Programación para Física y Astronomía

Departamento de Física.

Coordinadora: C Loyola
Profesores C Femenías / F Bugini / D Basantes

Primer Semestre 2025

Universidad Andrés Bello
Departamento de Física y Astronomía

Universidad Andrés Bello®

python powered

Resumen - Semana 3, Sesión 1 (Sesión 5)

Repaso y Contexto

Funciones en Python

Parámetros y Alcance

Módulos y Paquetes

Ejercicios Guiados

Actividad Práctica

Conclusiones

Repaso y Contexto

Repaso y Contexto ∋ Recapitulación de la Sesión Anterior (Sesión 5)

- **Semana 3, Sesión 1 (Sesión 5)** se centró en:
  - Laboratorio práctico con estructuras de control (`if`, `while`).
  - Problemas tipo: “Adivina el número”, “Calculadora de Calificaciones”, “Movimiento Discreto”.
  - Manejo de validaciones, contadores y salidas controladas de bucles.

Ejemplo: `def f(x): x=10` — la variable `x` vive solo en `f(x)`.

- **Comprender** la sintaxis y concepto de funciones en Python (`def`, parámetros, `return`).
- **Explorar** cómo organizar código en módulos y paquetes simples.
- **Aplicar** estos conceptos a pequeños proyectos para mejorar reutilización y claridad.
- **Conectar** funciones con los temas previos (estructuras de control, manejo de datos, etc.).

Funciones en Python

Funciones en Python ∋ ¿Por qué usar Funciones?

- **Reutilización de Código**: evitar escribir la misma lógica en múltiples lugares.
- **Organización**: encapsular tareas específicas en “cajas negras”.
- **Legibilidad**: el nombre de la función describe la operación que realiza.
- **Mantenibilidad**: cambiar la lógica en un solo lugar (la definición de la función).

```python
def nombre_de_funcion(param1, param2, ...):
    """
    Documentación opcional (docstring).
    Explica qué hace la función, los parámetros y el retorno.
    """
    # bloque de código
    # opcionalmente retornar un valor
    return resultado
```

- **Parámetros**: valores de entrada que la función utiliza.
- **return**: finaliza la función y opcionalmente devuelve un valor.
- **Mejor práctica**: Usar nombres descriptivos para parámetros y funciones.

Funciones en Python ∋ Ejemplo: Función para Calcular el Área de un Círculo

```python
import math

def area_circulo(radio):
    """
    Retorna el área de un círculo de radio 'radio'.
    Formula: pi * r^2
    """
    area = math.pi * (radio**2)
    return area

# Uso de la función
r = 5
a = area_circulo(r)
print("El área del círculo es:", a)
```

- **Encapsulamos** la operación en `area_circulo`.
- **Parámetro** `radio`.
- **Retorno** con `return area`.

Parámetros y Alcance

```python
def calcular_salario(base, horas_extra=0):
    """Calcula salario sumando base + 2000 por hora extra"""
    return base + 2000 * horas_extra

s1 = calcular_salario(50000)      # horas_extra por defecto: 0
s2 = calcular_salario(50000, 3)   # 3 horas extra
print(s1, s2)
```

- **Parámetros con valor por defecto** (ej. `horas_extra=0`).
- **Posicionales**: deben ir en orden; `calcular_salario(50000, 3)`.
- **Keyword arguments**: `calcular_salario(base=50000, horas_extra=3)`.

- **Local**: Variables definidas dentro de la función solo existen dentro de ella.
- **Global**: Variables definidas fuera de cualquier función son accesibles dentro, pero no se recomienda modificarlas sin razón.
- **Mejor práctica**: Mantener funciones “puras”, evitando depender de variables globales.

Ejemplo: `def f(): x=10` — la variable `x` vive solo en `f()`.

Parámetros y Alcance ∋ Conflicto Global vs Local

```python
x = 5  # global

def foo():
    x = 10  # local
    print("Dentro de foo, x =", x)

foo()
print("Fuera de foo, x =", x)
```

- Impresiones:
  - “Dentro de foo, x = 10”
  - “Fuera de foo, x = 5”
- **Evitar confusiones** usando nombres significativos y parámetros claros.

Módulos y Paquetes

- Un **módulo** es un archivo `.py` que contiene definiciones de funciones, variables y clases.
- **Ventaja**: podemos importar este archivo desde otros scripts y reutilizar el código.
- **Ejemplo**: Crear un archivo `mifunciones.py` con varias funciones y luego:

```
import mifunciones
```

```python
# mifunciones.py

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

def resta(a, b):
    return a - b
```

Uso en otro archivo:

```python
import mifunciones

print(mifunciones.suma(3, 4))
print(mifunciones.resta(10, 2))
```

Módulos y Paquetes ∋ Importaciones Específicas

```python
from mifunciones import suma

resultado = suma(5, 7)
print(resultado)
```

- Trae solo la función `suma` del módulo.
- **Cuidado**: peligro de colisiones de nombre (si hay otra `suma`).
- `from mifunciones import *` trae todo, también puede causar conflictos.

