NOTAS DE CLASE
EL LENGUAJE PYTHON

''' EL LENGUAJE DE PROGRAMACIÓN PYTHON '''

Jaime E. Montoya M.

NOTAS DE CLASE
EL LENGUAJE PYTHON

''' EL LENGUAJE DE PROGRAMACIÓN PYTHON '''

```
"""
* Versión 3.0
* Fecha: 2026
* Licencia software: GNU GPL
* Licencia doc: GNU Free Document License (GNU FDL)
"""


class Author:

    name = “Jaime E. Montoya M.”

    profession = “Ingeniero Informático”

    employment = “Docente y desarrollador”

    city = “Medellín - Antioquia - Colombia”

    year = 2026
```

Tabla de contenido

Introducción 6
Capítulo 1. Introducción al lenguaje Python 7
- Descarga e instalación 7
- Características de hardware y software 7
- Modo consola o intérprete 8
  - Crear scripts 9
  - Ejecutar scripts 9
- Convenciones de nombres 9
- Comentarios 9
  - Docstrings (cadenas literales de documentación) 10
- Dividir una sentencia en varias líneas 10
- Salida estándar 11
- Constantes y variables 11
- Tipos de datos 11
  - Numéricos 11
  - Cadenas 11
  - Booleanos 12
  - Secuencias 12
  - Función type() 12
- Entrada estándar 12
- Conversión de tipos o casteo de variables 12
  - Cadenas 12
  - Enteros 12
  - Reales (punto flotante) 13
  - Listas 13
  - Conjuntos 13
  - Binarios 13
  - Octales 13
  - Hexadecimales 13
- Operadores 14
  - Operadores aritméticos 14
    - Operación de módulo y división entera 14
  - Operadores relacionales o de comparación 15
  - Operadores lógicos (booleanos) 15
  - Operador de asignación 16
  - Otros operadores aritméticos de asignación 16
  - Operador de concatenación de cadenas 17
  - Operadores de pertenencia 17
  - Operadores de identidad 17

Plantillas literales 17
Establecer color al texto en entorno de terminal 18
Preguntas 19
Ejercicios 19
Capítulo 2. Estructuras de control 20
Condicionales 20
Condicional if else elif 20
Selector múltiple o estructura caso: sentencia match 22
Ciclos 23
Ciclo while 23
Ciclo for 25
Sentencias de bifurcación de control 27
break 27
continue 27
exit() 28
return 29
pass 29
Capítulo 3. Funciones 30
Funciones incorporadas o predefinidas 30
Métodos y funciones matemáticas 30
Funciones matemáticas. La clase math 30
Números pseudo aleatorios. La clase random 31
Números complejos. La clase complex 32
Métodos para la manipulación de cadenas de caracteres. La clase str 32
Métodos para la manipulación de fechas y horas. La clase datetime 33
Funciones definidas por el programador 34
Problemas resueltos 35
Preguntas 39
Ejercicios 39
Capítulo 4. Datos estructurados 40
Listas, tuplas, conjuntos y diccionarios 40
Listas 40
Declaración de listas 40
Métodos de listas 42
Tuplas 46
Declaración de tuplas 46
Métodos de tuplas 47
Desempaquetar una tupla (tuple unpacking) 48
Conjuntos 48
Diccionarios 50
Espacios de nombres 51
Módulos 51
Paquetes 51
Capítulo 5. Programación Orientada a Objetos (POO) 52

La Programación Orientada a Objetos en Python 52
Preguntas 55
Ejercicios 55
Capítulo 6. Archivos 56
Archivos en Python 56
Abrir archivos 56
Lectura de archivos 56
Escritura sobre archivos 57
Capítulo 7. Bases de Datos 58
Bases de Datos (BD) en Python 58
Instalar módulo MySQL 58
Conexión a una base de datos MySQL 58
Instalar mysql.connector 59
Capítulo 8. Frameworks web para Python 61
Framework Web Python Flask 61
Instalación de Flask 61
Creación de un proyecto Flask 61
Ejecutar el proyecto 61
Crear páginas HTML 62
Crear la página HTML 62
Modificar archivo app.py 63
Framework Web Python FastAPI 63
Instalación de FastAPI 64
Instalación de uvicorn 64
Preguntas 64
Ejercicios 64
Capítulo 9. Introducción al análisis de datos con Python 65
NumPy 65
Características y funciones principales de NumPy 65
Instalación 65
SciPy 66
Usos de SciPy 66
Instalación 66
Pandas 68
Características de Pandas 68
Instalación 68
Fuentes y referencias adicionales 70

Introducción

Este documento es un complemento a las clases presenciales y virtuales, y está basado en la bibliografía del curso, así como de otras fuentes adicionales que se indican a lo largo del texto, además de la experiencia del autor en su función docente en las áreas de desarrollo, así como por su labor como desarrollador de software en distintas empresas.

Capítulo 1. Introducción al lenguaje Python

Python es considerado un lenguaje de ***script* interpretado**. Esto significa que el código es ejecutado directamente sin tener que crear archivos objeto como lo hacen los lenguajes compilados.

Python es además un lenguaje orientado a objetos y fuertemente tipado, a pesar de no declarar variables de forma explícita, pero éstas toman asumen el tipo de dato con el primer valor que se les asigne.

Figura 1.1. Logo de Python[1]

Descarga e instalación

En Windows basta con descargar el ejecutable y seguir los pasos del asistente. En Linux depende de la distribución, aunque en todos los casos es muy similar y en la red se dispone de documentación al respecto. Para equipos Mac con sistemas operativo MacOS (una versión especial de Unix) también está la documentación acerca de los paquetes y cómo proceder.

El sitio de descargas oficial es:
Download Python

Características de hardware y software

Los requerimientos de Python son mínimos; a nivel de hardware basta con tener un equipo ajustado a los estándares actuales y el lenguaje correrá sin inconvenientes. En cuanto al software, Python es multiplataforma, por lo que puede instalarse en cualquier sistema operativo (los más usados: Windows, Mac, Unix, Linux).

[1] Figura tomada de: Archivo:Python-logo-notext.svg - Wikipedia, la enciclopedia libre

Modo consola o intérprete

Una vez instalado el lenguaje Python, podemos acceder al modo consola o terminal para realizar tareas desde allí con éste.

El indicador del sistema (prompt) muestra tres signos mayor (**>>>**).

Ejemplo 1.1

Ingresar al modo de comandos de Python desde una terminal y ejecutar algunas instrucciones.

Ingresar al modo consola

```
>py
```

Realizar operaciones y usar instrucciones del lenguaje desde el modo consola de Python

```
>>>4 + 5
9

>>>print("Hola")
Hola
```

Para salir del modo consola: se tienen dos formas:

1. *Ctrl+Z*
2. `>>>quit()`

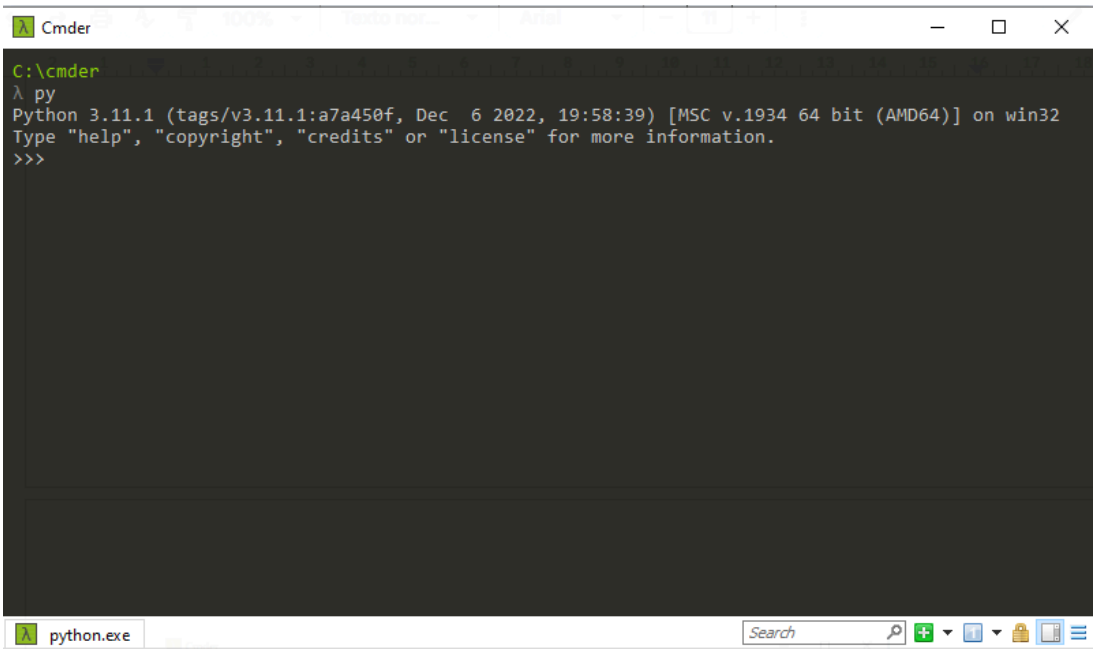

Figura 1.2. Modo *comandos* en Python desde una terminal ***cmder*** (*Commander*)

Crear *scripts*

Los archivos se crean con extensión **.py**. Siguiendo las convenciones del lenguaje, es buena práctica utilizar en el nombre del archivo la notación *snake_case* en minúscula.

Ejecutar *scripts*

Desde la terminal y ubicados en el directorio donde esté el archivo ingresamos el comando ***py*** seguido del nombre del archivo Python.

Ejemplo 1.2
Ejecutar un archivo llamado *script_python.py*.

```
>py script_python.py
```

Convenciones de nombres

Usar los caracteres permitidos para el nombre de identificadores según las reglas de programación.

A diferencia de otros lenguajes, en Python cuando el nombre de un identificador se compone de varias palabras, no se utiliza la notación *camelCase*, sino que cada palabra se separa con *guión bajo* (*underline*) (**_**)

La recomendación también sugiere escribir los nombres de identificadores en minúscula, a excepción de las clases que siguen la notación *UpperCamelCase*.

Notas (acerca de la guía de estilo del lenguaje)

- La *guía de estilo PEP 8*, recomienda una longitud de línea máxima de 72 caracteres.
- La indentación (sangría) debe ser de cuatro (4) espacios en blanco.
- Los nombres de clases siguen la notación *UpperCamelCase*.
- Las funciones se definen en minúscula en formato *snake_case*.
- Las constantes se definen en mayúscula formato *SNAKE_CASE*.
- Las variables se definen en minúscula en formato *snake_case*.
- Al igual que otros lenguajes, Python también es sensible a los caracteres.

Comentarios

Python, al igual que otros lenguajes, también maneja dos tipos de comentarios: para una línea y para varias líneas.

- Para una línea: se utiliza el carácter *numeral* (**#**)
- Para varias líneas: triple comilla (**"**) o apóstrofo (comilla simple: **'**) al abrir y cerrar

Ejemplo 1.3
Ilustración del uso de comentarios.

Para una sola línea

```
# Esto es un comentario
```

Para varias líneas

1. Usando doble comilla

```
"""
Este comentario aplica
para varias líneas
"""
```

2. Usando apóstrofos o comillas simples

```
'''
Este comentario también aplica
para varias líneas
'''
```

Docstrings (cadenas literales de documentación)

Son comentarios al inicio de las funciones que sirven para generar la documentación. Se utiliza el tipo de comentario de varias líneas con triple comilla doble o simple para la creación de los *Docstrings*.

