# PYTHON

## aplicado al mantenimiento industrial

Vanessa Lorena Valverde González
Félix Antonio García Mora
Eduardo Segundo Hernández Dávila

CIDE
EDITORIAL

# PYTHON

## aplicado al mantenimiento industrial

# PYTHON

## aplicado al mantenimiento industrial

**AUTORES:**

**Vanessa Lorena Valverde González**
**Félix Antonio García Mora**
**Eduardo Segundo Hernández Dávila**



espoch

Dirección de Publicaciones

Centro de Investigación y Desarrollo Ecuador
Tel.: + (593) 04 2037524
http.:/www.cidecuador.org

ISBN: 978-9942-636-33-1

https://doi.org/10.33996/cide.ecuador.PA2636331

**Filiación:**
Vanessa Lorena Valverde González
Félix Antonio García Mora
Eduardo Segundo Hernández Dávila
**Escuela Superior Politécnica de Chimborazo**

**Dirección editorial:** Lic. Pedro Misacc Naranjo, Msc.
**Coordinación técnica**: Lic. María J. Delgado
**Diseño gráfico**: Lic. Danissa Colmenares
**Diagramación**: Lic. Alba Gil
**Fecha de publicación**: octubre, 2023

La presente obra fue evaluada por pares académicos experimentados en el área.

**Catalogación en la Fuente**

Python. Aplicado al mantenimiento industrial / Vanessa Lorena Valverde González, Félix Antonio García Mora, Eduardo Segundo Hernández Dávila - Ecuador: Editorial CIDE, 2023.

244 p.: incluye tablas, figuras; 17,6 x 25 cm.

ISBN: 978-9942-636-33-1

# Índice

# Capítulo 3
# Manejo de datos

# Prólogo

El libro *"Python aplicado al mantenimiento industrial"* es un compendio de experiencias de los autores y recoge todo el recorrido que han amasado durante todo el tiempo en el que han aplicado el ejercicio docente. Cada autor aporta con su experiencia y relata de manera didáctica y simple, los conceptos y ejemplos que han aplicado en sus aulas de clases.

El libro está compuesto de tres capítulos bien organizados y agrupados con temáticas que abordan los ejes centrales que requiere el lector para seguir una línea que le permita capturar la esencia del documento y así lograr que el conocimiento se compacte y llega a entender de manera progresiva los objetivos del mismo.

En esta obra se abordan temas respecto a cómo la herramienta de programación Python se ajusta a las necesidades del proceso de mantenimiento industrial, aplicándolo a un caso de estudio, pero abriendo la puerta para que el lector pueda

aplicarlo en cualquier problema que se presente tanto en su vida académica como profesional.

El primer capítulo está orientado a iniciar al lector en la herramienta de desarrollo de *software Python* exponiendo de manera práctica, las características básicas de la herramienta, explicando variables, operadores lógicos, aritméticos, sentencias condicionales, ciclos, *arrays*, archivos, entre otros; todos los elementos fundamentales que el lector requiere para empezar a aplicar en sus programas o aplicaciones más adelante.

El capítulo dos se centra más en el manejo y creación de funciones; tan importante es capturar la escencia de estos temas que los autores de una manera simple para que el lector de un modo independiente se convierta en el hacedor de su propio conocimiento ya que aplicando los conceptos y ejercicios que el capítulo brinda, podrá crear recursos permanentes sobre lo que ha leído.

El último capítulo, permite interactuar al lector con un caso práctico utilizando todo lo que sistemáticamente venía aprendiendo en los apartados anteriores. Permite ir paso a paso

codificando el proceso de mantenimiento de un mecanismo motorizado como es el caso de una trituradora de madera; al final de la aplicación de todo el procedimiento ofrece una estimación del posible fallo que puede ocurrir, permitiendo de esta manera a la gerencia tomar las debidas decisiones para evitar inconvenientes en el futuro.

Este libro se convierte en una herramienta didáctica y práctica para que el lector pueda a partir de esta enseñanza, crear su propio conocimiento empleando esos criterios y conceptos, así como también, generar nuevas aplicaciones en el ámbito donde se esté desenvolviendo ya sea el académico o profesional.

*Gustavo X. Hidalgo Solórzano.*
**Ingeniero en Sistemas Informáticos.**
**Dirección de Tecnologías de la Información,**
**Comunicación y Procesos.**
**DTIC-ESPOCH**

# Introducción

En la actualidad, la tecnología está en todos los ambientes laborales, las computadoras se las emplea para diferentes áreas, lo que suscita, que conocer acerca de generación de código a pequeña o gran escala, sea una obligación y necesidad para cualquier ambiente laboral.

Las tecnologías de la comunicación y computación son parte de la ciencia conocida como informática; al ser una ciencia, es un requisito que los profesionales actuales tengan la necesidad de conocer la generación de código, es por esto que el presente libro está dirigido a las personas que necesitan continuar con la generación de código en el lenguaje de programación en *Python*, para la manipulación de datos.

El objetivo de este libro es brindar la información necesaria para el manejo de datos a través de estructuras, funciones y manejo de datos orientados a actividades del mantenimiento, implementado el código en *Python*, dirigido a diferentes tipos de lectores que tenga la necesidad de conocer un poco más sobre el

manejo de datos de forma digital, aunque los ejemplos estén orientados a conceptos o términos del Ingeniero en Mantenimiento Industrial, sus ejemplos se pueden implementar según sea la necesidad de docentes, estudiantes o profesionales de otras áreas.

Es por tal motivo, que el Capítulo 1 trata sobre las estructuras de datos, comenzando con una introducción de conceptos básicos necesarios para la programación, tales como: variables, tipos de datos que maneja el lenguaje, operadores necesarios para la programación, recordando también las sentencias condicionales necesarias para realizar consultas y sentencias cíclicas o repetitivas; una vez revisado los conceptos básicos se procede al desarrollo de las estructuras, como: *Listas, Tuplas, String Set, Diccionarios, Archivos y Data Frame*.

