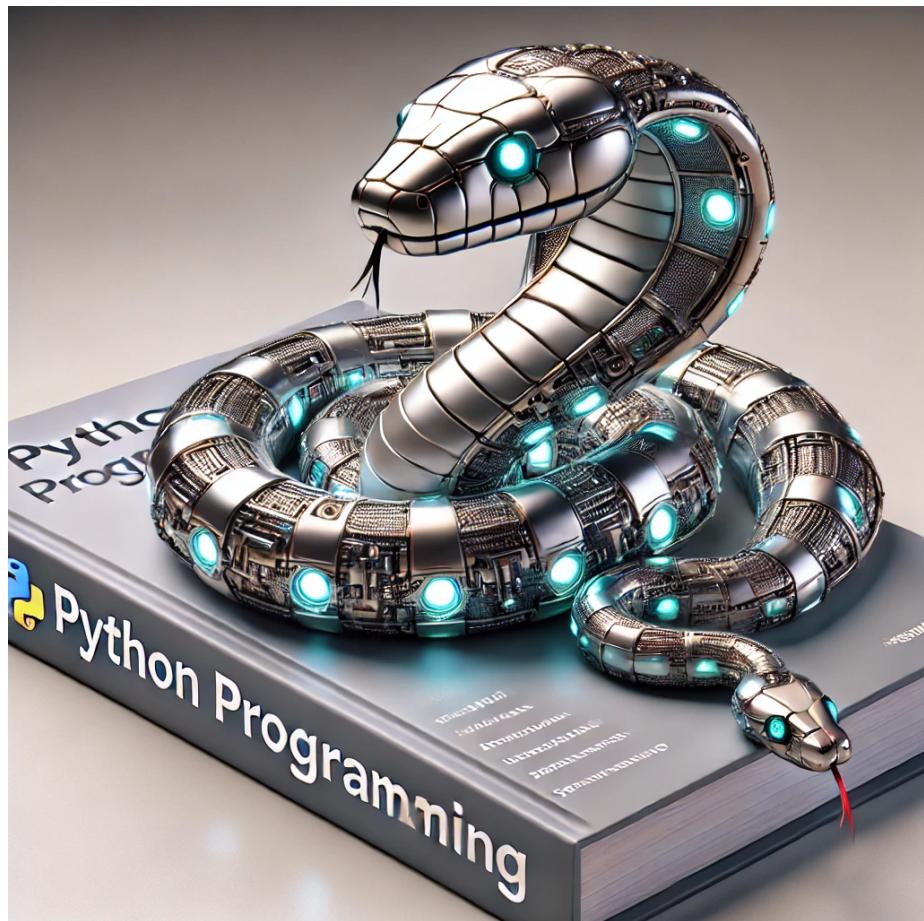


UNIDAD 2

EL LENGUAJE DE PROGRAMACIÓN PYTHON

Programación de Inteligencia Artificial
Curso de Especialización en Inteligencia Artificial y Big Data



CHATGPT prompt: Crea una imagen con un libro sobre programación en Python y que tenga sobre él una serpiente pitón robotizada

Carlos M. Abrisqueta Valcárcel
IES Ingeniero de la Cierva 2024/25

ÍNDICE

1. INTRODUCCIÓN A PYTHON	5
1.2. FILOSOFÍA: EL ZEN DE PYTHON	5
1.3. HISTÓRICO DE VERSIONES	6
1.3.1. Python 2.0	6
1.3.2. Python 3.0	6
1.3.3. Python 3.6	6
1.3.4. Python 3.8	6
1.3.5. Python 3.9	7
1.3.6. Python 3.10	7
1.3.7. Python 3.11	7
1.3.8. Python 3.12	7
1.3.9. Python 3.13	8
1.4. ENTORNOS DE DESARROLLO PARA PYTHON	8
1.4.1. Anaconda.....	8
1.4.2. PyCharm	8
1.4.3. Jupiter Notebooks y Google Colab.....	9
1.4.4. Importancia de los entornos virtuales (envs) en Python	9
1.4.5. Creación y gestión de entornos virtuales con Conda.....	9
1.5. VARIABLES, CONSTANTES Y TIPOS DE DATOS EN PYTHON	10
1.5.1. Variables y Constantes	10
1.5.2. Tipos de datos básicos en Python	10
1.5.3. Reglas de identificadores en Python.....	11
1.5.4. El Estilo CamelCase en los identificadores	11
1.5.5. Expresiones y Asignaciones en Python	12
1.5.6. Manejo de cadenas de texto en Python.....	14
1.5.7. Uso de literales de cadena en Python	16
1.5.8. Funciones para manipular cadenas	17
1.5.9. La función str en Python.....	19
1.5.10. Formateo de cadenas con el método format ()	21
1.5.11. Símbolos utilizados en el método format ()	22
1.5.12. La función print en Python.....	23
1.6. LAS FECHAS EN PYTHON	23
1.6.1. Utilizando la función import en Python.....	24
1.6.2. La función input en Python.....	25
1.6.3. Comentarios en Python	26
1.6.4. Utilización de Docstrings en IDEs de Python	26
1.6.5. Dividiendo una instrucción en varias líneas en Python.....	27
2. SENTENCIAS DE CONTROL EN PYTHON.....	28
2.1. EXPRESIONES BOOLEANAS EN PYTHON.....	28
2.1.1. Operador is en Python	28
2.2. CONDICIONALES EN PYTHON.....	29
2.3. SENTENCIAS ITERATIVAS EN PYTHON	30
2.3.1. Ejemplos de Bucles.....	31
2.3.2. Bucle for.....	32
2.3.3. Bucles Anidados	32
2.3.4. La función range	33
3. ESTRUCTURAS DE DATOS EN PYTHON.....	33
3.1. LISTAS	34
3.1.1. Listas con elementos de diferentes tipos de datos	35
3.1.2. Listas anidadas y estructuras de datos complejas	35
3.1.3. Comprensión de listas	36
3.1.4. Inicialización de matrices con bucles for	36

3.1.5. <i>Realizando Slicing en Listas</i>	36
3.1.6. <i>Concatenación de listas en Python</i>	36
3.2. DICCIONARIOS	37
3.2.1. <i>Recorrido de un diccionario</i>	38
3.2.2. <i>Métodos copy, fromkeys y get en Diccionarios Python</i>	39
3.2.3. <i>Relación entre diccionarios en Python y el formato Json</i>	40
3.3. TUPLAS.....	42
3.3.1. <i>Accediendo a elementos de la Tupla</i>	42
3.3.2. <i>Empaquetado y desempaquetado de tuplas</i>	42
3.3.3. <i>Métodos de las tuplas</i>	42
3.3.4. <i>Packing y unpacking en tuplas: Inicialización, salida y bucles</i>	43
3.3.5. <i>Uso de la función zip en Python</i>	44
3.4. CONJUNTOS EN PYTHON.....	45
3.4.1. <i>Operaciones con conjuntos</i>	45
3.4.2. <i>Métodos de conjuntos</i>	45
3.5. FUNCIONES PARA COLECCIONES EN PYTHON.....	46
3.5.1. <i>La función len ()</i>	46
3.5.2. <i>Las funciones min() y max()</i>	46
3.5.3. <i>La función sum ()</i>	46
3.5.4. <i>La función sort () y el método sorted()</i>	47
4. LAS FUNCIONES	48
4.1. DEFINICIÓN E INVOCACIÓN DE FUNCIONES.....	48
4.2. ARGUMENTOS Y VALOR DE RETORNO.....	48
4.3. RECURSIVIDAD	48
4.4. IMPORTANCIA DE LA RECURSIVIDAD EN INTELIGENCIA ARTIFICIAL.....	48
4.5. DOCUMENTACIÓN Y ANOTACIONES DE TIPO EN FUNCIONES	49
4.5.1. <i>Pasar argumentos a funciones en Python</i>	50
4.6. PASO POR VALOR Y PASO POR REFERENCIA.....	51
4.7. ÁMBITO DE LAS VARIABLES EN PYTHON	52
4.8. FUNCIONES LAMBDA	53
4.8.1. <i>Limitaciones de las funciones Lambda</i>	54
4.8.2. <i>Uso de filter, map, reduce y sorted con funciones lambda</i>	54
5. PANDAS DATAFRAMES	55
5.1. CREACIÓN DE DATAFRAMES	55
5.2. ANÁLISIS INICIAL DEL DATAFRAME CON <i>SHAPE, SIZE Y NDIM</i>	56
5.3. SELECCIÓN DE DATOS CON <i>ILOC, LOC Y COLUMNS</i>	57
5.4. OPERACIONES DML EN PANDAS	59
5.5. CARGA DE ARCHIVOS CSV Y JSON EN PANDAS	60
5.6. MANEJO DE DATOS INEXISTENTES EN PANDAS	61
6. LA LIBRERÍA NUMPY	62
6.1. TRATAMIENTO DE ARRAYS EN NUMPY	62
6.1.1. <i>Exploración y manipulación de dimensiones y forma en arrays con Numpy</i>	62
6.2. FILTRADO DE ARRAYS EN NUMPY	64
6.2.1. <i>Tipos de datos en NumPy</i>	65
6.2.2. <i>Copias y vistas (views) en NumPy</i>	66
6.2.3. <i>Recorrido de arrays en NumPy con nditer y ndenumerate</i>	67
7. PROGRAMACIÓN ORIENTADA A OBJETOS EN PYTHON	68
7.1. DEFINIENDO UNA CLASE EN PYTHON: MÉTODOS Y ATRIBUTOS	68
7.2. CONSTRUCTORES Y DESTRUCTORES EN PYTHON	70
7.3. HERENCIA EN PYTHON	70
7.4. MÉTODOS: ESTÁTICOS, DE INSTANCIA Y DE CLASE.....	71
7.5. DECORADORES Y @PROPERTY	72
7.6. VISIBILIDAD DE MIEMBROS DE UNA CLASE	72

7.7.	POLIMORFISMO EN PYTHON	73
7.8.	USO DE INTERFACES EN POO Y EJEMPLO CON EL PATRÓN OBSERVER.....	74
8.	MANEJO DE EXCEPCIONES EN PYTHON	75
8.1.	MANEJO DE MÚLTIPLES EXCEPCIONES	75
9.	TRATAMIENTO DE FICHEROS EN PYTHON	76
9.1.	LECTURA Y ESCRITURA EN FICHEROS	76
9.2.	MANEJO DE EXCEPCIONES CON FICHEROS EN PYTHON	77
10.	LOS GRÁFICOS EN PYTHON	78
10.1.	GRÁFICO DE LÍNEAS	78
10.2.	USO DE COLORES EN MATPLOTLIB	79
10.3.	GRÁFICOS DE DISPERSIÓN (SCATTER PLOTS) EN MATPLOTLIB	81
10.4.	GRÁFICO DE BARRAS	84
10.5.	HISTOGRAMA.....	85
10.6.	PIECHARTS	85
10.7.	EL MÉTODO SUBPLOTS	87
10.8.	VISUALIZACIÓN DE SERIES TEMPORALES CON OBJETOS <i>DATETIME</i> EN MATPLOTLIB	88

1. INTRODUCCIÓN A PYTHON

Python, concebido por Guido van Rossum y liberado en 1991, ha sido desarrollado bajo una filosofía que enfatiza la simplicidad y la legibilidad del código. Su sintaxis permite a los programadores expresar conceptos en menos líneas de código que en lenguajes como C++ o Java.

1.2. FILOSOFÍA: EL ZEN DE PYTHON

El Zen de Python, escrito por Tim Peters, enfatiza la belleza y la simplicidad del código. Algunos de sus aforismos incluyen: “Lo explícito es mejor que lo implícito”, “La simplicidad es mejor que la complejidad”. Estos principios guían el diseño y desarrollo del lenguaje. Las características de python son las siguientes:

Tipado Dinámico: Python es un lenguaje de tipado dinámico, lo que significa que el tipo de dato de una variable se determina en tiempo de ejecución, permitiendo una mayor flexibilidad al asignar y manipular variables.

Soporte Multiparadigma: Python es un lenguaje de programación multiparadigma que soporta programación orientada a objetos, imperativa y funcional, ofreciendo así diversas opciones para resolver problemas y desarrollar sistemas.

Portabilidad: La portabilidad de Python se refiere a la capacidad del lenguaje para ejecutarse en diversas plataformas y sistemas operativos sin necesidad de realizar modificaciones significativas en el código fuente.

Garbage Collection: Python cuenta con un recolector de basura (“garbage collector”) que gestiona automáticamente la memoria, liberando aquellos objetos que ya no están siendo utilizados y asegurando que la memoria utilizada sea la mínima necesaria.

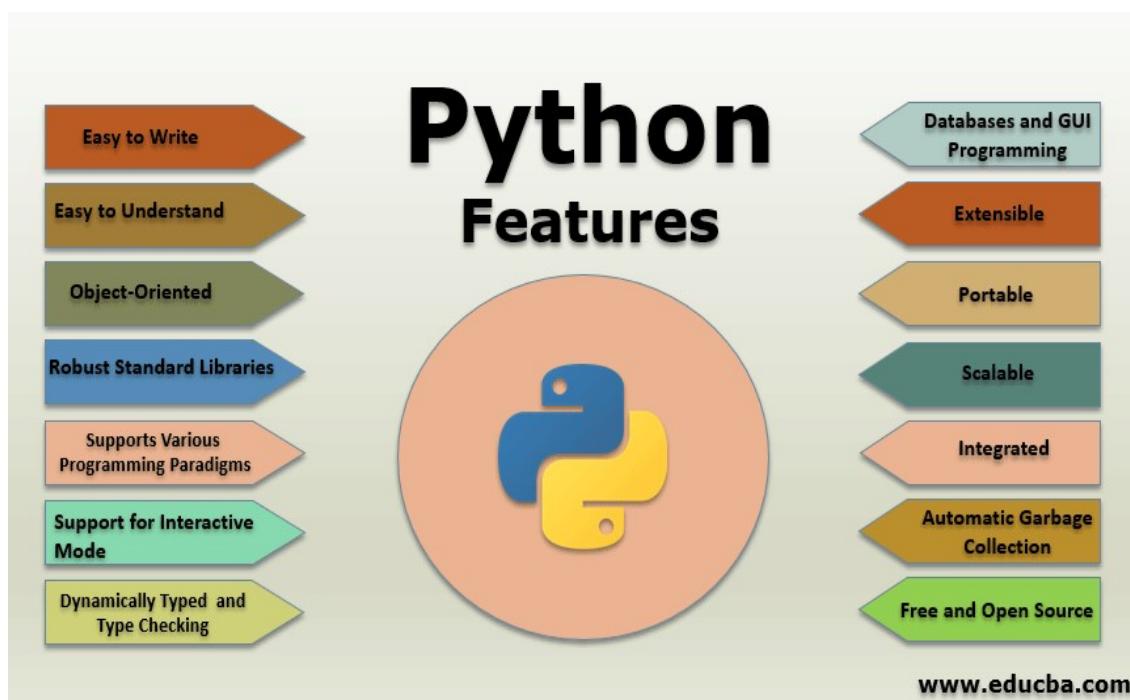


Figura 1: Las características de Python

1.3. HISTÓRICO DE VERSIONES

1.3.1. Python 2.0

Lanzado en el año 2000, Python 2.0 introdujo varias nuevas características, tales como la recolección de basura y soporte completo para Unicode. Durante su desarrollo, se colocó un énfasis significativo en la retrocompatibilidad con versiones anteriores.

Características notables:

- **List comprehensions:** proporcionando una sintaxis más concisa para crear listas.
- **Recolección de basura:** sistema de gestión automática de memoria.

1.3.2. Python 3.0

Python 3.0, lanzado en 2008, fue diseñado para rectificar los defectos percibidos del lenguaje. No se diseñó para ser compatible con las versiones anteriores, marcando un cambio importante en la evolución del lenguaje.

Características notables:

- `print()` se convierte en una función, alterando su uso en comparación con Python 2.
- Introducción de la sintaxis “`nonlocal`” para usar variables no locales en funciones anidadas.
- Nuevos operadores de división: `/` realiza la división clásica y `//` la división entera.

1.3.3. Python 3.6

Python 3.6, lanzado en 2016, introdujo varias características nuevas, optimizaciones y actualizaciones.

Características notables:

- **Formateo de cadenas de texto (f-strings):** una nueva forma de incrustar expresiones dentro de literales de cadena usando `f"texto {expresión}"`.
- **Tipo de dictado y anotaciones de variables:** permitiendo la especificación explícita de los tipos de datos.

1.3.4. Python 3.8

Lanzado en 2019, Python 3.8 trajo varias optimizaciones y algunas características nuevas significativas.

Características notables:

- **Asignación con expresión (operador Walrus):** permite asignar valores a variables como parte de una expresión usando la sintaxis `NAME := expr`.
- **Protocolos de tipos:** soporte mejorado para tipado estático y verificaciones de tipos en tiempo de ejecución.

1.3.5. Python 3.9

La versión 3.9 del lenguaje, lanzada en 2020, introdujo varias nuevas sintaxis y actualizaciones de la API.

Características notables:

- Operadores de fusión y actualización de diccionarios, que permiten unir y actualizar diccionarios con `|` y `+=`.
- Análisis sintáctico de Zonas Horarias: El módulo `zoneinfo` trae soporte para el análisis de zonas horarias IANA al estándar de la biblioteca.

1.3.6. Python 3.10

Lanzada en octubre de 2021, introdujo varias nuevas sintaxis y actualizaciones de la API.

Características notables:

- Pattern matching: Se introduce el equivalente a `switch/case`, pero con más potencia gracias a la sintaxis `match-case`.
- Introducción de las union-types

1.3.7. Python 3.11

Lanzado en octubre de 2022, Python 3.11 se enfocó en mejorar el rendimiento y la simplicidad en el manejo de excepciones y tipos.

Características notables:

- **Mejoras en el rendimiento:** Python 3.11 ofrece mejoras en el rendimiento, con una aceleración general de las ejecuciones en torno al 10-60% en varios casos de uso.
- Nuevas excepciones: Se introduce `ExceptionGroup`, una nueva clase que permite agrupar múltiples excepciones en una sola, y `except*` para manejar excepciones agrupadas.

1.3.8. Python 3.12

Lanzado en octubre de 2023, esta versión se centró en seguir mejorando el rendimiento y refinar la funcionalidad de los tipos y las excepciones introducidas en versiones anteriores.

Características notables:

- **Iteración y rendimiento:** Se continúan optimizando algunas operaciones comunes para que sean más rápidas, como la concatenación de cadenas y el manejo de estructuras de datos grandes.
- **Depuración avanzada:** Mejoras en las herramientas de depuración, que permiten un análisis más detallado del código, especialmente en entornos de desarrollo.

1.3.9. Python 3.13

Lanzado en octubre de 2024, Python 3.13 introduce varias funcionalidades que mejoran la experiencia del desarrollador, optimizan el rendimiento y amplían el soporte de plataformas.

Características notables:

- Una de las actualizaciones más notables es el nuevo intérprete interactivo, inspirado en *PyPy*. Ahora cuenta con edición de líneas múltiples, soporte para color y trazas de excepción coloreadas, lo cual facilita la depuración y mejora la experiencia de codificación en tiempo real.
- Compatibilidad y deprecaciones: Algunos métodos y funciones que han sido reemplazados en las últimas versiones comienzan a ser oficialmente eliminados, mejorando la consistencia del lenguaje a largo plazo.

Python release cycle

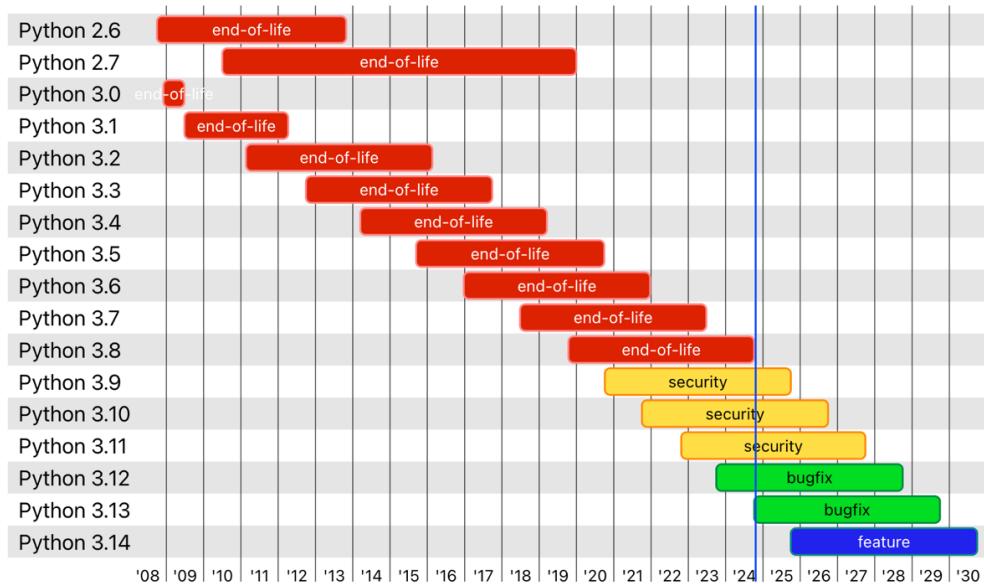


Figura 2: Evolución de las versiones de Python

1.4. ENTORNOS DE DESARROLLO PARA PYTHON

Existen diversos IDEs y distribuciones que facilitan la programación en Python, proporcionando herramientas útiles para desarrollo, prueba y depuración de código.

1.4.1. Anaconda

Anaconda es una distribución gratuita y de código abierto que busca simplificar el manejo de paquetes y su despliegue. Su plataforma *conda* facilita la gestión y despliegue de software en diferentes sistemas operativos.

1.4.2. PyCharm

Desarrollado por JetBrains, PyCharm es un IDE utilizado comúnmente para desarrollo en Python. Ofrece potentes herramientas para desarrollo web, ciencia de datos, inteligencia artificial y desarrollo de software en general. Con tu cuenta de alumno, tienes gratuitamente una licencia de

desarrollo, tan sólo tienes que utilizar el correo electrónico @alu.murciaeduca.es a la hora de descargarlo.

1.4.3. Jupiter Notebooks y Google Colab

Jupyter Notebooks es un entorno de desarrollo que facilita la ciencia de datos al permitir combinar código, gráficos y texto en un mismo documento. Google Colab, por su parte, permite ejecutar estos notebooks en la nube, facilitando el acceso a hardware potente y compartición de trabajo.

1.4.4. Importancia de los entornos virtuales (envs) en Python

Los entornos virtuales, también conocidos por sus abreviaciones env o venv, han surgido como una solución esencial para gestionar dependencias y versiones en los proyectos de Python. Un env en Python es un espacio aislado donde se pueden instalar paquetes y mantener dependencias sin interferir con el sistema de Python global o con otros proyectos.

Utilidades y ventajas:

- **Gestión de Dependencias:** Los envs permiten tener un control preciso sobre las versiones de los paquetes utilizados en un proyecto, asegurando que las dependencias documentadas son coherentes y reproducibles.
- **Aislamiento:** Cada proyecto puede tener sus propios requisitos y, gracias a los envs, es posible mantenerlos aislados para evitar conflictos entre versiones de paquetes.
- **Compatibilidad:** Al utilizar envs, los desarrolladores pueden trabajar con diferentes versiones de Python y sus respectivas bibliotecas sin enfrentarse a incompatibilidades.
- **Facilita la Colaboración:** Los colaboradores de un proyecto pueden replicar fácilmente el entorno de desarrollo al utilizar un env con las dependencias especificadas.