Ejemplo 1.4
Uso de otros operadores de asignación y su significado.

```
def function_example():
    """
    Este es un comentario tipo docstrings para documentar la función
    y también es utilizado por los programas de documentación para
    realizar la respectiva generación de ésta
    """
    print("Esto es una función")
```

Dividir una sentencia en varias líneas

Dado que muchas instrucciones pueden ser muy extensas, la guía de estilos sugiere una longitud máxima de 72 caracteres. Para no salirse de los estándares, Python permite partir o dividir una instrucción en varias líneas; para ello se utiliza el carácter ***backslash*** (\).

Ejemplo 1.5
Dividir una instrucción en varias líneas.

```
a = 2 + 3 + 5 + \
```

```
7 + 9 + 4 + \
6
```

Salida estándar

Para imprimir por pantalla se utiliza la instrucción ***print*()**. Más adelante veremos varias formas de uso de esta función.

Ejemplo 1.6
Ilustrar la salida estándar: imprimir un mensaje por pantalla.

```
>>>print("Hola")
```

Constantes y variables

Python no implementa el concepto de *constante* como en otros lenguajes, solo usa *variables*, sin embargo, usando las recomendaciones de la guía de estilos, podemos definir constantes usando variables en la notación *SNAKE_CASE*. Como se mencionó antes, las variables se definen usando la convención *snake_case*.

Notas

1. La constante del lenguaje ***None*** (valor del tipo ***NoneType***) permite determinar la ausencia de valor en una variable
2. Las constantes booleanas son: ***True*** y ***False***

Tipos de datos

Python es un lenguaje fuertemente tipado que no requiere declarar las variables y que cuenta con los siguientes tipos de datos (el tipo de la variable es asignado de manera implícita con el primer valor que tome ésta).

Numéricos

Python cuenta con los siguientes tipos para el manejo de números:

- Enteros: ***int***
- Reales (en punto flotante): ***float***
- Complejos: ***complex***

Cadenas

- ***str***

Booleanos

- ***bool***

Secuencias

- Tuplas: ***tuple***
- Conjuntos: ***set***
- Listas: ***list***
- Diccionarios: ***dict***

Función type()

Permite determinar el tipo de dato de una variable

Sintaxis

```
type(variable)
```

Entrada estándar

En el modo consola existe una forma para capturar datos desde el teclado; para ello se utiliza la función ***input*()**.

Ejemplo 1.7
Ingresar el nombre de una persona y mostrarlo por pantalla.

```
>>>print("Ingrese su nombre: ")
>>>name = input()
>>>print(f"Su nombre es: {name}")
```

Conversión de tipos o casteo de variables

La operación de conversión de tipos, también conocida como **casteo** de variables está presente en Python. Para convertir tipos de datos a otros o *castear variables*, se utilizan las siguientes funciones.

Cadenas

Función: **str()**

Enteros

Función: **int()**

Reales (punto flotante)

Función: **float()**

Listas

Función: **list()**

Conjuntos

Función: **set()**

Binarios

Función: **bin()**

Octales

Función: **oct()**

Hexadecimales

Función: **hex()**

Ejemplo 1.8
Ingresar dos números por teclado y sumarlos.

Solución
Para trabajar con números ingresados por teclado, es necesario realizar la conversión de tipos, ya que por defecto todo lo que viene del teclado se toma por defecto como cadena de caracteres. En el desarrollo del ejemplo también se incluye la *concatenación de cadenas* que se explica más adelante y que tiene como fin unir cadenas de caracteres. Dado que los números no son cadenas, también se hace necesario castearlos para mostrarlos en la salida.

Programa:

```
print("Ingrese x:", end = " ")
x = int(input())
print("Ingrese y:", end = " ")
y = int(input())
z = x + y
y = print(str(x) + str(y) + " = " + str(z))
```

Nota

La instrucción **end = " "** que aparece como segundo parámetro (argumento) de la función *print* hace que el cursor no salte en la terminal (similar a la instrucción *Sin Saltar* de *PSeInt*).

Operadores

Un *operador* es un símbolo usado en matemáticas para representar una operación a realizar, la cual puede ser unaria (con un *operando*) o binaria (con dos *operandos*). En la aritmética y en el álgebra se cuenta, entre otros, con varios operadores elementales; cada operador tiene una *prioridad* asignada, lo cual significa que los de mayor prioridad, se ejecutarán primero. Se dividen en tres grupos, de los cuales se muestra su representación en el lenguaje.

Operadores aritméticos

Utilizados para realizar las operaciones aritméticas básicas y otros cálculos (operaciones) matemáticos; tenemos los siguientes:

Nombre Operador	Símbolo en Python	Prioridad
Negación aritmética unaria	$-$	Alta
Potencia	**	Alta - media
Multiplicación	*	Media
División	/	
División entera	//	
Módulo	%	
Suma	+	Baja
Resta	$-$	

Notas

- La negación aritmética es una operación *unaria* que consiste en negar el símbolo del número (operando). Ejemplo: $-(+2)$, $-(8)$, $-(-5)$, -94
- La prioridad se refiere al orden en que los operadores se efectúan en una expresión aritmética: los de mayor prioridad se efectúan primero.
- Las operaciones encerradas entre paréntesis se efectúan primero, por lo que tienen mayor prioridad. Los paréntesis modifican la prioridad de los operadores en una expresión.
- Si hay dos operadores de igual prioridad, se ejecuta primero el que se encuentre más a la izquierda, esto es, se sigue el orden de izquierda a derecha.
- Para calcular cualquier raíz, se puede usar el operador de potencia aprovechando las propiedades del álgebra para los exponentes fraccionarios: $\sqrt[n]{a^m} = a^{\frac{m}{n}}$

Operación de módulo y división entera

El *módulo* es una división entera que devuelve el residuo de ésta. Se representa con el símbolo **%**; la división entera se encarga de devolver solo la parte entera de dividir dos números enteros.

Ejemplo 1.9

```
a. 7 % 5 = 2; 7 // 5 = 1
b. 17 % 2 = 1; 17 // 2 = 8
c. 48 % 4 = 0; 48 // 4 = 12
d. 57 % 6 = 3; 57 // 6 = 9
e. 49 % 5 = 4; 49 // 5 = 9
f. 9 % 20 = 9; 9 // 20 = 0
g. 55 % 11 = 0; 55 // 11 = 5
```

Operadores relacionales o de comparación

Son operadores binarios utilizados para comparar expresiones. El resultado de una comparación entre dos expresiones es un valor lógico (booleano), devolviendo o un verdadero (*true*) o un falso (*false*); estos son:

Nombre Operador	Símbolo en Python	Prioridad
Igual	$==$	Alta
Diferente	$!=$	
Mayor que	$>$	Media
Menor que	$<$	
Mayor o igual que	$>=$	Baja
Menor o igual que	$<=$	

Ejemplo 1.10
Resultados devueltos al realizar operaciones con los operadores relacionales.

a. $8 != 9 \rightarrow (True)$
b. $9 >= 9 \rightarrow (True)$
c. $7 != 14 / 2 \rightarrow (False)$
d. $9 * 2 <= 50 / 10 \rightarrow (False)$
e. $-8 = 8 \rightarrow (False)$

Operadores lógicos (booleanos)

Permiten conectar (unir) expresiones de comparación y realizar operaciones lógicas. El valor devuelto (verdadero o falso) depende del conectivo lógico utilizado, según las leyes del álgebra proposicional y booleana; estos son los utilizados en el lenguaje de programación Python:

Nombre Operador	Símbolo en Python	Prioridad
Negación lógica unaria	not	Alta
Conjunción	and	Media
Disyunción	or	Baja

Ejemplo 1.11
Resultados devueltos al realizar operaciones con los operadores booleanos.

a. $True\ and\ True\ \rightarrow (True)$
b. $False\ and\ True \rightarrow\ (False)$
c. $not\ True\ \rightarrow\ (False)$
d. $True\ or\ False \rightarrow\ (True)$
e. $False\ or\ False \rightarrow\ (False)$

Operador de asignación

Para asignar un valor (expresión) a una variable, utilizamos el operador *igual* (**=**).

Ejemplo 1.12
Ilustrar la asignación de valores a variables de diferente tipo y realizar operaciones aritméticas y lógicas.

```
age = 10
profession = "Ingeniero"
sw = True
bool = sw or 4 > 2
percentage = 1000 * 15.4 / 100
print("El tipo de dato de la variable age es: ", type(age))
print("15.4% de 1000 es: ", type(percentage))
print("Resultado operación lógica: ", type(bool))
```

Otros operadores aritméticos de asignación

Existen operadores para acumular sumas, diferencias, etc., similar a otros lenguajes; éstos permiten simplificar la escritura de algunas expresiones aritméticas. Estos son los usados en Python:

Nombre operador	Operador	Ejemplo
Más igual	**+=**	`x+=2 equivale a x = x + 2`
Menos igual	**-=**	`x-=2 equivale a x = x - 2`
Por igual	***=**	`x*=2 equivale a x = x * 2`
Dividido igual	**/=**	`x/=2 equivale a x = x / 2`
Módulo igual	**%=**	`x%=2 equivale a x = x % 2`
División entera igual	**//=**	`x//=2 equivale a x = x // 2`
Potencia igual	****=**	`x**=2 equivale a x = x ** 2`

Ejemplo 1.13
Uso de otros operadores de asignación y su significado.

```
a = 10
b = 3
c = a + b
c+=a #Equivale a escribir: c = c + a
b-=c #Equivale a escribir: b = b - c
```

Operador de concatenación de cadenas

Una **cadena de caracteres** es una secuencia compuesta por caracteres del código **ASCII** (*American Standard Code for Information Interchange*) y que están disponibles en todas las distribuciones comerciales en los distintos idiomas en que están los teclados. La **concatenación de cadenas** es una operación que consiste en unirlas; esta operación también se conoce como *suma de cadenas*. Python utiliza el operador *más* (**+**) para realizar esta operación, usado también en otros lenguajes de programación.

Ejemplo 1.14
Concatenar las variables nombre y apellido.

```
first_name = "Pedro"
last_name= "Gil"
full_name= first_name + last_name
print("Su nombre completo es: ", full_name)
```

Operadores de pertenencia

Se aplican a secuencias para determinar si un elemento se encuentra en ella. Devuelven un valor booleano en caso de que el elemento se encuentre o no en la secuencia. Estos operadores se tratarán más adelante.

- *in*
- *not in*

Operadores de identidad

Se aplican para saber si dos variables corresponden al mismo objeto. Estos operadores se tratarán más adelante.

- *is*
- *is not*

Plantillas literales

Similar a otros lenguajes, es posible incluir variables en los literales de cadenas de caracteres. Una **plantilla literal** en Python es una cadena de texto que permite la

interpolación de variables y expresiones. A partir de la versión 3.6 debe hacerse con *formatted string literals ("f-strings")*.

Nota
Python permite en su sintaxis varias formas para combinar cadenas literales con variables.

Ejemplo 1.15
Veamos diferentes formas de uso de la función **print** para mostrar literales de cadena junto con variables.
Programa:

```
print("Ingrese su nombre: ", end = " ")
name = input()
print("Ingrese su edad: ", end = " ")
age = int(input())
print("Ingrese su salario: ", end = " ")
salary = float(input())

# Forma clásica (usada también en algoritmia y otros lenguajes)
print("Nombre: ", name, "\nEdad", age, "\nSalario", salary)

# Usando f-strings (nueva implementación en Python)
print(f"Nombre: {name}\nEdad: {age}\nSalario: {salary}")

# Utilizando formatos (similar a lenguajes como C y PHP)
print("Nombre: %s \nEdad: %d \nSalario: %.2f" % (name, age, salary))
```

Establecer color al texto en entorno de terminal

Una forma fácil de establecer color al texto en los entornos de terminal, es usando los códigos estándar y asignarlos a constantes, como se muestra a continuación.

