



[LCD] MÓDULO 3

Bienvenido al **Módulo 3** del curso **Lenguaje de Ciencia de Datos I.**

INTRODUCCIÓN

Estimado estudiante:

En el presente módulo aprenderás los fundamentos de la programación modular en Python, comprendiendo cómo dividir un programa en partes más pequeñas y reutilizables. Analizarás el uso de funciones incorporadas, la biblioteca estándar, módulos especializados y la creación de funciones propias, además del manejo de excepciones para controlar errores de forma eficiente.

Te deseamos una excelente experiencia de aprendizaje en este curso.

¡Muchos éxitos!

 TEMA 3: PROGRAMACIÓN MODULAR

 CIERRE

TEMA 3: PROGRAMACIÓN MODULAR

PROGRAMACIÓN MODULAR

Llegó el momento para que te conviertas en el protagonista de tu aprendizaje.

- Descubre y analiza la información que se presenta en la lección interactiva para transformarla en nuevo conocimiento.

La programación modular permite crear sistemas complejos a partir de partes simples, mejorando la organización, la colaboración y la eficiencia en el desarrollo de software.



La programación modular constituye uno de los pilares fundamentales en la ingeniería de software, pues facilita la construcción de sistemas complejos mediante la división del código en módulos más pequeños, coherentes y reutilizables. Este enfoque no solo favorece la organización y el mantenimiento del código, sino que también promueve la colaboración entre equipos de trabajo, la escalabilidad de proyectos y la detección temprana de errores.

En este tema, se abordan los principios y herramientas de la programación modular en Python, incluyendo el uso de funciones incorporadas, la biblioteca estándar, los módulos especializados, la creación de funciones definidas por el usuario y el manejo de excepciones. El objetivo es que los estudiantes adquieran una visión integral y aplicada de esta estrategia de programación.

Funciones Incorporadas y Biblioteca Estándar

Python provee un conjunto de más de 60 **funciones incorporadas (built-in functions)** que permiten realizar operaciones frecuentes sin necesidad de importar módulos adicionales. Estas funciones abarcan tareas como la obtención de longitudes (`len()`), la suma de elementos (`sum()`), la conversión de tipos (`int()`, `float()`, `str()`), la búsqueda de valores máximos o mínimos (`max()`, `min()`), o la generación de secuencias (`range()`).

Al estar optimizadas en su implementación, ofrecen **alto rendimiento, portabilidad y confiabilidad**, lo que las convierte en herramientas esenciales para el desarrollo cotidiano en Python.

Funciones y Módulos de Python



Funciones incorporadas

Funciones que simplifican operaciones frecuentes, optimizadas para rendimiento.



Biblioteca Estándar

Módulos que amplían el alcance del lenguaje, incluyendo utilidades para diversas tareas.

Por otro lado, la [biblioteca estándar de Python](#) constituye una colección extensa de módulos preinstalados que amplían las capacidades del lenguaje. Permite realizar desde operaciones matemáticas avanzadas (math) y manejo de fechas (datetime), hasta tareas como el control de archivos y directorios (os, shutil), creación de números aleatorios (random), validación de texto mediante expresiones regulares (re) o incluso la configuración de servidores web básicos (http).

Su propósito es reducir la necesidad de reinventar soluciones, brindando componentes listos para usar y con documentación oficial actualizada.

Ejemplo de funciones incorporadas

A continuación se presentan algunos ejemplos de funciones incorporadas en Python que se utilizan con frecuencia para realizar operaciones básicas. Estas funciones permiten trabajar con distintos tipos de datos de manera sencilla y eficiente, facilitando el desarrollo de programas más claros y organizados.

Función	Uso	Resultado
<code>len('Python')</code>	Devuelve la longitud de la cadena	6
<code>sum([1,2,3])</code>	Suma elementos de una lista	6
<code>type(3.14)</code>	Devuelve el tipo de dato	<class 'float'>

<code>max([10, 25, 7])</code>	Devuelve el valor máximo	25
<code>round(8.567, 2)</code>	Redondea un número a dos decimales	8.57

Ejercicios prácticos

EJERCICIO 1: UTILIZA FUNCIONES INCORPORADAS PARA DETERMINAR

EJERCICIO 2: IMPORTA EL MÓDULO 'MATH' Y CALCULA EL VALOR DE

- a. La longitud de la palabra 'Modularidad'.
- b. El valor máximo y mínimo de la lista [3, 7, 2, 9, 5].
- c. La suma de los elementos en la lista anterior.