En el Capítulo 2 se ingresará en el fascinante mundo de las funciones en *Python;* estas son un componente esencial de la programación permitiendo modularizar el código, reutilizarlo y hacerlo más legible. A lo largo de este apartado, se explorará en detalle qué son las funciones y por qué son tan valiosas en el desarrollo de software.

En este sentido, se comenzará por definir qué es una función y se explorará las numerosas ventajas que ofrecen en la escritura de programas. Conocer la sintaxis básica para crear funciones en *Python* y aprender sobre los conceptos de parámetros y argumentos, incluyendo los diferentes tipos de argumentos que se pueden utilizar en las funciones.

Además, profundizar en el retorno de valores y cómo utilizar la sentencia "*return*" para devolver resultados desde una función. Explorar casos de retorno de valores múltiples y ver cómo funcionan las funciones que no tienen un retorno explícito.

En cuanto al contenido del Capítulo 3, el mismo se centra en la utilización de funciones específicas de *Python*, utilizadas en las matemáticas, estadística, análisis de datos y elaboración de gráficos como *numpy, pandas, scipy, sklearn, matplotlib y seaborn*, que combinadas con el empleo de las estructuras recursivas y las funciones anónimas y de orden superior, se desarrolla un caso práctico de análisis y tratamiento de datos orientados a la creación de estrategias y toma de decisiones en el campo de la Ingeniería del Mantenimiento Industrial.

Cada uno de los capítulos propone una lista de ejercicios propuestos, para que lector practique y continúe con la adquisición de conocimientos en el área de la programación. La importancia de este libro es que está orientado a personas que desean continuar fortaleciendo sus habilidades en la programación, estudiantes que cursan la asignatura de estadística o personas en general que necesitan conocer el manejo de datos de manera digital.

1

# Capítulo 1
## Estructuras de datos

Capítulo 1

# Estructuras de datos

## 1. Conceptos básicos

*Python* es un lenguaje de alto nivel multiparadigma, esto significa que tiene programación orientada a objetos y la hace muy fácil; es un lenguaje interpretado que le da la ventaja de ser flexible, lo que ayuda a no tener que declarar a una variable, es decir, a una misma variable se le puede asignar diferentes tipos de datos.

*Asimismo, "Python* es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un enfoque simple pero efectivo para la programación orientada a objetos. La sintaxis elegante y la tipificación dinámica" (*Python*, 2023, párr. 1).

Ejemplo de uso de una variable:

Código:

```
#asignando un texto a la variable
nombre=input("Ingrese su nombre")
print(nombre)
#asignando un número a la variable
nombre=123
print(123)
```

En el ejemplo se puede observar que la variable **nombre** no ha sido previamente declarada, sino que directamente se le utiliza para la asignación de la información ingresada por el usuario a través de la función **input,** una vez que se procede a la impresión de la variable a través de la función **print,** se le asigna a la variable **nombre** la cantidad **123** por lo que ahora la variable ya no contiene texto sino un tipo de dato entero. Este es el ejemplo de lo que ofrece *Python* al ser un lenguaje interpretado.

"De la misma manera en lenguaje de programación el tipado es dinámico, esto quiere decir que se pueden manejar diferentes tipos de datos sin que tengan que especificarse" (Trejos y Muñoz, 2021, p. 14).

Ahora bien*, Python* es multiplataforma, es decir que puede ser ejecutado en tonos operativos como *Linux, Windows o MacOS*, también se puede utilizar para programar sistemas móviles, aplicaciones de escritorios.

Para proceder a generar código es muy importante la utilización de operadores aritméticos, en especial cuando se desea tratar la solución de fórmulas o realizar algún proceso matemático, es por esto que en la Tabla 1.1 se muestra cada uno de los operadores que se puede implementar en *Python*.

**Tabla 1.1**

*Operadores aritméticos.*

| Signo | Descripción | Proceso |
|---|---|---|
| + | Suma | Suma=a+b |
| - | Resta | Resta=a-b |
| * | Multiplicación | Multi=a*b |
| / | División | Divi=a/b |
| // | Divide y da el resultado la parte entera | Entera=a//b |
| % | Devuelve el residuo de la división | Residuo=a&b |
| ** | Potencia | Potencia=a**b |
| = | Asignación | A=123 |

*Nota.* Los autores

Identificar cada tipo de dato que se va a implementar es muy importante para poder controlar el valor que se desea trabajar a la hora de la generación del código, es por esto que en la Tabla 1.2 se muestran los tipos de datos que se pueden aplicar en este lenguaje de programación.

**Tabla 1.2**

*Tipos de datos*.

| Signo | Descripción | Proceso |
|---|---|---|
| int | Entero | num=123 |
| float | Valores con punto decimal | num=12.3 |
| str | Cadena | Nombre="Mantenimiento Correctivo" |
| bool | Booleano | verdadero=True |
| complex | Complejo | Comp= 5j+2j |
| none | Vacío | nada=None |

*Nota.* Los autores

*Python* facilita funciones para mostrar y recibir información, en la Tabla 1.3 se identifican las funciones de entrada y salida de información para interactuar con el usuario:

**Tabla 1.3**

*Ingreso y salida de datos*.

| Signo | Descripción | Proceso |
|---|---|---|
| print | Salida | Print(“Este es un ejemplo”) |
| input | Entrada/Salida | num=(“Ingrese un número”) |

*Nota.* Los autores

En *Python* se puede implementar comillas simples o comillas dobles para mostrar información; para colocar una línea en comentario se utiliza el signo **#**, y para colocar un bloque en comentario, se utiliza tres comillas al inicio del bloque y tres comillas al final del bloque.