1.4.5. Creación y gestión de entornos virtuales con Conda.

Conda es un sistema de gestión de paquetes y también un sistema de gestión de entornos, proporcionando herramientas para instalar, correr y actualizar bibliotecas y dependencias dentro de diferentes entornos en tu sistema de desarrollo.

Ventajas de Conda:

- **Gestión de Paquetes:** Conda permite instalar y gestionar paquetes de diferentes versiones y configuraciones.
- **Manejo de Entornos:** Permite crear, exportar, listar, eliminar e intercambiar entornos que tienen diferentes versiones de Python y/o paquetes instalados.
- **Compatibilidad:** Asegura que los paquetes dentro de un entorno sean compatibles.
- **Reproducibilidad:** Los entornos y paquetes se pueden compartir, facilitando la reproducción de resultados y configuraciones en diferentes sistemas.

Creación de un Entorno Virtual con Conda:

Para crear un entorno virtual utilizando Conda, puedes utilizar la siguiente sintaxis básica:

```
conda create --name myenv # Crear un entorno llamado 'myenv'
```

En este caso, ‘myenv’ es el nombre del nuevo entorno. Si también deseas especificar la versión de Python, puedes hacerlo de la siguiente manera:

```
conda create --name myenv python=3.8 # Especificando versión de Python
```

Activación y Desactivación de Entornos:

Activar y desactivar entornos es esencial para gestionar múltiples proyectos y asegurar que las dependencias utilizadas no entren en conflicto.

```
conda info --envs      # listar los entornos disponibles  
conda activate myenv # Activar el entorno 'myenv'  
conda deactivate     # Desactivar el entorno activo
```

Gestión de Paquetes:

Conda también permite gestionar paquetes, permitiendo instalar, actualizar y eliminar paquetes:

```
conda install numpy    # Instalar un paquete (ej. numpy)  
conda update numpy    # Actualizar un paquete  
conda remove numpy    # Desinstalar un paquete
```

Los entornos virtuales y la gestión de paquetes son cruciales para un desarrollo de software efectivo y eficiente. Conda proporciona herramientas robustas para manejar tanto paquetes como entornos, asegurando que las dependencias sean manejadas de manera coherente y reproducible. Conda se ha destacado en la comunidad de ciencia de datos, pero su utilidad es amplia y aplicable a una variedad de proyectos y disciplinas en desarrollo de software en Python.

1.5. VARIABLES, CONSTANTES Y TIPOS DE DATOS EN PYTHON

Python ofrece una rica variedad de tipos de datos y estructuras de datos incorporadas para facilitar la manipulación de datos y la implementación de algoritmos.

1.5.1. Variables y Constantes

En Python, las variables no requieren declaración explícita. Una variable se crea en el momento que le asigna un valor por primera vez. Por otro lado, las constantes idealmente no deben cambiar durante la ejecución del programa.

```
variable = 15 # Ejemplo de variable  
CONSTANTE = 15 # Ejemplo de constante
```

1.5.2. Tipos de datos básicos en Python

Los tipos de datos numéricos en Python incluyen enteros, números de punto flotante y números complejos. Las cadenas de caracteres pueden ser definidas usando comillas simples o dobles y pueden contener una secuencia de caracteres de cualquier longitud.

```
entero = 10  
flotante = 20.5
```

```
complejo = 3 + 5j
cadena = "Hola, Python"
```

1.5.3. Reglas de identificadores en Python

Los identificadores en Python, que incluyen los nombres de variables, siguen un conjunto de reglas y convenciones definidas para garantizar que el código sea legible y evite conflictos con las palabras clave y elementos internos del lenguaje. A continuación, se describen las reglas básicas para formar identificadores de variables en Python:

- **Caracteres permitidos:** Los identificadores deben estar compuestos únicamente por letras (mayúsculas o minúsculas), dígitos y/o el carácter guion bajo (_).
- **Inicio del identificador:** No pueden comenzar con un dígito. Generalmente, un identificador debe comenzar con una letra (a-z, A-Z) o un guion bajo (_).
- **Sensibilidad a mayúsculas y minúsculas:** Python es sensible a mayúsculas y minúsculas, por lo que, Variable y variable serían dos identificadores distintos.
- **Evitar palabras clave:** Los identificadores no deben coincidir con las palabras clave de Python, como if, else, while, etc., ya que estas están reservadas para funcionalidades específicas del lenguaje.
- **Sin límite de longitud:** No hay un límite para la longitud de los identificadores, pero es recomendable utilizar nombres concisos y que indiquen claramente el propósito de la variable.
- **Convenciones:** Aunque no es una regla, es común usar el estilo snake case (todo en minúsculas con palabras separadas por guiones bajos) para los nombres de variables, mientras que CamelCase se utiliza para nombres de clases.

Ejemplos de identificadores válidos:

```
mi_variable = 10
otraVariable = 20
_variable3 = 30
```

Ejemplos de identificadores inválidos:

```
3variable = 40 # Inicia con un dígito
mi-variable = 50 # Contiene un carácter no permitido (-)
if = 60 # Coincide con una palabra clave
```

Estas reglas y convenciones ayudan a mantener el código claro y a evitar errores semánticos en la programación. Es vital para los programadores estar familiarizados con estas para escribir código Python efectivo y fácil de entender.

1.5.4. El Estilo CamelCase en los identificadores

El camelCase es una convención para la escritura de identificadores como variables, funciones y otras entidades en la programación y otros contextos relacionados. Es especialmente prominente en lenguajes como Java y JavaScript, aunque también se usa en Python principalmente para nombrar clases. La nomenclatura camelCase se caracteriza por las siguientes propiedades:

- **Sin espacios:** Los identificadores escritos en camelCase no contienen espacios.

- Capitalización: La primera letra de cada palabra, excepto la inicial (a veces), se escribe con mayúscula.
- Legibilidad: La capitalización de las iniciales de cada palabra ayuda a mejorar la legibilidad, haciendo que las fronteras de las palabras sean fácilmente identificables.

Ejemplo de uso:

```
miPrimeraVariable = 10
calcularAreaTriangulo = 0.5 * baseTriangulo * alturaTriangulo
```

En estos ejemplos, `miPrimeraVariable` y `calcularAreaTriangulo` están escritos en camelCase. Note cómo cada palabra después de la primera comienza con una letra mayúscula, lo que facilita la identificación de las palabras individuales que componen el identificador, incluso sin la presencia de espacios o guiones bajos.

Es esencial mencionar que diferentes comunidades y lenguajes de programación pueden tener preferencias diversas en cuanto a las convenciones de nomenclatura. En Python, por ejemplo, mientras que camelCase es común para los nombres de las clases, para variables y funciones se suele preferir el uso de snake case, donde las palabras se escriben en minúsculas y se separan por guiones bajos.

```
class MiClase:
    def metodoEjemplo(self): pass
    variable_ejemplo = 20
```

La elección de la convención a menudo depende del estilo de codificación adoptado por un proyecto o equipo en particular. Es crucial ser consistente en la adopción de estas convenciones para mantener la base de código clara y fácil de leer para todos los colaboradores.

1.5.5. Expresiones y Asignaciones en Python

Las expresiones en Python son combinaciones de valores y operadores que pueden ser evaluadas para obtener un resultado. Python proporciona una variedad de operadores que permiten construir expresiones para realizar operaciones matemáticas, de comparación, lógicas, y de asignación, entre otras.

Operadores Aritméticos Básicos:

- Suma: `+`
- Resta: `-`
- Multiplicación: `*`
- División: `/`
- Módulo: `%`
- Exponente: `**`
- División Entera: `//`

Por ejemplo,

```
result = 3 * (4 + 5) # result se evaluá como 27
```

Operadores de Comparación y Lógicos:

- Igual: ==
- Diferente: !=
- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=
- Y lógico: and
- O lógico: or
- No lógico: not

Operador de Asignación:

El operador de asignación (=) se utiliza para asignar un valor a una variable. La sintaxis general es:

```
variable = expresion
```

Donde el valor de la expresión se asigna a variable. Es imperativo notar que la asignación no es una relación de igualdad matemática, sino una operación que establece un vínculo entre un identificador (nombre de variable) y un valor o expresión.

Operadores de Asignación Compuesta:

Python también admite operadores de asignación compuesta, que combinan una operación aritmética y una operación de asignación en una única expresión.

- suma en asignación: +=
- resta en asignación: -=
- multiplicación en asignación: *=
- división en asignación: /=
- módulo en asignación: %=
- exponente en asignación: **=
- división entera en asignación: //=

```
# Ejemplo de uso de operadores y variables en Python

# Declaración de variables
a = 10
b = 20

# Operadores aritméticos
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b # Resultado: 0.5 (en Python 3.x)

# Operadores de comparación
```

```
es_igual = a == b
es_diferente = a != b
es_mayor = a > b
es_menor = a < b

# Operadores lógicos
y_logico = es_igual and es_diferente
o_logico = es_mayor or es_menor
negacion = not es_igual

# Operadores de asignación
a += 5 # Equivalente a: a = a + 5
b *= 2 # Equivalente a: b = b * 2

# Impresión de resultados
print("Suma:", suma)
print("Resta:", resta)
print("Y Logico:", y_logico)
print("O Logico:", o_logico)
```

La función `type()` en Python es utilizada para obtener el tipo de un objeto. Cuando se pasa un objeto como argumento a esta función, retorna el tipo del mismo. Ejemplo:

```
x = 10
print(type(x))      # <class 'int'>
```

Conversión de Tipos:

La conversión de tipos, por otro lado, se refiere al proceso de convertir un valor de un tipo de dato a otro. En Python, es posible realizar la conversión de tipos utilizando funciones predefinidas como `int()`, `float()`, y `str()` para convertir valores a enteros, números de punto flotante y cadenas de texto respectivamente.

Sin embargo, es crucial mencionar que no todas las conversiones de tipo son posibles. Si Python no puede determinar cómo convertir un valor de un tipo a otro, se generará un `ValueError` o un `TypeError`.

Ejemplo de una conversión de tipo inválida:

```
x = "texto"
y = int(x) # Generará un ValueError, ya que "texto" no puede convertirse
           # a entero
```

1.5.6. Manejo de cadenas de texto en Python

Las cadenas de texto (`strings`) en Python son arrays o vectores de caracteres y se encuentran entre los tipos de datos más utilizados. Python proporciona una amplia variedad de métodos y operadores para manipular cadenas, los cuales facilitan la realización de operaciones comunes de una manera fácil y directa.

Creación de Cadenas:

Las cadenas se pueden crear utilizando comillas simples o dobles, lo que permite incluir comillas y otros caracteres especiales dentro de ellas.

```
s1 = 'Hello, world!'
s2 = "It's a wonderful day!"
```

Cadenas Multilínea y Especiales:

Python permite la creación de cadenas multilínea y también el uso de caracteres especiales mediante el uso de secuencias de escape, tales como \n para nuevas líneas o \t para tabulaciones.

```
multiline_string = """This is a multiline
string spanning several lines."""
special_chars = "New line: \\n Tab: \\t"
```

Operaciones Básicas con Cadenas:

- Concatenación: Utilizando el operador +, se pueden unir dos o más cadenas.
- Repetición: Utilizando el operador *, se puede repetir una cadena un número específico de veces.

```
s3 = s1 + " " + s2          # Indexación: obtiene un carácter
s4 = "Repeat me! " * 3       # Segmentación: obtiene una subcadena
```

La indexación es una característica fundamental en Python para acceder a los elementos de estructuras de datos, tales como listas y cadenas, entre otros. La indexación permite acceder, modificar y referenciar los elementos de una secuencia mediante el uso de números enteros, los cuales representan la posición de un elemento dentro de la estructura.

Indexación Positiva:

La indexación en Python comienza desde 0, es decir, el primer elemento de una cadena tiene el índice 0, el segundo elemento tiene el índice 1, y así sucesivamente.

```
mi_cadena = "Programación"
primer_caracter = mi_cadena[0]    # Valor: 'P'
```

Indexación Negativa:

Python también permite la indexación negativa, donde el índice -1 se refiere al último elemento, -2 al penúltimo, y así sucesivamente.

```
ultimo_caracter = mi_cadena[-1] # Valor: 'n'
```

Indexación de Slicing:

Python permite rebanar cadenas usando la notación [inicio : fin : paso], donde inicio es el índice inicial, fin es el índice hasta donde se quiere obtener la subcadena (sin incluir), y paso es la distancia entre índices consecutivos.

```
sub_cadena = mi_cadena[1:4] # Valor: 'yth'  
  
mi_cadena = "Python"  
  
# Slicing con índices negativos  
sub_cadena = mi_cadena[-3:-1]  
print(sub_cadena) # Salida: ho
```

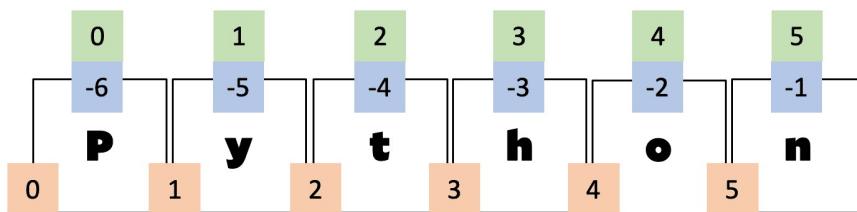


Figura 3: Python slicing

Es fundamental mencionar que el acceso a un índice fuera del rango de la estructura resultará en un error de tipo `IndexError`. Por lo tanto, es crucial asegurarse de que los índices utilizados estén dentro del rango válido para la estructura de datos con la que se está trabajando.

1.5.7. Uso de literales de cadena en Python

En Python, los `string literals`, o literales de cadena, son caracteres o secuencias de caracteres que representan valores de texto y son interpretados de manera especial cuando están precedidos por una barra invertida (\). Estos literales se utilizan para insertar caracteres especiales dentro de las cadenas de texto.

Uno de los literales de cadena más utilizados es \n, que se usa para insertar un nuevo renglón o salto de línea en una cadena de texto.

Ejemplo básico de uso:

```
print("Hola\nMundo")
```

La salida de este código sería:

```
Hola  
Mundo
```

Otro ejemplo común es \t, que inserta un tabulador horizontal (espaciado). Algunos literales de cadena comunes en Python incluyen:

- \n - Nueva línea
- \t - Tabulador horizontal
- \" - Comillas dobles
- \' - Comillas simples
- \\ - Barra invertida

Estos literales permiten a los desarrolladores manejar convenientemente caracteres especiales y formatos dentro de las cadenas de texto, permitiendo una mayor flexibilidad y control sobre la salida del texto en sus programas.

Nota: Es importante considerar que si se desea utilizar una barra invertida como carácter común y no como prefijo de un literal, esta debe ser escapada como \\ para evitar interpretaciones erróneas.

Los literales de cadena son herramientas esenciales, especialmente en situaciones donde el formato y la presentación del texto son críticos para la comunicación de información al usuario.

1.5.8. Funciones para manipular cadenas

Python ofrece una multitud de métodos para manipular y consultar cadenas, por ejemplo:

- `capitalize()`: Convierte el primer carácter de la cadena a mayúsculas y el resto a minúsculas.
- `swapcase()`: Invierte las mayúsculas por minúsculas y viceversa en toda la cadena.
- `replace(old, new)`: Reemplaza todas las ocurrencias del substring old por new.
- `split(separator)`: Divide la cadena en una lista de palabras utilizando separator como delimitador.
- `strip()`: Elimina los espacios en blanco en ambos extremos de la cadena.
- `rstrip()`: Elimina los espacios en blanco en el extremo derecho de la cadena.
- `lstrip()`: Elimina los espacios en blanco en el extremo izquierdo de la cadena.
- `find(substring)`: Retorna el índice de la primera ocurrencia de substring (-1 si no se encuentra).
- `index(substring)`: Similar a find, pero lanza una excepción si substring no se encuentra.
- `rindex(substring)`: Retorna el índice de la última ocurrencia de substring.
- `len()`: Retorna la longitud de la cadena.
- `title()`: Convierte el primer carácter de cada palabra a mayúsculas y el resto a minúsculas.
- `count(substring)`: Retorna el número de ocurrencias de substring en la cadena.
- `upper()`: Convierte todos los caracteres de la cadena a mayúsculas.
- `lower()`: Convierte todos los caracteres de la cadena a minúsculas.

Mira los ejemplos propuestos a continuación:

```
cadena_original = " Python es un lenguaje de programación. "

# capitalize(): Convierte el primer carácter en mayúsculas
cadena_capitalize = cadena_original.capitalize()

# swapcase(): Invierte las mayúsculas y minúsculas
```

```
cadena_swapcase = cadena_original.swapcase()

# replace(old, new): Reemplaza un substring por otro
cadena_replace = cadena_original.replace("Python", "Java")

# split(separator): Divide la cadena en una lista utilizando un
separador
cadena_split = cadena_original.split(" ")

# strip(): Elimina espacios en blanco en ambos extremos de la cadena
cadena_strip = cadena_original.strip()

# rstrip() y lstrip(): Elimina espacios a la derecha e izquierda
respectivamente
cadena_rstrip = cadena_original.rstrip()
cadena_lstrip = cadena_original.lstrip()

# find(substring): Retorna el indice de la primera ocurrencia de un
substring (-1 si no se encuentra)
indice_find = cadena_original.find("lenguaje")

# index(substring): Similar a find, pero lanza una excepción si no se
encuentra el substring
indice_index = cadena_original.index("lenguaje")

# rindex(substring): Retorna el índice de la última ocurrencia de un
substring
indice_rindex = cadena_original.rindex("a")

# len(): Retorna la longitud de la cadena
longitud_cadena = len(cadena_original)

# title(): Convierte la primera letra de cada palabra a mayúsculas
cadena_title = cadena_original.title()

# count(substring): Cuenta las ocurrencias de un substring en la cadena
conteo_a = cadena_original.count("a")

# upper() y lower(): Convierte la cadena a mayúsculas y minúsculas
respectivamente
cadena_upper = cadena_original.upper()
cadena_lower = cadena_original.lower()

# Impresión de algunos resultados
print("Original:", cadena_original)
# Salida: " Python es un lenguaje de programación. "

print("Capitalize:", cadena_capitalize)
# Salida: " python es un lenguaje de programación. "
```

```
print("Swapcase:", cadena_swapcase)
# Salida: " pYTHON ES UN LENGUAJE DE PROGRAMACIÓN. "

print("Replace:", cadena_replace)
# Salida: " Java es un lenguaje de programación. "

print("Split:", cadena_split)
# Salida: ['', '', '', 'Python', 'es', 'un',
# 'lenguaje', 'de', 'programación.', '', '', '']

print("Strip:", cadena_strip)
# Salida: "Python es un lenguaje de programación."
```

1.5.9. La función str en Python

La función `str()` en Python se utiliza para convertir un objeto dado en una cadena de texto (`string`). Esta función es especialmente útil en situaciones donde es necesario realizar operaciones de concatenación de texto o simplemente para mejorar la legibilidad de los datos de salida en la consola o interfaz de usuario.

La sintaxis básica de la función es la siguiente:

```
str(objeto)
```

donde `objeto` es el objeto que se desea convertir a una cadena de texto.

Ejemplo básico de uso:

```
numero = 123
texto_numero = str(numero)
print("El número convertido a texto es: " + texto_numero)
```

En este código, el número 123 es convertido en la cadena de texto "123" utilizando la función `str()`. Posteriormente, se concatena con otro texto para formar una cadena más larga que es impresa en la consola.

Uso con diferentes tipos de objetos:

```
mi_lista = [1, 2, 3]
texto_lista = str(mi_lista)
print("La lista convertida a texto es: " + texto_lista)
```

Aquí, la lista `[1, 2, 3]` se convierte en la cadena de texto "[1, 2, 3]" y posteriormente se concatena con otro `string` para imprimir el resultado en la consola.

Es importante mencionar que la función `str()` utiliza el método `str()` definido en el objeto para realizar la conversión. Si un objeto no tiene este método, Python intentará utilizar el método `repr()` como respaldo para obtener una representación en `string` del objeto.

La función `str()` es fundamental para la manipulación de texto en Python, permitiendo a los desarrolladores trabajar de manera más flexible con diferentes tipos de datos al convertirlos en su representación de texto.

Formato de Cadenas:

El formateo de cadenas permite insertar datos variables en una cadena de manera estructurada.

```
name = "World"  
greeting = f"Hello, {name}!"
```

Las cadenas con formato, o `f-strings`, introducidas en Python 3.6, proporcionan una forma expresiva y cómoda de incrustar expresiones dentro de cadenas literales, utilizando una sintaxis minimalista. La sintaxis básica de las `f-strings` es preceder la cadena con la letra `f` o `F` y utilizar llaves `{}` para encerrar las expresiones que serán evaluadas en tiempo de ejecución y formateadas utilizando el formato especificado.

Ejemplo Básico:

```
name = "World"  
greeting = f"Hello, {name}!"
```

En este caso, la variable `name` es referenciada directamente dentro de la cadena utilizando las llaves. Python evaluará la expresión dentro de las llaves y la insertará en la cadena en ese lugar.

Especificando la Precisión:

Las `f-strings` permiten especificar la precisión con la que se desea mostrar un número flotante, utilizando la sintaxis `{variable : .nf}`, donde `n` indica el número de decimales deseados.

```
pi = 3.141592653589793  
formatted_pi = f"Pi, rounded to three decimal places: {pi:.3f}"
```

Expresiones Complejas:

Las `f-strings` también pueden contener expresiones más complejas, incluyendo operaciones aritméticas y llamadas a métodos.

```
radius = 5  
area = f"Area of a circle with radius {radius} is: {3.14159 * (radius  
** 2):.2f}"
```

Alineación y Relleno:

Es posible especificar la alineación y el carácter de relleno de las expresiones dentro de una `f-string`. Por ejemplo, alinear a la derecha con un ancho de 10 caracteres y llenar con ceros.

```
number = 42  
padded_number = f"{number:>010}"
```

Las *f-strings* en Python proporcionan una forma poderosa y flexible de trabajar con cadenas y expresiones, facilitando la generación de salidas formateadas y la interpolación de variables y expresiones dentro de cadenas de texto.

1.5.10. Formateo de cadenas con el método `format()`

El método `format()` en Python permite un control extenso sobre el formateo y alineación de cadenas de caracteres, facilitando la inserción y maquetación de variables y expresiones dentro de una cadena de texto. Este método puede ser particularmente útil para situaciones donde se requiere estructurar el texto de manera precisa o para mejorar la legibilidad del código.

Sintaxis Básica:

```
template_string = "Hello, {}!"  
output_string = template_string.format("World")
```

En este caso, las llaves `{}` dentro de la cadena sirven como marcadores de posición para los argumentos proporcionados al método `format()`.

Formateo Posicional:

Es posible especificar el índice de los argumentos proporcionados para controlar su posición en la cadena de salida.

```
template_string = "From {1} to {0}"  
output_string = template_string.format("B", "A")
```

Aquí, los números dentro de las llaves indican la posición de los argumentos a insertar.