Solución:

```
print(len('Modularidad')) # 11
print(max([3, 7, 2, 9, 5])) # 9
print(min([3, 7, 2, 9, 5])) # 2
print(sum([3, 7, 2, 9, 5])) # 26
```

EJERCICIO 1: UTILIZA FUNCIONES INCORPORADAS PARA DETERMINAR

EJERCICIO 2: IMPORTA EL MÓDULO 'MATH' Y CALCULA EL VALOR DE

- a. La raíz cuadrada de 144.
- b. El valor del coseno de 0 radianes.

Solución:

```
import math
```

```
print(math.sqrt(144)) # 12.0  
print(math.cos(0)) # 1.0
```

Las funciones incorporadas y la biblioteca estándar representan la base operativa de Python. Gracias a ellas, los programadores pueden resolver problemas comunes de manera rápida y confiable, sin depender de librerías externas. Comprender su uso permite desarrollar código más eficiente, legible y profesional, sentando las bases para aprovechar posteriormente módulos especializados y técnicas de programación más avanzadas.

CONTINUAR

Módulos Especializados

Cuando la biblioteca estándar no cubre todas las necesidades, Python permite instalar módulos externos desde el repositorio PyPI. Estos módulos especializados son ampliamente usados en áreas como ciencia de datos, inteligencia artificial, ingeniería y desarrollo web.

Entre los más destacados se encuentran:

NumPy

Operaciones con
vectores y matrices
de gran tamaño.

Pandas

Manipulación y
análisis de datos en
estructuras de tipo
DataFrame.

Matplotlib y Seaborn

Visualización de
datos mediante
gráficos.

Requests

Consumo de
servicios web y
APIs.

Flask y Django

Frameworks para el
desarrollo de
aplicaciones web.



Su ventaja radica en que son mantenidos por comunidades activas que actualizan y mejoran continuamente las funcionalidades, lo cual refleja el valor de la colaboración en el ecosistema Python.

Ejercicios prácticos

EJERCICIO 1: USANDO NUMPY, CREA UN ARREGLO CON LOS NÚMEROS DEL 1 AL 5 Y CALCULA SU PROMEDIO.

EJERCICIO 2: UTILIZANDO PANDAS, CREA UN DATAFRAME CON LAS COLUMNAS 'PRODUCTO' Y 'PRECIO', E IMPRIME EL PRECIO PROMEDIO.

Solución:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])  
print(arr.mean()) # 3.0
```

EJERCICIO 1: USANDO NUMPY, CREA UN ARREGLO CON LOS NÚMEROS DEL 1 AL 5 Y CALCULA SU PROMEDIO.

EJERCICIO 2: UTILIZANDO PANDAS, CREA UN DATAFRAME CON LAS COLUMNAS 'PRODUCTO' Y 'PRECIO', E IMPRIME EL PRECIO PROMEDIO.

Solución:

```
import pandas as pd  
data = {'Producto': ['A', 'B', 'C'], 'Precio': [10, 20, 30]}  
df = pd.DataFrame(data)  
print(df['Precio'].mean()) # 20.0
```

CONTINUAR

Funciones definidas por el usuario

Más allá de las funciones incorporadas, los programadores pueden **crear sus propias funciones** para resolver problemas específicos o repetir tareas de forma controlada. Este enfoque fomenta la **abstracción**, la **organización del código** y la **reutilización**, ya que una función puede ser llamada varias veces sin necesidad de reescribir instrucciones.

Una función definida por el usuario en Python incluye:

- **Un nombre identificador** (que describe su propósito).
- **Parámetros** o variables de entrada.
- **Un bloque de instrucciones** que ejecuta una tarea.
- **Un valor de retorno** mediante la palabra clave return, que devuelve el resultado al programa principal.



Estructura general