Cada vez que se necesita realizar una consulta, pregunta o plantear una condición es necesario manejar el tipo de dato booleano, que emplea dos estados: verdadero o falso. En la Tabla 1.4 se muestra su implementación:

**Tabla 1.4**

*Ingreso y salida de datos*.

| Signo | Descripción | Uso |
|---|---|---|
| and | y | (123==123) and (verdad=True) |
| or | o | (123==123) or (verdad=True) |
| not | negación | Not(True) |

*Nota.* Los autores

Si se necesita implementar condiciones y tener resultados booleanos es necesario utilizar en estas consultas, los operadores lógicos. En la Tabla 1.5 se facilitan algunos posibles ejemplos de sus usos:

**Tabla 1.5**

*Operadores lógicos*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| > | Mayor | 5>4 | True |
| < | Menor | 5<4 | False |
| >= | Mayor igual | 5>=4 | True |
| <= | Menor igual | 5<=4 | False |
| == | Igual | 5==4 | False |
| != | Diferente | 5!=4 | True |

*Nota.* Los autores

## 1.1 Sentencias condicionales y cíclicas

Como en todo lenguaje de programación, *Python* contiene sentencias condicionales como son el **if, if-else, if-elif-else,** además, a partir de la versión 10, se puede implementar la sentencia **match.**

Para proceder a implementar condicionales con ejemplos de Mantenimiento Industrial, se tomará el concepto de Tipos de

Mantenimiento según UNE EN 13306 (2018) quien indica que, "Correctivo: son los que se aplican una vez aparecida la falla. Preventivo: los que tratan de prevenir las consecuencias de las fallas, Mejorado: los que tratan de mejorar la fiabilidad, mantenibilidad y seguridad" (p. 16).

Tomando estas definiciones, se plantea el ejemplo de la sentencia **if-elif-else** aplicada al mantenimiento:

A partir de un menú se le mostrará al usuario que tipo de mantenimiento deberá implementar:

Código:

```
print("Escoja la opción de que se adapta más a su situación")
print("1. Son los que se aplican una vez aparecida la falla")
print("2. los que tratan de prevenir las consecuencias de las fallas")
print("3. los que tratan de mejorar la fiabilidad, mantenibilidad y
seguridad")
falla=input()

if falla=="1":
    print("Aplicar el Mantenimiento Correctivo")
elif falla=="2":
    print("Aplicar el Mantenimiento Preventivo")
elif falla=="3":
    print("Aplicar el Mantenimiento Mejorado")
else:
    print("No seleccionó ninguna de las opciones")
```

En el ejemplo se procede a mostrarle al usuario un menú de opciones de las cuales sólo podrá seleccionar una. La sentencia comienza por el **if**, si esta condición se cumple, entonces ingresará y mostrará que el usuario tendrá que realizar un mantenimiento correctivo, sino es así, pasará la sentencia a consultar a través del **elif** si se cumple la condición; si no se cumple, pasará al siguiente **elif** y si de igual forma la condición no es verdadera, entonces ingresará al **else** donde mostrará en pantalla que el usuario no ingresó ninguna de las opciones que se le mostró en el menú.

La función **input** recibe texto es por eso que en cada condición se le coloca al número entre comillas, para así proceder a realizar la comparación de texto con texto; si se deseara comparar número en la condición se debe obviar las comillas a los números y en la función **input** se debe proceder a realizar una conversión de texto a número, por ejemplo:

Código:

```
falla=int(input())
```

En la línea de código se realiza la conversión de texto a entero.

A partir de *Python* versión 10 se puede trabajar con la sentencia condicional **match**, el ejemplo anterior de los tipos de mantenimiento se lo muestra con la sentencia **match**:

Código:

```
print("Escoja la opción de que se adapta más a su situación")
print("1. Se ha implementado y se acaba de generar la falla")
print("2. Aún no se muestra ninguna falla")
print("3. Intenta eliminar la falla")
falla=int(input())

match falla:
    case 1:
        print("Aplicar el Mantenimiento Correctivo")
    case 2:
        print("Aplicar el Mantenimiento Predictivo")
    case 3:
        print("Aplicar el Mantenimiento Modificativo")
    case _:
        print("No seleccionó ninguna de las opciones")
```

La sentencia match, es un equivalente a la sentencia condicional **if-elif-else,** pero resulta más práctico a la hora de implementar menús en la programación.

Ejemplo, aplicando MTBF:

Código:

```
print("Cálculo del MTBF entre dos equipos")
horas_operativas=float(input("Ingrese el número de horas operativas de la máquina"))

print("Ingreso de Datos del primer equipo")
horas1=float(input("Ingrese las horas que estuvo inactivo"))
fallas1=float(input("Ingrese el número de paradas"))
MTBF1=horas_operativas-horas1/fallas1

print("Ingreso de Datos del segundo equipo")
horas2=float(input("Ingrese las horas que estuvo inactivo"))
fallas2=float(input("Ingrese el número de paradas"))
MTBF2=horas_operativas-horas2/fallas2


if MTBF1>MTBF2:
    print(f"MTBF de la máquina 1 es: {fallas1} y el de la máquina 2 es: {fallas2} ")
    print("Al obtener la máquina 1 un mayor valor hace más fiable el funcionamiento de la máquina")
elif MTBF1==MTBF2:
    print(f"MTBF de la máquina 1 es: {fallas1} y el de la máquina 2 es: {fallas2} ")
    print("Al obtener ambas máquinas el mismo valor se deberá calcular la desviación stándar de los datos")
else:
    print(f"MTBF de la máquina 1 es: {fallas1} y el de la máquina 2 es: {fallas2} ")
    print("Al obtener la máquina 2 un mayor valor hace más fiable el funcionamiento de la máquina")
```

En el ejemplo se emplea una conversión en el **input** de texto a tipo de dato real, además se procede en los **print** a colocar un reformateado con el formato **f**, que permite introducir al texto del **print** una operación o variable, también aplica para la función **input**.

En *Python* también se pueden implementar sentencias repetitivas como son la **while** y **for.**

Sentencia **while**: permite realizar una condición o pregunta que mientras está sea verdadera ingresará al bloque de la sentencia su estructura la siguiente:

Código:

```
while True:
    #bloque de instrucciones
```

Siempre hay que tomar en cuenta que si por un cierto tiempo no se modifica la condición entonces se volverá un ciclo infinito.