Ejemplo 1.16
Crear una lista de colores con nombres a partir de sus códigos estándar para entorno de terminal. Mostrar un mensaje con un color diferente al establecido por defecto.

Programa:

```
BLACK = '\033[30m'
RED = '\033[31m'
GREEN = '\033[32m'
YELLOW = '\033[33m' # orange on some systems
BLUE = '\033[34m'
MAGENTA = '\033[35m'
CYAN = '\033[36m'
LIGHT_GRAY = '\033[37m'
DARK_GRAY = '\033[90m'
```

```
BRIGHT_RED = '\033[91m'
BRIGHT_GREEN = '\033[92m'
BRIGHT_YELLOW = '\033[93m'
BRIGHT_BLUE = '\033[94m'
BRIGHT_MAGENTA = '\033[95m'
BRIGHT_CYAN = '\033[96m'
WHITE = '\033[97m'

print(f"{BRIGHT_CYAN}Menú de opciones{WHITE}")
```

Preguntas

1. ¿Qué tipo de lenguaje es Python? ¿Procedimental u orientado a objetos? ¿Débil o fuertemente tipado? ¿Compilado o interpretado?
2. ¿Qué es un lenguaje de script? ¿Por qué Python entra en esta categoría?
3. ¿Cuál encuentra más interesante: Python o PSeInt? ¿Cuál es más “fácil”, por qué?
4. ¿Se puede escribir un programa en la terminal de Python?
5. ¿Cómo se finaliza la consola de Python?
6. ¿Qué es una guía de estilos y cuál es la qué se sigue para Python?
7. ¿Cuál es la sugerencia para la longitud de línea en Python?
8. ¿Cuál es la sugerencia para la notación de las variables en Python?
9. ¿Python declara variables y cómo se definen los tipos de datos de ellas?
10. ¿En qué consiste el *casteo* de variables?
11. Si en Python hago lo siguiente: `a = 3` ¿Qué significado tiene realizar luego la operación `a+=5` y luego `a-=4`?
12. ¿Cómo concateno una cadena con un número entero o real en Python?
13. ¿Cuáles son las constantes booleanas en Python?
14. ¿Cómo es la salida y entrada estándar en Python?
15. ¿Qué es un literal de cadena? ¿Qué es una plantilla literal?

Ejercicios

Capítulo 2. Estructuras de control

En los lenguajes de programación, las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con ellas se puede:

- De acuerdo a una condición (comparación), ejecutar un grupo u otro de sentencias
- Ejecutar un grupo de sentencias un número determinado de veces
- Interrumpir la ejecución normal del programa

Todas las estructuras de control tienen un único punto de entrada y un único punto de salida. Éstas se pueden clasificar en: decisión, iteración y de control avanzadas.

Condicionales

Python permite el uso de condicionales y selectores múltiples, así como la variante **elif** para simplificar las líneas de código en la toma de decisiones. Comparado desde el inicio de Python (1991), el selector múltiple (sentencia **match**) es relativamente nuevo en el lenguaje, por lo que antes debía solucionarse cualquier decisión mediante condicionales *if else elif*; afortunadamente ya existe la implementación respectiva que permite simplificar el uso de éstos.

Condicional if else elif

Evalúa una determinada condición o expresión de comparación, en caso de ser verdadera, se ejecuta un bloque de instrucciones. Si dicha condición no se cumple, esto es, es falsa, entonces ninguna de las instrucciones es ejecutada a no ser que se especifique la cláusula *else*.

Sintaxis

```
if condición:
    instrucciones si la condición es verdadera
[else:
    instrucciones si la condición es falsa]
```

Ejemplo 2.1
Leer el nombre y edad de una persona y determinar si es mayor de edad.

Programa:

```
print("Nombre:", end = " ")
name = input()
print("Edad:", end = " ")
age = int(input())
```

```
print(f'Su nombre es: {name}')
print(f"Su edad es: {age}")
if age > 18:
    message = "Mayor de edad"
else:
    message = "Menor de edad"
print(f"Usted es {message}")
```

Nota

Similar a otros lenguajes con la sentencia *elseif*, en Python está su análoga ***elif***. Esta sentencia es útil para simplificar escritura de código en el programa.

Sintaxis

```
if condición_1:
    instrucciones si la condición_1 es verdadera
[elif condición2:
    instrucciones si la condición_2 es verdadera
elif condición3:
    instrucciones si la condición_3 es verdadera
...
elif condición_N:
    instrucciones si la condición_N es verdadera
else:
    instrucciones si todas las condiciones anteriores son falsas]
```

Ejemplo 2.2

Leer un número por teclado y determinar si es positivo, negativo o cero. Este programa ilustra el uso de la sentencia ***elif***.

Programa:

```
print("Ingrese un número: ", end = " ")
number = float(input())
if number > 0:
    message = "Número positivo"
elif number < 0:
    message = "Número negativo"
else:
    message = "Número igual a 0"
print(f"Dato ingresado {number}: {message}")
```

Selector múltiple o estructura caso: sentencia match

Permite la ejecución de un bloque de instrucciones en función del valor que tome una expresión. Es utilizada en lógica de programación como alternativa al condicional cuando se tienen más de dos salidas lógicas. También se conoce como la estructura "*Según*". Esta estructura de control recibe diferentes nombres en distintos lenguajes de programación, el más conocido quizá, es el *switch*, de lenguajes como C/C++, PHP, Java, C# y Javascript, entre otros.

Esta estructura es muy útil en la creación de menús y casos donde una variable tome más de dos valores. Veamos cómo se puede implementar.

Sintaxis

```
match variable:
    case valor_1:
        instrucciones si la variable coincide con valor_1
    case valor_2:
        instrucciones si la variable coincide con valor_2
…
    case valor_N:
        instrucciones si la variable coincide con valor_N
    [case _:
        instrucciones si la variable no coincide con ningún valor
```

Observe como el caso "**_**" es la equivalente en lógica de programación al caso "*En Otro Caso*" o la opción *default* de lenguajes como los mencionados arriba, utilizada opcionalmente para indicar algo en caso de que no se cumple ninguno de los otros casos.

Ejemplo 2.3

Ingresar un número y determinar: raíz cuadrada, si es par o impar y su valor absoluto. Resolver utilizando un menú de opciones.

En este ejemplo se hará uso de la **librería math** para usar algunas de sus funciones matemáticas. También se muestra una forma de asignar color al texto desplegado en un entorno de terminal con códigos estándar

Programa:

```
import math

num = 0
opt = ""
while opt != "0":
    print("\033[96mMenú de opciones\033[97m")
    print("0. Salir")
    print("1. Ingresar número")
    print("2. Raíz cuadrada")
    print("3. Par o impar")
```

```
    print("4. Valor absoluto")
    opt = input("Ingrese su opción: ")

    match opt:
        case "0":
            print("¡Hasta pronto!")
        case "1":
            num = float(input("Ingrese un número: "))
        case "2":
            if num >= 0:
                square_root = math.sqrt(num)
                print(f"Raíz cuadrada de {num}: {square_root}")
            else:
                print("Raíz imaginaria")
        case "3":
            if num % 2 == 0:
                print(f"{num} es par")
            else:
                print(f"{num} es impar")
        case "4":
            print(f"Valor absoluto de {num}: {abs(num)}")
        case _:
            print("Opción no válida")
```

Ciclos

Los ciclos o bucles son estructuras de control que repiten un grupo de instrucciones mientras se cumpla una condición. Python solo implementa los ciclos *while* (*mientras*) y *for* (*para*). El bucle *for* es usado además para iterar listas, diccionarios y otros elementos del lenguaje.

Ciclo while

El ciclo while en Python se utiliza de forma muy similar a otros lenguajes, pero con su estilo particular para las estructuras de control.

Sintaxis

```
while condición:
    instrucciones del ciclo
```

Ejemplo 2.4

Mostrar los términos de la serie de Fibonacci menores o iguales a `n`. La serie de Fibonacci está formada por los términos: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...[2]

Programa:

```
print("Ingrese n: ", end = " ")
n = int(input())
a = 0
b = 1
c = a + b
print(a)
print(b)
while c <= n:
    print(c)
    a = b
    b = c
    c = a + b
```

Notas

- Observe la sangría (indentación) que se debe mantener para que el lenguaje sepa que está dentro del ciclo. Una vez comience a escribir en la misma columna de la palabra *while*, dichas instrucciones estarán por fuera del ciclo.
- En Python existe una variante del ciclo *while* conocida como ***while else***; el bloque de código del ***else*** se ejecuta siempre y cuando la condición del ciclo se evalúe a false y no se haya ejecutado una sentencia *break* en el ciclo. Esta alternativa es similar a usar suiches o banderas.

Sintaxis

```
while condición:
    instrucciones del ciclo
[else:
    instrucciones del ciclo si condición es False y no se ejecuta break]
```

Ejemplo 2.5

Se ingresan varios números por teclado; indicar si se ingresa el número 2 y finalizar el ciclo. El programa también finaliza si se ingresa un 0.

Programa: solución usando *banderas* o *suiches*

```
num = -1
sw = False #Supuesto: el número 2 nunca será ingresado
while num != 0:
    print('Ingrese un número')
```

[2] Leonardo de Pisa, más conocido como Fibonacci, fue un matemático italiano (1173 - 1241) que dio a conocer en Occidente la serie numérica que lleva su nombre observando la forma como se reproducen los conejos, aunque se sabe que esta sucesión ya estaba descrita en la matemática de la India desde mucho antes. Puede leer más al respecto en Leonardo de Pisa - Wikipedia, la enciclopedia libre y Sucesión de Fibonacci - Wikipedia, la enciclopedia libre

```
    num = int(input())
    if num == 2:
        sw = True
if sw:
    print(f'Elemento 2 encontrado en la lista')
else:
    print('El elemento 2 no se encuentra en la lista')
```

Programa: solución usando *while else*

```
num = -1
while num != 0:
    print('Ingrese un número')
    num = int(input())
    if num == 2:
        print(f'Elemento 2 encontrado en la lista')
        break
else:
    print('El elemento 2 no se encuentra en la lista')
```

Ciclo for

El ciclo ***for*** varía un poco en Python respecto a otros lenguajes; es particularmente usado como el *foreach* de otros lenguajes, en este caso para objetos iterables como listas, tuplas, conjuntos y diccionarios que se verán más adelante. Con la clase ***range()*** podemos implementar un ciclo **for** basado en una secuencia numérica.

Sintaxis

```
for variable in iterable:
    instrucciones del ciclo
```

Ejemplo 2.6
Imprimir los números del 0 al 10.

Programa:

```
for i in range(11):
    print(i)
```

Aquí `i` comenzará en `0` y el ciclo hará su último ingreso en el valor `10`.

La función **range** permite especificar varios parámetros, para indicar los límites inicial y final de la secuencia, así como el incremento.

En general, se puede invocar a range de la siguiente manera

- **`range(max)`**: iterable que comienza en `0` y termina en `max - 1`
- **`range(min, max)`**: iterable que comienza en `min` y termina en `max - 1`

- **`range(min, max, step)`**: iterable que comienza en `min`, termina en *`max - 1`* y avanza cada *step* pasos, es decir, nos permite especificar el *incremento*

Ejemplo 2.7
Mostrar los números impares del 1 al 100.