```
def nombre_funcion(parametros):  
    # Bloque de código  
    return resultado
```



Ejemplo práctico

```
def calcular_promedio(lista):  
    return sum(lista) / len(lista)  
  
notas = [15, 18, 12, 20]  
print('Promedio:', calcular_promedio(notas))
```



Salida

```
Promedio: 16.25
```



Las funciones también pueden incluir documentación interna (docstrings) para explicar su propósito:

```
def es_par(numero):  
    """Verifica si un número es par."""  
  
    return numero % 2 == 0  
  
print(es_par(10))  # True
```



Además, pueden usarse con parámetros opcionales o valores por defecto, lo que brinda flexibilidad al programador:

```
def saludar(nombre='estudiante'):  
    print('Hola,', nombre)  
  
saludar()          # Hola, estudiante  
saludar('Claudia') # Hola, Claudia
```



Definir funciones propias permite al programador estructurar el código en bloques reutilizables, mejorar la claridad del programa y facilitar su mantenimiento. En proyectos más grandes, esta práctica es clave para lograr un diseño modular, donde cada función cumple una responsabilidad específica dentro del sistema, contribuyendo así a la eficiencia y calidad del desarrollo en Python.

CONTINUAR

Manejo de Excepciones

En programación, los errores son inevitables y forman parte natural del proceso de ejecución. Python introduce el **manejo de excepciones** como un mecanismo que permite **detectar, controlar y responder a errores en tiempo de ejecución**. Su

propósito es evitar que el programa se detenga de forma abrupta y ofrecer mensajes claros al usuario o al programador.

Las excepciones se activan cuando ocurre un error —por ejemplo, intentar dividir entre cero o convertir texto en número— y se pueden **capturar mediante la estructura try-except**, que brinda control sobre el flujo del programa.



Estructura general

```
try:  
    # Código que puede generar un error  
except TipoDeError:  
    # Código que maneja el error  
else:
```

```
# Código que se ejecuta si no ocurre ningún error
finally:
    # Código que siempre se ejecuta (ocurra o no un error)
```

Ejemplo básico

```
try:
    numero = int(input('Ingrese un número: '))
    print('Número al cuadrado:', numero**2)
except ValueError:
    print('Error: debe ingresar un número entero')
finally:
    print('Ejecución finalizada')
```

Ejemplo con múltiples excepciones

A veces, un programa puede enfrentar distintos tipos de errores. En estos casos, es posible manejar varios tipos de excepciones:

```
try:
    num = int(input('Ingrese un número divisor: '))
    resultado = 10 / num
    print('Resultado:', resultado)
except ValueError:
    print('Error: debe ingresar un número válido')
except ZeroDivisionError:
    print('Error: no se puede dividir entre cero')
else:
    print('Operación realizada correctamente')
```