Código:

```
numero_equipos=int(input("Ingrese el número de equipos que desea calcular el MTBF"))
i=1
while numero_equipos>0:

    print(f"Cálculo del MTBF del equipo: {i} ")
    horas_operativas=float(input("Ingrese el número de horas operativas de la máquina"))

    horas1=float(input("Ingrese las horas que estuvo inactivo"))
    fallas1=float(input("Ingrese el número de paradas"))
    print(f"El valor del MTBF de la máquina {i} es: {horas_operativas-horas1/fallas1}")
    print()

    numero_equipos=numero_equipos-1
    i=i+1
```

En el ejemplo se ha implementado la misma fórmula para calcular el MTBF pero implementado con un **while**; se solicita al usuario que ingrese la cantidad de máquinas de las cuales desea realizar el cálculo del MTBF, la sentencia **while** procederá a la repetición del ciclo mientras se cumpla la condición es decir, que cuando el valor del número de máquinas llegue a cero se saldrá del ciclo, pero mientras se cumpla la condición este se repetirá; dentro de la sentencia se ha colocado las solicitudes de ingreso de información para proceder a aplicar la fórmula, cada recorrido

dentro del ciclo se decrementa el número de máquinas ingresado por el usuario y se incrementa un contador de nombre **i** que sirve para indicar en cada recorrido del **input** y salida del **print**, el número de máquina con la que se está trabajando.

Calculando MTBF con la sentencia for:

Código:

```
numero_equipos=int(input("Ingrese el número de equipos que desea calcular el MTBF"))

for i in range(numero_equipos):

    print(f"Cálculo del MTBF del equipo: {i+1} ")
    horas_operativas=float(input("Ingrese el número de horas operativas de la máquina"))

    horas1=float(input("Ingrese las horas que estuvo inactivo"))
    fallas1=float(input("Ingrese el número de paradas"))
    print(f"El valor del MTBF de la máquina {i+1} es: {horas_operativas-horas1/fallas1}")
    print()
```

En el ejemplo mostrado, se ha implementado el cálculo del MTBF pero con la sentencia **for,** con lo cual se puede apreciar que ha eliminado tres líneas de código a diferencia de la implementación del mismo ejercicio con una sentencia **while;** la

sentencia **for** se ejecutará siempre que el contador **i** tenga valores que tomar; este conteo empieza a partir del valor cero, es por este motivo que en las funciones **input** y **print** se ha tenido que sumarle el valor de uno a cada recorrido de este bloque, para dar la impresión que el conteo del contador **i** es desde uno.

El ciclo **for** es una sentencia iterable, la misma que tiene distintas formas de implementarlo. A continuación, en la Tabla 1.6 se muestra algunos de los más usados:

**Tabla 1.6**

*Iteraciones con for*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| range() | Rango | for i in range(4) | Recorre 4 veces y el contador **i** toma los valores 0,1,2,3 |
| [1,2,3] | Lista | for i in [1,2,3] | Recorre 3 veces y el contador **i** toma los valores 1,2,3 |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| "Hola" | Cadena | for i in "Hola" | Recorre 4 veces y el contador **i** toma los valores H,o,l,a |
| zip | dos iterables | for i, j in zip([1,2,3],["a","b","c"]) | Recorre 3 veces y el contador **i** toma el valor 1 mientras que el contador j toma el valor *"a"* y así sucesivamente. |
| for.. else.. | ciclo con condicional | for i in [1,2,4]:<br>if (i==3):<br>break<br>else: | Recorre 3 veces, el contador **i** nunca toma el valor de 3 por lo que se ingresará al **else**. |

*Nota.* Los autores

Por lo general se utiliza la función **range** en una sentencia **for** para indicar que se requiere que la variable tome valores en forma ascendente hasta que llegue al valor que se le indica restado uno. Por ejemplo:

Código:

```
for i in range(2):
```

En este caso la variable i tomará valores de cero hasta uno, pero **range** con un incremento de uno**.**

Código:

```
for i in range(2,6):
```

La variable **i** toma valores desde dos hasta cinco con un incremento de uno, que está por defecto, *Python* lo sobreentiende así el programador no lo haya colocado.

Código:

```
for i in range(2,10,2):
```

La variable **i** toma valores desde 2 hasta 8 con un incremento de dos.

La sentencia **for** facilita que en un solo ciclo se pueda implementar más de un contador y más de una iteración, lo cual ayuda a reducir código.

Ejemplo de la sentencia **for** con uso de la función **zip**, en el ejemplo se va a recorrer dos listas predefinidas; la primera lista

tiene registrado tres herramientas para el mantenimiento hospitalario y en la segunda lista se tiene información de cada herramienta.

Los conceptos de cada alicate fueron tomados de Rebollar (2021) donde indica cada concepto de la siguiente forma "Alicate universal: sujetar, cortar o moldear, Alicate de punta plana: sujetar y hacer torsión, Alicate puntas redondas: cortar, posicionar y dar forma a cables" (párr. 4).

Código:

```
herramientas=["alicate universal","alicate de punta plana", "alicate punta redonda" ]

usos=["sujetar,cortar,moldear","sujetar y hacer torsión","cortar, posicionar y dar forma a los cables"]

for i,j in zip(herramientas,usos):
    print("Herramienta ", i, " uso: ", j)
```

Para la implementación de la sentencia **for** con un **else,** se realizará una lista que contendrá una cantidad de motores, si de esta lista no existe coincidencia con el motor ingresado por el usuario entonces se pasará al **else** para continuar con la sentencia y posterior salir de la misma.

Código:

```
motores=["motor alternativo", "Motor monocilindrico", "motor de doble
cilindro"]

for i in motores:

    if i=="motor de doble cilindro":
        print(f"No se ha implementado el mantenimiento del motor {i}")
        break
    print(f"se ha generado el mantenimiento del {i}")

else:

    print("Todo los procesos 100% ejecutados")
```

En el ejemplo se ha procedido a implementar una lista con tres tipos de motores; la sentencia **for** recorre la lista hasta que en la sentencia condicional se cumpla la igualdad, en este caso ingresará al condicional **if** donde muestra por pantalla que no se ha implementado información para dicho motor y ejecuta el **break** con lo que rompe el ciclo, si acaso no existiera coincidencia al trabajarlo con una lista dinámica entonces pasaría a la sentencia **else.**

## 1.2 Estructuras de almacenamiento

### 1.2.1 Listas

Las estructuras de almacenamiento, permiten el almacenamiento de varios tipos de datos, se les conoce como listas. Las listas pueden almacenar muchos tipos de datos a diferencia de la variable cuya capacidad es sola para una.