Formateo de Números:

El método `format()` también permite un control detallado sobre la representación de números, incluyendo especificación de decimales y formateo como porcentaje.

```
pi = 3.141592653589793  
formatted_string = "Value of Pi to 3 decimal places: {:.3f}".format(pi)
```

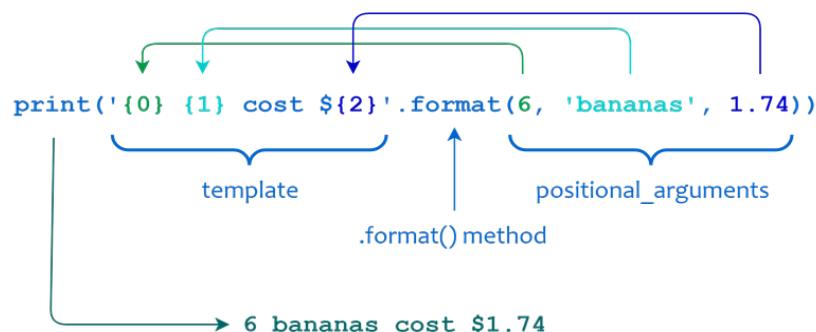


Figura 4: Ejemplo de uso de *format strings*

1.5.11. Símbolos utilizados en el método `format()`

El método `format()` en Python permite a los programadores especificar el formato de los valores de salida mediante el uso de varios símbolos y caracteres especiales, conocidos como especificadores de formato. Algunos de los más relevantes incluyen:

- `{}` Llaves utilizadas para definir campos de sustitución.
- `:` Dos puntos, introduce la especificación de formato.
- `.` Punto, especifica la precisión decimal en los formatos de punto flotante.
- `,` Coma, utilizada como separador de mil en números.
- `%` Porcentaje, convierte el número a una representación porcentual.
- `b` Binario, forma el número como binario.
- `o` Octal, forma el número como octal.
- `x/X` Hexadecimal, forma el número como hexadecimal.
- `e/E` Notación científica.
- `f/F` Punto flotante.
- `<, >, ^` Símbolos para alineación izquierda, derecha y centro respectivamente.
- `+` Fuerza la aparición del signo.
- `-` Solo signo negativo.
- Espacio, inserta un espacio en blanco antes de números positivos.
- `#` Agrega un prefijo que indica la base de números.

Por ejemplo:

```
number = 1234567.890123456
formatted_number = "{:,}.".format(number) # 1,234,567.890123456

print("binary value of 255 is {0:b}.".format(255))
>>>binary value of 255 is 11111111
```

Es pertinente anotar que ciertos especificadores son exclusivos para ciertos tipos de formato y pueden ser combinados para ajustarse a las necesidades específicas del programador. La documentación oficial de Python proporciona una guía exhaustiva sobre estos símbolos y su utilización en diferentes contextos.

Alineación y Relleno:

Es posible controlar la alineación y caracteres de relleno de los valores insertados.

```
aligned_string = "|{:<10}|{:^10}|{:>10}|".format("Left", "Center",
"Right")
```

En el ejemplo, los caracteres dentro de las llaves son utilizados para especificar la alineación de las cadenas a la izquierda, centro, y derecha respectivamente, mientras que el número ‘10’ indica el ancho del campo.

A través del adecuado uso del método `format()`, los desarrolladores pueden crear cadenas de salida estructuradas y legibles, asegurando que la presentación de los datos sea clara y coherente.

1.5.12. La función `print` en Python

La función `print()` es una de las funciones más utilizadas en Python y sirve para imprimir datos en la salida estándar, comúnmente la pantalla. Acepta varios parámetros que permiten configurar la impresión de los datos.

- **sep:** Este parámetro determina el separador entre los elementos impresos y por defecto es un espacio en blanco.
- **end:** Define qué se imprimirá al final de la llamada a la función. Por defecto es un salto de línea.
- **file:** Permite definir un flujo de salida diferente al estándar.
- **flush:** Un booleano que indica si se debe forzar la escritura de la salida (por defecto es False).

Ejemplo básico de uso:

```
print("Hola", "Mundo", sep="-", end="!")
```

En este caso, se imprime “Hola-Mundo!” ya que hemos configurado el separador como ‘-’ y el final de la impresión como ‘!’.

1.6. LAS FECHAS EN PYTHON

Python proporciona la biblioteca `datetime` para trabajar con fechas y tiempos. Esta biblioteca incluye varios tipos de datos para manejar tanto fechas como combinaciones de fechas y tiempos.

A continuación, se ofrece una breve introducción sobre cómo usar algunos de los componentes principales de la biblioteca `datetime`:

```
import datetime

# Crear un objeto date (fecha)
fecha = datetime.date(2023, 10, 23)
print(fecha) # Salida: 2023-10-23

# Obtener la fecha actual
hoy = datetime.date.today()
print(hoy)

# Crear un objeto datetime (fecha y hora)
fecha_hora = datetime.datetime(2023, 10, 23, 14, 30)
print(fecha_hora) # Salida: 2023-10-23 14:30:00

# Sumar y restar fechas con timedelta
dias = datetime.timedelta(days=5)
nueva_fecha = hoy + dias
print(nueva_fecha)

# Formatear fechas
```

```
fecha_str = hoy.strftime("%d-%m-%Y")
print(fecha_str) # Salida: 23-10-2023

# Convertir cadena a fecha
fecha_obj = datetime.datetime.strptime("23-10-2023", "%d-%m-%Y")
print(fecha_obj.date())
```

Los puntos clave de la biblioteca `datetime` incluyen:

- `date`: Representa una fecha (año, mes, día).
- `datetime`: Representa una combinación de fecha y hora.
- `timedelta`: Representa una duración, la diferencia entre dos fechas o tiempos.
- `strftime`: Método para formatear objetos de fecha/hora como cadenas.
- `strptime`: Método para convertir cadenas en objetos de fecha/hora según un formato especificado.

1.6.1. Utilizando la función `import` en Python

En Python, la función `import` es una declaración que permite incluir módulos y paquetes, brindando acceso a recursos, funciones y clases adicionales que no están disponibles en el espacio de nombres principal del script. La utilización de `import` no sólo facilita la organización del código al permitir la segregación de funciones en diferentes módulos, sino que también enriquece los programas al proporcionar acceso a una amplia gama de módulos tanto integrados en Python como de terceros.

Sintaxis básica:

```
import modulo
```

Donde `modulo` es el nombre del módulo que se desea importar.

Ejemplo básico de uso:

```
import math

# Uso de una función del módulo math
resultado = math.sqrt(25)
print(resultado) # Salida: 5.0
```

En este ejemplo, el módulo `math` es importado, permitiendo el uso de sus funciones, en este caso, `sqrt()`, para calcular la raíz cuadrada de un número.

La función `import` también se puede utilizar con las declaraciones `from ... import ...`, permitiendo importar específicamente una función o clase de un módulo y reduciendo la necesidad de utilizar el nombre del módulo como prefijo al acceder a sus funciones.

Ejemplo de uso de `from ... import ...`:

```
from math import sqrt
```

```
# Uso directo de la función sqrt
resultado = sqrt(25)
print(resultado) # Salida: 5.0
```

Renombrando importaciones:

Se puede renombrar un módulo o función importada usando la palabra clave *as* para mejorar la legibilidad del código o evitar conflictos de nombres.

```
import math as m

resultado = m.sqrt(25)
print(resultado) # Salida: 5.0
```

1.6.2. La función *input* en Python

La función *input()* es una de las funciones integradas en Python utilizada para obtener entrada del usuario en forma de cadena de texto (*string*). Esta función puede aceptar un argumento opcional, el cual es la cadena que se desea imprimir en pantalla como mensaje o indicación para el usuario, pero es comúnmente usado para proporcionar un mensaje de indicación o *prompt*.

Sintaxis básica de la función:

```
input([prompt])
```

donde *[prompt]* es el mensaje que se desea mostrar. Es importante mencionar que la respuesta del usuario siempre se trata como una cadena de texto, incluso si el usuario escribe un número.

Ejemplo básico de uso:

```
nombre = input("Por favor, introduce tu nombre: ")
print("Hola,", nombre)
```

En este ejemplo, se le pide al usuario que introduzca su nombre. La función *input()* espera a que el usuario escriba su entrada y la retorna como una cadena de texto, que en este caso se almacena en la variable *nombre*. Posteriormente, se utiliza la función *print()* para saludar al usuario utilizando el nombre ingresado.

Cuando se necesita convertir la entrada del usuario a un tipo de dato diferente (por ejemplo, a un número entero o flotante), se puede utilizar funciones de conversión de tipos como *int()* o *float()*.

Ejemplo de conversión de tipo de dato:

```
edad = int(input("Por favor, introduce tu edad: "))
```

En este caso, se le pide al usuario su edad y se convierte la cadena de texto proporcionada por el usuario a un número entero antes de asignarlo a la variable edad. Si el usuario ingresa un valor que no se puede convertir a un número entero, Python generará una excepción de tipo `ValueError`.

1.6.3. Comentarios en Python

Los comentarios en Python son fragmentos de texto que no son ejecutados como parte del código. Son esenciales para explicar y aclarar el código para que los desarrolladores que lo lean puedan entender fácilmente su funcionalidad y propósito.

En Python, los comentarios se crean utilizando el símbolo `#` para los comentarios de una sola línea y las comillas triples `''' '''` o para los comentarios de múltiples líneas.

- **Comentarios de una sola línea:** Se utilizan para hacer anotaciones cortas y concisas.
- **Comentarios de múltiples líneas:** Se utilizan para proporcionar descripciones más detalladas o para bloquear temporalmente bloques de código durante la depuración y el desarrollo.

A continuación se presentan ejemplos:

```
# Este es un comentario de una sola linea en Python

'''

Este es un comentario
de múltiples líneas en Python
'''


"""

También este es un comentario de múltiples líneas en Python """

def funcion_ejemplo():
    # Este comentario describe la siguiente linea de código
    print("¡Hola, mundo!") # Este comentario está al final de la linea
```

Es crucial usar comentarios para mejorar la legibilidad del código, explicando la funcionalidad de segmentos de código complejos o describiendo la lógica de negocio subyacente. Los comentarios también son útiles para explicar las decisiones de codificación y cualquier limitación conocida del código. Al escribir comentarios, es importante ser claro y conciso, proporcionando información útil sin ser demasiado exhaustivo (verborrágico).

1.6.4. Utilización de Docstrings en IDEs de Python

La documentación es un pilar fundamental en el desarrollo de software, facilitando la comprensión y utilización de código por parte de otros desarrolladores. En Python, las docstrings (cadenas de documentación) representan una forma estándar y accesible de documentar el código. Una docstring es una cadena de texto que se encuentra en la primera línea de una función, método, clase o módulo y se utiliza para describir su propósito y comportamiento.

```
def ejemplo_funcion():
    """
    Esta es una docstring que describe la función ejemplo_funcion.
```

```
"""
pass
```

Los IDEs (Entornos de Desarrollo Integrado) de Python capitalizan activamente las docstrings. Al integrarlas, los IDEs permiten a los desarrolladores acceder a la documentación directamente desde el código fuente, facilitando así la comprensión y la utilización del código de una manera eficiente y efectiva.

Funcionalidades proporcionadas por los IDEs a través de docstrings:

- **Autocompletado Inteligente:** Los IDEs proponen automáticamente nombres de funciones, métodos y variables al escribir código, utilizando docstrings para mostrar información relevante acerca de los objetos sugeridos.
- **Ayuda Contextual:** Al colocar el cursor sobre una función o método, algunos IDEs muestran la docstring correspondiente en una ventana emergente, proporcionando información inmediata sobre su uso y requerimientos.
- **Generación Automática de Documentación:** Herramientas como Sphinx pueden utilizar docstrings para generar automáticamente la documentación del código en formatos accesibles como HTML o PDF.
- **Control de Calidad del Código:** Algunos IDEs y herramientas de control de calidad del código verifican la existencia y calidad de las docstrings, promoviendo las mejores prácticas en la documentación del código.

La utilización de docstrings no solo mejora la legibilidad del código, sino que también enriquece la experiencia de desarrollo al proporcionar acceso instantáneo a la documentación, directamente dentro del IDE. Es una práctica recomendada documentar sistemáticamente funciones, métodos, clases y módulos utilizando docstrings para aprovechar estas capacidades integradas y mejorar la sostenibilidad y comprensión del código a lo largo del tiempo.

1.6.5. Dividiendo una instrucción en varias líneas en Python

Python permite dividir una instrucción en varias líneas de código para mejorar la legibilidad y organización del mismo. Esto puede ser especialmente útil cuando se trabaja con expresiones o sentencias muy largas. Existen varias maneras de realizar esta división.

Usando el Carácter de Barra Invertida (*textbackslash*). En Python, podemos utilizar el carácter de barra invertida (\) para indicar que una línea continuá en la siguiente. Veamos un ejemplo con una expresión aritmética larga:

```
longExpressionResult = 3 + 4 + 5 - 6 + 7 + 8 - 9 + \
                      10 - 11 + 12 + 13 + 14 - 15
```

Usando Paréntesis, Corchetes o Llaves. Cuando una expresión está entre paréntesis (), corchetes [] o llaves {}, la expresión puede continuar en la siguiente línea sin necesidad de usar la barra:

```
longSumResult = (
    3+4+5-6+7+8-9+
    10 - 11 + 12 + 13 + 14 - 15
)
```

```
longList = [  
    1, 2, 3,  
    4, 5, 6,  
    7, 8, 9  
]  
  
longDictionary = {  
    "keyOne": 1,  
    "keyTwo": 2,  
    "keyThree": 3,  
    "keyFour": 4  
}
```

En ambos ejemplos, la expresión o declaración es legible y clara, y Python entiende que la instrucción se extiende a través de varias líneas debido al uso de paréntesis y corchetes/llaves, respectivamente.

2. SENTENCIAS DE CONTROL EN PYTHON

Las sentencias de control, también conocidas como estructuras de control, se utilizan en la programación para manejar el flujo de ejecución del código. Permiten que el código tome decisiones y repita bloques de código bajo ciertas condiciones. Hay dos tipos principales de estructuras de control: las estructuras de control de flujo condicional y las estructuras de control de flujo iterativo (bucles).

2.1. EXPRESIONES BOOLEANAS EN PYTHON

Las expresiones booleanas en Python se utilizan para realizar operaciones que involucran valores verdaderos o falsos (*True* o *False*). Python ofrece operadores booleanos típicos como AND, OR y NOT que permiten realizar operaciones lógicas.

```
# Ejemplo de expresiones booleanas en Python  
x = True  
y = False  
  
# Operadores AND , OR y NOT  
print(x and y) # Salida: False  
print(x or y) # Salida: True  
print(not x) # Salida: False
```

2.1.1. Operador *is* en Python

El operador *is* se utiliza para comparar si dos variables apuntan al mismo objeto, mientras que el *is not* se utiliza para verificar si dos variables no apuntan al mismo objeto. Es importante mencionar que *is* no es lo mismo que *==*, ya que este último compara los valores de las variables y no sus identidades.

```
# Ejemplo de uso del operador 'is'
```

```
x = [1, 2, 3]
y = [1, 2, 3]

# 'is' compara identidades
print(x is y) # Salida: False

# '==' compara valores
print(x == y) # Salida: True

# Ejemplo de uso de 'is' con 'type'
print(type(x) is list) # Salida: True

# Ejemplo de uso de 'is not'
z=x
print(z is not y) # Salida: True
```

2.2. CONDICIONALES EN PYTHON

Las sentencias condicionales en Python nos permiten ejecutar bloques específicos de código basados en una condición dada. La sintaxis básica para una sentencia condicional en Python es la siguiente:

```
if condicion:
    # Código a ejecutar si la condición es verdadera
```

La sintaxis para una sentencia condicional doble en Python es la siguiente:

```
if condicion:
    # Código a ejecutar si la condición es verdadera
else
    # Código a ejecutar si la condición es falsa
```

La sintaxis para una sentencia condicional múltiple en Python es la siguiente:

```
if condicion1:
    # Código a ejecutar si condición1 es verdadera
elif condicion2:
    # Código a ejecutar si la condición2 es verdadera
elif ...
    # Código a ejecutar si la condiciónN es verdadera
else
    # Código a ejecutar si las anteriores condiciones fueron falsas
```

En este fragmento, *condicion* es una expresión que se evalúa como *True* o *False*. Si *condicion* es *True*, entonces el bloque de código indentado debajo de la sentencia *if* se ejecutará. A continuación, un ejemplo práctico:

```
studentGrade = 85

if studentGrade >= 90:
```

```
letterGrade = "A"
elif studentGrade >= 80:
    letterGrade = "B"
elif studentGrade >= 70:
    letterGrade = "C"
elif studentGrade >= 60:
    letterGrade = "D"
else:
    letterGrade = "F"

print("The letter grade is: " + letterGrade)
```

Explicación:

La variable `studentGrade` es verificada contra varias condiciones utilizando la estructura `if/elif/else`. Dependiendo de la primera condición que sea `True`, el bloque de código asociado se ejecutará, asignando un valor correspondiente a `letterGrade`. Si `studentGrade` es 85, entonces `letterGrade` se asignará como "B" y la salida será: "The letter grade is: B".

2.3. SENTENCIAS ITERATIVAS EN PYTHON

Las sentencias iterativas, también conocidas como bucles o ciclos, se utilizan en programación para repetir un bloque de código múltiples veces. Python proporciona dos estructuras de bucle principales: el bucle `for` y el bucle `while`.

Bucle `for`:

La sintaxis del bucle `for` es

```
for variable in iterable:
    # Cuerpo del bucle
    # ...
```

Aquí, `variable` es la variable de iteración que toma el valor de cada elemento en `iterable` en cada iteración del bucle.

Bucle `while`:

La sintaxis del bucle `while` es

```
while condicion:
    # Cuerpo del bucle
    # ...
```

Bucle `while` con `else`:

Python también permite el uso de una cláusula `else` con el bucle `while`, que se ejecuta cuando la condición del `while` se vuelve `false`.

```

while condicion:
    # Cuerpo del bucle
    #
    # ...
else:
    # Código a ejecutar cuando
    # la condición del while es False
    #
    # ...

```

El bloque de código dentro de la cláusula `else` se ejecutará cuando el bucle `while` finalice, es decir, cuando `condicion` se evalúe como `False`. Nota que si el bucle termina con un `break`, el bloque `else` no se ejecutará.

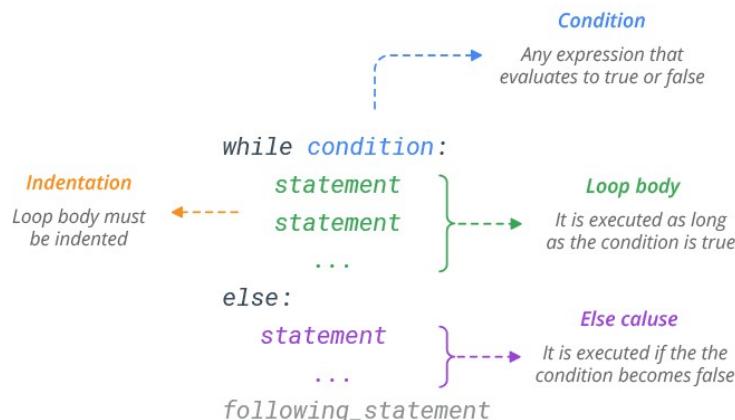


Figura 5: Sintaxis del bucle `while`

Control de Bucles:

Python también proporciona declaraciones de control de bucles, `break` y `continue`, que permiten una gestión más fina del comportamiento de los bucles.

```

for variable in iterable:
    if condicion_para_salir:
        break
    elif condicion_para_continuar:
        continue
    # Cuerpo del bucle
    #
    # ...

```

La instrucción `break` permite salir anticipadamente del bucle, mientras que `continue` salta al siguiente ciclo del bucle, omitiendo el resto del código que sigue en la iteración actual.

2.3.1. Ejemplos de Bucles

El bucle `while` repite un bloque de código mientras una condición dada sea verdadera

```

# Ejemplo de bucle while
count = 0

```

```

while count < 5:
    print("Iteración número:", count)
    count += 1
# EJEMPLO: sacar las tablas de multiplicar del 1 al 10
# Bucle para iterar a través de los números del 1 al 10 (inclusive)
# para generar las tablas de multiplicar
for i in range(1, 11):
    print(f"\nTabla del {i}:")

    # Bucle anidado para calcular e imprimir
    # cada línea de la tabla de multiplicar
    for j in range(1, 11):
        print(f"{i} x {j} = {i*j}")

```

En este ejemplo, el bucle `while` seguirá ejecutándose mientras la variable `count` sea menor que 5, imprimiendo el número de iteración en cada paso y aumentando el valor de `count` por 1 en cada iteración.

2.3.2. Bucle `for`

El bucle `for` es utilizado para iterar sobre una secuencia iterable:

```

# Ejemplo de bucle for con range
for number in range(5):
    print("Número: ", number)

```

Aquí, la función `range(5)` genera una secuencia de números desde 0 hasta 4, y el bucle `for` imprime cada número en su respectiva iteración.

2.3.3. Bucles Anidados

Los bucles también pueden estar anidados dentro de otros bucles.

```

# Ejemplo de bucles anidados
for i in range(3):
    for j in range(2):
        print("i:", i, ", j:", j)

```

En este ejemplo, para cada valor de `i` en el bucle externo, el bucle interno se ejecutará completamente, generando todas las posibles combinaciones de `i` y `j` y imprimiéndolas.

```

# Ejemplo de combinación de for y while
for i in range(3):
    print("i:", i)
    j=0
    while j < 2:
        print(" j:", j)
        j += 1

```

En este ejemplo, para cada iteración del bucle `for`, el bucle `while` se ejecuta mientras `j` sea menor que `2`, imprimiendo los valores de `i` y `j`.

2.3.4. La función `range`

La función `range` en Python es una función incorporada que se utiliza para generar una secuencia de números, la cual es especialmente útil para iterar sobre un bloque de código un número específico de veces mediante un bucle `for`. La función `range` puede aceptar entre uno y tres argumentos numéricos: `start`, `stop`, y `step`.

- **start:** Especifica desde qué número comenzar la secuencia. Si se omite, la secuencia comenzará desde `0`.
- **stop:** Especifica hasta qué número generar la secuencia. Este número no está incluido en la salida.
- **step:** Especifica la diferencia entre cada número en la secuencia. Si se omite, la diferencia será `1`.

Si se proporciona un solo argumento a `range`, se utilizará como el valor de `stop`, y la secuencia generada será desde `0` hasta `stop-1`.

```
# Uso básico de range con un argumento
for i in range(5):
    print(i) # Salida: 0 1 2 3 4
```

Cuando se proporcionan dos argumentos, se utilizan como los valores de `start` y `stop` respectivamente, generando números desde `start` hasta `stop-1`

```
# Uso de range con dos argumentos
for i in range(2, 5):
    print(i) # Salida: 2 3 4
```

Cuando se utilizan tres argumentos, se interpretan como los valores de `start`, `stop`, y `step`, respectivamente. La secuencia generada comenzará en `start`, incrementará en `step`, y terminará justo antes de llegar a `stop`.

```
# Uso de range con tres argumentos
for i in range(1, 10, 2):
    print(i) # Salida: 1 3 5 7 9
```

3. ESTRUCTURAS DE DATOS EN PYTHON

“Una estructura de datos es un conjunto de objetos, con operaciones definidas sobre ellos, un comportamiento especificado para estas operaciones, y relaciones definidas entre los objetos y las operaciones”.

— Niclaus Wirth

Python incluye varias estructuras de datos incorporadas: listas, diccionarios, tuplas y conjuntos, las cuales permiten crear, almacenar y manipular datos de manera eficiente y fácil.