- Un **paquete** es una carpeta que contiene un archivo `__init__.py` y varios módulos `.py`.
- Estructura típica: mypackage/modulo1.py
  mypackage/modulo1.py
- Para usarlo: `import mipaquete.modulo1`.

Uso: Organizar proyectos grandes en submódulos lógicamente separados.

Ejercicios Guiados

Ejercicios Guiados ∋ Ejercicio 1: Suma Condicional

Enunciado

Crear una función que reciba dos números y realice lo siguiente:

- Si ambos números son positivos, retornar su suma.
- Si uno de los números es negativo, retornar su división.
- Si ambos números son negativos, retornar su producto.

Conceptos: Uso de estructuras condicionales (`if`, `elif`, `else`).

Ejercicios Guiados ∋ Ejercicio 2: Clasificación de Números

Enunciado

Crear una función que reciba una lista de números y clasifique cada número como:

- Positivo.
- Negativo.
- Cero.

La función debe imprimir el resultado para cada número y retornar la cantidad de números positivos, negativos y ceros.

Conceptos: Uso de ciclos `for` y estructuras condicionales.

Ejercicios Guiados ∋ Ejercicio 3: Suma de Números con Validación

Enunciado

Crear una función que reciba un número entero *n* y realice lo siguiente:

- Solicitar al usuario *n* números enteros.
- Validar que cada número ingresado sea mayor que cero. Si no lo es, volver a solicitar el número.
- Retornar la suma de los *n* números ingresados.

Conceptos: Uso de ciclos `while` y validaciones.

Ejercicios Guiados ∋ Ejercicio 4: Serie de Fibonacci

Enunciado

Crear una función que reciba un número entero *n* y retorne los primeros *n* términos de la serie de Fibonacci.

- La serie de Fibonacci comienza con 0 y 1.
- Cada término siguiente es la suma de los dos anteriores.

Conceptos: Uso de ciclos `for` y listas.

Ejercicios Guiados ∋ Ejercicio 5: Verificación de Número Primo

Enunciado

Crear una función que reciba un número entero y determine si es primo o no.

- Un número primo es divisible únicamente por 1 y por sí mismo.
- La función debe retornar `True` si el número es primo y `False` en caso contrario.

Conceptos: Uso de ciclos `for` y estructuras condicionales.

Ejercicios Guiados ∋ Solución 1 de Referencia: Suma Condicional ✓

```python
def suma_condicional(a, b):
    """
    Realiza operaciones según los valores de a y b:
    - Si ambos son positivos, retorna su suma.
    - Si uno es negativo, retorna su división.
    - Si ambos son negativos, retorna su producto.
    """
    if a > 0 and b > 0:
        return a + b
    elif a < 0 and b < 0:
        return a * b
    else:
        return a / b

# Ejemplo de uso
resultado = suma_condicional(5, -3)
print("Resultado:", resultado)
```

Discusión: ¿Qué sucede si $b = 0$ en la división? ¿Cómo manejar este caso?

Ejercicios Guiados ∋ Solución 2 de Referencia: Clasificación de Números ✓

```python
def clasificar_numeros(lista):
    """
    Clasifica números en positivos, negativos y ceros.
    Retorna la cantidad de cada tipo.
    """
    positivos = 0
    negativos = 0
    ceros = 0

    for num in lista:
        if num > 0:
            print(f"{num} es positivo")
            positivos += 1
        elif num < 0:
            print(f"{num} es negativo")
            negativos += 1
        else:
            print(f"{num} es cero")
            ceros += 1

    return positivos, negativos, ceros

# Ejemplo de uso
resultado = clasificar_numeros([3, -1, 0, 7, -5])
print("Positivos, Negativos, Ceros:", resultado)
```

Discusión: ¿Cómo manejar listas vacías o valores no numéricos?

Ejercicios Guiados ∋ Solución 3 de Referencia: Suma de Números con Validación ✓

```python
def suma_con_validacion(n):
    """
    Solicita n números positivos al usuario y retorna su suma.
    Valida que cada número ingresado sea mayor que cero.
    """
    suma = 0
    contador = 0

    while contador < n:
        num = float(input(f"Ingrese el número {contador + 1}: "))
        if num > 0:
            suma += num
            contador += 1
        else:
            print("El número debe ser mayor que cero. Intente de nuevo.")

    return suma

# Ejemplo de uso
resultado = suma_con_validacion(3)
print("Suma total:", resultado)
```

Discusión: ¿Cómo manejar entradas no numéricas o interrupciones del usuario?