```
finally:  
    print('Fin del programa')
```

En este ejemplo, el programa distingue entre un error de tipo de dato (*ValueError*) y una división inválida (*ZeroDivisionError*), garantizando una respuesta adecuada para cada situación.

Importancia en la programación modular

El manejo de excepciones es un **componente esencial en la programación modular**, ya que cada módulo o función puede incluir su propio control de errores. Esto mejora la **robustez, confiabilidad y estabilidad del sistema**, al evitar que un fallo local afecte la ejecución completa del programa. Además, facilita la depuración y el mantenimiento, al registrar errores de manera controlada en archivos de log o en la consola.

El manejo de excepciones permite diseñar programas seguros, claros y resistentes a fallos, manteniendo la integridad del flujo de ejecución. Integrar este mecanismo en cada módulo es una práctica profesional que refleja la madurez del programador, especialmente en proyectos colaborativos o de gran escala.

En conjunto con las funciones y la modularidad, las excepciones convierten a Python en un lenguaje poderoso para construir soluciones

confiables, escalables y fáciles de mantener.

CONTINUAR



Sistema modular de control de inventarios

Una librería universitaria maneja un inventario de libros y desea automatizar tareas básicas con Python. El sistema debe ser modular: separar lectura de datos, validaciones, lógica de negocio, reportes y manejo de errores. Este caso permite integrar funciones propias, uso de módulos estándar y excepciones.

Objetivos de aprendizaje

- Aplicar programación modular para estructurar un programa de tamaño medio.
- Diseñar y reutilizar funciones con parámetros y valores de retorno.

- Usar módulos de la biblioteca estándar (csv, datetime, statistics, pathlib) de forma pertinente.
- Implementar manejo de excepciones para entradas inválidas y archivos ausentes.
- Generar reportes tabulares y métricas básicas del inventario.

Requisitos funcionales

El programa debe:

1. Leer un archivo csv con las columnas:
codigo, isbn, título, autor, categoría, stock, precio.
2. Validar tipos de datos y reglas ($\text{stock} \geq 0$, $\text{precio} > 0$).
3. Permitir:
 - a. Buscar libro por título o código,
 - b. Registrar venta (disminuye stock),
 - c. Reponer stock,
 - d. Calcular valor total del inventario,
 - e. Generar un top 5 por categoría por precio.
4. Registrar en un log de texto los eventos relevantes (fecha/hora, operación, resultado).
5. Manejar excepciones (archivo no encontrado, csv malformado, valores no numéricos, stock insuficiente).

Datos de ejemplo (inventario.csv)

Puedes crear un archivo csv con el siguiente contenido para pruebas:

Código	isbn	Título	Autor	Categoría	Stock	Precio
L001	9780135166307	Clean Code	Robert C. Martin	software	8	150.00
L002	9781492051367	Fluent Python	Luciano Ramalho	software	5	220.00
L003	9780132350884	Clean Architecture	Robert C. Martin	software	3	180.00
L004	9780262033848	Introduction to Algorithms	Cormen et al.	algoritmos	4	300.00
L005	9780596007126	Head First Design Patterns	Freeman	software	6	190.00
L006	9780131101630	The C Programming Language	Kernighan & Ritchie	programación	2	130.00

Arquitectura propuesta (módulos y responsabilidades)

- datos.py: funciones de acceso a datos (leer_csv, escribir_log).
- validacion.py: validadores (es_numero_positivo, validar_fila).
- negocio.py: operaciones de dominio (buscar, vender, reponer, valor_inventario, top5_categoria).
- ui.py (opcional): interfaz simple por consola para orquestar el flujo.



Para el caso, se presenta una implementación en un solo archivo con secciones modulares.

Solución de referencia (script único con secciones modulares)

```
from pathlib import Path
import csv
from datetime import datetime
from statistics import mean

# ----- utils de logging -----
def escribir_log(msg, ruta='inventario.log'):
    marca = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    with open(ruta, 'a', encoding='utf-8') as f:
        f.write(f"[{marca}] {msg}\n")
```

```

# ----- acceso a datos -----
def leer_csv(ruta):
    try:
        with open(ruta, newline='', encoding='utf-8') as f:
            lector = csv.DictReader(f)
            datos = [fila for fila in lector]
    if not datos:
        raise ValueError('el archivo está vacío')
    return datos
except FileNotFoundError as e:
    escribir_log(f'ERROR: archivo no encontrado -> {ruta}')
    raise e
except Exception as e:
    escribir_log(f'ERROR al leer csv: {e}')
    raise e

# ----- validación -----
def es_float_pos(v):
    try:
        return float(v) > 0
    except:
        return False

def es_int_no_neg(v):
    try:
        return int(v) >= 0
    except:
        return False

def validar_fila(f):
    req = all(k in f for k in ('codigo','isbn','titulo','autor','categoria','stock','precio'))
    tipos = es_int_no_neg(f['stock']) and es_float_pos(f['precio'])
    if not (req and tipos):
        raise ValueError(f"fila inválida: {f}")

# ----- lógica de negocio -----
def normalizar(regs):
    for r in regs:
        validar_fila(r)
        r['stock'] = int(r['stock'])
        r['precio'] = float(r['precio'])
    return regs

def buscar(regs, termino):
    t = termino.lower()
    return [r for r in regs if t in r['titulo'].lower() or t == r['codigo'].lower()]

def vender(regs, codigo, cantidad=1):
    for r in regs:
        if r['codigo'].lower() == codigo.lower():
            if r['stock'] < cantidad:
                raise ValueError('stock insuficiente')

```