Se puede trabajar con la lista de dos formas, asignándoles directamente el conjunto de datos, inicializándola sin ningún dato como, por ejemplo:

Código:

```
lista=[]

lista1=["Mantenimiento",123,"1",lista]
```

En el primer ejemplo se ha procedido a generar una lista; primero una lista vacía a través de los **[ ]** que no contienen información y se le asigna a la variable de nombre **lista.** Para el segundo caso se ha procedido a generar una lista con 4

elementos, donde el primer elemento es un texto **"Mantenimiento"**, el segundo elemento es un número de tres cifras **123**, como tercer elemento es un carácter "**1"** y como cuarto elemento es una lista vacía **lista**.

Lo más común en el ámbito de la programación es trabajar con lista dinámicas, es decir que el usuario en tiempo de ejecución agregue la información; para poder trabajar de esta manera, *Python* ofrece una serie de funciones como se indica en la Tabla 1.7.

**Tabla 1.7**

*Propiedades para listas*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| append | anexar | lista.append(5) | Se agrega a la lista el valor de 5. |
| extend | extendido | lista.extend([1,2,3]) | Agrega al final de la lista, cada uno de los valores como elementos independientes. |
| insert | insertar | lista.insert(2,100) | Insertará en la posición 2 de la lista el valor de 100. |
| pop | remover | lista.pop(1) | Elimina el elemento en la posición 1 de la lista. |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| remove | remover | lista.remove (“Mantenimiento”) | Elimina de la lista la primera coincidencia que encuentre con el texto ingresado en la función. |
| del | eliminar | del lista[2] | Elimina el elemento en ubicado en el índice 2 de la lista. |
| clear | limpiar | lista.clear() | Elimina todos los elementos de la lista. |
| + | concatenar | lista3=lista1+lista2 | Los elementos de la lista 1 con los elementos de lista 2, formarán una nueva listas 3. |
| * | duplicar | lista=lista*2 | Los elementos de la lista se van a duplicar. |
| reverse | invertir | lista.reverse() | Invierte el contenido de una lista. |

*Nota.* Los autores

Ejemplo de registro de trabajadores que forman parte de una empresa para el área de mantenimiento:

Código:

```
lista_trabajadores=[]

num=int(input("Ingrese la cantidad de trabajadores que desea registrar"))

for i in range(num):
    lista_trabajadores.append(input(f"Ingrese el número de cédula del trabajador {i+1} : ").upper())
    lista_trabajadores.append(input(f"Ingrese el nombre del trabajador
```

```
{i+1} : ").upper())
    lista_trabajadores.append(input(f"Ingrese el apellido del trabajador
{i+1} : ").upper())
    lista_trabajadores.append(input(f"Ingrese la dirección del trabajador
{i+1} : ").upper())
    lista_trabajadores.append(input(f"Ingrese el correo del trabajador {i+1}
: ").upper())

i=0
for j in range (num):
    print (f"Datos del trabajador {j+1}")
    print(lista_trabajadores[i])
    i=i+1
    print(lista_trabajadores[i])
    i=i+1
    print(lista_trabajadores[i])
    i=i+1
    print(lista_trabajadores[i])
    i=i+1
    print(lista_trabajadores[i])
    print()
    i=i+1
```

En el ejemplo se inicializa una lista de trabajadores vacía con el nombre **lista_trabajadores**, se solicita al usuario las cantidades de registros que desea ingresar y procede al llenado de la lista con la sentencia **for**, en la función **input** se ha procedido a agregar la función **upper()**, función que cambia el ingreso a mayúscula. Una vez llenada la lista con la información requerida, se procede a mostrar por pantalla cada uno de los datos ingresados por registro, es aquí donde se procede a

mostrar con un contador adicional **i** que incremente con cada posición de la lista; en este ejemplo se imprime la lista recorrida por posición, es decir manejo de índex o índice.

Existen ocasiones que no sólo se desea manejar una simple lista, sino que se desea manejar una lista que contenga otra lista; a este concepto se le conoce como *lista anidada* o en la programación tradicional *matriz*.

Por ejemplo, si un ingeniero de mantenimiento desea generar una tabla con cierta información del mantenimiento de alguna máquina, esta información la puede almacenar en una matriz o en *Python* en una lista anidada.

Código:

```
matriz=[]

num=int(input("Ingrese la cantidad de máquinas que desea registrar"))

mayor=0

for i in range(1):
    lista=[]
    for j in range(1):
        lista.append("N.M")
```

```
        lista.append("N.I")
        lista.append("N.A")
        lista.append("TOTAL")
    matriz.append(lista)

for fila in matriz:
    print(fila)

for i in range(1,num+1):
    lista=[]
    print()
    print("**************************************************")
    print(f"Ingrese los datos de la máquina {i}")
    for j in range(1):
        lista.append(input("Ingrese el nombre de la máquina"))
        lista.append(int(input("Ingrese el nivel de interés (1-10)")))
        lista.append(int(input("Ingrese el nivel de análisis (1-10)")))
        lista.append(lista[1]+lista[2])
    valor=lista[3]
    if mayor<valor:
        mayor=valor

    matriz.append(lista)

for fila in matriz:
    print(fila)

print("La mauina con prioridad más alta y que se debe intervenir es la
que tiene un total de:",mayor)
```

Para poder realizar la lista que contenga otras listas de la misma dimensión, se procede a generar una lista vacía de nombre **matriz** para crear otra lista de nombre **lista** que también está inicialmente vacía; con la ayuda de la sentencia **for** se forma un ciclo que va a representar el recorrido de sólo una fila de la matriz para pasar al siguiente **for** que genera un ciclo que va a representar el recorrido de las columnas que va a tener la matriz; Con este proceso se ha creado la cabecera de la tabla con información importante para el usuario; estas sentencias anidadas cíclicas sólo llenan la primera fila de la tabla. Luego se continúa a mostrar en pantalla estas cabeceras.