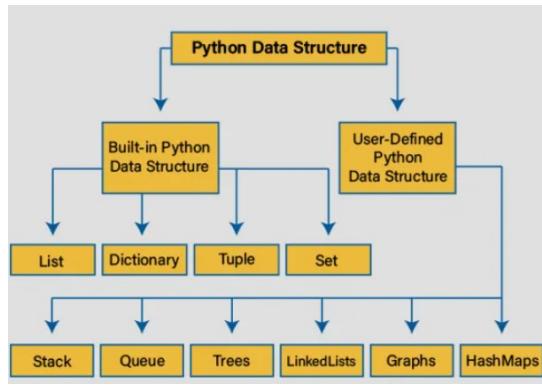


Figura 6: Estructuras de datos en Python

3.1. LISTAS

Las listas en Python son colecciones ordenadas de elementos que pueden ser de cualquier tipo y son mutables, es decir, los elementos pueden ser cambiados después de que la lista ha sido creada.

```

myList = [1, 2, 3, 4, 5]
myList[1] = 10 # Modificar un elemento
myList.append(6) # Añadir un elemento
    
```

Las listas en Python son una de las estructuras de datos más versátiles y utilizadas. Consisten en una colección ordenada de elementos, los cuales pueden ser de cualquier tipo, como enteros, cadenas, y hasta otras listas. Las listas son mutables, lo que significa que los elementos pueden ser modificados incluso después de que la lista ha sido creada.

Creación y acceso a las listas:

Para crear una lista, se utilizan corchetes `[]` y se separan los elementos con comas. Para acceder a los elementos de la lista, se utilizan índices, que comienzan desde 0.

```

myList = [1, 2, 3, 4, 5] # Crear una lista
firstElement = myList[0] # Acceder al primer elemento
    
```

Métodos importantes en listas:

A continuación, se presentan algunos métodos útiles que están disponibles para las listas en Python.

- `append(element)`: Añade un elemento al final de la lista

```

myList.append(6) # [1, 2, 3, 4, 5, 6]
    
```

- `insert(index, element)`: Inserta un elemento en la posición especificada

```

myList.insert(0, 0) # [0, 1, 2, 3, 4, 5, 6]
    
```

- `remove(element)`: Elimina la primera ocurrencia del elemento especificado

```
myList.remove(0) # [1, 2, 3, 4, 5, 6]
```

- `pop(index)`: Elimina y devuelve el elemento en la posición especificada

```
myList.pop(0) # [2, 3, 4, 5, 6]
```

- `reverse()`: Invierte el orden de los elementos en la lista

```
myList.reverse() # [6, 5, 4, 3, 2]
```

- `sort()`: Ordena los elementos de la lista

```
myList.sort() # [2, 3, 4, 5, 6]
```

- `count(element)`: Cuenta las apariciones de un elemento en la lista

```
#¿Cuántas veces aparece cada elemento en la lista?  
numbers = [0, 1, 1, 2, 2, 2, 3, 3, 3]  
  
counted = []  
for element in numbers:  
    if element not in counted:  
        counted.append(element)  
        print("El elemento {} aparece {} veces".format(element,  
numbers.count(element)))
```

3.1.1. Listas con elementos de diferentes tipos de datos

En Python, las listas pueden contener elementos de diferentes tipos de datos, incluyendo números, cadenas y objetos.

```
mixedList = [1, "two", 3.0, ['another', 'list'], {'key': 'value'}]
```

En el ejemplo anterior, `mixedList` contiene un entero, una cadena, un número de punto flotante, otra lista y un diccionario, demostrando la flexibilidad y la naturaleza dinámica de las listas en Python.

3.1.2. Listas anidadas y estructuras de datos complejas

Las listas pueden contener otras listas como elementos, permitiendo la creación de estructuras de datos más complejas, como matrices bidimensionales.

```
nestedList = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`nestedList` se comporta como una matriz 3x3 en la que se accede a los elementos mediante dos índices: el primero para la sublista y el segundo para el elemento dentro de esa sublista.

3.1.3. Comprepción de listas

En Python, es posible utilizar estructuras de control de flujo `for` e `if` en una única línea, a menudo a través de técnicas conocidas como “comprepción de listas” (list comprehension), “comprepción de conjuntos” (set comprehension) o “comprepción de diccionarios” (dictionary comprehension).

Utilizando una sola línea de código, es posible generar listas en Python a través de operaciones y filtros aplicados directamente en la definición de la lista.

```
squaresOfEvens = [x**2 for x in range(10) if x % 2 == 0]
print(squaresOfEvens) #SALIDA: [0, 4, 16, 36, 64]

words = ["apple", "banana", "cherry", "kiwi"]
longWords = [word for word in words if len(word) > 3]
print(longWords) #Salida: ['apple', 'banana', 'cherry', 'kiwi']
```

3.1.4. Inicialización de matrices con bucles `for`

Se pueden utilizar bucles `for` para inicializar matrices y trabajar con listas anidadas de una manera más dinámica.

```
matrix = [[1 if col==row else 0 for col in range(3)] for row in range(3)]
print(matrix) #salida [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

matrix = [[col+(row+1)*10 for col in range(3)] for row in range(3)]
print(matrix) #salida [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
```

3.1.5. Realizando Slicing en Listas

El slicing permite acceder a subconjuntos de una lista utilizando índices positivos y negativos. El índice positivo en el slicing empieza desde el principio de la lista (0), mientras que el índice negativo empieza desde el final de la lista (-1). También puedes utilizar un tercer índice como paso para el slicing.

```
myList=[10, 20, 30, 40, 50, 60, 70, 80, 90]

# Slicing usando índices positivos
subList1 = myList[1:4] # Output: [20, 30, 40]
# Slicing usando índices negativos
subList2 = myList[-4:-1] # Output: [60, 70, 80]

# Slicing con pasos
subList3 = myList[::-2] # Output: [10, 30, 50, 70, 90]
```

3.1.6. Concatenación de listas en Python

La concatenación de listas en Python es una operación que une dos o más listas en una nueva lista. Esto se puede lograr utilizando el operador de concatenación `+`, que permite unir dos listas de

manera secuencial, creando una nueva lista como resultado. Es importante notar que la concatenación de listas genera una nueva lista, y las listas originales no se modifican.

```
# Ejemplo básico de concatenación de listas
listA = [1, 2, 3]
listB = [4, 5, 6]
concatenatedList = listA + listB # [1, 2, 3, 4, 5, 6]
```

La concatenación no se limita a dos listas, pueden concatenarse tantas listas como se deseen en una sola operación, simplemente uniendo las listas con el operador `+`.

```
# Ejemplo de concatenación de múltiples listas
listC = [7, 8, 9]
concatenatedList = listA + listB + listC #[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Una alternativa para agregar elementos de una lista a otra sin crear una nueva lista es utilizando el método `extend()`. Este método agrega los elementos de la lista proporcionada como argumento al final de la lista que invoca el método, modificando esta última.

```
# Ejemplo de uso de extend ()
listA.extend(listB)      # listA ahora es [1, 2, 3, 4, 5, 6]
```

3.2. DICCIONARIOS

Los diccionarios en Python son una estructura de datos mutable que almacena pares de clave-valor. A diferencia de las listas, los datos en los diccionarios son accesibles mediante claves únicas y no mediante índices. Un diccionario se define utilizando llaves {}, y los pares de clave-valor están separados por comas.

```
myDict = {"apple": 1, "banana": 2, "cherry": 3}
myDict["banana"] = 10      # Modificar un elemento
myDict["date"] = 4        # Añadir un elemento
```

Creación y acceso a los diccionarios

Un ejemplo simple de creación de un diccionario y acceso a sus elementos es:

```
myDict = {'name': 'John', 'age': 30, 'city': 'New York'}
print(myDict['name'])      # Output: John
```

Se puede utilizar el método `dict` para inicializar los datos de un diccionario:

```
dicc1 = dict(x = 0, y = 1, z = -1)
print(dicc1)

dicc2 = dict({"x": 0, "y": 1, "z": -1})
print(dicc2)

dicc3 = dict({"x": 0}, y = 1, z = -1)
print(dicc3)
```

Métodos importantes en diccionarios

Los diccionarios en Python proporcionan varios métodos útiles para trabajar con ellos. A continuación, se presentan algunos de estos métodos:

- *keys ()*: Devuelve una vista de todas las claves en el diccionario.

```
print(myDict.keys()) # Output: dict_keys(['name', 'age', 'city'])
```

- *values ()*: Devuelve una vista de todos los valores en el diccionario

```
print(myDict.values()) # Output: dict_values(['John', 30, 'New York'])
```

- *items ()*: Devuelve una vista de todos los pares clave-valor en el diccionario

```
print(myDict.items())
# Output: dict_items([('name', 'John'), ('age', 30), ('city', 'New
York')))
```

- *get(key[, default])*: Devuelve el valor para key si key está en el diccionario, si no, devuelve default

```
print(myDict.get('name')) # Output: John
print(myDict.get('country', 'USA')) # Output: USA
```

- *update([other])*: Actualiza el diccionario con los pares clave-valor de other, sobrescribiendo las claves existentes

```
myDict.update({'name': 'Jane', 'country': 'USA'})
```

- *clear()*: Elimina todos los elementos del diccionario

```
myDict.clear()
```

3.2.1. Recorrido de un diccionario

Es común necesitar iterar o “recorrer” los diccionarios para acceder a sus claves y valores. Python proporciona varios métodos que facilitan la iteración a través de los diccionarios, permitiendo acceder tanto a las claves como a los valores durante la iteración.

```
# Ejemplo de un diccionario simple
myDict = {'one': 1, 'two': 2, 'three': 3}
```

Iteración a través de las Claves

Por defecto, cuando se itera a través de un diccionario usando un bucle *for*, se accede a sus claves

```
# Iteración a través de las claves del diccionario
for key in myDict:
```

```
print(key)
# Imprime: one, two, three
```

Iteración a través de los Valores

Para iterar directamente a través de los valores del diccionario, se utiliza el método `values()`.

```
# Iteración a través de los valores del diccionario
for value in myDict.values():
    print(value)
# Imprime: 1, 2, 3
```

Iteración a través de Claves y Valores simultáneamente

El método `items()` permite iterar a través de las claves y los valores del diccionario simultáneamente

```
# Iteración a través de claves y valores del diccionario
for key, value in myDict.items():
    print(key, value)
# Imprime: one 1, two 2, three 3
```

Estos diferentes enfoques de iteración proporcionan una gran flexibilidad y facilitan el manejo de los diccionarios en Python, permitiendo seleccionar el método más apropiado según las necesidades específicas del caso en el que se esté trabajando.

3.2.2. Métodos `copy`, `fromkeys` y `get` en Diccionarios Python

Los diccionarios en Python ofrecen una variedad de métodos que facilitan su manipulación y acceso a los datos. A continuación se exploran los métodos `copy`, `fromkeys` y `get`, proporcionando ejemplos de su uso y prácticas recomendadas.

Método `copy`

El método `copy()` se utiliza para crear una copia superficial del diccionario. Es útil cuando se quiere trabajar con un conjunto de datos sin modificar el diccionario original

```
originalDict = {'a': 1, 'b': 2, 'c': 3}
copiedDict = originalDict.copy()

# Resultado: copiedDict es {'a': 1, 'b': 2, 'c': 3}
```

Es vital mencionar que las copias superficiales realizadas con `copy()` no crean copias independientes de los objetos mutables que se almacenan en el diccionario, como listas o diccionarios anidados.

Método `fromkeys`

El método `fromkeys()` es un método de clase que permite crear un nuevo diccionario con claves especificadas y un valor común (o `None` si no se especifica el valor).

```
keys = ['a', 'b', 'c']
default_value = 0
newDict = dict.fromkeys(keys, default_value)

# Resultado: newDict es {'a': 0, 'b': 0, 'c': 0}
```

Este método es particularmente útil cuando se quiere inicializar un diccionario para realizar conteos o acumulaciones sobre un conjunto conocido de claves.

Método `get`

El método `get()` se utiliza para recuperar el valor de una clave específica del diccionario sin lanzar un error si la clave no existe. Adicionalmente, permite especificar un valor por defecto que será retornado en caso de que la clave no esté presente.

```
myDict = {'a': 1, 'b': 2, 'c': 3}
valueA = myDict.get('a', 0)      # Retorna 1
valueD = myDict.get('d', 0) # Retorna 0, ya que 'd' no existe

# Sin usar un valor por defecto
valueD = myDict.get('d') # Retorna None
```

3.2.3. Relación entre diccionarios en Python y el formato Json

La relación entre los diccionarios de Python y el formato JSON (JavaScript Object Notation) es estrecha y natural debido a su similitud sintáctica y estructural. JSON es un formato de intercambio de datos que es fácilmente legible por humanos y se utiliza comúnmente para la comunicación entre servidores y aplicaciones web. Los diccionarios de Python pueden ser fácilmente convertidos a una cadena en formato JSON y viceversa, facilitando la manipulación, almacenamiento y transmisión de datos estructurados.

Conversión de un Diccionario a JSON

Para convertir un diccionario de Python en una cadena JSON, se utiliza la función `dumps()` del módulo `json`.

```
import json

myDict = {'name': 'John', 'age': 30, 'city': 'New York'}
myJson = json.dumps(myDict)

# Resultado: '{"name": "John", "age": 30, "city": "New York"}'
```

Conversión de una Cadena JSON a Diccionario

Para convertir una cadena en formato JSON a un diccionario de Python, se utiliza la función `loads()` del módulo `json`

```
import json

myJson = '{"name": "John", "age": 30, "city": "New York"}'
myDict = json.loads(myJson)

# Resultado: {'name': 'John', 'age': 30, 'city': 'New York'}
```

También se puede cargar, fácilmente, un archivo JSON en un diccionario. Suponga que tenemos un fichero llamado `data.json` con el siguiente contenido

```
{
    "name": "Jane",
    "age": 28,
    "city": "London"
}
```

Este JSON puede ser cargado en un diccionario Python utilizando el método `load()` del módulo `json`, de la siguiente manera

```
import json

# Cargar JSON desde un fichero with open('data.json', 'r') as file:

data = json.load(file)

# Resultado: {'name': 'Jane', 'age': 28, 'city': 'London'}
```

Aquí, `open()` es usado para abrir el fichero `data.json` en modo lectura ('`r`'). El contexto `with` se utiliza para asegurar que el fichero se cierre automáticamente después de que los datos sean leídos. El método `json.load(file)` lee los datos del fichero y los convierte automáticamente en un diccionario Python.

Estas conversiones facilitan la manipulación de datos entre Python y otras tecnologías web, ya que JSON es ampliamente reconocido y utilizado en diversas plataformas y lenguajes de programación. Además, JSON puede representar estructuras de datos más complejas, incluyendo listas y anidamientos, lo que lo hace compatible con una amplia variedad de aplicaciones.

Consideraciones Importantes

Es crucial tener en cuenta que aunque la sintaxis de los diccionarios de Python y el formato JSON son muy similares, existen algunas diferencias, como la representación de las cadenas (en JSON siempre se utilizan comillas dobles) y los tipos de datos permitidos. Asegurarse de manejar estos detalles es vital para garantizar la correcta serialización y deserialización de los datos.

3.3. TUPLAS

Las tuplas en Python son similares a las listas, pero son inmutables, es decir, una vez que una tupla es creada, sus elementos no pueden ser cambiados.

Para definir una tupla, se utilizan paréntesis () y los elementos de la tupla se separan por comas.

A diferencia de las listas, las tuplas son inmutables, lo que significa que una vez que una tupla ha sido creada, no es posible modificar sus elementos ni su tamaño.

```
myTuple = (1, 2, 3, 4, 5)  
# myTuple[1] = 10      # Esto generará un error
```

3.3.1. Accediendo a elementos de la Tupla

Para acceder a los elementos de una tupla, se utiliza el índice del elemento deseado entre corchetes [].

```
firstElement = myTuple[0]      # Salida: 1
```

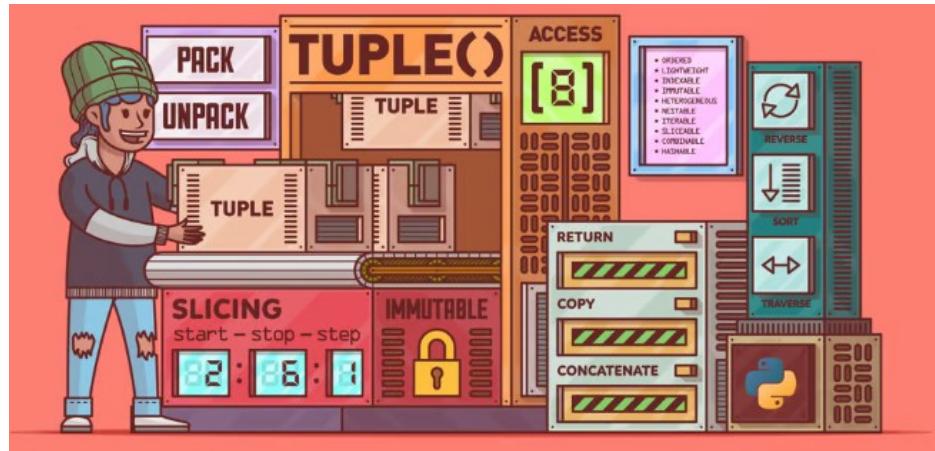


Figura 7: Las tuplas en Python

3.3.2. Empaquetado y desempaquetado de tuplas

En Python, el empaquetado de tuplas se refiere a la creación de una tupla, mientras que el desempaquetado de tuplas es la operación inversa, es decir, extraer los valores de la tupla en variables.

```
# Empaquetado de tuplas  
myTuple = (1, 2, 3)  
  
# Desempaquetado de tuplas  
a, b, c = myTuple
```

3.3.3. Métodos de las tuplas

A pesar de que las tuplas son inmutables, existen algunos métodos disponibles para trabajar con ellas:

- `count(x)`: Retorna el número de veces que x aparece en la tupla.
- `index(x)`: Retorna el índice de la primera aparición de x en la tupla.

```
myTuple = (1, 2, 3, 3, 4, 5)

# Uso de count
numThrees = myTuple.count(3) # Salida: 2

# Uso de index
indexThree = myTuple.index(3) # Salida: 2
```

Aunque las tuplas ofrecen menos funcionalidades que las listas, su inmutabilidad las hace útiles en situaciones donde se desea garantizar que los datos no serán alterados, como por ejemplo, para usarlas como claves en diccionarios.

3.3.4. Packing y unpacking en tuplas: Inicialización, salida y bucles

El manejo de tuplas en Python permite técnicas eficientes de empaquetado y desempaquetado de valores, facilitando la asignación múltiple y la iteración a través de estructuras de datos complejas.

Empaquetado (Packing)

El empaquetado de tuplas ocurre cuando varios valores se agrupan en una única tupla, permitiendo manipular o retornar múltiples valores de manera compacta.

```
# Empaquetado
coordinates = (4, 5)
print(coordinates) # Salida: (4, 5)
```

Desempaquetado (Unpacking)

Por otro lado, el desempaquetado extrae los valores contenidos en una tupla, permitiendo asignarlos a variables individuales de manera simultánea y eficiente.

```
# Desempaquetado
x, y = coordinates
print(f"x: {x}, y: {y}") # Salida: x: 4, y: 5
```

Iteración y Desempaquetado en Bucles

Al combinar las tuplas con bucles, es posible iterar a través de listas de tuplas y desempaquetar sus contenidos de forma limpia y expresiva.

```
# Lista de tuplas
points = [(1, 2), (3, 4), (5, 6)]

# Iteración y desempaquetado
for x, y in points:
    print(f"Point: ({x}, {y})")
```

En este caso, cada tupla dentro de la lista `points` es desempaquetada en las variables `x` e `y` durante cada iteración del bucle, permitiendo acceder directamente a los valores contenidos en las tuplas.

Uso Práctico: Intercambio de Valores

El desempaquetado de tuplas también permite un intercambio de valores entre variables de manera directa y sin la necesidad de una variable temporal.

```
# Intercambio de valores
a=5
b = 10
a, b = b, a
print(f"a: {a}, b: {b}")      # Salida: a: 10, b: 5
```

El empaquetado y desempaquetado de tuplas ofrece una sintaxis expresiva y minimalista, permitiendo desarrollar código Python limpio y legible, especialmente al manipular estructuras de datos y variables.

3.3.5. Uso de la función `zip` en Python

La función `zip` en Python permite combinar dos o más iterables (por ejemplo, listas o tuplas) elemento a elemento, creando un nuevo iterable que genera tuplas que contienen los elementos correspondientes de los iterables originales.

Básicos de `zip`

La función `zip` toma como argumentos dos o más iterables, y retorna un iterador que genera tuplas. Cada tupla contiene los elementos n-ésimos de los iterables de entrada.

```
names = ["Alice", "Bob", "Charlie"]
ages = [23, 34, 45]

paired = zip(names , ages)
# Resultado: [('Alice', 23), ('Bob', 34), ('Charlie', 45)]
```

Uso de `zip` con Bucles

`zip` es especialmente útil en bucles `for`, donde puede simplificar la iteración sobre múltiples listas en paralelo.

```
for name , age in zip(names , ages):
    print(f"{name} is {age} years old.")
```

En este ejemplo, `name` y `age` toman los valores de las tuplas generadas por `zip` en cada iteración del bucle.

Desempaquetado con `zip`

La función `zip` también puede ser usada en conjunción con el operador `*` (asterisco) para desempaquetar los iterables y realizar operaciones inversas, como convertir filas en columnas (y viceversa) en contextos de estructuras bidimensionales.

```
pairs = [('Alice', 23), ('Bob', 34), ('Charlie', 45)]  
  
# Desempaquetando pares a dos listas  
names, ages = zip(*pairs)
```

En este caso, los pares de nombres y edades son desempaquetados en dos listas separadas, replicando los iterables originales.

El uso de la función `zip` agrega una capa de flexibilidad y expresividad en el código, facilitando la manipulación de datos organizados de manera tabular y la iteración paralela sobre múltiples iterables.

3.4. CONJUNTOS EN PYTHON

Un conjunto en Python es una colección desordenada de elementos únicos. Los conjuntos son útiles para realizar operaciones matemáticas como la unión, intersección, diferencia y diferencia simétrica. Se definen utilizando llaves `{}` o mediante la función `set()`.

```
mySet = {1, 2, 3}      # {1, 2, 3}  
anotherSet = set([2, 3, 4])    # {2, 3, 4}
```

Puedes usar llaves para definir un conjunto de esta manera: `1, 2, 3`. Sin embargo, si dejas las llaves vacías de esta manera, `{}`, Python creará un diccionario vacío en su lugar. Así que para crear un conjunto vacío, utiliza `set()`.

3.4.1. Operaciones con conjuntos

- `union()` o `|`: devuelve un conjunto que contiene todos los elementos de los conjuntos involucrados.
- `intersection()` o `&`: devuelve un conjunto que contiene todos los elementos comunes a todos los conjuntos involucrados.
- `difference()` o `-`: devuelve un conjunto que contiene todos los elementos del primer conjunto que no están en los conjuntos especificados.
- `symmetric difference()` o `^`: devuelve un conjunto que contiene elementos que están en un conjunto o en el otro, pero no en ambos.

```
# Uso de operaciones con conjuntos  
unionSet = mySet | anotherSet      # {1, 2, 3, 4}  
intersectionSet = mySet & anotherSet    # {2, 3}
```

3.4.2. Métodos de conjuntos

Algunos de los métodos que proporcionan los conjuntos son:

- `add(x)`: añade el elemento x al conjunto.
- `remove(x)`: elimina el elemento x del conjunto. Si x no existe, se genera un error.
- `discard(x)`: elimina el elemento x del conjunto si existe.
- `clear()`: elimina todos los elementos del conjunto.
- `pop()`: elimina y devuelve un elemento arbitrario del conjunto. Si el conjunto está vacío, se genera un error.
- `update(...)`: actualiza el conjunto, añadiendo elementos de todos los conjuntos proporcionados.

```
# Uso de métodos con conjuntos
mySet.add(4)    # {1, 2, 3, 4}
mySet.remove(1)  # {2, 3, 4}
mySet.discard(2) # {3, 4}
mySet.clear()   # set()
```

Cada estructura de datos en Python tiene sus propias características y es adecuada para diferentes tipos de aplicaciones, dependiendo de los requerimientos del problema a resolver. Seleccionar la estructura de datos adecuada es crucial para implementar soluciones eficientes y efectivas.