Programa:
```
print("Números pares de 0 a 100")
for i in range(1, 101, 2):
    print(i)
```

Nota
Python también dispone de un ciclo ***for else***. De forma análoga a como sucede con el ciclo while, esta forma simplifica el uso de suiches o banderas. La sentencia **else** se ejecuta si dentro del **for** nunca se ejecuta una sentencia *break*.

Sintaxis

```
for variable in iterable:
    instrucciones del ciclo
else:
    Instrucciones sino se ejecutó la sentencia break en el bloque for
```

Ejemplo 2.8
Leer el nombre de una lista de `n` estudiantes. Determinar si hay una alumna de nombre "Luz Patricia".

Programa: solución usando *banderas* o *suiches*
```
sw = False //#Supuesto: no se encuentra la persona
print("Total estudiantes:", end = " ")
n = int(input())
for i in range(1, n + 1, 1):
    print("Ingrese nombre:", end = " ")
    name = input()
    if name == "Luz Patricia":
        sw = True
if sw:
    print("Se encontró una alumna de nombre 'Luz Patricia'")
else:
    print("No se encontró una alumna de nombre 'Luz Patricia'")
```

Programa: solución con *for else*
```
print("Total estudiantes:", end = " ")
n = int(input())
for i in range(1, n + 1, 1):
    print("Ingrese nombre:", end = " ")
    name = input()
```

```
    if name == "Luz Patricia":
        print("Se encontró una alumna de nombre 'Luz Patricia'")
        break
else:
      print("No se encontró una alumna de nombre 'Luz Patricia'")
```

Sentencias de bifurcación de control

La Bifurcación de control está conformada por un grupo de instrucciones que permite hacer saltos sobre ciertas partes del código de un programa, interrumpiendo su flujo normal. Al igual que en otros lenguajes, Python cuenta con las sentencias para interrumpir el flujo de un programa, entre las que se tiene: ***break***, ***continue, exit()*** y ***return***.

break

Permite romper o finalizar la estructura de control donde se encuentre pasando a la instrucción que le sigue. En particular, si la instrucción se encuentra en un ciclo, lo finaliza sin importar la condición de control y continúa con la instrucción siguiente al ciclo.

Ejemplo 2.9

Leer el nombre de un grupo de personas y contar cuantas se registraron; el último nombre en ingresar es un registro centinela con nombre igual a “*”.

El ciclo contendrá una condición que en teoría genera un bucle infinito, pero dentro de éste se encontrará una lectura de datos que llevará en algún momento a un registro centinela permitiendo ejecutar la sentencia break que da por terminado el ciclo.

Programa:

```
c = 0
while True:
    print("Ingrese un el nombre (* para terminar):", end = " ")
    name = input()
    if name == "*":
        break
    c+=1
print(f"Total personas: {c}")
```

continue

Utilizada en los ciclos, hace que salte el resto de instrucciones de éste desde donde se encuentra la instrucción, pasando a la siguiente iteración.

Ejemplo 2.10

Imprimir cada carácter de un texto ingresado por teclado, excepto si éste es una “a”.

Las cadenas son objetos iterables, por lo que podemos usar un ciclo *for* para mostrar cada carácter de éstas.

Programa:

```
print("Ingrese un texto:", end = " ")
string = input()
for letter in string:
    if letter == "a":
        continue
    print(letter)
```

Nota

La diferencia de *continue* con *break*, es que esta última sentencia rompe el ciclo y sale de éste ignorando las instrucciones que estuvieran pendientes de ejecutar; mientras que la primera también ignora las instrucciones que le siguen, pero dirige el flujo del programa a una nueva iteración del ciclo.

exit()

Python implementa esta sentencia de bifurcación de control como una función. **exit()** interrumpe el *script* que se esté ejecutando, cancelando totalmente el programa e ignorando las instrucciones que se encuentren después de ella.

Ejemplo 2.11

Leer el código de un grupo de empleados que inician labores y contar cuantas llegaron a trabajar; una vez ingresen todos, se ingresa el registro centinela con código igual a “*”.

El ciclo contendrá una condición que en teoría genera un bucle infinito, pero dentro de éste se encontrará una lectura de datos que llevará en algún momento a un registro centinela permitiendo ejecutar la función **exit()** que da por terminado el programa ignorando el código que continúe luego de ella.

Programa:

```
c = 0
while True:
    print("Ingrese código (* para terminar):", end = " ")
    code = input()
    if code == "*":
        exit()
    c+=1
# Observe que la siguiente línea no se ejecuta, a diferencia de break
# que continúa el flujo del programa luego del ciclo
print(f"Total empleados: {c}")
```

return

Devuelve el control del programa a una *función* o porción de programa que *llama*; a su vez puede devolver un valor a dicha función. Esta instrucción al igual que las anteriores, interrumpe el flujo normal de ejecución, devolviendo el control a otra parte del programa e ignorando las instrucciones que pudieran continuar. En la siguiente subsección se ilustra el uso de esta sentencia muy utilizada en funciones.

pass

Python implementa esta sentencia para evitar errores de código vacío. Puede ser útil al definir un encabezado de función u otros bloques de construcción de código donde se piensa en "**código a futuro**". No es propiamente una sentencia de bifurcación de control, sino que la usa el lenguaje para evitar errores.

Capítulo 3. Funciones

Funciones incorporadas o predefinidas

Python al igual que otros lenguajes de programación, dispone de una vasta serie de funciones y métodos para realizar tareas diversas y que tienen como objetivo, además de solucionar ciertos problemas, facilitar al programador el tema de construir alguna funcionalidad. Es así como se dispone de funciones matemáticas, para el tratamiento de cadenas de caracteres, para el manejo de fechas entre muchos otros muchos casos.

Métodos y funciones matemáticas

Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la resta o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando funciones incorporadas. Dichas funciones están optimizadas y de no contar con ellas, se haría necesario construirlas para obtener los cálculos deseados, como el coseno trigonométrico, una raíz cuadrada, etc. Las más comunes se muestran a continuación; Python las agrupa en distintos módulos y algunas están como funciones “libres”.

Funciones matemáticas. La clase *math*

La mayor parte de las funcionalidades matemáticas se encuentran agrupadas en **métodos** de la clase ***math***, sin embargo, algunas están como **funciones** libres, por lo cual se usan sin llamar a esta clase. Aunque la clase es nativa de Python, es necesario importarla para trabajar con ella.

Ejemplo 2.12
Aplicar algunas funciones y métodos de la clase math de Python.

Programa:

```
import math

print("Métodos y funciones matemáticas")
print("Ingrese número 1: ")
x = float(input())
print("Ingrese número 2: ")
y = float(input())
z = math.pow(x, y)
print(f"Potencia {x} ^ {y}: {z}")
if x >= 0:
    z = math.sqrt(x)
    print(f"Raíz cuadrada {x}: {z}")
else:
```

```
    print("No se puede calcular la raíz cuadrada")

z = math.trunc(x)
print(f"Redondeo trunc (0 decimales): {z}")

z = round(x, 1)  //round no está en la clase math, es una función
print(f"Redondeo round (1 decimal): {z}")

z = math.floor(x)
print(f"Redondeo por debajo: {z}")

z = math.ceil(x)
print(f"Redondeo por encima: {z}")

z = math.sin(y)
print(f"Radianes: sen({y}): {z}")
print(f"Grados: sen({y}): {math.degrees(z)}")

z = math.cos(y)
print(f"Radianes: cos({y}): {z}")
print(f"Grados: cos({y}): {math.degrees(z)}")
```

Números pseudo aleatorios. La clase *random*

Python por su lado incluye una librería llamada **random** que dispone de una serie de métodos para trabajar con números pseudo aleatorios[3]. Por ejemplo, para generar un aleatorio (entero) entre 1 y 10, indicamos el método randint de la clase random especificando el intervalo respectivo por medio de dos parámetros: `random.randint(1, 10)`.

Ejemplo 2.13

Otros métodos interesantes de esta clase son los que operan sobre secuencias: **`choice`**, el cual elige aleatoriamente un elemento de una secuencia; **`shuffle`**, que devuelve una lista con los elementos distribuidos aleatoriamente:

```
data_list = [0, 5, 4, 8]
print(f"Lista: ", data_list)
print(f"choice: ", random.choice(data_list))
random.shuffle(data_list)
print(f"Lista shuffle: ", data_list)
```

[3] Puede consultar más acerca de esta librería en: Generate pseudo-random numbers — Python 3.13.1 documentation y en Python Random Module

Números complejos. La clase *complex*

Python permite de una forma sencilla el manejo de números complejos. Recordemos que un número complejo es un número de la forma `z = a + bi`, donde `a, b` son números enteros e `i` es la unidad imaginaria: $i = (-1)^{0.5}$. Python reemplaza `i` por `j` y debe ponerse siempre que se quiera expresar un número complejo acompañada de un coeficiente, así sea la unidad imaginaria, por lo cual hay que escribirla como `1j`. Un número complejo queda definido dentro de la clase **complex**, la cual no tiene que importarse para trabajar con estos elementos.

Los números complejos admiten los operadores conocidos, por los que pueden sumarse, restarse, multiplicarse, dividirse, etc. El conjunto de los números complejos contiene al conjunto de los números reales: $R \subset C$

Es posible extraer la parte real e imaginaria con las propiedades `real` e `imag`, respectivamente, o hallar el conjugado con el método `conjugate`.

Veamos algunas operaciones con números complejos.

Ejemplo 2.14
Números complejos en Python.

Programa:

```
#Crear un complejo con la función complex
c = complex(-9, 3)
#Crear un complejo con la notación z = a + bi
z = 3 + 2j
print(c)
print(z)
z = c + (-5 - 1j)
print(z)
print(c.real) #Obtener la parte real
print(c.imag) #Obtener la parte imaginaria
print(z.conjugate()) #Hallar el conjugado de z = a + bi: z = a - bi
```

Nota: la clase *cmath*
La clase **cmath** proporciona una serie de funcionalidades para el trabajo con números complejos que puede emplearse en caso de que la clase **complex** no disponga de todas las herramientas en un problema dado.

Métodos para la manipulación de cadenas de caracteres. La clase *str*

Una **cadena de caracteres** es una secuencia compuesta por caracteres del código **ASCII** (*American Standard Code for Information Interchange*) y que están disponibles en todas las distribuciones comerciales en los distintos idiomas en que están los teclados.

Python define las cadenas de caracteres en la clase ***str*** y dispone de una serie de métodos y funciones para su manipulación que se encuentran agrupados en esta clase.

Las cadenas deben estar encerradas entre comillas dobles o simples (apóstrofos), y pueden alternarse para mostrar por pantalla comillas cuando así se requiera. También se pueden escapar caracteres para casos más complejos en el manejo de comillas: `"\"cadena escapada\""`.

Cada carácter de una cadena tiene una posición asociada dentro de ella; en Python las cadenas también son consideradas secuencias y pueden ser manipuladas como tal (similar a los arreglos), donde el primer carácter ocupa la posición cero, el segundo la uno y así sucesivamente.

Ejemplo 2.15
Recorrer cadenas en Python.

Programa: (desde terminal)

```
>>> nom = "Pedro"
>>> for p in nom:
...     print(p)
...
P
e
d
r
o
>>> for i in range(len(nom)):
...     print(nom[i])
...
P
e
d
r
o
>>>
```

Métodos para la manipulación de fechas y horas. La clase *datetime*

Python también dispone de una serie de métodos en la clase **datetime** para el tratamiento y manipulación de fechas y horas.

