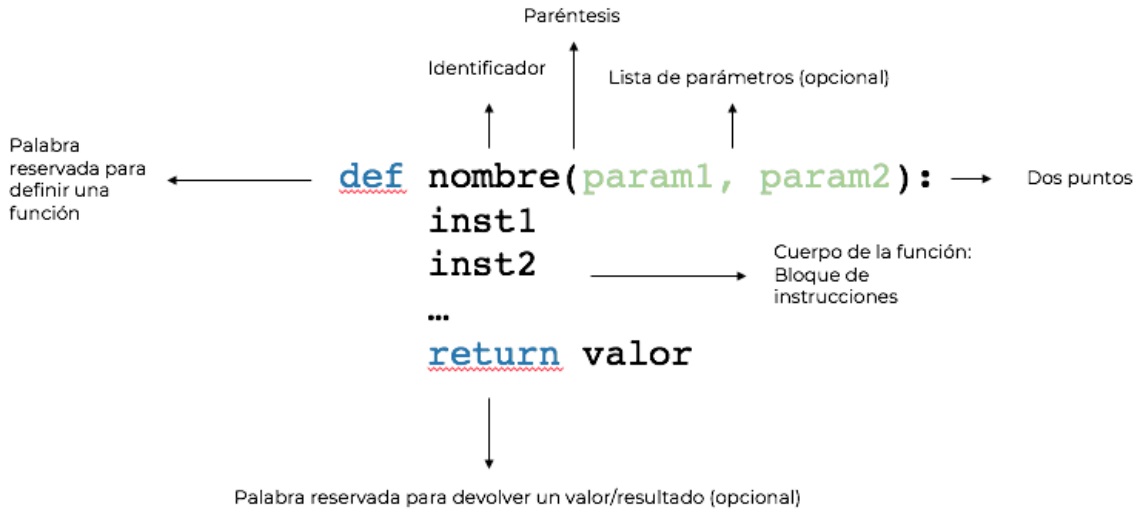
Funciones personalizadas

Al igual que en otros lenguajes de programación, en Python también es posible definir nuestras propias funciones. Para ello, se utiliza la palabra clave ***def***, que indica al lenguaje que se va a crear una función con un nombre y un comportamiento específicos. Esto permite agrupar un conjunto de instrucciones bajo un mismo bloque, facilitando su reutilización en diferentes momentos del programa.

Esta estructura se compone de varias partes fundamentales para su correcta ejecución, entre ellas:

- **def** → Indica que estás definiendo una función.
- **nombre_funcion** → El nombre que le das a tu función.
- **(parámetros)** → Datos que la función puede recibir (opcionales).
- **(argumentos)** → Es el valor real que le pasas a una función cuando la llamas, para que esa función pueda trabajar con él.
- **:** → Marca el inicio del bloque de código de la función.
- **return** → Devuelve un resultado al terminar (opcional).

En este gráfico se muestra la estructura básica para definir una función en Python utilizando la palabra clave ***def.***



A continuación, se presentan algunos ejemplos de cómo utilizar *def* con ***return*** y **sin *return*** para crear una nueva función:

Ejemplo de función con return (retorna valor)

```python
def calcular_iva(precio):
    """Calcula el IVA del 19% sobre un precio"""
    iva = precio * 0.19
    return iva  # Retorna el valor calculado

# Uso de la función con return
precio_producto = 10000
monto_iva = calcular_iva(precio_producto)  # Capturamos el valor retornado
total = precio_producto + monto_iva

print(f"IVA calculado: ${monto_iva:,.0f}")
print(f"Total a pagar: ${total:,.0f}")
```

Ejemplo de Función sin return (procedimiento)

```python
def mostrar_menu():
    """Muestra un menú de opciones (no retorna valor)"""
    print("\nMENÚ PRINCIPAL")
    print("1. Consultar saldo")
    print("2. Realizar depósito")
    print("3. Salir")

# Uso de la función sin return
mostrar_menu()  # Solo ejecuta el código interno
opcion = input("Seleccione una opción: ")
```

Recursividad

La recursividad es una técnica donde una función se llama a sí misma para resolver un problema, que se puede dividir en subproblemas más pequeños del mismo tipo. Cualquier función recursiva tiene dos secciones de código divididas:

- Por un lado, tenemos la sección en la que la función se llama a sí misma.
- Por otro lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse. Es muy importante, porque de lo contrario, la función se llamaría de forma infinita.

Ejemplo de recursividad con cálculo de factorial

```
def factorial(n):
    # Caso base: factorial de 0 o 1 es 1
    if n == 0 or n == 1:
        return 1
    # Caso recursivo: n! = n * (n-1)!
    else:
        return n * factorial(n - 1)

# Prueba la función
numero = 5
print(f"El factorial de {numero} es {factorial(numero)}")
# Salida: El factorial de 5 es 120
```

Conclusión

Las estructuras de datos y las funciones en Python son pilares fundamentales para el desarrollo de programas eficientes y organizados. Las estructuras de datos, permiten almacenar y manipular información de manera flexible, adaptándose a diferentes necesidades. Por otro lado, las funciones facilitan la reutilización de código y mejoran la legibilidad, mientras que la recursividad ofrece una solución para problemas que pueden dividirse en subproblemas más pequeños.

Dominar estos conceptos no solo optimiza el rendimiento de los programas, sino que también sienta las bases para abordar desafíos más

complejos en el desarrollo de software, con una sintaxis clara y una amplia gama de herramientas.

Referencias:

- Python. (2025). *Data Structures*. Recuperado el 29 de julio de 2025, de https://docs.python.org/3/tutorial/datastructures.html
- Python. (2025). *Efficient arrays of numeric values.* Recuperado el 29 de julio, de 2025 de https://docs.python.org/3/library/array.html
- Python. (2025). *Built -in Types*. Recuperado el 30 de julio de 2025, de https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
- Python. (2025). *Built-in Functions.* Recuperado el 31 de julio de 2025, de https://docs.python.org/3/library/functions.html
- Python. (2025). *More Control Flow Tools*. Recuperado el 31 de julio de 2025, de https://docs.python.org/3/tutorial/controlflow.html#defining-function
- Carrillo, E. (2021). *Funciones en Python*. Recuperado el 31 de Julio de 2025, de https://aulavirtual.espol.edu.ec/courses/4558/pages/funciones-en-python

Elaboró contenido: Sebastián Romero Zapata