Ejercicios Guiados ∋ Solución 4 de Referencia: Serie de Fibonacci ✓

```python
def fibonacci(n):
    """
    Retorna los primeros n términos de la serie de Fibonacci.
    """
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    serie = [0, 1]
    for i in range(2, n):
        siguiente = serie[-1] + serie[-2]
        serie.append(siguiente)

    return serie

# Ejemplo de uso
resultado = fibonacci(10)
print("Serie de Fibonacci:", resultado)
```

Discusión: ¿Cómo optimizar para valores grandes de n?

Ejercicios Guiados ∋ Solución 5 de Referencia: Verificación de Número Primo ✓

```python
def es_primo(num):
    """
    Determina si un número es primo.
    Retorna True si es primo, False si no.
    """
    if num < 2:
        return False
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

# Ejemplo de uso
resultado = es_primo(13)
print("¿Es primo?", resultado)
```

Discusión: ¿Cómo manejar números negativos o muy grandes?

Actividad Práctica

Actividad Práctica ∋ Actividad Extra 1: Función Avanzada

Enunciado

Crear una función que calcule el factorial de un número de forma recursiva (que la función se llame a sí misma).

Objetivo: Aplicar recursión.

Actividad Práctica ∋ Actividad Extra 2: Módulo Personalizado

Enunciado

Crear un módulo con funciones para calcular el área y perímetro de figuras geométricas básicas (círculo, cuadrado, triángulo).

Objetivo: Modularizar el código y practicar importaciones.

Actividad Práctica ∋ Actividad Extra 3: Proyecto Integrador

Enunciado

Diseñar un programa que utilice funciones y módulos para resolver un problema práctico, como una calculadora científica básica.

Objetivo: Integrar conceptos de funciones, módulos y estructuras de control.

Actividad Práctica ∋ Actividad Extra 4: Creando Tus Módulos

- **Objetivo**: Organizar funciones útiles en un módulo y probar su importación.
- **Instrucciones**:
  1. Crea un archivo `utilidades.py` con al menos 3 funciones (ej. `factorial(n)`, `es_primo(n)`, etc.).
  2. En un **notebook de Colab** o un script `main.py`, importa `utilidades` y prueba dichas funciones.
  3. (Opcional) Añade una cuarta función con un **parámetro por defecto** (`def hola_mundo(nombre="Mundo")`, etc.).

Sugerencia: Comenta tu código y explica la lógica de cada función.
Reflexión: ¿Cómo ayuda la modularización a proyectos más grandes?

Actividad Práctica ∋ Trabajo Colaborativo

- Formar **parejas o tríos**.
- Diseñar **un pequeño módulo** de funciones:
  - Matemáticas, Estadística básica, Conversión de unidades, etc.
- Utilizar dichas funciones en **otro script** o **notebook**.
- **Opcional**: Explorar la creación de una carpeta con `__init__.py` para armar un paquete simple.

Actividad Práctica ∋ Espacio para Dudas y Apoyo

- ¿Problemas al importar el módulo en Colab?
- ¿Cómo organizar los archivos en Google Drive?
- ¿Errores de `ModuleNotFoundError`?

Consulta en voz alta o pide ayuda a tus compañeros.

Actividad Práctica ∋ Retroalimentación

- ¿Se entendió la **separación** entre la lógica (en un módulo .py) y el código principal?
- ¿Ventajas de tener todo en un solo archivo vs. múltiples módulos?
- ¿Dudas sobre parámetros y valores por defecto?

Conclusiones

- Apreciamos la **modularización** del código para mejor legibilidad y mantenimiento.
- Vimos **funciones**: sintaxis `def`, parámetros, `return`, docstring.
- Exploramos la creación de **módulos** y su importación en otros scripts o notebooks.
- Ahora estamos preparados para proyectos más grandes y ordenados.

- **Sesion 8 (Semana 4)**: Continuaremos con el uso de módulos y paquetes, y profundizaremos en el uso de `pip` y librerías externas.
- **Tarea sugerida**:
  - Crear un pequeño proyecto con un paquete `mipaquete/` y varios módulos (ej. `mipaquete/calculos.py`, `mipaquete/utiles.py`, etc.).
  - Probar importarlos y usarlos en un script principal.

¡Gracias y hasta la próxima sesión!

- Guarda tus **notebooks** y archivos `.py` en Google Drive.
- Explora la documentación oficial de Python (sobre **funciones** y **módulos**).
- ¡Sigue practicando!