Ejemplo 2.16
Manejo de fechas y horas en Python.

Programa: (desde terminal)

```
>>> from datetime import date
>>> date.today()
```

```
datetime.date(2023, 11, 5)
>>> f=date.today()
>>> f
datetime.date(2023, 11, 5)
>>> f.year
2023
>>> f.day
5
>>> f.month
11
>>> from datetime import datetime
>>> datetime.now()
datetime.datetime(2023, 11, 5, 15, 50, 2, 74204)
```

Funciones definidas por el programador

Al desarrollar aplicaciones nos vemos en la necesidad de utilizar la misma porción de código en varias ocasiones; para optimizar la programación y evitar tener que repetir segmentos de código, los lenguajes implementan los subprogramas, secuencias de código a las que podemos invocar pasándole de `0` a `N` parámetros y que tras ejecutar una serie de operaciones, nos pueden devolver `0, 1` o más valores. Normalmente, a nivel teórico, cuando devuelven un valor se les llama *funciones*, y en los demás casos, *procedimientos*. Python trata todos los subprogramas como funciones y todas las variables son pasadas por *valor*, excepto las listas y otros tipos de colecciones que son pasados por *referencia*. Por consiguiente, no podremos modificar el valor de una variable primitiva de manera "global" desde una función en Python, aunque sí podremos utilizar una función que no devuelva algún valor. Esto se debe a que Python es un lenguaje orientado a objetos, por los que el concepto de "variable global" pierde sentido, siendo todo tratado como un objeto. Más adelante al tratar el tema de *POO* se hablará más al respecto.

Se definen los subprogramas con la palabra reservada ***def***, el nombre de la función y entre paréntesis los parámetros que contendrá. En el caso de que devuelva la función algún valor, se utiliza la sentencia return seguida de dicho valor.

Sintaxis

```
def nombre_función([parametros]) [-> tipo_dato]:
     cuerpo de la función
    [return valor_retorno]
```

Ejemplo 2.17
Dados dos números, cree una función que los sume y devuelva el resultado.

Programa:

```
# Función sum con los parámetro x e y
def sum(x, y) -> int:
    return x + y
```

```
#Programa principal que llama a la función sum
print("Ingrese número 1:", end =" ")
n1 = int(input())
print("Ingrese número 2:", end =" ")
n2 = int(input())
print(f"Suma {n1} + {n2} = {sum(n1, n2)}", end =" ")
```

Problemas resueltos

Los siguientes ejercicios de programación resueltos incluyen los temas del manejo de variables, operadores, asignaciones, entrada y salida de información, estructuras de control (ciclos, condicionales, bifurcación de control) y funciones.

Problema 2.1

Una empresa con cinco (3) sucursales desea conocer cuántos empleados participarán de una rifa. La actividad es informal, por lo que las personas que están recolectando la información no tienen claridad del número de empleados por sucursal ni de toda la empresa. Ellos, aprovecharán esta encuesta para saber el total de empleados de la empresa y por sucursal.

Programa: cycle_nested.py

```
cpe = 0 #Contador personas empresa
cp_yes = 0 #Contador personas participan
cp_no = 0 #Contador personas no participan
for i in range(1, 4, 1):
    cps = 0 #Contador personas sucursal
    print("--- Sucursal: ", i, " ---")
    print("Ingrese su nombre (* para terminar):", end = " ")
    name = input()
    while name != '*':
        cps = cps + 1
        print("¿Participa en la rifa? (s/n):", end = " ")
        response = input()
        if response == 's':
            cp_yes = cp_yes + 1
        else:
            cp_no = cp_no + 1
        print("Ingrese su nombre (* para terminar)", end = " ")
        name = input()
    print(f"Total empleados sucursal {i}: {cps}")
    cpe = cpe + cps
print(f"Total empleados empresa: {cpe}")
print(f"Total empleados que participan: {cp_yes}")
print(f"Total empleados que no participan: {cp_no}")
```

Problema 2.2

Versión de la calculadora en Python empleando un menú de opciones y haciendo el uso de funciones..

Programa: calculator_functions.py

```
from os import system

# Ejemplo return y funciones

# Función sum con los parámetro x e y
def sum(x, y):
    return x + y

def substract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y

def show(message):
    print(message)

# Programa principal que llama a la función sum
option = ""
n1 = 0
n2 = 0
while option != "8":
    print("1. Ingresar números")
    print("2. Sumar")
    print("3. Restar")
    print("4. Multiplicar")
    print("5. Dividir")
    print("6. Números actuales")
    print("7. Limpiar pantalla")
    print("8. Terminar")
    print("Ingrese su opción:", end =" ")
    option = input()
    if option == "1":
        print("Ingrese número 1:", end =" ")
        n1 = int(input())
        print("Ingrese número 2:", end =" ")
        n2 = int(input())
    elif option == "2":
        msg = "Suma: " + str(n1) + " + " + str(n2) + " = " \
```

```
            + str(sum(n1, n2))
        show(msg)
    elif option == "3":
        msg = "Resta: " + str(n1) + " - " + str(n2) + " = " \
            + str(substract(n1, n2))
        show(msg)
    elif option == "4":
        msg = "Multiplicación: " + str(n1) + " * " + str(n2) + " = " \
            + str(multiply(n1, n2))
        show(msg)
    elif option == "5":
        if n2 != 0:
            msg = "División: " + str(n1) + " / " + str(n2) + " = " \
                    + str(divide(n1, n2))
            show(msg)
        else:
            show("¡Error, división por 0!")
    elif option == "6":
        msg = "Número 1: " + str(n1) + " \tNúmero 2: " + str(n2)
        show(msg)
    elif option == "7":
        system('cls')
    elif option == "8":
        show("Programa terminado")
```

Problema 2.3

Crear un programa para convertir bases entre sistemas numéricos: de base 10 a base 2, 8 y 16, y viceversa.

Programa: conversion_bases.py

```
def conversion_number(num, base) -> str:
    num_baseN = ""
    div = num
    while div > 0:
        mod = div % base
        div = div // base
        if mod < 10:
            num_baseN = str(mod) + num_baseN
        else:
            match mod:
                case 10:
                    digit = "A"
                case 11:
                    digit = "B"
                case 12:
                    digit = "C"
                case 13:
```

```
                    digit = "D"
                case 14:
                    digit = "E"
                case 15:
                    digit = "F"
            num_baseN = digit + num_baseN
    return num_baseN if num_baseN != "" else "0"

def baseN_to_10(string_num, base) -> int:
    sum = 0
    i = len(string_num) - 1
    for d in string_num:
        match d:
            case "A":
                digit = 10
            case "B":
                digit = 11
            case "C":
                digit = 12
            case "D":
                digit = 13
            case "E":
                digit = 14
            case "F":
                digit = 15
            case _:
                digit = d
        sum += int(digit) * base ** int(i)
        i -= 1
    return sum

op = ""
num = 0
string_num = "0"

while op != "0":
    print("Menú de opciones")
    print("0. Terminar")
    print("1. Solicitar valor decimal")
    print(f"2. Valor decimal actual: {num}")
    print(f"3. Convertir {num} a base 2")
    print(f"4. Convertir {num} a base 8")
    print(f"5. Convertir {num} a base 16")
    print(f"6. Solicitar número en alguna base")
    print(f"7. Convertir {string_num} (base 2) a base 10")
    print(f"8. Convertir {string_num} (base 8) a base 10")
    print(f"9. Convertir {string_num} (base 16) a base 10")
```

```
    op = input("Ingrese su opción: ")

    match op:
        case "0":
            print("Hasta pronto")
        case "1":
            num = int(input("Ingrese un número en decimal: "))
        case "2":
           print("-" * 30)
           print(f"|Valor decimal actual: {num}   |")
           print("-" * 30)
        case "3":
            num_baseN = conversion_number(num, 2)
            print(f"Base 2: {num_baseN}")
        case "4":
            num_baseN = conversion_number(num, 8)
            print(f"Base 8: {num_baseN}")
        case "5":
            num_baseN = conversion_number(num, 16)
            print(f"Base 16: {num_baseN}")
        case "6":
            string_num = input("Ingrese un número en la base deseada:
").upper()
        case "7":
            string_base10 = baseN_to_10(string_num, 2)
            print(f"Base 10: {string_base10}")
        case "8":
            string_base10 = baseN_to_10(string_num, 8)
            print(f"Base 10: {string_base10}")
        case "9":
            string_base10 = baseN_to_10(string_num, 16)
            print(f"Base 10: {string_base10}")
        case _:
            print("-" * 18)
            print("|Opción no válida|")
            print("-" * 18)
```

Preguntas

Ejercicios

Capítulo 4. Datos estructurados

Listas, tuplas, conjuntos y diccionarios

Además de los tipos de datos primitivos bool, int, float, complex y str, Python también implementa una serie de tipos de datos complejos, los cuales permiten agrupar en una misma variable varios datos a la vez. Estos datos también nos permiten representar secuencias de elementos, mapas o conjuntos. Veamos cada uno de ellos.

Listas

Las **listas** en Python son objetos de la clase **list** que se usan para almacenar conjuntos de elementos de tipo heterogéneo. Junto a las clases **tuple**, **range** y **str**, **list** es uno de los tipos de secuencia en Python.

Las listas en Python equivalen a los arreglos (*arrays*) en otros lenguajes de programación.

Declaración de listas

Para crear una lista en Python, asignamos a una variable corchetes vacíos ([]) para indicar que la lista inicia vacía o con los elementos especificados dentro de ellos.

Sintaxis

1.
```
variableLista = [elementos]
```

2.
```
variableLista = list(iterable)
```

Donde:
elementos: es la secuencia o conjunto de elementos separados por comas de la lista. Si no se indica, la lista se crea vacía
iterable: objeto iterable reconocible por Python, tal como listas, tuplas o cadenas. Si no se especifica, la lista se inicializará vacía

Ejemplo 3.1
Crear una lista con números enteros y acceder a sus elementos.

El ciclo **for** es implementado al estilo ***foreach*** de otros lenguajes aprovechando que las listas son elementos iterables.

Programa:
```
# Acceso a los elementos de una lista
array_data = [3, 5, 1]
```

```
print(array_data)
x = array_data[0] # x = 3
y = x + array_data[1] * array_data[2]
for d in array_data:
    print(d)
```

Ejemplo 3.2
Crear una lista con números enteros; sumar sus datos y mostrar los resultados.

El ciclo **for** puede ser implementado con la clase range() accediendo a los elementos de la lista tal y como se accede a los elementos de un arreglo unidimensional o vector.

Programa:

```
numbers = [1, 2, 7, 3, 5, 8, 6]
sum = 0
for i in range(len(numbers )):
    sum += numbers [i]
print("Suma vector: ", sum)
```

Ejemplo 3.3
Buscar el elemento 3 en la lista dada que almacena números enteros usando un ciclo **for else**.

Programa:

```
numbers = [1, 2, 4, 3, 5, 8, 6]
for n in numbers:
    if n == 3:
        break
else:
    print('No se encontró el número 3')
```

Ejemplo 3.4
Buscar el elemento 2 en la lista dada que almacena números enteros usando un ciclo **while else**.