Para proceder con el llenado ya de toda la tabla se genera otras sentencias anidadas de **for**, donde el primer **for** de igual forma representa las filas de la matriz y en el segundo **for**, representa las columnas; en este bloque de **for**, se procede a solicitar la información al usuario tales como: el nombre de la máquina, el valor del nivel de interés y el nivel de análisis; luego se procede a sumar el nivel de interés y el nivel de análisis, información que se agrega como último elemento de la lista.

*Proceso1, primer bloque de **for**:*

Llenado de la lista.

lista[“N.M”,”N.I”,”N.A”,”TOTAL”]

Primera fila agregada a la lista **matriz:**

matriz[ [“N.M”,”N.I”,”N.A”,”TOTAL”],

]

*Proceso2, segundo bloque de **for**:*

Llenado de la lista con valores ingresados por el usuario.

lista[“Máquina …”, 5, 6, 11]

Segunda fila agregada a la lista **matriz:**

matriz[ [“N.M”,”N.I”,”N.A”,”TOTAL”],

[“Máquina …”, 5, 6, 11],

]

Cada elemento de la lista **matriz** es una lista que se genera desde cero cada vez que la sentencia **for** de variable se genera. Con la sentencia condicional **if**, se procede a identificar cuál es el valor total mayor de cada fila, con lo que deriva a mostrar un mensaje por pantalla, la máquina que requiere ser atendida con mayor prioridad.

Para mostrar la lista con cada uno de sus elementos se aconseja proceder a imprimirla con una sentencia **for**, para que, de esta manera, se pueda apreciar cada elemento de la matriz como una fila de la misma. Con la sentencia **for**, lo que se logra es que cada recorrido tomará una fila.

Suponga que durante todo el año la empresa ha procedido a dar mantenimiento a **n** equipos de la empresa, entonces se requiere conocer la máquina con el mayor número total de mantenimiento al año para tomar ciertas decisiones, por lo que elaborará una matriz de cada mes del año en donde se registrará cada mantenimiento de cada mes por máquina y se identificará la máquina con la cantidad de mantenimientos que le ha realizado al año.

Código:

```
matriz=[]

num=int(input("Ingrese la cantidad de máquinas que desea registrar"))

mayor=0

for i in range(1):
    lista=[]
    for j in range(1):
```

```
        lista.append("MAQ.")
        lista.append("ENE.")
        lista.append("FEB.")
        lista.append("MAR.")
        lista.append("ABR.")
        lista.append("MAY.")
        lista.append("JUN.")
        lista.append("JUL.")
        lista.append("AGO.")
        lista.append("SEP.")
        lista.append("OCT.")
        lista.append("NOV.")
        lista.append("DIC.")
    matriz.append(lista)

for i in range(1,num+1):
    lista=[]
    print()
    print("**************************************************")
    print(f"Ingrese los datos de la máquina {i}")
    for j in range(1):
        lista.append(input("Ingrese el nombre de la máquina"))
        lista.append(int(input("Cantidad de mantenimientos realizados en
ENERO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
FEBRERO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
MARZO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
ABRIL ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
MAYO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
JUNIO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
JULIO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
```

```
AGOSTO ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
SEPTIEMBRE ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
OCTUBRE ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
NOVIEMBRE ")))
        lista.append(int(input("Cantidad de mantenimientos realizados en
DICIEMBRE ")))
    matriz.append(lista)

    for _ in range(1):
        suma = sum([lista[fila] for fila in range(1,13)])

    if mayor<suma:
        mayor=suma
        a=i


for fila in matriz:
    print(fila)

print("La máquina con la mayor cantidad de mantenimiento realizados en
el año es",matriz[a])
```

En el ejemplo se ha procedido a realizar los mismos pasos para generar una **matriz**, algo adicional con la manipulación de los datos es la función **sum**:

Código:

```
for _ in range(1):
        suma = sum([lista[fila] for fila in range(1,13)])
```

En este bloque de Código se recorre la lista generada con los valores de cada mes del año, datos ingresados por el usuario a través de la sentencia **for**, luego se procede a recorrer la única fila creada hasta el momento, en vez de generar un **for** fuera de la función se ha procedido a generarla dentro de la misma línea de código, lo cual ayuda a reducir líneas de código; el **for** interno de igual forma recorre cada elemento de **lista** y con el manejo de los índex de la lista, la función **suma**, procede a sumarlos.

### 1.2.2 Tuplas

Las tuplas son estructuras de almacenamiento muy similares a las listas, la diferencia es que las tuplas una vez generadas no se las puede modificar; para proceder a declarar una tupla se utilizan los paréntesis.

Código:

```
tupla=("Vane", 123)

print(tupla)

print(tupla[0])
```

En el ejemplo se muestra la generación de una tupla con dos elementos: el primero en la posición cero el nombre “Vane” y el segundo elemento en la posición uno con la cantidad 123, las tuplas simplemente se las puede mostrar a través de la función **print** con lo que presentará en pantalla todos los elementos de la tupla o utilizar la función **print,** pero por cada posición o *índex* que maneja la tupla; en el ejemplo se utiliza corchetes para indicar la posición cero del elemento que se desea enseñar, para el ejemplo sólo va a mostrar “Vane”.

También se puede generar una tupla colocando cada elemento separado por una coma sin la necesidad de utilizar paréntesis.

Código:

```
tupla="Vane", 123

print(tupla)

print(tupla[0])
```

Al ser una estructura de almacenamiento, se maneja similares conceptos, como por ejemplo, una tupla puede contener diferentes tipos de datos y también otras estructuras, es

decir, la tupla puede contener listas. Las tuplas permiten asignar a diferentes variables cada uno de sus elementos sin la necesidad de generar varias líneas de código, lo cual ayuda mucho en la programación, por ejemplo:

Código:

```
tupla="Vane", 123,[1,2,3]

a,b,c=tupla

print(a,b,c)
```

Las tuplas al igual que las listas utilizan propiedades las cuales se describen en la Tabla 1.8.