3.5. FUNCIONES PARA COLECCIONES EN PYTHON

En Python, algunas funciones predefinidas como `len()`, `min()`, `max()`, `sum()` y `sort()` se utilizan comúnmente para realizar operaciones en colecciones de datos como listas, conjuntos, tuplas y diccionarios.

3.5.1. La función `len()`

La función `len()` retorna la longitud (número de elementos) de una colección.

```
myList = [1, 2, 3]
length = len(myList)    # Salida: 3
```

3.5.2. Las funciones `min()` y `max()`

`min()` y `max()` devuelven, respectivamente, el elemento más pequeño y más grande de una colección.

```
mySet = {1, 2, 3}
minValue = min(mySet)    # Salida: 1
maxValue = max(mySet)    # Salida: 3
```

3.5.3. La función `sum()`

`sum()` retorna la suma de todos los elementos de una colección.

```
myTuple = (1, 2, 3)
total = sum(myTuple)    # Salida: 6
```

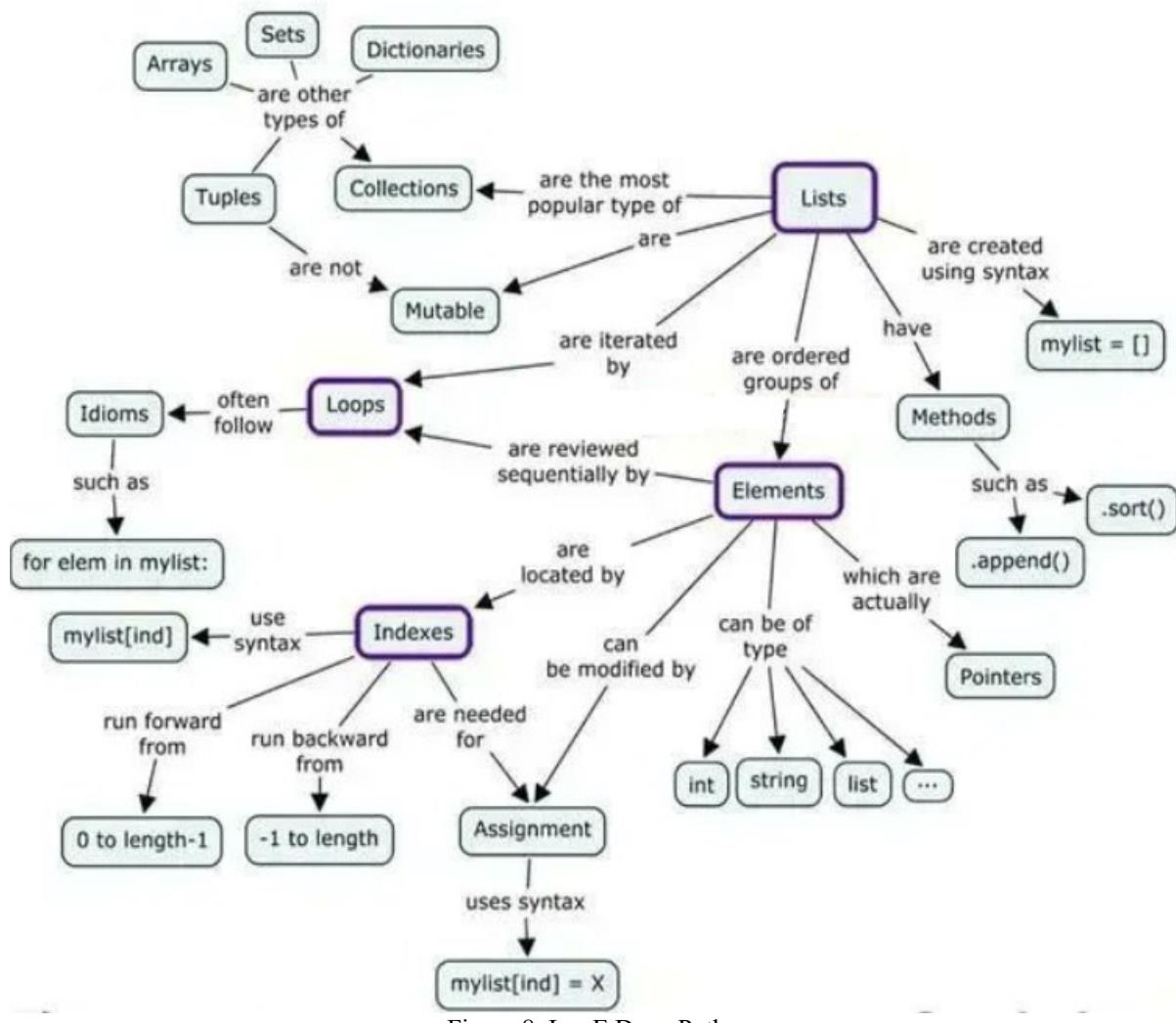


Figura 8: Las E.D. en Python

3.5.4. La función `sort()` y el método `sorted()`

`sort()` es un método de las listas que las ordena in situ, mientras que `sorted()` retorna una nueva colección ordenada a partir de la original.

```

myList = [3, 1, 2]
myList.sort()      # myList es ahora [1, 2, 3]

myTuple = (3, 1, 2)
sortedTuple = tuple(sorted(myTuple))      # sortedTuple es (1, 2, 3)

```

Todas estas funciones pueden aplicarse a listas, conjuntos y tuplas. Sin embargo, con los diccionarios, actúan de forma predeterminada sobre las claves del diccionario. Si deseas aplicarlas a los valores, debes utilizar el método `values()` del diccionario.

```

myDict = {"one": 1, "two": 2, "three": 3}
minKey = min(myDict)      # Salida: "one"
minValue = min(myDict.values())      # Salida: 1

```

4. LAS FUNCIONES

Las funciones en Python son bloques de código reutilizables que se definen con la palabra clave `def` y se invocan utilizando su nombre seguido de paréntesis. Las funciones pueden aceptar argumentos, retornar valores y son fundamentales para escribir código limpio, modular y eficiente.

4.1. DEFINICIÓN E INVOCACIÓN DE FUNCIONES

Una función en Python se define utilizando la sintaxis siguiente y se invoca llamándola por su nombre.

```
def greet(name):
    print(f"Hello, {name}!")

# Invocación de la función
greet("Alice")
```

4.2. ARGUMENTOS Y VALOR DE RETORNO

Las funciones pueden aceptar argumentos que permiten modificar su comportamiento y pueden retornar valores para ser utilizados posteriormente.

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

# Invocación con argumentos y captura del valor returned
result = add(3, 4)
print(result)      # Salida: 7
```

4.3. RECURSIVIDAD

La recursividad implica que una función se llame a sí misma en su definición. Es especialmente útil para resolver problemas que se pueden descomponer en subproblemas de una naturaleza similar.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
# Cálculo factorial de 5: 5 * 4 * 3 * 2 * 1 = 120
print(factorial(5))      # Salida: 120
```

4.4. IMPORTANCIA DE LA RECURSIVIDAD EN INTELIGENCIA ARTIFICIAL

Las funciones y, por ende, la recursividad, desempeñan un papel crucial en el campo de la Inteligencia Artificial. Proporcionan una manera de estructurar el código para que pueda adaptarse y evolucionar, facilitando la implementación de algoritmos complejos y modelando procesos de decisión, como los que se encuentran en algoritmos de búsqueda y optimización.

- **Modularidad:** Permiten dividir problemas complejos en subproblemas manejables.

- **Reusabilidad:** Facilitan la reutilización del código en diferentes partes del programa o en diferentes proyectos.
- **Abstracción:** Ofrecen una manera de ocultar los detalles de implementación, permitiendo que los científicos de datos e ingenieros de IA se centren en el problema de alto nivel.

Por ejemplo, los algoritmos de aprendizaje profundo utilizan funciones de activación, pérdida y optimización, que son implementadas típicamente como funciones debido a su reusabilidad y flexibilidad, permitiendo a los desarrolladores y científicos experimentar con diferentes combinaciones y ajustes de manera eficiente y organizada.

A través de este mecanismo de funciones, Python facilita el desarrollo y experimentación en IA, proporcionando las herramientas necesarias para construir, entrenar, y evaluar modelos, facilitando así la exploración de soluciones innovadoras y efectivas en el campo.

4.5. DOCUMENTACIÓN Y ANOTACIONES DE TIPO EN FUNCIONES

Las funciones en Python pueden (y deben) ser documentadas para proporcionar una descripción clara de su propósito, los parámetros que aceptan y los valores que retornan. Una técnica comúnmente utilizada para documentar funciones es la inclusión de una cadena de documentación (docstring) justo después de la definición de la función.

Adicionalmente, desde la versión 3.5, Python ha introducido las anotaciones de tipo, permitiendo a los desarrolladores especificar el tipo de valor que una función espera recibir y el tipo de valor que devolverá. Estas anotaciones de tipo se establecen utilizando el operador `->` y los dos puntos (`:`).

```
def greet(name: str) -> str:  
    """  
    This function greets the person passed in as a parameter.  
  
    Parameters:  
    - name (str): The name of the person to greet.  
  
    Returns:  
    str: A greeting string.  
    """  
    return f"Hello, {name}!"
```

En este ejemplo, la función `greet` acepta un parámetro `name` que debería ser de tipo `str` (`string`) y también retornará un valor de tipo `str`. La cadena de documentación, encerrada entre triple comillas dobles, describe el propósito de la función, los parámetros que acepta y el tipo de valor que retorna.

Las anotaciones de tipo y las cadenas de documentación son herramientas valiosas que permiten a los desarrolladores entender rápidamente el uso previsto de una función, sin necesariamente tener que leer y entender el código fuente de la misma. Además, aunque Python es un lenguaje de tipado dinámico y las anotaciones de tipo no afectan la ejecución del código, las mismas facilitan la detección de posibles errores mediante el uso de herramientas de verificación de tipo estático, así como mejoran la calidad de las sugerencias de funciones y métodos en entornos de desarrollo integrados (IDEs) y editores de texto.

4.5.1. Pasar argumentos a funciones en Python

En Python, los argumentos pueden ser pasados a las funciones de varias formas, lo que permite flexibilidad al llamar a las funciones y mejora la legibilidad del código. A continuación, se describe cómo se pueden pasar los argumentos a una función:

Argumentos Posicionales

Los argumentos posicionales son aquellos que se pasan por posición, lo que significa que el orden en el que se pasan los argumentos es crucial.

```
def function(a, b):  
    return a + b  
  
result = function(3, 4)      # a=3, b=4
```

Argumentos por Palabra Clave (keyword)

Los argumentos también pueden ser pasados utilizando palabras clave, especificando directamente el valor que cada parámetro debe tomar.

```
result = function(a=3, b=4)      # a=3, b=4  
result = function(b=4, a=3)      # a=3, b=4
```

Argumentos con Valores por Defecto

Las funciones pueden definir valores por defecto para uno o más de sus argumentos.

```
def function(a, b=4):  
    return a + b  
  
result = function(3)      # a=3, b=4
```

Uso de `*args`

El operador `*` permite que una función acepte un número arbitrario de argumentos posicionales, que se recibirán como una tupla.

```
def functionWithArgs(*args):  
    return sum(args)  
  
result = functionWithArgs(3, 4, 5)      # args=(3, 4, 5)  
# Salida: 12
```

Uso de `**kwargs`

De manera similar, `**kwargs` permite aceptar un número arbitrario de argumentos por palabra clave, que se recibirán como un diccionario.

```
def functionWithKwargs(**kwargs):
```

```
return sum(kwargs.values())

result = functionWithKwargs(a=3, b=4, c=5)
# kwargs={'a': 3, 'b': 4, 'c': 5}
# Salida: 12
```

4.6. PASO POR VALOR Y PASO POR REFERENCIA

Cuando se habla de pasar argumentos a funciones, es fundamental comprender la diferencia entre el paso por valor y el paso por referencia, ya que el comportamiento de los argumentos dentro de la función puede variar notablemente según se aplique una u otra modalidad. En general, estos conceptos se refieren a cómo se gestionan la memoria y la asignación de valores en el contexto de las funciones.

Paso por valor

En el paso por valor, se pasa una copia del valor actual de la variable. Los cambios realizados en este valor dentro de la función no afectan a la variable original fuera de la función.

```
def modifyValue(x):
    x=x+1
    return x a=5

b = modifyValue(a)
# a = 5, b = 6
```

En este ejemplo, aunque `x` se modifica dentro de la función, `a` permanece inalterada, ya que lo que se ha pasado a la función es una copia del valor de `a`, no `a` misma.

Paso por referencia

En el paso por referencia, se pasa un alias o referencia a la variable original, de manera que los cambios en la variable dentro de la función afectan también a la variable original fuera de la función.

```
def modifyList(lst):
    lst.append(4)

myList = [1, 2, 3]
modifyList(myList)
# myList = [1, 2, 3, 4]
```

En este caso, al modificar `lst` dentro de la función, también estamos modificando `myList` fuera de la función, ya que hemos pasado la referencia a la lista original, no una copia de ella.

Notas Importantes: En Python, los objetos inmutables como los números, las cadenas y las tuplas se pasan por valor (más exactamente, se pasa una referencia al objeto, pero al ser este inmutable, en la práctica es como si se pasase el valor). En contraste, los objetos mutables, como listas y diccionarios, se pasan por referencia. Este comportamiento puede tener implicaciones

importantes cuando se diseñan funciones y métodos, especialmente si modifican objetos mutables que se pasan como argumentos.

4.7. ÁMBITO DE LAS VARIABLES EN PYTHON

El ámbito de una variable se refiere al área del programa en la que una variable es reconocida y puede ser utilizada. En Python, las variables se asocian con determinados ámbitos, que se definen por el lugar en el cual la variable es asignada. A continuación se describen algunos conceptos clave sobre este tema.

Variable Local

Una variable definida dentro de una función se llama variable local a esa función, ya que su ámbito está limitado a ella. Cuando la función finaliza su ejecución, la variable local se destruye y no puede ser accedida fuera de la función.

```
def exampleFunction():
    localVariable = "I am local"
    print(localVariable) # Output: I am local

exampleFunction()
# print(localVariable)
# Error: NameError
```

Variable Global

Las variables definidas fuera de todas las funciones, en el nivel principal del script, son globales. Estas pueden ser leídas desde cualquier parte del código, pero para modificarlas dentro de una función es necesario declararlas como global dentro de la función.

```
globalVariable = "I am global"

def exampleFunction():
    global globalVariable
    globalVariable = "I am modified"
    print(globalVariable)    # Output: I am modified

exampleFunction()
print(globalVariable)    # Output: I am modified
```

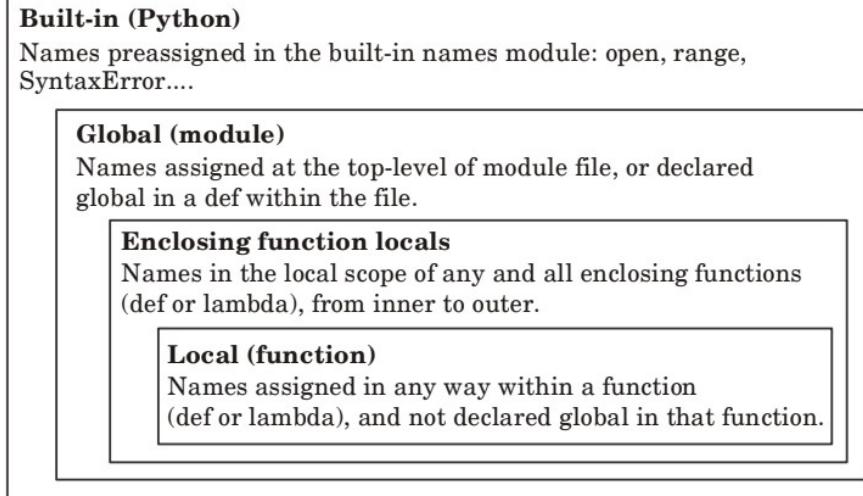


Figura 9: Alcance de las variables en Python

Variable No Local

En el caso de funciones anidadas, para modificar una variable de la función contenedora en la función anidada, se utiliza la palabra clave nonlocal.

```
def outerFunction():
    nonLocalVariable = "I am non-local"

    def innerFunction():
        nonlocal nonLocalVariable
        nonLocalVariable = "I am modified non-local"
        print(nonLocalVariable)      # Output: I am modified non-local

    innerFunction()
    print(nonLocalVariable)      # Output: I am modified non-local

outerFunction()
```

Estos conceptos acerca del ámbito de las variables son fundamentales para escribir código que sea tanto funcional como fácil de entender y mantener.

4.8. FUNCIONES LAMBDA

Las funciones lambda, también denominadas funciones anónimas, son una característica distintiva de Python, permitiendo la creación de funciones de manera rápida y concisa sin necesidad de utilizar la palabra clave *def* para su definición. La sintaxis general de una función lambda es:

```
lambda arguments: expression
```

Donde *arguments* son los parámetros que recibe la función, y *expression* es la operación que realiza, devolviendo su resultado. Es crucial notar que las funciones lambda pueden tener cualquier número de parámetros, pero solo una expresión.

A continuación se presentan algunos ejemplos básicos utilizando funciones lambda.

```
# Ejemplo 1: Función Lambda que suma dos números
suma = lambda x, y: x + y
print(suma(5, 3))      # Output: 8

# Ejemplo 2: Función Lambda que eleva un número al cuadrado
cuadrado = lambda x: x ** 2
print(cuadrado(4))    # Output: 16

# Ejemplo 3: Función Lambda que calcula el máximo entre dos números
maximo = lambda x, y: x if x > y else y
print(maximo(10, 15)) # Output: 15
```

Las funciones lambda son particularmente útiles cuando deseamos realizar operaciones sencillas sin la necesidad de definir una función completa mediante `def`. Es común utilizarlas en funciones como `filter`, `map`, y `reduce`, así como en definiciones de funciones de orden superior o cuando se requieren funciones temporales para operaciones como el ordenamiento de listas.

4.8.1. Limitaciones de las funciones Lambda

A pesar de su utilidad y concisión, las funciones lambda presentan algunas limitaciones:

- Sólo pueden contener una expresión y no bloques de código.
- No pueden incluir anotaciones de tipo.
- Su legibilidad puede ser menor en comparación con las funciones definidas con `def`, especialmente cuando la expresión es compleja.

Por lo tanto, mientras que las funciones `lambda` son una herramienta poderosa para tareas pequeñas y específicas, las funciones definidas tradicionalmente mediante `def` pueden ser preferibles en situaciones que requieran mayor claridad y funcionalidad.

4.8.2. Uso de `filter`, `map`, `reduce` y `sorted` con funciones lambda

Filter

La función `filter(función, secuencia)` filtra los elementos de una secuencia. La función debe retornar `True` o `False` y sólo los elementos para los que la función retorna `True` son incluidos en el resultado.

```
# Ejemplo con filter
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))    # Salida: [2, 4, 6, 8]
```

Map

La función `map(función, secuencia)` aplica la función a cada elemento de la secuencia y retorna un objeto `map` que puede ser convertido a una lista o tupla.

```
# Ejemplo con map
numbers = [1, 2, 3, 4, 5]
```

```
squares = map(lambda x: x**2, numbers)
print(list(squares))      # Salida: [1, 4, 9, 16, 25]
```

Reduce

La función `reduce(función, secuencia)` aplica la función acumulativa a los elementos de la secuencia de izquierda a derecha, reduciendo la secuencia a un único valor.

```
from functools import reduce
# Ejemplo con reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)      # Salida: 120
```

Sorted

La función `sorted(secuencia, key=función)` retorna una nueva lista ordenada de los elementos de la secuencia, opcionalmente utilizando una función clave.

```
# Ejemplo con sorted
words = ['banana', 'apple', 'kiwi', 'orange']
sorted_words = sorted(words, key=lambda x: x[1])
print(sorted_words)      # Salida: ['banana', 'kiwi', 'apple', 'orange']
```

5. PANDAS DATAFRAMES

Pandas es una biblioteca de Python que proporciona estructuras de datos y herramientas de análisis de datos. Los DataFrames son una parte central de Pandas y se utilizan ampliamente en la manipulación de datos.

5.1. CREACIÓN DE DATAFRAMES

Los DataFrames pueden ser creados de diversas maneras, utilizando los múltiples constructores que tiene.

```
import pandas as pd

# Desde listas
data_from_lists = pd.DataFrame([[1, 'Alice'], [2, 'Bob'], [3, 'Charlie']], columns=['ID', 'Name'])
print(data_from_lists)

# Desde diccionarios
data_from_dict = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
print(data_from_dict)

# Usando zip
names = ['Alice', 'Bob', 'Charlie']
ids = [1, 2, 3]
```

```

        data_from_zip = pd.DataFrame(list(zip(ids, names)), columns=['ID',
'Name'])
print(data_from_zip)

# salidas
#   ID Name
# 0  1 Alice
# 1  2 Bob
# 2  3 Charlie
    
```

5.2. ANÁLISIS INICIAL DEL DATAFRAME CON *SHAPE*, *SIZE* Y *NDIM*

Shape

El atributo *shape* de un DataFrame en Pandas devuelve una tupla que representa la dimensionalidad del DataFrame. Específicamente, *shape* retorna una tupla de la forma *(rows, columns)*, permitiéndonos conocer rápidamente el tamaño del DataFrame.

```

import pandas as pd

# Creación de un DataFrame
data = {'Name': ['Anna', 'Bob', 'Charlie', 'Diana'],
        'Age': [23, 34, 45, 36],
        'Salary': [70000, 80000, 120000, 110000]}
df = pd.DataFrame(data)

# Uso de shape
df_shape = df.shape      # Retorna: (4, 3)
    
```

Size

Por otro lado, *size* devuelve el número total de elementos dentro del DataFrame, es decir, el producto de la cantidad de filas por la cantidad de columnas.

```

# Uso de size
df_size = df.size      # Retorna: 12
    
```

Es importante notar que *size* cuenta todos los elementos del DataFrame, incluso los *Nan*.

Ndim

El atributo *ndim* retorna el número de ejes/dimensiones del DataFrame. Para un DataFrame regular, siempre devolverá el valor 2, dado que es una estructura bidimensional (tiene filas y columnas).

```

# Uso de ndim
df_ndim = df.ndim      # Retorna: 2
    
```

5.3. SELECCIÓN DE DATOS CON *iLoc*, *Loc* Y *Columns*

La selección de datos es una operación fundamental cuando trabajamos con DataFrames en pandas. A continuación, se presenta una explicación detallada de los métodos *iLoc*, *loc* y el uso de *columns* para acceder y seleccionar datos.