Programa:

```
values = [5, 1, 9, 2, 7, 4]
index = 0
length = len(values)
while index < length:
    values = values[index]
    if values == 2:
        print(f'Elemento 2 encontrado en posición: {index}')
        break
    else:
        index += 1
else:
```

```
    print('El elemento 2 no se encuentra en la lista de valores')
```

Métodos de listas

Para manipular las listas, existen una serie de métodos que proporciona el lenguaje. Veamos algunos ejemplos con los más usados.

Sean:
`array`: una lista
`dato`: una variable
`posicion`: variable que guarda un índice de la lista

Se pueden aplicar los siguientes métodos a una lista en Python:

- Adicionar un elemento al final: `array.append(dato)`.
- Obtener el total de elementos con la función `len`: `len(array)`.
- Insertar en una posición específica (índice): `insert(posicion, dato)`.
- Eliminar el elemento de la última posición: `array.pop()`.
- Eliminar un elemento en una posición dada (índice): `array.pop(posición)`.
- Eliminar un elemento por el valor: `array.remove(dato)`.
- Eliminar todos los elementos de la lista: `array.clear()`.
- Buscar un dato y obtener su posición: `array.index(dato[, inicio, fin])`.
- Buscar con el operador `in`: `sw = dato in array //True: si se encuentra`.
- Ordenar la lista en orden ascendente: `array.sort()`.
- Ordenar la lista en orden descendente: `array.sort(reverse=True)`.

Ejemplo 3.5
Crear una lista con números (enteros o reales) y realizar las operaciones de agregado, búsqueda (secuencial), eliminación, actualización e inserción implementando funciones con el objetivo adicional de que puedan reutilizarse. Realizar otras operaciones sobre los elementos de la lista: suma, promedio, mayor, menor

Para el uso de colores y estilos, instalamos el paquete ***colorama***.

```
>pip install colorama
```

La apariencia en la interfaz de comandos usando este paquete puede verse como se ilustra en la siguiente figura:

Figura 3.1. Interfaz con estilos aplicados en entorno de consola

Programa: arrays.py

```
from os import system
from colorama import Fore, Back, Style

def addList(array, datum) -> None:
    array.append(datum)

def showList(array) -> None:
    print(array)
    print("")
    l =  len(array)
    for i in range(l):
        print("-----", end = "")
    print("\n|", end = "")
    for d in array:
        print(d, "|", end = "")
    print("")
    for i in range(l):
        print("-----", end = "")

def searchList(array, datum) -> int:
    index = 0
    position = -1
    length = len(array)
    while index < length and position == -1:
```

```
        if array[index] == datum:
            position = index
        else:
            index += 1
    return position

def sumArray(array):
    sum = 0
    for index, value in enumerate(array):
       sum += array[index]
    return sum

def maxArray(array) -> float:
    m = array[0]
    for index, value in enumerate(array, 1):
       if value > m:
           m = value
    return m

def minArray(array) -> float:
    m = array[0]
    for index, value in enumerate(array, 1):
       if value < m:
           m = value
    return m

system('cls')
values = []
option = ""
while option.upper() != "X":
    print(Fore.LIGHTGREEN_EX + Back.BLACK)
    print("=========================================")
    print("||           Menú de opciones           ||")
    print("=========================================")
    print(Fore.LIGHTBLUE_EX)
    print("            1. Agregar")
    print("            2. Mostrar")
    print("            3. Buscar")
    print("            4. Actualizar")
    print("            5. Eliminar")
    print("            6. Sumar datos")
    print("            7. Promedio datos")
    print("            8. Mayor dato")
    print("            9. Menor dato")
    print("            0. Limpiar pantalla")
    print("            X. Salir")
    print(Fore.LIGHTYELLOW_EX)
```

```
    option = input("            Digite opción: ")

    if option == "1":
        datum = float(input("\nDato a agregar: "))
        addList(values, datum)
    elif option == "2":
        if len(values) > 0:
            showList(values)
        else:
            print('No hay datos en la lista')
    elif option == "3":
        if len(values) > 0:
            datum = float(input("\nDato a buscar: "))
            pos = searchList(values, datum)
            if pos > -1:
                print(f'\nDato encontrado en posición: {pos}')
            else:
                print('El dato no se encuentra en la lista')
        else:
            print('No hay datos en la lista')
    elif option == "4":
        if len(values) > 0:
            datum = float(input("\nDato a modificar: "))
            pos = searchList(values, datum)
            if pos > -1:
                datum = float(input("\nNuevo dato: "))
                values[pos] = datum
                print(f'\nDato en posición {pos} actualizado')
            else:
                print('El dato no se encuentra en la lista')
        else:
            print('No hay datos en la lista')
    elif option == "5":
        if len(values) > 0:
            datum = float(input("\nDato a eliminar: "))
            pos = searchList(values, datum)
            if pos > -1:
                values.pop(pos)
                print(f'\nDato {datum} eliminado')
            else:
                print('El dato no se encuentra en la lista')
        else:
            print('No hay datos en la lista')
    elif option == "6":
        if len(values) > 0:
            print(f'\nSuma datos lista: {sumArray(values)}')
        else:
```

```
            print('No hay datos en la lista')
    elif option == "7":
        if len(values) > 0:
            prom = sumArray(values) / len(values)
            print(f'\nPromedio datos lista: {prom}')
        else:
            print('No hay datos en la lista')
    elif option == "8":
        if len(values) > 0:
            print(f'\nMayor dato lista: {maxArray(values)}')
        else:
            print('No hay datos en la lista')
    elif option == "9":
        if len(values) > 0:
            print(f'\nMenor dato lista: {minArray(values)}')
        else:
            print('No hay datos en la lista')
    elif option == "0":
        system('cls')
    elif option.upper() == "X":
        print(Fore.LIGHTGREEN_EX, '\n          Programa finalizado')
        print(Style.RESET_ALL)
    else:
        print("")
        print(Back.LIGHTWHITE_EX + "
")
        print(Fore.LIGHTRED_EX + Back.LIGHTWHITE_EX, '           Opción
no válida           ')
        print(Back.LIGHTWHITE_EX + "
")
```

Tuplas

La clase **tuple** en Python permite instanciar objetos secuenciales llamados **tuplas**, similares a los objetos de la clase **list**, pero, a diferencia de éstos, las tuplas son un tipo de secuencia inmutable, lo cual significa que no puede ser modificada en tiempo de ejecución.

Declaración de tuplas

Para crear una tupla en Python, asignamos a una variable una secuencia válida separada por comas; el uso de paréntesis es opcional.

Sintaxis

1.

```
variableTupla = elemento, elemento2, ..., elementoN
```

2.

```
variableTupla = (elemento, elemento2, ..., elementoN)
```

Ejemplo 3.6
Crear una tupla y acceder a sus elementos.

Programa:

```
# Creación y acceso a los elementos de una tupla
tupla = 3, 5, 1
print(tupla)
x = tupla[0] # x = 3
y = x + tupla[1] * tupla[2]
for d in tupla:
    print(d)
```

Ejemplo 3.7
Pedir un número de 1 a 7 al usuario y mostrar el día equivalente. El primer día es el domingo y el último el sábado.

Programa:

```
# Mostrar día
diasSemana = (
    'domingo',
    'lunes',
    'martes',
    'miércoles',
    'jueves',
    'viernes',
    'sábado'
)
dia = input('Ingrese un número correspondiente al día del semana (1-7)')
print(f'El día es {diasSemana[dia] - 1}')
for d in diasSemana:
    print(d)
```

Métodos de tuplas

Dado que las tuplas son inmutables, su manipulación es simple, ya que se limita a buscar elementos en ellas.

Sean:
`array`: una tupla
`dato`: una variable
`posicion`: variable que guarda un índice de la tupla

Se pueden aplicar los siguientes métodos a una tupla en Python:

- Obtener el total de elementos con la función `len`: `len(array)`.
- Buscar un dato y obtener su posición: `array.index(dato[, inicio, fin])`.
- Buscar con el operador `in`: `sw = dato in array //True: si se encuentra`.

Nota: acerca de la modificación de tuplas
Las tuplas son objetos inmutables, sin embargo, ellas pueden contener objetos mutables como las listas, que pueden ser modificados.

Ejemplo 3.8
Modificar los elementos mutables de una tupla.

Programa:
```
# Modificar elemento mutable de una tupla
tupla = ('a', 'b', '[1, 2, 3]', 'x', 'y', 'z')
print(tupla)
tupla[2].append(4)
print(tupla)
```

Desempaquetar una tupla (*tuple unpacking*)

En realidad, el desempaquetado se puede aplicar sobre cualquier objeto de tipo secuencia, aunque se usa comúnmente con las tuplas, y consiste en llevar a variables cada elemento de la tupla, por lo cual se requerirán tantas variables como elementos tenga la tupla separadas por comas.

Sintaxis
1.
```
var1, var2, …, varN = variableTupla
```

Ejemplo 3.9
Desempaquetar una tupla.

Programa:
```
# Desempaquetar tupla
tupla = ('a', 'b', 'c')
print(tupla)
a, b, c = tupla
print(a, b, c)
```

Conjuntos

Un conjunto es un elemento similar a una lista, pero que tiene la misma propiedad de los conjuntos matemáticos, y es que cada elemento que lo compone es único. Los conjuntos admiten información heterogénea

Sintaxis

1.

```
variableConjunto = {elementos}
```

2.

```
variableConjunto = set(iterable)
```

Donde:
elementos: es la secuencia o conjunto de elementos separados por comas. Si no se indica, el conjunto se crea vacío
iterable: objeto iterable reconocible por Python, tal como listas, tuplas o cadenas. Si no se especifica, el conjunto se inicializará vacío

Con los conjuntos de Python es posible realizar algunas de las operaciones matemáticas entre ellos. Algunas de las siguientes operaciones entre conjuntos se pueden realizar en Python:

Creamos los conjuntos a y b:

```
>>> a = {1, 2, 3}
>>> a
{1, 2, 3}
```

```
>>> b = set([0, 5, 1])
>>> b
{0, 1, 5}
```

Unión, intersección y diferencia de los conjuntos a y b:

```
>>> c = a.union(b)
>>> c
{0, 1, 2, 3, 5}
>>> c = a.intersection(b)
>>> c
{1}
```

```
>>> a
{0, 1, 3}
>>> b
{0, 1, 5}
>>> c = a | b
>>> c
{0, 1, 3, 5}
>>> c = a & b
>>> c
{0, 1}
>>> c = a - b
>>> c
{3}
```

Sean:
`a, b, c`: conjuntos
`dato`: una variable
`posicion`: variable que guarda un índice de la tupla

Se pueden aplicar los siguientes métodos a un conjunto en Python:

- Obtener el total de elementos con la función `len`: `len(a)`.
- Agregar un elemento: `a.add(dato)`.
- Buscar con el operador `in`: `sw = dato in a # True: si se encuentra`.
- Eliminar un elemento: `a.discard(dato)`.

```
>>> len(a)
3
>>> a.add(2)
>>> a
{1, 2, 3}
>>> a.add(0)
>>> a
{0, 1, 2, 3}
>>> a.discard(2)
>>> a
{0, 1, 3}
```

Diccionarios

Es una estructura de datos muy útil para el almacenamiento de datos que asocia un **valor** a una **clave**. Se podría decir que es similar a los objetos *JSON* del lenguaje Javascript.

Un diccionario puede contener información de cualquier tipo, desde datos primitivos hasta estructurados, por lo que éstos no sólo soportan enteros o caracteres, sino también listas, tuplas y otros diccionarios, entre otros.

La *clave* de un diccionario es **inmutable** y **única**, ya que actúa como una especie de variable dentro de una estructura, pero no así su *valor*, el cual irá cambiando a medida que las operaciones del programa actúen sobre éste a través de su clave.