**Tabla 1.8**

*Propiedades para tuplas*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| count() | Consulta | **print**(tupla.**count**(123)) | Cuenta cuántas veces está en la tupla el valor. |
| index() | Consulta | **print**(tupla.**index**(123)) | Muestra el número del índice donde está ubicado el elemento en la tupla. |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| tuple | Conversión | tupla=**tuple**(lista) | La lista se convierte en una tupla. |
| list | Conversión | lista=**list**(tupla) | La tupla se convierte en una lista. |
| len | Consulta | **print**(**len**(tupla)) | Muestra el número de elementos que tiene la tupla. |

*Nota.* Los autores

La tupla se le tiene que predefinir desde el inicio del código, porque estas no pueden agregar información después de que estén definidas; por lo tanto, no admite la propiedad **append** y tampoco se puede eliminar ni modificar ninguno de sus elementos. Lo que sí se puede hacer es mostrar y realizar alguna búsqueda en particular, algo muy similar a las listas es utilizar los índices de las posiciones como se muestra en la Tabla 1.9:

**Tabla 1.9**

*Índex para listas y tuplas.*

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| [1] | Tupla=(“Vane”,123,[1,2,3]) | print(tupla[1]) | Muestra el segundo elemento de la lista o tupla. |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| [-1] | Tupla=(“Vane”,123,[1,2,3]) | print(tupla[-1]) | Muestra el último elemento de la lista o tupla. |
| [:-1] | Tupla=(“Vane”,123,[1,2,3]) | print(tupla[:-1]) | Muestra desde la posición -1 de la lista o tupla hasta el primer elemento. |
| [1:-1] | Tupla=(“Vane”,123,[1,2,3]) | print(tupla[1:-1]) | Muestra desde la posición 1 hasta la posición -1, sin incluir el elemento en la posición -1 del elemento de la tupla o lista. |
| [1:-1:2] | Tupla=(“Vane”,123,[1,2,3],3,4) | **print**(lista[1:4:2]) | Muestra desde el elemento que está en la posición 1 hasta el 4 sin incluir, con un salto de 2 3n 2, en la tupla o lista. |
| # in tupla | Tupla=(“Vane”,123,[1,2,3]) | **print**(123 in tupla) | Muestra un True o False, depende de que si el número es elemento de la tupla. |

*Nota.* Los autores

Para entender mejor el manejo de los elementos en posiciones o índices de una lista o tupla, se ha procedido a realizar la siguiente figura:

**Figura 1.1**

*Manejo de Índex.*

| Índex | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **Lista o Tupla** | **M** | A | N | T |
| | **-4** | -3 | -2 | -1 |

Índex Negativo

*Nota.* Los autores

En el siguiente ejemplo se va a colocar una lista predefinidas de herramientas que por lo general los ingenieros de mantenimiento suelen utilizar; el usuario va a registrar una cantidad de herramientas, si la herramienta ya existe no se la agrega a la lista final, caso contrario, se ingresa y se muestra el total de herramientas registradas.

Código:

```
herramientas_mmto="LLAVE", "MARTILLO","PINZA" , "DESTORNILLADOR"

lista=[]

num=int(input("Cuantas herramientas desea aumentar a la lista"))

for i in range(num):
   herramienta_nueva=input("Ingrese la nueva herramienta").upper()
```

```
    if herramienta_nueva in herramientas_mmto:
        print("La herramienta ya esta en agregada")
    else:
        lista.append(herramienta_nueva)

total=len(herramientas_mmto)+len(lista)

print(f"En total estan en la lista: {total} herramientas y estas son:")
print(lista+list(herramientas_mmto))
```

Se ha generado una tupla de nombre **herramientas_mmto**, en la cual se ha preestablecido una lista de herramientas, luego se solicita al usuario la cantidad de herramientas que desea registrar a través de la sentencia **for**; se procede a solicitar la cantidad de herramientas que indicó el usuario; dentro de este ciclo, se genera un bloque con la solicitud al usuario de que ingrese la herramienta a través de la sentencia condicional; si se consulta si la herramienta que ingresó el usuario está contenida en la tupla **herramientas_mmto,** si la condición es verdadera se procede a informar al usuario que la herramienta ya ha sido ingresada en la lista, caso contrario, la nueva herramienta se ingresará en una lista, ya que las tuplas no permiten alteraciones en sus definiciones, sólo consultas.

Con la función **len**, se procede a calcular el número de elementos de la tupla y de la lista, para poder mostrar al usuario

la cantidad de herramientas con la que cuenta; finalmente, la tupla **herramientas_mmto**, se transforma a lista para proceder a concatenarla con la lista donde se ingresaron las herramientas digitadas por el usuario y así mostrar en pantalla una sola lista.

### 1.2.3 String

Los *string* son una cadena de caracteres, pues cada elemento de un *string* o cadena, está constituido por uno o varios caracteres, por ejemplo:

**Tabla 1.10**

*Cadenas y caracteres*.

| Descripción | Uso |
|---|---|
| Cadena 1 | "Va123" |
| Cadena 2 | "123" |
| Cadena 3 | "Hola mundo" |
| Cadena 4 | "El valor de a es 23" |
| Cadena 5 | "a * b" |
| Carácter 1 | "A" |
| Carácter 2 | "1" |
| Carácter 3 | "a" |
| Carácter 4 | " " |

*Nota.* Los autores

En la Tabla 1.10 se puede apreciar que una cadena está compuesta por 1 o más caracteres, en *Python* se lo interpreta como una estructura, es decir un *string* se puede iterar.

Para poder iterar una lista se puede aplicar el siguiente ejemplo:

Código:

```
frase=input("Ingrese la frase")

for m in frase:
    print(m)

print(frase)
```

Entonces en el ejemplo se puede apreciar que cualquier frase que ingrese el usuario, a través de la sentencia **for,** la variable **m** va a ir tomando una a una las letras que son parte de la frase, para al final mostrar la frase ingresada a través de la función **print**.

También se puede manejar las posiciones del *string*, como, por ejemplo:

Código:

```
frase=input("Ingrese la frase")

for m in range(len(frase)):
    print(frase[m], end="")
```

A los string se les puede aplicar la función **len,** como se muestra en el ejemplo y en el bloque del **for,** se procede a mostrar cada letra de la frase por la posición que es representada a través de corchetes, además se ha agregado a la función **print**, **end= " ",** esto lo que le indica a la función es que no realice un salto de línea, consiguiendo así mostrar toda la frase en una sola línea de impresión.