Utilizando *iLoc*

El método *iLoc* se utiliza para la selección basada en índices enteros. Permite seleccionar elementos de un DataFrame por posición usando enteros, similar a cómo se accede a los elementos en arrays de NumPy.

```
import pandas as pd

data = pd.DataFrame({
    'Name': ['Anna', 'Bob', 'Charlie', 'Diana'],
    'Age': [23, 34, 45, 36],
    'Salary': [70000, 80000, 120000, 110000]
})

# Seleccionar la primera fila
first_row = data.iloc[0]

# Seleccionar las primeras dos filas y columnas
subset = data.iloc[:2, :2]
```

Se puede especificar tanto un rango de filas como de columnas para seleccionar, utilizando la notación de *slicing* de Python (por ejemplo, `data.iloc[1:3, 0:2]`).

Utilizando *loc*

Mientras que *iLoc* selecciona datos basándose en su posición numérica, *loc* selecciona filas y columnas utilizando sus etiquetas. Este método permite seleccionar datos basándose en etiquetas de índice y nombres de columnas.

```
# Seleccionar la fila para 'Bob' por etiqueta
bob_row = data.loc[1]

# Seleccionar un rango de filas y columnas específicas por
etiqueta/nombre
subset = data.loc[1:3, ['Name', 'Salary']]
```

loc es particularmente útil cuando los índices del DataFrame son etiquetas en lugar de números enteros.

	loc[] - By Label	iloc[] - By Index
Select Single Row	df.loc['r2']	df.iloc[1]
Select Single Column	df.loc[:, "Courses"]	df.iloc[:, 0]
Select Multiple Rows	df.loc[['r2','r3']]	df.iloc[[1,2]]
Select Multiple Columns	df.loc[:, ["Courses","Fee"]]	df.iloc[:, [0,1]]
Select Rows Range	df.loc['r1':'r4']	df.iloc[0:4]
Select Columns Range	df.loc[:, 'Fee':'Discount']	df.iloc[:, 1:4]
Select Alternate Rows	df.loc['r1':'r4':1]	df.iloc[0:4:1]
Select Alternate Columns	df.loc[:, 'Fee':'Discount':1]	df.iloc[:, 1:4:1]
Using Condition	df.loc[df['Fee'] >= 24000]	df.iloc[list(df['Fee'] >= 24000)]
Using Lambda Function	df.loc[lambda x: x[3]]	df.iloc[lambda x: x[3]]

Figura 10: loc vs iloc

Utilizando Nombres de Columnas

Pandas también permite seleccionar datos directamente utilizando el nombre de la columna, proporcionando una forma intuitiva y fácil de acceder a columnas específicas de datos.

```
# Seleccionar la columna 'Age'
age_column = data['Age']

# Seleccionar múltiples columnas usando una lista de nombres de columnas
subset = data[['Name', 'Salary']]
```

Este método es especialmente útil para seleccionar columnas específicas rápidamente, pero debe utilizarse con cuidado para evitar confusiones y errores en scripts y análisis más grandes y complejos.

Cada uno de estos métodos ofrece diferentes ventajas y desventajas dependiendo del contexto y los requisitos de tu análisis. El uso adecuado de estos métodos de selección puede mejorar significativamente la legibilidad y eficiencia de tu código en Python utilizando pandas.

	loc	iloc
A value	A single label or integer e.g. loc[A] or loc[1]	A single integer e.g. iloc[1]
A list	A list of labels e.g. loc[[A, B]]	A list of integers e.g. iloc[[1,2,3]]
Slicing	e.g. loc[A:B], A and B are included	e.g. iloc[n:m], n is included, m is excluded
Conditions	A bool Series or list	A bool list
Callable function	loc[lambda x: x[2]]	iloc[lambda x: x[2]]

Figura 11: loc vs iloc (II)

5.4. OPERACIONES DML EN PANDAS

En esta sección, se presenta un conjunto de operaciones básicas comúnmente realizadas en DataFrames utilizando la librería Pandas. A través de la explicación detallada de varios fragmentos de código, se pretende ofrecer una visión clara y comprensiva de las funcionalidades de estos métodos.

- `copy()`: Crea una copia del DataFrame para preservar los datos originales.
- `head(n)`: Retorna las primeras n filas del DataFrame.
- `tail(n)`: Retorna las últimas n filas del DataFrame.

```
# copy , head , tail
data_copy = data_from_lists.copy()
first_rows = data_from_lists.head(2)      # Primeras dos filas
last_rows = data_from_lists.tail(2)       # Últimas dos filas
```

- `rename()`: Renombra las columnas especificadas.
- `columns`: Reasigna los nombres de todas las columnas.
- `insert()`: Inserta una nueva columna en la posición especificada.

```
# rename , columns , insert
data_renamed = data_from_lists.rename(columns={'ID': 'Identifier'})
data_from_lists.columns = ['Identifier', 'First Name']
data_from_lists.insert(2, 'Age', [30, 25, 35])
```

- `drop()`: Elimina la(s) columna(s) especificada(s).
- `unique()`: Obtiene valores únicos de una columna.
- `duplicated()`: Identifica filas duplicadas.
- `drop_duplicates()`: Elimina filas duplicadas.

```
# drop , unique , duplicated , drop_duplicates
data_dropped = data_from_lists.drop(columns=['Age'])
unique_names = data_from_lists['First Name'].unique()
duplicated_rows = data_from_lists.duplicated()
data_no_duplicates = data_from_lists.drop_duplicates()
```

- `nsmallest(n, 'Column')`: Obtiene las n filas con los valores más pequeños en 'Column'.
- `nlargest(n, 'Column')`: Obtiene las n filas con los valores más grandes en 'Column'.

```
# nsmallest , nlargest
youngest = data_from_lists.nsmallest(1, 'Age')
oldest = data_from_lists.nlargest(1, 'Age')
```

- `dtypes`: Retorna los tipos de datos de cada columna.
- `iterrows()`: Itera a través de las filas del DataFrame.
- `itertuples()`: Itera a través de las filas y proporciona un acceso más rápido comparado a `iterrows()`.

- `iteritems()`: Itera a través de las columnas del DataFrame.

```
# dtypes , iterrows , itertuples , iteritems
data_types = data_from_lists.dtypes
for index, row in data_from_lists.iterrows():
    pass
for row in data_from_lists.itertuples():
    pass
for column_name, column in data_from_lists.iteritems():
    pass
```

5.5. CARGA DE ARCHIVOS CSV Y JSON EN PANDAS

La biblioteca Pandas proporciona utilidades poderosas para cargar datos desde diferentes formatos de archivo directamente en un DataFrame. Los métodos `read_csv` y `read_json` permiten una fácil importación de datos desde archivos CSV y JSON, respectivamente, al proporcionar una variedad de parámetros para controlar el proceso de carga, haciendo que la ingestión de datos sea flexible y eficiente.

```
import pandas as pd

# Cargar un archivo CSV
data_csv = pd.read_csv('file_path.csv')

# Cargar un CSV especificando el separador, y las columnas a cargar
data_csv_custom = pd.read_csv('file_path.csv', \
    sep=';', usecols=['col1', 'col3'])
```

El método `read_csv` es utilizado comúnmente para leer archivos CSV, que son uno de los formatos de datos tabulares más comunes. El primer argumento es la ubicación del archivo. El parámetro opcional `sep` permite especificar un separador de campo personalizado (el predeterminado es la coma). `usecols` es otro parámetro útil que permite seleccionar qué columnas cargar en el DataFrame, lo cual es especialmente útil cuando se trata con conjuntos de datos de gran tamaño.

```
# Cargar un archivo JSON
data_json = pd.read_json('file_path.json')

# Cargar un JSON con orientación de registros, y convertir la columna
'date' en datetime
data_json_custom = pd.read_json('file_path.json', \
    orient='records', convert_dates=['date'])
```

Por otro lado, `read_json` es utilizado para cargar datos desde archivos JSON. El primer argumento es, nuevamente, la ubicación del archivo. JSON puede almacenar datos en diversas orientaciones (estructuras), como columnar o como una serie de registros. El parámetro `orient` permite especificar esta orientación, siendo las opciones más comunes `split`, `records`, `index`, `columns` y `values`. El parámetro `convert_dates` puede ser utilizado para convertir automáticamente campos específicos a objetos `datetime`.

Ambos métodos, `read_csv` y `read_json`, poseen una amplia variedad de parámetros adicionales que te permiten gestionar situaciones como hojas de cálculo con múltiples tablas, limpieza de datos incompletos o malformados, y mapeo de datos categóricos, entre otros. Para más detalles y opciones de uso, se recomienda revisar la documentación oficial de Pandas.

5.6. MANEJO DE DATOS INEXISTENTES EN PANDAS

Para el preprocesamiento de datos, Pandas proporciona una variedad de herramientas para manejar datos nulos o inexistentes representados generalmente por `NaN` (Not a Number). Esta sección muestra ejemplos de métodos como `isnull`, `notnull`, `dropna`, `fillna`, `replace` e `interpolate`, los cuales permiten identificar, eliminar o imputar estos datos inexistentes.

- `isnull()`: Genera una máscara booleana indicando posiciones de datos inexistentes.
- `notnull()`: Genera una máscara booleana indicando posiciones de datos existentes.

```
import pandas as pd

# Crear un DataFrame de ejemplo
data = pd.DataFrame({'A': [1, 2, np.nan, 4],
                     'B': [5, np.nan, np.nan, 8],
                     'C': [9, 10, 11, 12]})

# Identificar datos inexistentes
is_null = data.isnull()
not_null = data.notnull()
```

- `dropna()`: Elimina las filas o columnas que contienen `NaN`, dependiendo del eje especificado.

```
# Eliminar filas que contienen datos inexistentes
data_dropped_row = data.dropna()

# Eliminar columnas que contienen datos inexistentes
data_dropped_col = data.dropna(axis=1)
```

- `fillna()`: Imputa los datos inexistentes utilizando un valor específico o un método (como '`ffill`' o '`bfill`').

```
# Rellenar con un valor específico
data_filled = data.fillna(0)

# Utilizar el método 'ffill' para llenar hacia adelante
data_ffill = data.fillna(method='ffill')

# Utilizar el método 'bfill' para llenar hacia atrás
data_bfill = data.fillna(method='bfill')
```

- `replace()`: Sustituye los valores especificados por otros deseados.

- `interpolate()`: Realiza una interpolación para llenar datos inexistentes, utilizando varios métodos, como lineal o polinómico.

```
# Reemplazar datos inexistentes con un valor específico
data_replaced = data.replace(np.nan, -1)

# Interpolan los datos inexistentes linealmente
data_interpolated = data.interpolate()
```

6. LA LIBRERÍA NUMPY

NumPy es una biblioteca central en Python para la computación científica, ofreciendo soporte para arrays multidimensionales, operaciones matemáticas de alto nivel y la capacidad de realizar operaciones de manera eficiente en bloques enteros de datos sin necesidad de ciclos `for` en Python puro.

6.1. TRATAMIENTO DE ARRAYS EN NUMPY

Creación e inicialización

Se puede crear un array de *NumPy* con la función `array`, que recibe una lista de elementos. Para crear e inicializar el array de *NumPy* al mismo tiempo, se pueden utilizar las funciones `zeros` y `ones`. La función `linspace` retorna números espaciados con un intervalo específico.

```
import numpy as np

# Crear un array a partir de una lista
array_from_list = np.array([1, 2, 3, 4, 5])

# Crear un array de ceros
zeros_array = np.zeros((3, 4))

# Crear un array de unos
ones_array = np.ones((2, 5))

# Crear un array con valores espaciados uniformemente
linspace_array = np.linspace(0, 1, num=5)
```

6.1.1. Exploración y manipulación de dimensiones y forma en arrays con Numpy

Función `ndim`

La función `ndim` en *NumPy* se utiliza para obtener el número de dimensiones de un array. La dimensión aquí se refiere a la cantidad de índices necesarios para seleccionar un elemento particular del array. Un array unidimensional (1D) tiene una dimensión, un array bidimensional (2D) tiene dos, y así sucesivamente.

```
import numpy as np
```

```
array_1d = np.array([1, 2, 3])
print(array_1d.ndim) # Salida: 1
```

En el código anterior, `array_1d` es un array de una dimensión, por lo que la salida será 1.

Función `shape`

La propiedad `shape` de un objeto array de *NumPy* proporciona una tupla de los tamaños de cada dimensión del array. La longitud de la tupla `shape` es, por lo tanto, el número de dimensiones, `ndim`.

```
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2d.shape) # Salida: (2, 3)
```

En este ejemplo, `array_2d` es un array de 2 filas y 3 columnas, por lo que `shape` devuelve la tupla (2,3).

Función `reshape`

La función `reshape` se utiliza para cambiar la forma del array sin cambiar sus datos. Específicamente, `reshape` permite reorganizar los datos del array en una nueva forma especificada por una tupla que representa la nueva dimensión deseada.

```
array_1d = np.array([1, 2, 3, 4, 5, 6])

array_2d = array_1d.reshape((2, 3))

print(array_2d)
# Salida:
# [[1 2 3]
#  [4 5 6]]
```

Aquí, el array unidimensional `array_1d` se ha transformado en un array bidimensional `array_2d` con 2 filas y 3 columnas mediante el uso de `reshape((2, 3))`. La nueva forma debe ser compatible con la cantidad de elementos en el array, es decir, el producto de las dimensiones de la nueva forma debe ser igual al número total de elementos en el array original.

En casos donde se desea que *NumPy* determine automáticamente el tamaño de una de las dimensiones, se puede usar `-1` como uno de los parámetros en la función `reshape`. Por ejemplo:

```
array_1d = np.array([1, 2, 3, 4, 5, 6])
array_2d = array_1d.reshape((-1, 2))
print(array_2d)
# Salida:
# [[1 2]
#  [3 4]
#  [5 6]]
```

Aquí, `array_1d` se transforma en un array 2D (`array_2d`) con un número de columnas fijado en 2. La dimensión de las filas se calcula automáticamente como `-1` para cumplir con la cantidad total de elementos en el array.

6.2. FILTRADO DE ARRAYS EN NUMPY

En el ámbito de la ciencia de datos y la programación numérica, a menudo es fundamental poder filtrar datos basándonos en ciertas condiciones. *Numpy* facilita esta tarea mediante una técnica conocida como *Boolean Indexing*, que permite seleccionar o modificar elementos de un array basándose en criterios condicionales.

Filtrado básico

El filtrado de datos en *Numpy* se basa comúnmente en la construcción de matrices de booleanos que se utilizan para indexar el array original. Para empezar, veamos un ejemplo simple de cómo podemos seleccionar elementos que cumplen con una condición específica.

```
import numpy as np

# Crear un array 1D
array_1d = np.array([1, 2, 3, 4, 5, 6])

# Crear un filtro booleano
bool_filter = array_1d > 3

# Aplicar el filtro al array
filtered_array = array_1d[bool_filter]

# Resultado: [4 5 6]
```

En el ejemplo anterior, `bool_filter` es un array booleano que tiene el mismo tamaño que `array_1d`, y contiene `True` o `False` dependiendo de si el elemento correspondiente cumple o no con la condición.

Combinando condiciones

Para construir filtros más complejos, se pueden combinar condiciones utilizando operadores lógicos bit a bit, como `&` (AND), `|` (OR) y `!` (NOT).

```
# Combinando condiciones
complex_filter = (array_1d > 3) & (array_1d < 6)

# Aplicar el filtro combinado
filtered_array = array_1d[complex_filter]

# Resultado: [4 5]
```

Cuando se combinan condiciones, es crucial encapsular cada condición con paréntesis para evitar problemas de precedencia de operadores.

Boolean indexing en Arrays Multidimensionales

El *Boolean indexing* no está limitado a arrays unidimensionales. También es aplicable a arrays multidimensionales.

```
# Crear un array 2D
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Crear un filtro booleano
bool_filter = array_2d % 2 == 0

# Aplicar el filtro al array
filtered_array = array_2d[bool_filter]

# Resultado: [2 4 6 8]
```

Cuando aplicamos el filtrado booleano a un array multidimensional, obtenemos un array unidimensional de elementos que cumplen con la condición.

Modificación con filtros booleanos

Además de seleccionar elementos, los filtros booleanos también pueden utilizarse para modificar un array.

```
# Modificar elementos mayores que 5
array_1d[array_1d > 5] = 0

# Resultado en array_1d: [1 2 3 4 5 0 0]
```

El filtrado de arrays utilizando indexación booleana ofrece una poderosa herramienta para seleccionar y manipular datos de forma rápida y eficiente sin necesidad de utilizar bucles explícitos, favoreciendo la claridad y la eficiencia en el código.

6.2.1. Tipos de datos en Numpy

Los datos en *Numpy* son almacenados en arrays multidimensionales. Cada item en un array de *Numpy* es del mismo tipo, conocido como *dtype* (*data-type*). Algunos ejemplos comunes de *dtype* incluyen *int*, *float*, *bool* y *complex*.

```
import numpy as np

# Crear un array especificando el dtype como int
array_int = np.array([1, 2, 3, 4], dtype=int)

# Crear un array especificando el dtype como float
array_float = np.array([1, 2, 3, 4], dtype=float)

# Crear un array especificando el dtype como bool
array_bool = np.array([True, False, True], dtype=bool)
```

```
# Crear un array especificando el dtype como complex
array_complex = np.array([1 + 2j, 2 - 3j], dtype=complex)
```

La propiedad `dtype` de un array puede ser consultada para conocer el tipo de dato de los elementos del array.

```
# Obtener el dtype del array
data_type = array_int.dtype      # dtype('int64')
```

Numpy intentará inferir el tipo de datos si no se especifica `dtype`, pero especificarlo manualmente puede ser útil para asegurar la precisión y la eficiencia del almacenamiento de datos.

6.2.2. Copias y vistas (views) en Numpy

En *Numpy*, es esencial entender la diferencia entre copias y vistas de un array para evitar problemas al manejar grandes cantidades de datos y para evitar alterar datos originales cuando no se desea hacerlo.

```
# Crear un array simple
original_array = np.array([1, 2, 3, 4, 5])

# Crear una vista del array
array_view = original_array[1:4]

# Crear una copia del array
array_copy = original_array.copy()
```

Una vista (`view`) es simplemente una referencia o vista del array original, lo que significa que los datos no se copian en la memoria. Cualquier cambio en la vista se reflejará en el array original.

```
# Modificar un elemento de la vista
array_view[0] = 99

# Los cambios se reflejan en el array original
# original_array es ahora [1, 99, 3, 4, 5]
```

Por otro lado, una copia (`copy`) es una copia completa de los datos del array en una nueva ubicación de la memoria. Los cambios a la copia no afectan al array original.

```
# Modificar un elemento de la copia
array_copy[0] = 99

# Los cambios no se reflejan en el array original
# original_array permanece como [1, 2, 3, 4, 5]
```

6.2.3. Recorrido de arrays en Numpy con `nditer` y `ndenumerate`

En *Numpy*, la iteración a través de arrays multidimensionales se realiza comúnmente con los métodos `nditer` y `ndenumerate`, los cuales permiten recorrer cada elemento del array de manera eficiente y, opcionalmente, obtener el índice de los elementos recorridos.

Iterando con `nditer`

El método `nditer` es un iterador multidimensional que permite recorrer todos los elementos de un array sin preocuparse por su forma o dimensionalidad. Se utiliza comúnmente cuando se desea aplicar una operación a cada elemento del array.

```
import numpy as np

# Creando un array 2D para demostración
array_2d = np.array([[1, 2], [3, 4]])

# Utilizando nditer para iterar a través del array
for item in np.nditer(array_2d):
    # Aquí 'item' representa cada elemento del array print(item)
    print(array_2d)

#salida
#1
#2
#3
#4
#[[1 2]
#[3 4]]
```

El método `nditer` puede ser configurado para modificar el array original durante la iteración, especificando el modo de operación como '`readwrite`'.

```
# Iterando y modificando elementos con nditer
for item in np.nditer(array_2d , op_flags=['readwrite']):
    item[...] = item * item # Elevando cada elemento al cuadrado

print(array_2d)
#salida
#[[ 1  4]
#[ 9 16]]

#item[...] es una notación de indexación de NumPy que se utiliza para
# realizar una operación in-place sobre el elemento actual al que se
# apunta en la iteración, modificando el array original.
```

Iterando con `ndenumerate`

Por otro lado, el método `ndenumerate` es utilizado para obtener tanto el índice como el valor de cada elemento mientras se itera a través del array, lo cual es particularmente útil cuando la posición del elemento dentro del array es relevante para las operaciones que se están realizando.

```
# Utilizando ndenumerate para iterar a través del array
for idx, item in np.ndenumerate(array_2d):
    # 'idx' contiene la tupla de índices y 'item' el valor del elemento
    print("Index:", idx, "Value:", item)
```

La tupla `idx` proporciona los índices del elemento en cada dimensión del array, facilitando operaciones que dependen de la ubicación de los elementos.

7. PROGRAMACIÓN ORIENTADA A OBJETOS EN PYTHON

La *Programación Orientada a Objetos* (POO) es un paradigma de programación que utiliza objetos y clases para estructurar el código de manera modular y reutilizable. En este enfoque, los objetos son instancias de clases, que actúan como plantillas que encapsulan atributos (datos) y métodos (funciones). La POO permite que los desarrolladores organicen su software alrededor de entidades del mundo real y sus interacciones, promoviendo el principio de modularidad, el ocultamiento de información y la reutilización de código a través de la herencia y la composición.

7.1. DEFINIENDO UNA CLASE EN PYTHON: MÉTODOS Y ATRIBUTOS

En la Programación Orientada a Objetos en Python, una clase se define como un prototipo para crear objetos (instancias), proporcionando atributos y comportamientos básicos que se encapsulan en métodos.

Estructura básica de una Clase

La definición de una clase generalmente involucra:

- **Atributos:** Variables que almacenan datos relacionados con la instancia.
- **Métodos:** Funciones que realizan operaciones sobre los atributos de la instancia o realizan otras operaciones relevantes para la clase.

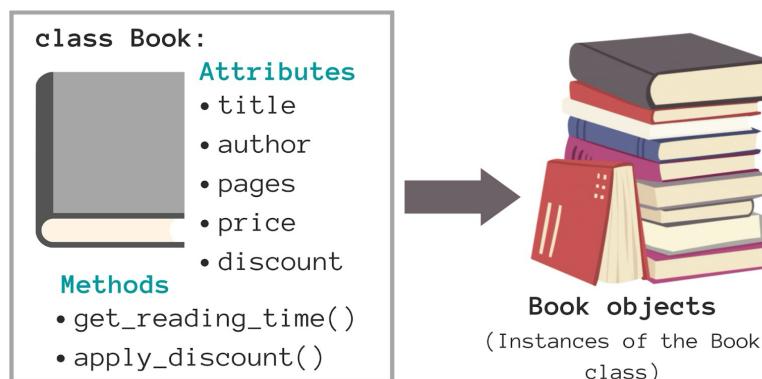


Figura 12: Ejemplo de clases y objetos en POO

Un simple ejemplo en Python sería la definición de una clase Vehículo:

```
class Vehiculo:
    def __init__(self, color, tipo):
        self.color = color
        self.tipo = tipo

    def describir(self):
        return f"Este es un vehículo {self.color} de tipo {self.tipo}"

    def cambiar_color(self, nuevo_color):
        self.color = nuevo_color
```

- `__init__`: Este método especial se llama constructor y se utiliza para inicializar los atributos de la clase. Cuando se crea una nueva instancia de la clase, `__init__` se ejecuta automáticamente.
- `self`: Es la primera variable que todos los métodos toman, y se refiere a la instancia objeto.
- `describir`: Este método devuelve una cadena que describe el vehículo.
- `cambiar_color`: Este método acepta un argumento (`nuevo_color`) y utiliza este valor para cambiar el atributo de color del vehículo.