Una aplicación útil, es crear listas de diccionarios, de esta forma, es posible manejar diversa información en una misma estructura que contiene a su vez otras estructuras.

Veamos un ejemplo del uso de diccionarios con el que se ilustrará como crearlos y acceder a sus elementos.

Ejemplo 3.
Crear un diccionario con las claves ‘Nombre’ y ‘Edad’. Agregar datos en el diccionario y mostrarlo por pantalla. Acceder a sus datos de forma individual e iterar por sus claves, sus valores y por las claves y valores.

Programa: dicts.py

```
name = input('Nombre: ')
age = int(input('Edad: '))
dict = {'Nombre': name, 'Edad': age}
print(dict)
print(f"Su nombre es: {dict['Nombre']}")
print(f"Su edad es: {dict['Edad']}")

for key in dict:
      print(f"Clave: {key}")

for value in dict.values():
      print(f"Valor: {value}")

for key, value in dict.items():
      print(f"Clave: {key} - Valor: {value}")
```

Ejemplo 3.

Crear una lista de diccionarios con las claves ‘nombre’ y ‘edad’. Agregar datos en el diccionario y mostrarlo para luego adicionarlo a una lista. Iterar la lista para mostrar sus datos.

Programa: dicts_list.py

```
list1 = []
for i in range(2):
      name = input('Nombre: ')
      age = int(input('Edad: '))
      list1.append({'nombre': name, 'edad': age})

print("-----" * 5)
for d in list1:
      print(f"Nombre: {d['nombre']}")
      print(f"Edad: {d['edad']}")
print(list1)
```

Espacios de nombres

Módulos

Paquetes

Capítulo 5. Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos en Python

Python es un lenguaje **orientado a objetos**, con soporte de primer nivel para la creación de **clases**, aunque no se requiere hacer uso de ellas para crear un programa en el lenguaje.

Suponemos que los conceptos relacionados con la **POO** ya han sido estudiados en otras asignaturas, por lo que se usarán sobre dicho supuesto enfocando las explicaciones a las implementaciones en Python.

Aunque Python es un lenguaje orientado a objetos y además fuertemente tipado, no define el concepto de atributo o método privado, por lo que todos son tratados como públicos. Podemos crear nuestras propias técnicas y usar otras recomendaciones del lenguaje para tratar adecuadamente los datos de los objetos.

El siguiente ejemplo ilustra la creación de clases e instanciación de objetos en Python para su uso en aplicaciones.

Ejemplo 3.
Crear una clase llamada *Person* con las propiedades *name* y *age*. La clase debe contener los métodos para agregar datos, devolverlos y otro que determine si la persona es mayor de edad. Crear luego un objeto tipo Person y determinar si es mayor de edad y mostrar sus datos.

Crear una clase (superclase) llamada *Person* con las propiedades *nombre* y *edad*. La clase debe contener los métodos para agregar datos, devolverlos y otro que determine si la persona es mayor de edad. Realizar además lo siguiente:

- Crear dos objetos de tipo Persona e ingresar la información respectiva
- Mostrar los datos de las personas
- Determinar cuál de éstas personas es mayor
- A partir de la clase Persona, crear otra clase para gestionar empleados llamada *Employee* con los atributos *salario mínimo* y *salario*. Además de los métodos de asignación y devolución, crear otro para determinar si un empleado gana el salario mínimo. El constructor de la clase debe permitir cargar opcionalmente el salario mínimo.

Programa: class.py

```
# Super clase Person
class Person:
    # Atributos de clase
    name = ""
    age = 0
```

```
    # Constructor
    def __init__(self) -> None:
        pass

    # Métodos de clase
    def set_name(self, name) -> None:
        self.name = name

    def set_age(self, age) -> None:
        self.age = age

    def get_name(self) -> str:
        return self.name

    def get_age(self) -> int:
        return self.age

    def adult(self) -> bool:
        if (self.age >= 18):
            return True
        else:
            return False

# La clase Employee hereda de la clase Person
class Employee(Person):
    # Atributos
    minimum_salary = 0
    salary = 0

    # Constructor
    def __init__(self, min_sal = 0) -> None:
        self.minimum_salary = min_sal

    # Métodos
    def set_minimum_salary(self, salary) -> None:
        self.minimum_salary = salary

    def get_minimum_salary(self) -> float:
        return self.minimum_salary

    def set_salary(self, salary) -> None:
        self.salary = salary

    def get_salary(self) -> float:
        return self.salary
```

```
    def salary_basic(self) -> bool:
        if self.salary == self.minimum_salary:
            return True
        else:
            return False

# Programa principal
per = Person()
per.set_name(input("Nombre: "))
per.set_age(int(input("Edad: ")))
print("Datos ingresados")
print(f"Nombre: {per.get_name()}")
print(f"Edad: {per.get_age()}")

if per.adult():
    print(f"{per.get_name()} es mayor de edad")
else:
    print(f"{per.get_name()} es menor de edad")

emp = Employee(1000000)
emp.set_name(input("Nombre empleado: "))
emp.set_age(int(input("Edad empleado: ")))
emp.set_salary(float(input("Salario empleado: ")))
print("Datos ingresados")
print(f"Nombre empleado: {emp.get_name()}")
print(f"Edad empleado: {emp.get_age()}")
print(f"Salario empleado: {emp.get_salary()}")

if emp.get_minimum_salary() > 0:
    print("Salario mínimo actual: ", emp.get_minimum_salary())
    print("Salario actual: ", emp.get_salary())
    if emp.salary_basic():
        print("Gana el mínimo")
    else:
        print("Salario diferente al mínimo")
else:
    print("No ha indicado el salario mínimo")
```

Preguntas

Ejercicios

Capítulo 6. Archivos

Archivos en Python

Las funciones para el manejo de archivos, son: ***open, close, write***

Abrir archivos

Un archivo puede ser abierto para lectura, escritura o ambas cosas.

Lectura de archivos

Este modo permite acceder al contenido del archivo sin alterarlo.

Sintaxis

1. Lectura línea a línea

```
with open('path', 'mode') as file:
      for line in file:
            # Instrucciones
```

2. Lectura completa del archivo

```
with open('path', 'mode') as variable_file:
      variable_string = variable_file.read()
```

Donde:

- *path*: ruta del archivo
- *mode*: modo apertura en modo texto (*r, w, a, x, r+*). Para modo binario agregar *b* (*rb, wb, …*)
- *file/variable_file*: es una instancia -*objeto*- de la clase *File* e incluye el carácter de escape \n -*retorno de carro*-
- *variable_string*: variable de tipo *string* (cadena) que guarda el contenido del archivo

Ejemplo 4.1

Programa:

```
print("----------Lectura general----------")
with open('file.txt', 'r') as file:
    content = file.read()
    print(content)

print("***** Lectura línea a línea*****")
```

```
with open('file.txt', 'r') as f:
    for line in f:
        print(line)
```

Escritura sobre archivos

Este modo de apertura permite la edición del archivo.

Ejemplo 4.2

Programa:

```
f = open('file', 'w')
try:
    # Instrucciones para escribir en el archivo
finally:
    f.close()
```

Capítulo 7. Bases de Datos

Bases de Datos (BD) en Python

Instalar módulo MySQL

```
> pip  install mysqlclient
```

Conexión a una base de datos MySQL

Ejemplo 4.3
Conectarse a una base de datos MySQL y hacer un *CRUD* a una tabla.

Programa:

```
import MySQLdb

DB_HOST = 'localhost'
DB_USER = 'root'
DB_NAME = 'academic-ai-001'
DB_PASS = ''

def run_query(query = ''):
    data = [DB_HOST, DB_USER, DB_PASS, DB_NAME]
    conn = MySQLdb.connect(*data)
    cursor = conn.cursor()
    cursor.execute(query)

    if query.upper().startswith('SELECT'):
        data = cursor.fetchall()
    else:
        conn.commit()
        data = None

    cursor.close()
    conn.close()

    return data

run_query("INSERT INTO subject (description) values('Cálculo I') ")
```

Instalar mysql.connector

```
> pip install mysql-connector
```

Ejemplo 4.4
Conectarse a una base de datos MySQL y hacer un *CRUD* a una tabla usando mysql.connector.

Programa:

```
import mysql.connector

def connection() -> object:
    conn = mysql.connector.connect(
        host = "localhost",
        port = 3307,
        user = "root",
        passwd = "",
        database = "testing"
    )
    return conn

def run_query(query = '') -> object:
    conn = connection()
    cursor = conn.cursor()
    cursor.execute(query)

    if query.upper().startswith('SELECT'):
        data = cursor.fetchall()
    else:
        conn.commit()
        data = None

    cursor.close()
    conn.close()

    return data

def show_records(data) -> None:
    print("Registros")
    print(f"Total registros: {len(data)}")

    for record in data:
        print(record[0])
        print(record[1])
```

```
def main_menu() -> None:
    op = ""
    while op != "0":
        print("Menú de opciones")
        print("0. Salir")
        print("1. Listar")
        print("2. Insertar")
        op = input("Ingrese su opción: ")
        match op:
            case "0":
                print("Programa finalizado")
            case "1":
                data = run_query("SELECT * FROM table1")
                show_records(data)
            case "2":
                run_query(
                    "INSERT INTO table1 (description, value)  \
                    VALUES('Cálculo IV', 1)"
                )

main_menu()
```

Capítulo 8. Frameworks web para Python

Framework Web Python Flask

Flask es un framework para Python, más simple y liviano que Django, un framework robusto y muy popular entre los amantes de este lenguaje; Flask es de hecho un micro framework basado en Wekzeug y Jinja 2.

Instalación de Flask

Desde la terminal ejecutamos el comando pip

```
>pip install flask
```

Creación de un proyecto Flask

Creamos una carpeta llamada `FlaskApp` (puede ser otro nombre) y dentro de ella creamos el archivo `app.py` y añadimos el siguiente código:

Programa: app.py

```
# Importar módulo flask
from flask import Flask
app = Flask(__name__)

# Ruta básica
@app.route("/")

# Manejador de solicitud
def main():
    return "Hola, bienvenido a Flask"

# Ejecutar programa principal
if __name__ == "__main__":
    app.run()
```

Ejecutar el proyecto

En la terminal y ubicados en la carpeta donde se encuentra el archivo, escribimos el comando:

```
>python app.py
```

Se muestra algo similar a lo ilustrado en la imagen:

```
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [27/Oct/2024 20:38:14] "GET / HTTP/1.1" 200 -
```

Esta url se escribe en la barra de direcciones de algún navegador web y se debe obtener algo como esto:

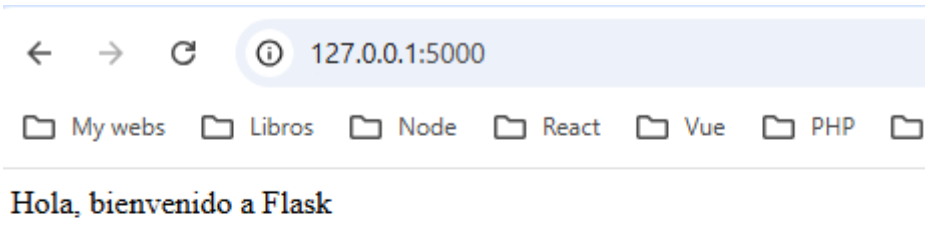

Imagen: Publicación del proyecto FlaskApp en un navegador Chrome

Este comando en realidad inicia un **servidor web** de pruebas que permite la publicación de una **página web** en un **navegador o cliente web**.