```

        r['stock'] -= cantidad
        escribir_log(f'venta: {codigo} x{cantidad}')
        return r
    raise LookupError('código no encontrado')

def reponer(regs, codigo, cantidad=1):
    for r in regs:
        if r['codigo'].lower() == codigo.lower():
            r['stock'] += cantidad
            escribir_log(f'reposición: {codigo} +{cantidad}')
            return r
    raise LookupError('código no encontrado')

def valor_inventario(regs):
    return round(sum(r['stock']*r['precio'] for r in regs), 2)

def top5_categoria(regs, categoria):
    items = [r for r in regs if r['categoria'].lower()==categoria.lower()]
    return sorted(items, key=lambda r: r['precio'], reverse=True)[:5]

# ----- orquestación mínima -----
def main(ruta='inventario.csv'):
    try:
        regs = normalizar(leer_csv(ruta))
        print('total de ítems:', len(regs))
        print('valor del inventario:', valor_inventario(regs))
        print('buscar "clean":', [r['titulo'] for r in buscar(regs, 'clean')])
        vender(regs, 'L001', 2)
        reponer(regs, 'L003', 1)
        print('valor actualizado:', valor_inventario(regs))
        print('top 5 software:', [r['titulo'] for r in top5_categoria(regs, 'software')])
    except Exception as e:
        print('ocurrió un error:', e)

if __name__ == '__main__':
    main()

```

Pruebas y resultados esperados

- Al iniciar, debe mostrar el total de ítems y el valor del inventario con 2 decimales.
- La búsqueda de 'clean' devuelve títulos que contengan esa palabra (p. ej., 'Clean Code', 'Clean Architecture').

- Vender L001 x2 reduce el stock de 8 a 6; reponer L003 +1 aumenta de 3 a 4.
- El valor del inventario debe actualizarse tras las operaciones.
- El top 5 por categoría 'software' lista los cinco libros de mayor precio.

Extensiones opcionales (retos)

- Persistencia de cambios: guardar el inventario actualizado en un nuevo csv.
- Reporte por consola en tabla alineada (módulo 'textwrap' o f-strings con anchuras fijas).
- Interfaz de menú y argumentos por línea de comandos (módulo 'argparse').
- Exportar un resumen a json (módulo 'json') y graficar top categorías (usar matplotlib).

CONTINUAR

La programación modular es un pilar esencial en el diseño de software moderno. Su aplicación en Python permite construir sistemas eficientes, escalables y sostenibles mediante el uso de funciones, módulos y estructuras de control de errores. Al dominar estos principios,

el estudiante no solo adquiere habilidades técnicas, sino también una visión integral de cómo estructurar soluciones informáticas limpias, colaborativas y de calidad profesional.



[CONTINUAR](#)

Para finalizar el tema, recordemos algunas ideas

claves.

1

La programación modular permite dividir un sistema complejo en componentes más pequeños, llamados módulos, que pueden desarrollarse, probarse y mantenerse de manera independiente. Este enfoque mejora la organización del código, facilita el trabajo en equipo y reduce los errores durante la ejecución.

2

Las **funciones incorporadas** y la **biblioteca estándar de Python** proporcionan herramientas listas para usar que optimizan tareas comunes, como cálculos, manejo de archivos y operaciones con datos. Su uso eficiente constituye la base para construir programas más sólidos y reutilizables.

3

Las **funciones definidas por el usuario** permiten personalizar soluciones a problemas específicos. Su correcta implementación fomenta la reutilización del código, la claridad en la estructura del programa y la capacidad de adaptación a distintos contextos de desarrollo.

4

El **manejo de excepciones** garantiza la estabilidad del software al anticipar y controlar errores en tiempo de ejecución. Integrarlo dentro de cada módulo favorece la creación de programas robustos, seguros y capaces de responder de forma controlada ante imprevistos.

CONTINUAR

CIERRE