Cuando se instancia la clase `Vehiculo`, se pueden utilizar sus métodos para interactuar con los atributos de la instancia o realizar otras operaciones pertinentes definidas en la clase.

```
mi_vehiculo = Vehiculo("rojo", "deportivo")
print(mi_vehiculo.describir())
mi_vehiculo.cambiar_color("azul")
print(mi_vehiculo.describir())
```

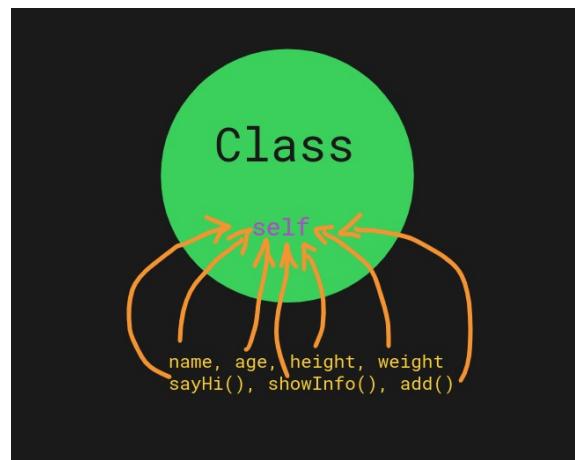


Figura 13: El atributo `self`

7.2. CONSTRUCTORES Y DESTRUCTORES EN PYTHON

Los constructores y destructores son métodos especiales en la programación orientada a objetos de Python que se utilizan para inicializar y des-inicializar respectivamente los objetos de una clase.

Constructor: `__init__()`

El método constructor, denominado `__init__()`, se utiliza para inicializar los atributos de un objeto. Es el primer método que se ejecuta automáticamente cuando se crea un objeto de una clase.

Destructor: `__del__()`

Por otro lado, el destructor, `__del__()`, se encarga de liberar los recursos antes de que el objeto sea destruido por el recolector de basura. Este método es llamado automáticamente cuando todas las referencias al objeto han sido eliminadas.

```
class Ejemplo:
    def __init__(self, valor):
        self.valor = valor
        print(f"Constructor ejecutado , valor = {valor}")

    def __del__(self):
        print(f"El objeto con valor {self.valor} será destruido")

    # Creando una instancia del objeto
    obj = Ejemplo(10)

    # Referenciando al objeto con otro nombre
    ref_obj = obj

    # Eliminando las referencias
    del obj
    del ref_obj
```

En este código, primero se define una clase llamada `Ejemplo` con un constructor que inicializa un atributo y muestra un mensaje. Luego, se define un destructor que simplemente imprime un mensaje notificando que el objeto será destruido. A continuación, se crea un objeto de la clase `Ejemplo`, se crea una referencia adicional al objeto y finalmente, se eliminan ambas referencias. Cuando no hay referencias al objeto, el destructor `__del__()` se ejecuta automáticamente.

7.3. HERENCIA EN PYTHON

La herencia es un mecanismo de la programación orientada a objetos que permite que una clase (denominada subclase) herede atributos y métodos de otra clase (denominada superclase). La herencia permite reutilizar y modificar el código de forma eficiente, al mismo tiempo que establece una relación jerárquica entre las clases.

Sintaxis básica

La sintaxis básica para establecer herencia en Python se muestra a continuación, donde SubClase hereda de SuperClase:

```
class SubClase(SuperClase):  
    \ ...
```

Uso del método `__init__()` en herencia

Cuando se define el método `__init__()` en una subclase, esta no hereda automáticamente el constructor de su superclase. Si se desea invocar el constructor de la superclase desde la subclase, se utiliza la función `super()`.

```
class Animal:  
    def __init__(self, especie):  
        self.especie = especie  
  
    def mostrar_especie(self):  
        print(f"Soy un {self.especie}")  
  
class Perro(Animal):  
    def __init__(self, nombre, raza):  
        super().__init__("Perro")  
        self.nombre = nombre  
        self.raza = raza  
  
    def mostrar_informacion(self):  
        print(f"Mi nombre es {self.nombre} y soy un {self.raza}")  
  
# Uso de las clases  
perro1 = Perro("Boby", "Beagle")  
perro1.mostrar_especie()  
perro1.mostrar_informacion()
```

En este código, `Animal` es una superclase que tiene un constructor `__init__()` para inicializar el atributo `especie` y un método para mostrar dicho atributo. La subclase `Perro` hereda de `Animal`.

7.4. MÉTODOS: ESTÁTICOS, DE INSTANCIA Y DE CLASE

Los métodos de instancia, de clase y estáticos tienen diferentes niveles de acceso a los atributos de la clase y son declarados utilizando los decoradores `@staticmethod` y `@classmethod` respectivamente.

```
class ExampleClass:  
    class_var = "I'm a class variable"  
  
    def __init__(self, parameter):
```

```
        self.parameter = parameter

    @staticmethod
    def static_method():
        print("Static Method Called")

    @classmethod
    def class_method(cls):
        print("Class Method Called:", cls.class_var)
```

7.5. DECORADORES Y `@property`

`@property` permite a los métodos ser accedidos como atributos.

```
class ExampleClass:
    def __init__(self, parameter):
        self._parameter = parameter

    @property
    def parameter(self):
        return self._parameter
```

7.6. VISIBILIDAD DE MIEMBROS DE UNA CLASE

En Python, los miembros de una clase (atributos y métodos) pueden ser públicos, privados o protegidos, lo cual determina su visibilidad y accesibilidad desde otras clases.

- **Public:** Pueden ser accedidos desde cualquier parte del código. No llevan ningún prefijo especial.
- **Protected:** Indicados por un prefijo de un guion bajo `_`, y por convención no deben ser accedidos fuera de la clase.
- **Privados:** Indicados por un prefijo de dos guiones bajos `__`. El nombre de cualquier miembro precedido por dos guiones bajos se traduce internamente a `__nombreClase__nombreMiembro`, lo que se denomina *name mangling*. Esto se hace para hacer la variable inaccesible desde el exterior de la clase y se utiliza para crear miembros privados.

```
class EjemploAcceso:
    def __init__(self):
        self.publico = "miembro publico"
        self._protegido = "miembro protegido"
        self.__privado = "miembro privado"

    def mostrar_atributos(self):
        print(f"Publico: {self.publico}")
        print(f"Protegido: {self._protegido}")
        print(f"Privado: {self.__privado}")

# Instancia de la clase
obj = EjemploAcceso()
```

```
# Acceso permitido
print(obj.publico)

# Acceso no recomendado pero permitido
print(obj._protegido)

# Acceso no permitido
# print(obj.__privado) # Descomentar esta línea genera un error

# Acceso utilizando name mangling
print(obj._EjemploAcceso__privado)
```

En el código proporcionado, se definen tres atributos con diferentes niveles de acceso en la clase *EjemploAcceso*: publico, protegido y privado. Aunque el acceso al atributo protegido (*protegido*) no es recomendado, es posible. El acceso al atributo privado (*privado*) se realiza mediante name mangling, siendo visible de esta manera como *_EjemploAcceso__privado*.

7.7. POLIMORFISMO EN PYTHON

El polimorfismo es un principio de la programación orientada a objetos que permite que objetos de diferentes tipos sean utilizados a través de una misma interfaz. En Python, el polimorfismo se manifiesta comúnmente permitiendo que diferentes clases definan métodos con el mismo nombre.

Métodos Polimórficos

Un método es considerado polimórfico si puede operar o aplicarse en diferentes tipos de objetos. Python, al ser un lenguaje de tipado dinámico, permite un alto grado de polimorfismo ya que no requiere que los tipos de los objetos se declaren explícitamente.

Polimorfismo con funciones

Se puede aprovechar el polimorfismo en funciones que pueden aceptar cualquier objeto, realizando operaciones dependiendo de su tipo o clase, siempre y cuando estas operaciones sean válidas para los objetos pasados.

```
class Perro:
    def sonido(self):
        return "Ladra"

class Gato:
    def sonido(self):
        return "Maulla"

def emitir_sonido(animal):
    print(f"El animal hace: {animal.sonido()}")

# Instancias de los objetos
perro = Perro()
gato = Gato()
```

```
# Uso polimórfico de los objetos
emitir_sonido(perro)
emitir_sonido(gato)
```

En este código, las clases *Perro* y *Gato* definen un método *sonido()* que retorna el sonido que cada animal emite. La función *emitir_sonido(animal)* acepta cualquier objeto que tenga un método *sonido()* y lo utiliza para imprimir el sonido del animal, demostrando un uso polimórfico de los objetos perro y gato, a pesar de ser de diferentes clases. Es válido usar la función con cualquier objeto que implemente el método *sonido()*, lo que muestra la flexibilidad y reutilización del código gracias al polimorfismo.

7.8. USO DE INTERFACES EN POO Y EJEMPLO CON EL PATRÓN OBSERVER

Las interfaces en la Programación Orientada a Objetos (POO) sirven para definir un contrato para las clases, garantizando que implementen un conjunto específico de métodos. En lenguajes como Java, las interfaces son una construcción del lenguaje. Sin embargo, en Python, donde no existe una construcción nativa para interfaces, se pueden simular utilizando clases abstractas y métodos abstractos.

El patrón de diseño *Observer*, por otro lado, es un patrón de comportamiento que define una dependencia uno a muchos entre objetos, de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. A continuación se muestra un ejemplo en Python que utiliza una interfaz y el patrón Observer.

```
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, message: str):
        pass

class ConcreteObserver(Observer):
    def update(self, message: str):
        print(f"Observer received message: {message}")

class Subject:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer: Observer):
        self._observers.append(observer)

    def notify_observers(self, message: str):
        for observer in self._observers:
            observer.update(message)

# Ejemplo de uso
subject = Subject()
observer = ConcreteObserver()
```

```
subject.add_observer(observer)
subject.notify_observers("The subject state has changed!")
```

En el ejemplo, la clase abstracta ‘*Observer*’ actúa como interfaz, estableciendo un contrato que garantiza que cualquier observador implementará el método ‘*update*’. ‘*ConcreteObserver*’ es una implementación concreta de un observador que simplemente imprime el mensaje recibido.

La clase ‘*Subject*’ mantiene una lista de observadores y tiene un método para notificarles los cambios, es decir, ‘*notify_observers*’ que informa a todos los observadores registrados cuando su estado cambia, llamando a su método ‘*update*’.

8. MANEJO DE EXCEPCIONES EN PYTHON

El manejo de excepciones en Python se realiza mediante bloques *try/except*, que permiten al programa gestionar errores (excepciones) de una manera flexible y controlada. Al escribir código, es crucial anticiparse a los posibles errores y gestionarlos de manera adecuada para evitar que el programa termine abruptamente y para proporcionar mensajes de error útiles.

- *try*: El bloque de código que puede lanzar una excepción.
- *except*: Bloque que manejará la excepción lanzada en el bloque *try*.
- *else*: (Opcional) Bloque que se ejecutará si no se lanza ninguna excepción en el bloque *try*.
- *finally*: (Opcional) Bloque que se ejecutará siempre, independientemente de si se lanzó una excepción o no.

El siguiente ejemplo práctico muestra cómo gestionar una excepción común al realizar una división por cero en Python.

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("No es posible dividir por cero.")
else:
    print(f"El resultado es: {resultado}")
finally:
    print("Esta línea se ejecutará siempre.")
```

8.1. MANEJO DE MÚLTIPLES EXCEPCIONES

Python permite manejar múltiples excepciones de manera individual para proporcionar respuestas específicas a diferentes tipos de errores.

```
try:
    numero = int(input("Introduce un número: "))
    resultado = 10 / numero
except ZeroDivisionError:
    print("No es posible dividir por cero.")
except ValueError:
```

```
    print("Debes introducir un número entero.")  
else:  
    print(f"El resultado es: {resultado}")
```

En este caso, se gestiona tanto la excepción que se produce al intentar dividir por cero (*ZeroDivisionError*) como la que ocurre al intentar convertir una cadena de texto que no representa un número entero (*ValueError*).

Utilizar un manejo de excepciones adecuado y preciso es esencial para desarrollar software robusto y resistente a errores, así como para proporcionar retroalimentación útil al usuario cuando las cosas no salen según lo previsto.

9. TRATAMIENTO DE FICHEROS EN PYTHON

El manejo de ficheros es una tarea común en la programación y Python proporciona una interfaz sencilla y eficiente para realizar operaciones de lectura y escritura en archivos. Para trabajar con ficheros, Python utiliza un objeto *file*, que se obtiene mediante la utilización de la función *open*. A continuación, se describen brevemente los modos más comunes en los que un archivo puede ser abierto:

- ‘r’: Lectura (default).
- ‘w’: Escritura (sobrescribe si el archivo existe).
- ‘a’: Añadir (escribe al final del archivo si este existe).
- ‘b’: Modo binario.
- ‘x’: Crea el archivo, retorna un error si el archivo ya existe.

9.1. LECTURA Y ESCRITURA EN FICHEROS

Para escribir en un fichero, se utiliza el método *write*. A continuación, se presenta un ejemplo de cómo escribir en un archivo en Python.

```
# Escritura en un fichero en Python  
with open('example.txt', 'w') as file:  
    file.write('Hola, mundo!')
```

El uso de *with* asegura que el archivo se cierre correctamente una vez que se haya terminado con él, incluso si ocurre una excepción durante las operaciones de E/S.

Para la lectura de ficheros, se pueden usar métodos como *read*, *readline* o iterar directamente sobre el objeto *file*. A continuación, un ejemplo práctico.

```
# Lectura de un fichero en Python  
with open('example.txt', 'r') as file:  
    content = file.read()  
    print(content)
```

En este caso, *content* contendrá la totalidad del texto presente en *example.txt* y posteriormente se imprime en la consola.

Es importante manejar las excepciones que pueden surgir durante las operaciones de lectura y escritura de archivos, como *FileNotFoundException* o *IOError*, para garantizar que el programa pueda gestionar de forma adecuada situaciones de error sin provocar un cierre inesperado.

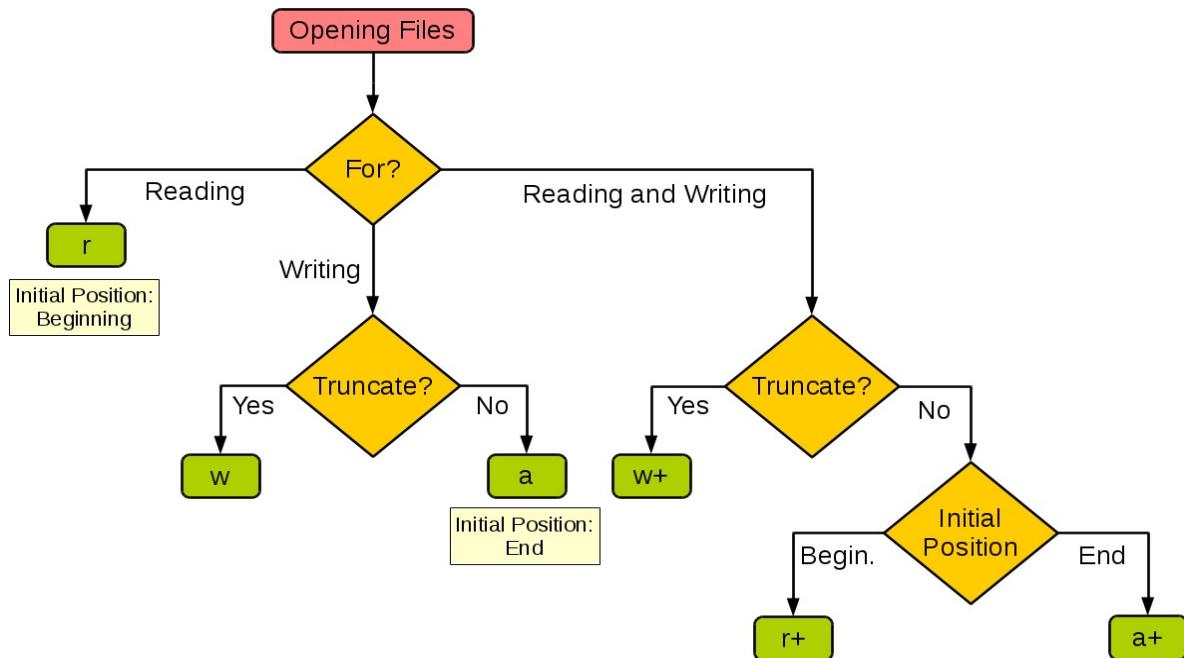


Figura 14: Modos de apertura en Python

9.2. MANEJO DE EXCEPCIONES CON FICHEROS EN PYTHON

En operaciones de E/S (entrada/salida) con ficheros, es fundamental implementar un manejo de excepciones robusto para tratar posibles errores de manera adecuada y asegurar la estabilidad del programa. Las excepciones permiten gestionar situaciones anómalas, como la ausencia del fichero, problemas con los permisos de lectura o escritura, y otros posibles errores relacionados con el fichero durante la ejecución del código.

Excepciones comunes en operaciones con ficheros:

- *FileNotFoundException*: El fichero no puede ser encontrado.
- *PermissionError*: No hay permisos para acceder al fichero.
- *IOError*: Error de E/S general (p.ej., no se puede escribir en un medio de almacenamiento lleno).

A continuación, se presentan ejemplos prácticos sobre cómo se pueden gestionar estas excepciones en operaciones de lectura y escritura de ficheros.

```

try:
    with open('non_existent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("El fichero no ha sido encontrado.")
except IOError:
    print("Error de E/S al leer el fichero.")
else:
    
```

```
print(content)

try:
    with open('read_only_file.txt', 'w') as file:
        file.write('Hola, mundo!')
except PermissionError:
    print("No tienes permiso para escribir en el fichero.")
except IOError:
    print("Error de E/S al escribir en el fichero.")
```

Estos ejemplos ilustran cómo se puede manejar de manera elegante y clara las excepciones asociadas a operaciones con ficheros, proporcionando mensajes de error informativos y previniendo terminaciones inesperadas del programa. Es esencial integrar estrategias de manejo de excepciones efectivas para asegurar que el software es robusto y confiable, especialmente en contextos donde los fallos pueden tener consecuencias importantes.

10. LOS GRÁFICOS EN PYTHON

Matplotlib es una de las bibliotecas más utilizadas en Python para la generación de gráficos a partir de datos contenidos en listas o arrays, tales como gráficos de línea, barras, error, dispersión, etc. Es particularmente poderosa para la creación de visualizaciones estáticas, animadas e interactivas en Python.

- *plot*: Para la creación de gráficos de líneas.
- *scatter*: Para gráficos de dispersión.
- *bar*: Para gráficos de barras verticales.
- *barh*: Para gráficos de barras horizontales.
- *hist*: Para histogramas.
- *pie*: Para gráficos de tarta.

Asegúrate de instalar e importar *Matplotlib* antes de utilizarla, lo cual puede hacerse mediante *pip*:

```
pip install matplotlib
```

10.1. GRÁFICO DE LÍNEAS

Los gráficos de líneas son útiles para representar la evolución de una variable en función de otra. Utilizando el método *plot*:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

plt.plot(x, y)
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title('Gráfico de Líneas')
```

```
plt.show()
```

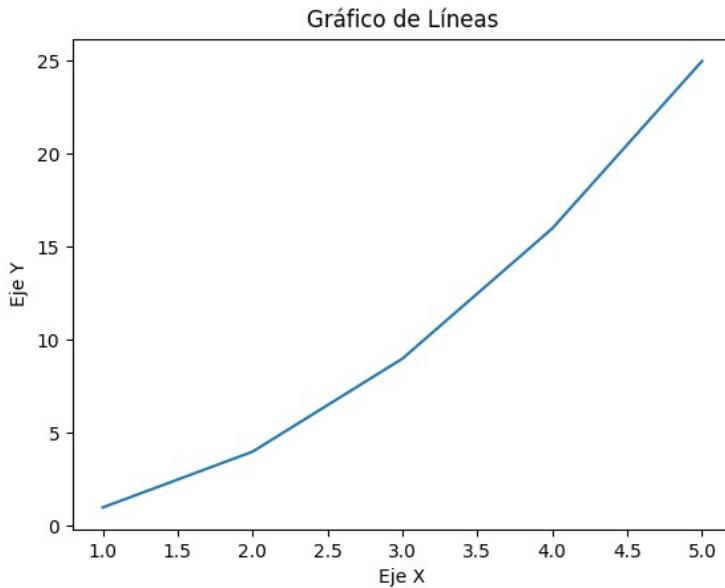


Figura 15: Gráfico de líneas

10.2. USO DE COLORES EN MATPLOTLIB

Matplotlib permite una amplia variedad de opciones para la especificación de colores en los gráficos, facilitando la personalización de las visualizaciones generadas. Podemos especificar los colores usando un nombre de color, un código de color hexadecimal, una tupla RGB, entre otros métodos. A continuación, se describen algunas maneras comunes de cómo se pueden usar los colores en *Matplotlib*.

Especificación de colores por nombre:

Matplotlib reconoce los nombres de los colores y sus respectivas abreviaturas para facilitar su uso. Algunos ejemplos son:

Cuadro 1: Colores en <i>matplotlib</i>		
Nombre	Abreviatura	Color
blue	b	azul
green	g	verde
red	r	rojo
cyan	c	cian
magenta	m	magenta
yellow	y	amarillo
black	k	negro
white	w	blanco

Ejemplo de uso en un gráfico de líneas:

```
import matplotlib.pyplot as plt
```

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'b-')
plt.show()
```

Es posible especificar colores utilizando su código hexadecimal, proporcionando una gran variedad de opciones de colores.

Ejemplo de uso:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], '#4CAF50')
plt.show()
```

Puedes obtener más información sobre la función plot en la documentación:

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

CSS Colors

black	bisque	forestgreen	slategrey
dimgray	darkorange	limegreen	lightsteelblue
dimgrey	burlywood	darkgreen	cornflowerblue
gray	antiquewhite	green	royalblue
grey	tan	lime	ghostwhite
darkgray	navajowhite	seagreen	lavender
darkgrey	blanchedalmond	mediumseagreen	midnightblue
silver	papayawhip	springgreen	navy
lightgray	moccasin	mintcream	darkblue
lightgrey	orange	mediumspringgreen	mediumblue
gainsboro	wheat	mediumaquamarine	blue
whitesmoke	oldlace	aquamarine	slateblue
white	floralwhite	turquoise	darkslateblue
snow	darkgoldenrod	lightseagreen	mediumslateblue
rosybrown	goldenrod	mediumturquoise	mediumpurple
lightcoral	cornsilk	azure	rebeccapurple
indianred	gold	lightcyan	blueviolet
brown	lemonchiffon	paleturquoise	indigo
firebrick	khaki	darkslategray	darkorchid
maroon	palegoldenrod	darkslategrey	darkviolet
darkred	darkkhaki	teal	mediumorchid
red	ivory	darkcyan	thistle
mistyrose	beige	aqua	plum
salmon	lightyellow	cyan	violet
tomato	lightgoldenrodyellow	darkturquoise	purple
darksalmon	olive	cadetblue	darkmagenta
coral	yellow	powderblue	fuchsia
orangered	olivedrab	lightblue	magenta
lightsalmon	yellowgreen	deepskyblue	orchid
sienna	darkolivegreen	skyblue	mediumvioletred
seashell	greenyellow	lightskyblue	deppink
chocolate	chartreuse	steelblue	hotpink
saddlebrown	lawngreen	aliceblue	lavenderblush
sandybrown	honeydew	dodgerblue	palevioletred
peachpuff	darkseagreen	lightslategrey	crimson
peru	palegreen	lightslategrey	pink
linen	lightgreen	slategray	lightpink

Figura 16: Colores css en `matplotlib`

Ejemplo de uso especificando el parámetro `fmt (*-b)`, `linewidth(lw)` y `markerfacecolor(mfc)`:

```
import random
import matplotlib.pyplot as plt

# Tus valores aleatorios ya generados
height = [random.uniform(150, 200) for _ in range(10)]
weight = [random.uniform(50, 100) for _ in range(10)]

plt.title("Alturas vs. Pesos")
plt.xlabel("Alturas (cm)")
plt.ylabel("Pesos (kg)")
plt.plot(sorted(height), weight, \
         "*-b", lw=2, markersize=20, mfc="m")
plt.show()
```

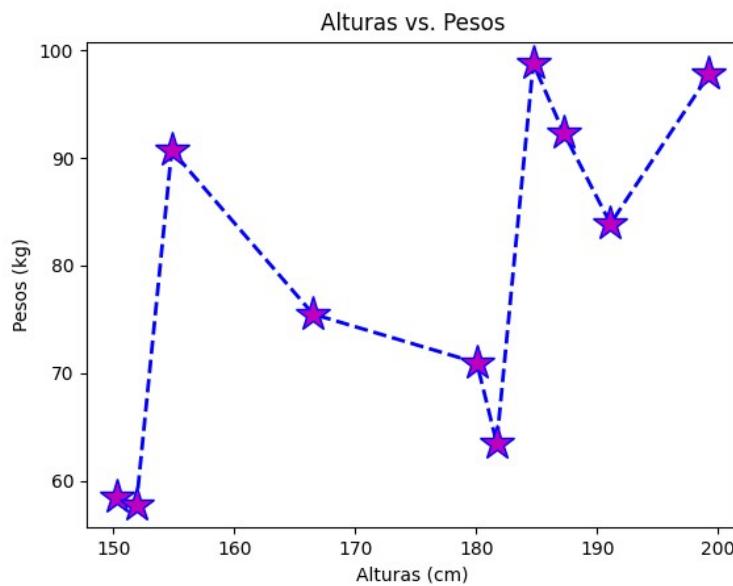


Figura 17: Uso de parámetros con el método `plot`

10.3. GRÁFICOS DE DISPERSIÓN (SCATTER PLOTS) EN MATPLOTLIB

Los gráficos de dispersión, o *scatter plots*, son un tipo de gráfico que utiliza coordenadas cartesianas para mostrar valores de dos variables para un conjunto de datos. En *Matplotlib*, los *scatter plots* son realizados mediante la función *scatter*.