Crear páginas HTML

Dentro de la carpeta `FlaskApp` creamos la carpeta `templates`, Flask se encarga de buscar archivos en dicha carpeta en donde se almacenan las plantillas de la aplicación. El primer archivo a crear será la **página de inicio** (`index.html`).

Crear la página HTML

Dentro de la carpeta templates creamos las páginas de la aplicación

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Python Flask Bucket List App</title>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
css" rel="stylesheet"
```

```
integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6h
W+ALEwIH" crossorigin="anonymous">
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle
.min.js"
integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN
7N6jIeHz" crossorigin="anonymous"></script>

    <link href="../static/styles.css" rel="stylesheet">

  </head>
  <body>

      <footer class="footer">
        <p>&copy; Company 2015</p>
      </footer>
    </div>
  </body>
</html>
```

Modificar archivo app.py

Modificamos el archivo `app.py` en las partes que se indican. Una vez hecho esto, se reinicia el servidor y se actualiza la página en el navegador.

```
from flask import Flask, render_template
```

…

```
def main():
    # return "Hola, bienvenido a Flask"
    return render_template('index.html')
```

Framework Web Python FastAPI

FastAPI es un framework para Python que tiene como finalidad la creación de APIs. Es rápido, fácil de usar, de alto rendimiento y está preparado para la producción. Se basa en el servidor web **Starlette** y en las anotaciones de tipos estándar de Python.

Algunas de sus características son:

- Rápido: muy alto rendimiento, a la par con NodeJS y Go (gracias a Starlette y Pydantic)
- Validación automática de datos
- Gestión de errores
- Documentación interactiva de la API

- Soporte nativo async compatible con OpenAPI y JSON Schema

Instalación de FastAPI

Desde la terminal ejecutamos el comando pip para instalar FastAPI. Se necesita Python 3.7+ y el servidor ***ASGI uvicorn***, por lo que debe revisarse que la versión de Python corresponda a la adecuada e instalar el servidor uvicorn en caso de no tenerlo.

```
>pip install fastapi
```

Instalación de uvicorn

Desde la terminal ejecutamos el comando pip para instalar el servidor ***ASGI uvicorn***.

```
>pip install uvicorn
```

Preguntas

Ejercicios

Capítulo 9. Introducción al análisis de datos con Python

NumPy

Es una biblioteca para el lenguaje de programación Python que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas.

NumPy sirve para realizar cálculos numéricos eficientes en Python, especialmente con grandes volúmenes de datos, mediante su estructura principal llamada `ndarray` (arreglo multidimensional), que permite operaciones matemáticas y lógicas rápidas en matrices y vectores, siendo fundamental en ciencia de datos, IA y computación científica. Facilita la manipulación, análisis y modelado de datos numéricos, siendo la base para otras librerías como Pandas y Scikit-learn.

Características y funciones principales de NumPy

- Manejo de arreglos (Arrays): Permite crear y manipular estructuras de datos multidimensionales (vectores, matrices, tensores) de manera más eficiente que las listas de Python.
- Operaciones matemáticas: Ofrece funciones optimizadas para álgebra lineal, transformadas de Fourier, y cálculos estadísticos avanzados sobre estos arreglos.
- Eficiencia: Reduce el consumo de memoria y acelera drásticamente el procesamiento de datos numéricos gracias a su implementación en C.
- Análisis de datos: Es esencial para el preprocesamiento de datos, la creación de modelos de Machine Learning y la visualización (junto con Matplotlib).
- Vectorización: Permite aplicar operaciones a todos los elementos de un arreglo de una sola vez, en lugar de iterar, lo que mejora enormemente el rendimiento.
- Base de ecosistema: Es el cimiento sobre el que se construyen otras librerías cruciales en el análisis de datos y la IA, como Pandas, SciPy, Scikit-learn y TensorFlow.
- Rapidez: Sus operaciones son mucho más rápidas y eficientes que las listas nativas de Python para cálculos numéricos.
- Versatilidad: Se utiliza en campos como finanzas, ingeniería, investigación científica y procesamiento de imágenes.

Instalación

En consola ingresamos:

```
> pip install numpy
```

SciPy

Esta librería sirve para cálculos científicos y técnicos de alto nivel, ofreciendo herramientas eficientes para optimización, álgebra lineal, integración, interpolación, procesamiento de señales e imágenes, y estadísticas, todo construido sobre NumPy para facilitar tareas complejas en ciencia de datos e ingeniería.

Usos de SciPy

- Optimización: Resolver problemas de minimización y maximización de funciones.
- Álgebra Lineal: Operaciones avanzadas como la descomposición de matrices y cálculo de valores propios.
- Integración Numérica: Calcular integrales definidas numéricamente.
- Interpolación: Ajustar curvas y encontrar valores entre puntos de datos existentes.
- Estadística: Generar números aleatorios, describir distribuciones y realizar pruebas estadísticas (módulo scipy.stats).
- Procesamiento de Señales (scipy.signal): Filtrado y análisis de señales.
- Procesamiento de Imágenes (scipy.ndimage): Manipulación y análisis de imágenes.
- Cálculos Espaciales: Estructuras de datos y algoritmos para datos geográficos y robótica (búsqueda de vecinos, etc.).

SciPy es base para la Ciencia en Python ya que es fundamental en el ecosistema de Python para tareas científicas, junto con NumPy, Matplotlib y Pandas. Algunas características adicionales es que es una extensión de NumPy, ya que se construye sobre NumPy, aprovechando su eficiencia para el manejo de arrays multidimensionales. Además, implementa comandos de alto nivel que proporcionan funciones intuitivas y eficientes que simplifican tareas complejas sin tener que programar algoritmos desde cero.

Instalación

En consola ingresamos:

```
> pip install scipy
```

Ejemplo

Programa:

```
import numpy as np
from scipy import stats

# Tratamiento de arreglos con Numpy

# Medidas estadísticas sobre los arreglos
```

```
# En vectores
vector = np.array([2, 3, 1, -3, 5, 7, 2, 2, 9])
print(f"Vector: {vector}")
print(f"Elementos en el vector: {np.size(vector)}")
print(f"Orden del vector: {np.shape(vector)}")
print(f"Distintos de 0 en el vector: {np.count_nonzero(vector)}")
print(f"Suma vector: {np.sum(vector)}")
print(f"Promedio vector: {np.mean(vector)}")
print(f"Raíz cuadrada vector: {np.sqrt(vector)}")
print(f"Desviación estándar: {np.std(vector)}")
print(f"Mediana: {np.median(vector)}")
print(f"Moda (objeto devuelto): {stats.mode(vector)}")
print(f"Moda (valor): {stats.mode(vector).mode}")
print(f"Total que repiten moda: {stats.mode(vector).count}")

# En matrices
matrix = np.array(
    [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9],
        [-1, -2, -3]
    ]
)

print(f"Matriz:\n {matrix}")
print(f"Elemento en posición [3, 2]: {matrix[3, 2]}")
print(f"Elementos en la matriz: {np.size(matrix)}")
print(f"Orden de la matriz: {np.shape(matrix)}")
print(f"Promedio matriz: {np.mean(matrix)}")

# Ejemplo con una matriz de datos heterogéneos
sales = np.array(
    [
        ["Shampoo", 100, 24000],
        ["Crema de manos", 50, 13000],
        ["Jabón", 75, 6000],
        ["Cera para el cabello", 120, 18000],
        ["Protector solar", 60, 82000]
    ]
)

print(f"Ventas: {sales}")
sub_total_sales = sales[:, 1].astype(int) * sales[:, 2].astype(float)
print(f"Subtotal ventas: {sub_total_sales}")

sts = np.array([0, 0, 0, 0, 0])
```

```
for i in range(0, 5, 1):
    sts[i] = int(sales[i, 1]) * float(sales[i, 2])
print(f"Sub total ventas 2: {sts}")

print(f"Total ventas: {np.sum(sub_total_sales)}")
print(f"Promedio ventas: {np.mean(sts)}")

star_product = sales[np.argmax(sales[:, 1].astype(int)), 0]
print(f"Producto con más ventas : {star_product}")
```

Pandas

Es una biblioteca de Python que se usa para manipular y analizar datos. Es una herramienta fundamental para la ciencia de datos y el análisis de datos.

El nombre de Pandas proviene de ”Panel Data”, un término econométrico para conjuntos de datos estructurados.

Características de Pandas

Esta librería se usa, entre otras, cosas para:

- Limpiar y tratar datos
- Realizar análisis exploratorio de datos (EDA)
- Apoyar actividades de Machine Learning
- Realizar consultas (*queries*) en bases de datos relacionales
- Visualizar datos
- Realizar *web scraping*

Instalación

En consola ingresamos:

```
> pip install pandas
```

Ejemplo

Programa:

```
import pandas as pd

series = pd.Series([1, 2, 3, 4], index = ['a','b','c','d'])
print(f"Serie (Array asociativo) \n{series}")
print(f"Acceso a elemento de array asociativo: {series['a']}")

data = {'Name':['Ana', 'Boby', 'Carlos'], 'Age':[25, 30, 35]}
```

```
print(f"Diccionario \n{data}")
print(f"{data['Name'][1]} tiene {data['Age'][1]} años")
df = pd.DataFrame(data)
print(f"Dataframe \n{df}")
print(f"Personas: {data['Name']}")
print(f"Edades \n{df['Age']}")
print(f"Fila 2 \n{df.iloc[2]}")
```

Fuentes y referencias adicionales

Python
En Internet
Python - Wikipedia, la enciclopedia libre

The Python Language Reference
En Internet
The Python Language Reference

Referencia del Lenguaje Python
En Internet
Referencia del Lenguaje Python

El tutorial de Python
En Internet
El tutorial de Python

PEP 8 – Style Guide for Python Code
En Internet
PEP 8 – Style Guide for Python Code

Tutorial de Python. Guía básica de Python en español
En Internet
Tutorial de Python en español - Primeros pasos con Python

El Libro De Python
En Internet
El Libro De Python

Python Tutorial
En Internet
Python Tutorial

Python para principiantes
En Internet
Python para principiantes

Entrada por teclado: la función input()
En Internet
Entrada por teclado: la función input()

Formatted string literals
En Internet
2. Lexical analysis — Python 3.12.0 documentation

Is there a Python equivalent to template literals in JavaScript?
En Internet
Is there a Python equivalent to template literals in JavaScript? - Stack Overflow

Cast en Python
En Internet
Casting Python

Instalar paquete colorama
En Internet
ModuleNotFoundError: No module named 'colorama' in Python | bobbyhadz

Print Colors in Python terminal
En Internet
Print Colors in Python terminal - GeeksforGeeks

La función enumerate()
En Internet
La función enumerate() - Recursos Python

Install mysql-python (Windows)
En Internet
Install mysql-python (Windows) - Stack Overflow

Conectarse a la base de datos y ejecutar consultas
En Internet
12.2. Conectarse a la base de datos y ejecutar consultas (Python para principiantes)

Framework Web Django (Python)
En Internet
Framework Web Django (Python) - Aprende desarrollo web | MDN

Framework Web Python Flask
En Internet
Creando una Aplicación Web Desde Cero Usando Python Flask y MySQL | Envato Tuts+

81 - MySQL : Base de datos desde Python
En Internet
81 - MySQL : Base de datos desde Python

Importar módulos Python
En Internet
Importar módulos Python

Print colored text to terminal with Python
En Internet
Print colored text to terminal with Python | Sentry