Uso básico de *Scatter Plots*:

La función *scatter* requiere dos argumentos, que corresponden a las coordenadas *x* e *y* de los puntos a dibujar. Cada punto representa una observación en el conjunto de datos.

Ejemplo de uso:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
```

```
y = [2, 3, 5, 7, 11]

plt.scatter(x, y)
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title('Scatter Plot Básico')
plt.show()
```

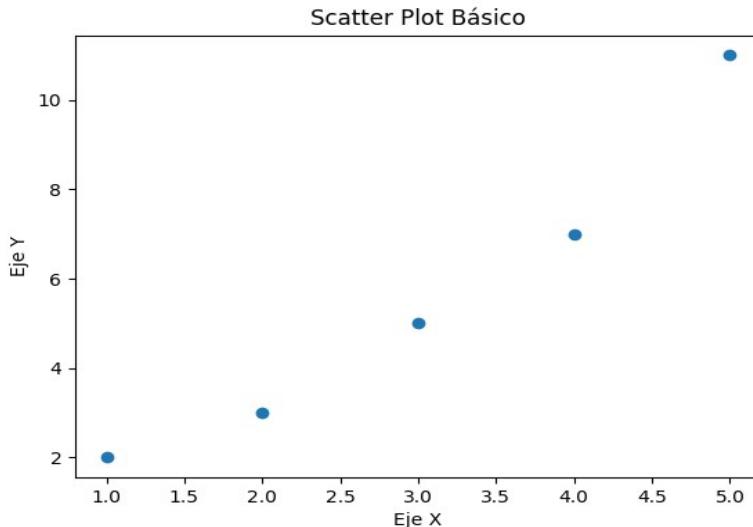


Figura 18: Gráficos de dispersión

Personalización de *Scatter Plots*:

Matplotlib permite una gran variedad de personalizaciones en los *scatter plots*, incluyendo el color, tamaño y forma de los marcadores.

Ejemplo de un scatter plot personalizado:

```
plt.scatter(x, y, color='red', marker='o', s=100, alpha=0.5)
plt.xlabel('Eje X')
plt.ylabel('Eje Y')
plt.title('Scatter Plot Personalizado')
plt.show()
```

Donde:

- *color*: Define el color de los puntos.
- *marker*: Define la forma de los puntos.
- *s*: Define el tamaño de los puntos.
- *alpha*: Define la transparencia de los puntos.

Un ejemplo, un poco más completo, es el que se muestra a continuación:

```
import matplotlib.pyplot as plt
import numpy as np

# Datos de ejemplo
```

```
np.random.seed(42) # para reproducibilidad
x = np.random.rand(100) * 10
y = x * 2 + np.random.randn(100) * 2

# Usar scatter con varios parámetros
plt.scatter(x,
            y,
            c=y,                 # Color basado en el valor de 'y'
            s=x*20,              # Tamaño basado en el valor de 'x'
            alpha=0.6,             # Transparencia,
            edgecolors="w",        # Color del borde de los marcadores,
            linewidth=0.5,         # Ancho del borde de los marcadores,
            cmap='viridis'        # Mapa de color
            )

plt.show()
```

En el ejemplo anterior:

- **c=y**: El color de cada punto se basa en su valor de ‘y’
- **s=x*20**: El tamaño de cada punto se basa en su valor de ‘x’
- **alpha=0.6**: Transparencia de los puntos
- **edgecolors=“w”**: Color blanco alrededor de cada punto para que se destaquen
- **linewidth=0.5**: Ancho del borde blanco alrededor de cada punto
- **cmap=’viridis’**: Especifica el mapa de colores a utilizar

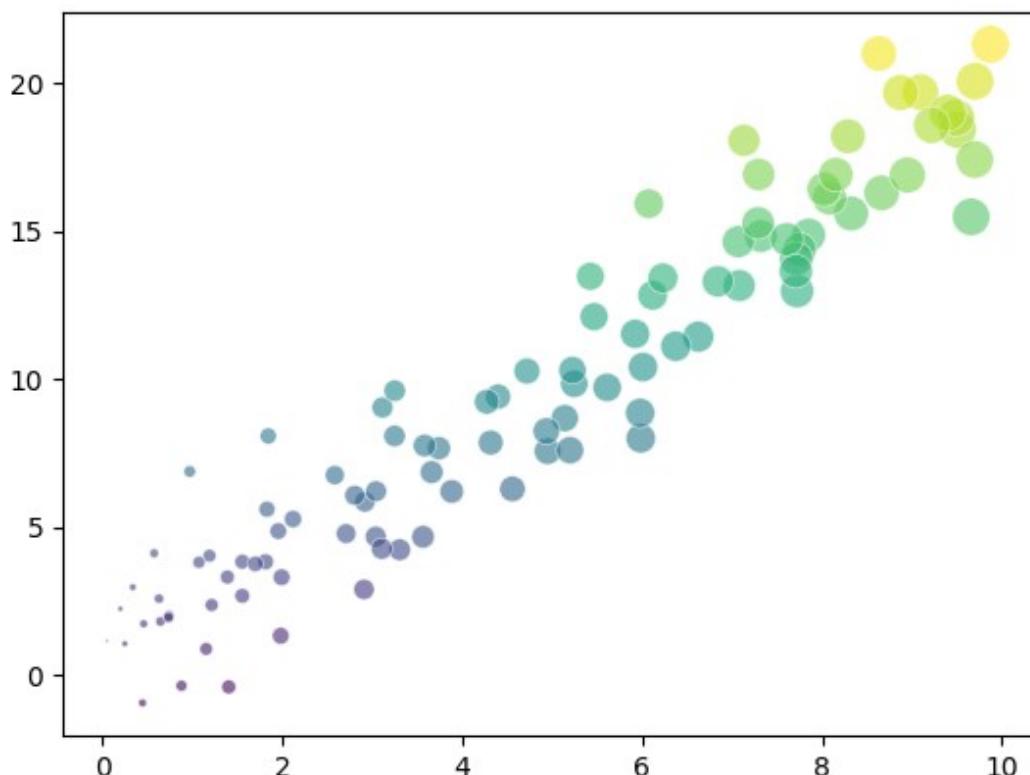


Figura 19: Gráficos de dispersión

10.4. GRÁFICO DE BARRAS

Los gráficos de barras se pueden generar utilizando `bar` para barras verticales o `barh` para barras horizontales.

```
import matplotlib.pyplot as plt

# Datos de ejemplo
labels = ['A', 'B', 'C', 'D', 'E']
values = [3, 7, 2, 5, 8]
error = [0.5, 0.4, 0.3, 0.6, 0.4] # Barras de error para añadir detalle

# Crear gráfico de barras con barras de error
bar_colors = ['red', 'green', 'blue', 'cyan', 'yellow']
bars = plt.bar(labels, values, yerr=error, color=bar_colors,
edgecolor='black', capsized=7)

# Añadir detalles al gráfico
plt.title('Gráfico de barras detallado')
plt.xlabel('Categorías')
plt.ylabel('Valores')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

# Añadir etiquetas de texto en la parte superior de cada barra
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + 0.1, round(yval, 2), ha='center', va='bottom')

# Añadir leyenda
from matplotlib.lines import Line2D
legend_elements = [Line2D([0], [0], color='gray', lw=4, label='Barra'),
                   Line2D([0], [0], color='black', lw=4, label='Borde',
                   linestyle='--'),
                   Line2D([0], [0], color='black', lw=0, label='Barras de error',
                   marker='|', markersize=15, markeredgewidth=1.5)]
plt.legend(handles=legend_elements, loc='upper left')

# Mostrar el gráfico
plt.show()
```

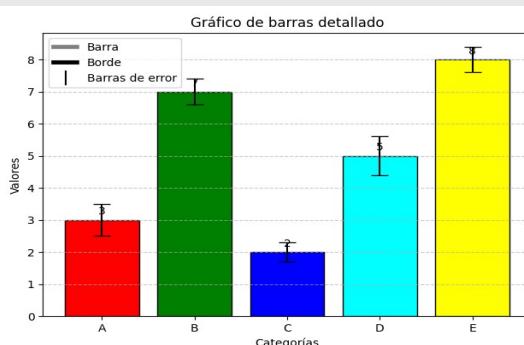


Figura 20: Gráficos de barras

10.5. HISTOGRAMA

El método `hist` es utilizado para la creación de histogramas, que son útiles para visualizar la distribución de un conjunto de datos. Se utiliza para representar distribuciones de datos continuos o cuantitativos. Un histograma divide el rango de valores en una serie de intervalos (o “bins”) y cuenta cuántos datos caen en cada intervalo. Proporciona una vista general de la distribución de una variable numérica (por ejemplo, ¿cómo se distribuyen las edades de un grupo de personas?).

Ejes de un histograma:

- **Eje X:** Representa los intervalos de datos o “bins”.
- **Eje Y:** Representa la frecuencia (número de observaciones) que caen en cada intervalo.

```
import matplotlib.pyplot as plt

data = [2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 6]

plt.hist(data, bins=5, color = "#fe3451", edgecolor='white')
plt.xlabel('Valor')
plt.ylabel('Frecuencia')
plt.title('Histograma')
plt.show()
```

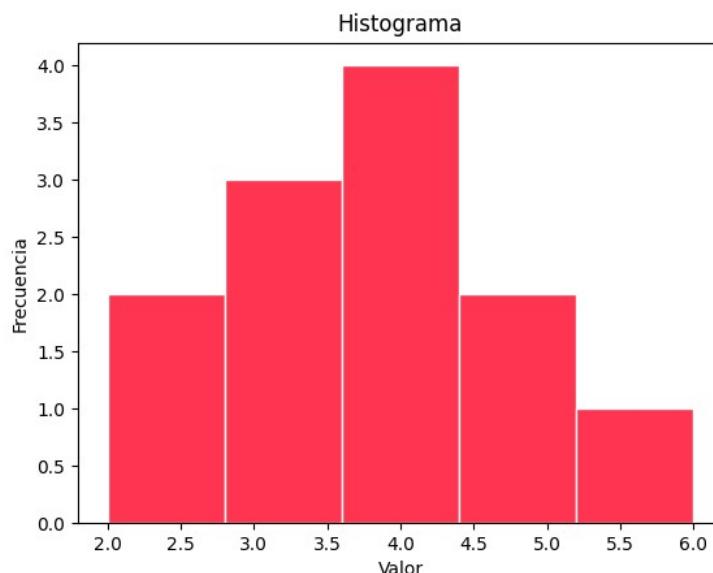


Figura 21: Histograma

10.6. PIECHARTS

Un gráfico de tarta, o *pie chart*, es una representación gráfica circular que divide una tarta o pastel en segmentos proporcionales, representando porcentajes o proporciones del total. *Matplotlib* proporciona la función `pie()` en el módulo `pyplot` para facilitar la creación de estos gráficos.

Veamos un ejemplo básico de cómo crear un gráfico de tarta en *Matplotlib*:

```
import matplotlib.pyplot as plt
# Datos de ejemplo
```

```
etiquetas = ['A', 'B', 'C']
valores = [15, 30, 25]

# Crear gráfico de tarta
plt.pie(valores, labels=etiquetas, startangle=140,
         colors=['red', 'blue', 'green'],
         explode=(0.2, 0, 0.1), shadow = True)

# Título del gráfico
plt.title("Distribución de valores por categoría")

# Mostrar el gráfico
plt.show()
```

En el ejemplo anterior, los principales parámetros de la función `pie()` son:

- `labels`: Lista de etiquetas para cada segmento
- `startangle`: Ángulo de inicio del gráfico de tarta
- `colors`: Lista de colores para cada segmento.
- `explode`: Una tupla que indica cuánto resaltar cada segmento. En el ejemplo, el primer segmento se resalta un poco al alejarlo del centro.



Figura 22: Piechart

Los gráficos de tarta son especialmente útiles para mostrar proporciones relativas y distribuciones, aunque se recomienda su uso con precaución, ya que pueden ser menos precisos que otros tipos de gráficos para comparaciones detalladas entre categorías.

10.7. EL MÉTODO SUBPLOTS

En *matplotlib*, los *subplots* permiten mostrar múltiples gráficos en una sola figura. Estos gráficos pueden estar organizados en una cuadrícula de filas y columnas, permitiendo comparar visualmente diferentes conjuntos de datos o diferentes vistas del mismo conjunto de datos.

La función principal para crear *subplots* en *matplotlib* es *subplots()*, que devuelve una figura y un conjunto de ejes, que corresponden a las áreas individuales del gráfico.

A continuación, se muestra un ejemplo de cómo crear una figura con cuatro subplots:

```
import matplotlib.pyplot as plt
import numpy as np

# Datos de ejemplo
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Crear figura y ejes para subplots
fig, ax = plt.subplots(2, 2) # 2 filas, 2 columnas

# Primer subplot
ax[0, 0].plot(x, y1, 'r')
ax[0, 0].set_title('Seno de x')

# Segundo subplot
ax[0, 1].plot(x, y2, 'b')
ax[0, 1].set_title('Coseno de x')

# Tercer subplot (solo como ejemplo, se duplicará el gráfico del seno)
ax[1, 0].plot(x, y1, 'g')
ax[1, 0].set_title('Seno de x (verde)')

# Cuarto subplot (solo como ejemplo, se duplicará el gráfico del coseno)
ax[1, 1].plot(x, y2, 'y')
ax[1, 1].set_title('Coseno de x (amarillo)')

# Ajustar el espaciado entre subplots
fig.tight_layout()

# Mostrar la figura
plt.show()
```

El método *tight_layout()* ajusta automáticamente el espaciado entre *subplots* para que no se superpongan los ejes y los títulos. Es especialmente útil cuando se tiene un gran número de *subplots* en una sola figura.

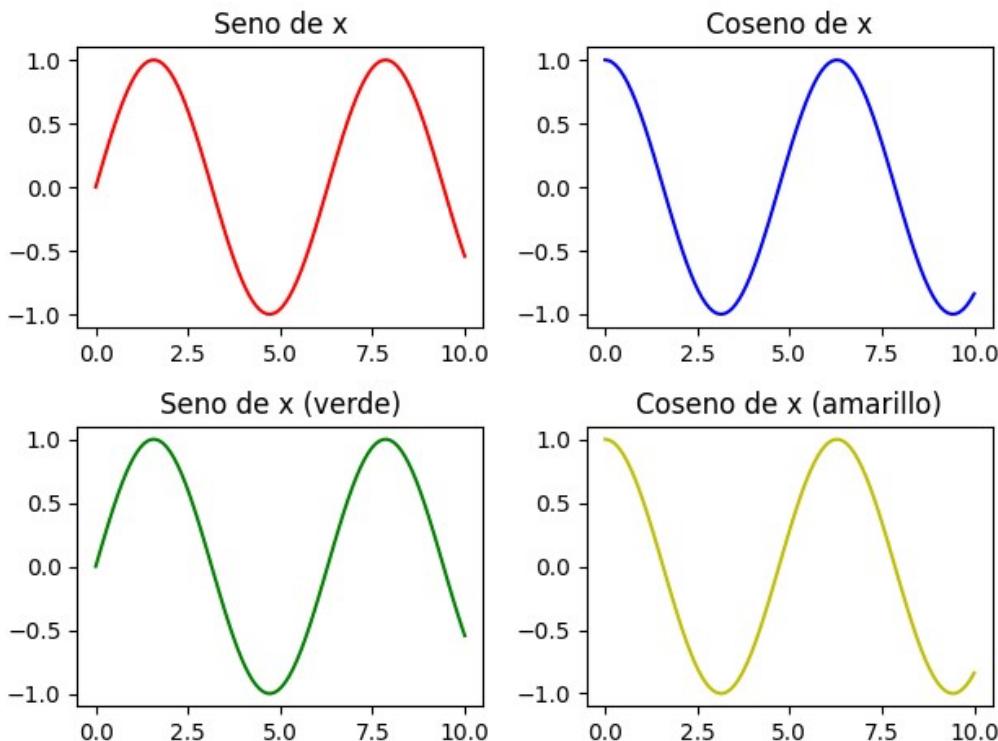


Figura 23: subplots

10.8. VISUALIZACIÓN DE SERIES TEMPORALES CON OBJETOS `DATETIME` EN MATPLOTLIB

La visualización de series temporales desempeña un papel crucial en numerosos campos académicos y profesionales. Estas series, que registran observaciones a través del tiempo, son fundamentales en áreas como la economía, finanzas y ciencias ambientales. La biblioteca *matplotlib* de Python proporciona herramientas robustas para representar gráficamente estas secuencias de datos temporales.

Una serie temporal se define, esencialmente, por dos componentes: un conjunto de marcas temporales y una serie correspondiente de valores. En una representación gráfica, las marcas temporales, por su carácter continuo y ordenado, generalmente se sitúan en el eje x, mientras que los valores observados se representan en el eje y.

Para ilustrar cómo se visualizan las series temporales con *matplotlib* y objetos *datetime*, se detalla a continuación un procedimiento básico:

```
import matplotlib.pyplot as plt
import datetime
import numpy as np

# Datos de ejemplo: generamos una serie de fechas y valores asociados
base = datetime.datetime.today()
num_dias = 100
fechas = [base - datetime.timedelta(days=x) for x in range(num_dias)]
valores = np.sin(np.linspace(0, 10, num_dias))

# Creación del gráfico
```

```

plt.figure(figsize=(10, 6))
plt.plot(fechas, valores, label='Valor observado')
plt.title('Serie Temporal de Ejemplo con Objetos datetime')
plt.xlabel('Fecha')
plt.ylabel('Valor')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.gcf().autofmt_xdate() # Optimiza la presentación de las fechas en el eje x

# Mostrar el gráfico
plt.show()

```

Elementos destacados en la visualización de series temporales con objetos *datetime* incluyen:

- **Objetos temporales:** Utilizar objetos *datetime* permite una representación precisa y una manipulación flexible de las marcas temporales
- **Formato de fecha:** Al trabajar con *datetime*, es recomendable optimizar la presentación de las fechas en el eje x para garantizar legibilidad, lo cual se logra con *autofmt_xdate()*
- **Anotaciones y resaltados:** Las fechas específicas o períodos pueden ser resaltados o anotados para señalar eventos significativos o patrones de interés

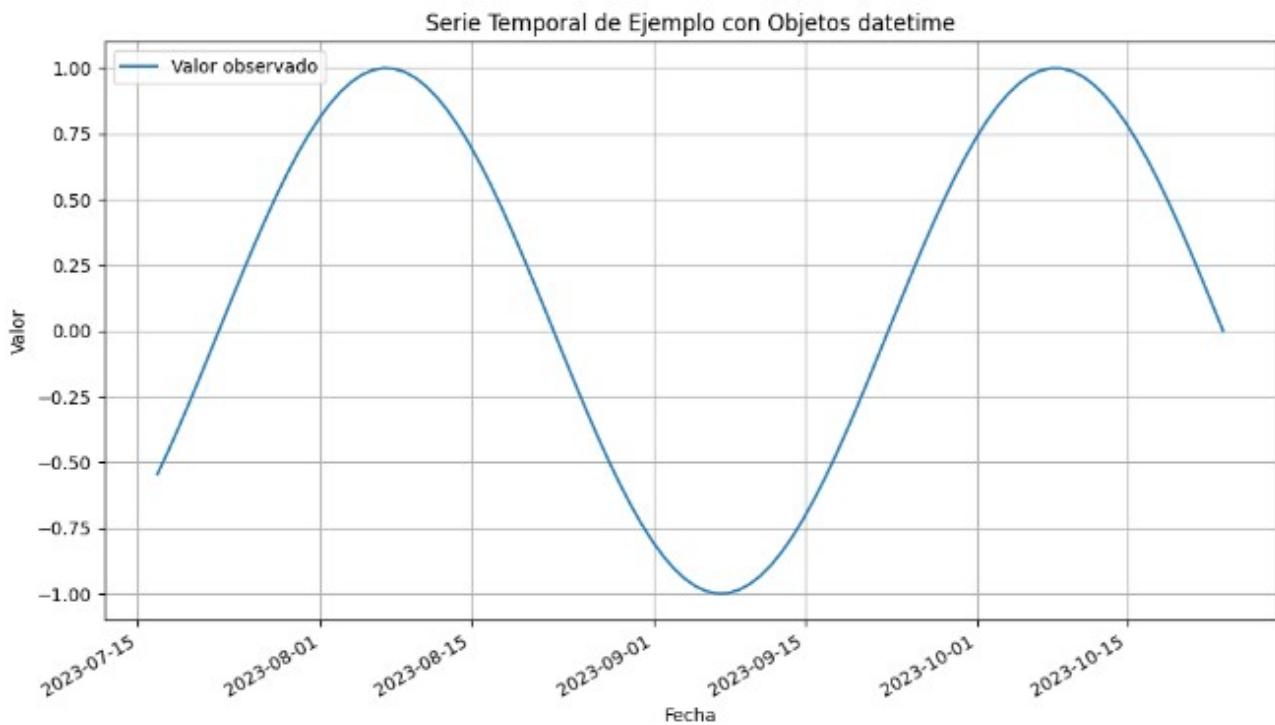


Figura 24: series temporales