Los *string* por ser caracteres se les puede aplicar las funciones, **upper** y l**ower** que lo que hacen es colocar al *string* en mayúscula y minúscula respectivamente.

Código:

```
frase=input("Ingrese la frase").upper()
frase=input("Ingrese la frase").lower()
```

### 1.2.4 Set

Es una estructura de datos similar a una lista, la diferencia es que no admite datos duplicados, su principal función es eliminar duplicados e identificar si algún elemento ya está contenido en la lista y ordena los elementos al momento de mostrar por pantalla cuando se maneja cantidades.

Para definir un **set** se realiza el siguiente proceso:

Código:

```
lista=set(
   ["Motor1", "Motor2", "Motor3"]
   )
print(lista)
```

A una variable lista se le ha asignado una lista generada con **set**, como **set** no admite datos duplicados, se muestra el siguiente ejemplo con datos duplicados:

Código:

```
lista=set(
   ["Motor1", " Motor2", " Motor3", " Motor3"]
   )
print(lista)
```

En este caso la lista sin ningún problema ha sido generada, pero al momento de imprimir la lista mostrará todos los elementos y los que estén duplicados, procederá a eliminar el duplicado para mostrar solo uno de ellos.

{'Motor1', 'Motor2', 'Motor3'}

Para entenderlo mejor se puede aplicar la teoría de conjuntos.

Si tiene el conjunto dado por ingenieros de mantenimiento en dos áreas:

**Figura 1.2**

*Diagrama de Ven.*


*Nota.* Los autores**.** Adaptado de Microsoft Visio

Se desea identificar los elementos que están en la **Lista 1** y que no están en la **Lista 2.**

Código:

```
lista1=set(
   ["Maria", "Sofía", "Karla", "Diego", "Lucas"]
   )
lista2=set(
   ["Carlos", "Andrés", "Nicolás","Diego", "Lucas"]
   )
print(lista1-lista2)
```

Como salida se obtendrá:

{'Karla', 'Maria', 'Sofía'}

Si lo que desea es mostrar los elementos que están en ambas listas entonces se utilizará la intersección que en programación es el signo **&.**

Código:

```
lista1=set(
   ["Maria", "Sofía", "Karla", "Diego", "Lucas"]
```

```
    )
lista2=set(
    ["Carlos", "Andrés", "Nicolás","Diego", "Lucas"]
    )
print(lista1 & lista2)
```

Como salida se tendrá:

{'Diego', 'Lucas'}

Pero si lo que se desea es realizar la unión de las listas se utilizará el signo **|.**

Código:

```
lista1=set(
    ["Maria", "Sofía", "Karla", "Diego", "Lucas"]
    )
lista2=set(
    ["Carlos", "Andrés", "Nicolás","Diego", "Lucas"]
    )
print(lista1 | lista2)
```

Como salida se tendrá:

{'Sofía', 'Andrés', 'Karla', 'Nicolás', 'Lucas', 'María', 'Carlos', 'Diego'}

También permite la manipulación de *strings* por ejemplo:

Código:

```
lista1=set( "Mantenimiento")

print(lista1)
```

Cada carácter del *string* lo pasará a un elemento de la lista y además elimina los caracteres repetidos.

Como salida se tendrá:

{'a', 'n', 'm', 'o', 't', 'M', 'i', 'e'}

**Set** es mutable, lo que quiere decir que sí se puede manipular sus elementos y además se puede ingresar más elementos al conjunto así ya esté generado. En la Tabla 1.11 se muestra alguna de las propiedades con las que puede trabajar **set.**

**Tabla 1.11**

*Propiedades set.*

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| add() | lista1=**set**("Mantenimiento") | lista1.**add**(1) | Ingresa a la lista el elemnto 1. |
| update() | lista1=**set**("Mantenimiento") | lista1.**update**(lista2) | Agrega los elementos de la lista2 a la lista1. |
| remove | lista1=**set**(["Maria", "Sofía", "Karla", "Diego", "Lucas"]) | lista1.**remove**("Maria") | Elimina el elemento Maria de la lista1. |

*Nota.* Los autores

El siguiente ejemplo es tratar de identificar los elementos repetidos de una lista y mostrar la cantidad de elemento con los que cuenta realmente al eliminar los repetidos. La lista contendrá los ingenieros de mantenimiento de una empresa.

Código:

```
lista=[]

num=int(input("Ingrese la cantidad de herramientas que desea
ingresar"))

for i in range(num):
    lista.append(input(f"Ingrese la {i+1} herramienta: ").upper())
```

```
conjunto=set(lista)
cantidad=len(conjunto)

print(f"La cantidad de herramientas con las que cuenta la empresa es:
{cantidad}")
print("listado de herramientas")
print(conjunto)
```

Para el ejemplo se ha procedido a generar una lista, la cual contendrá el listado de herramientas que ingresará el usuario cuya cantidad será controlada por la sentencia **for,** para poder eliminar los elementos duplicados de la lista, se ha convertido la lista a un conjunto y con la función **len**, se procede a contar los elementos del conjunto, por lo que el presente ejemplo se convierte en una evidencia clara como los conjuntos ayudan en la programación.

### 1.2.5 Diccionarios

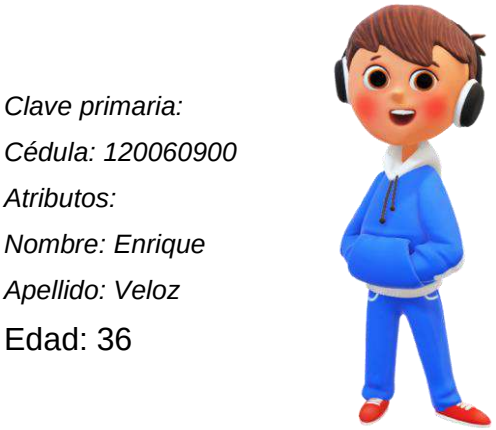
Son estructuras que permiten almacenar valores, donde intervienen claves primarias y atributos.

Las claves primarias representan de forma única al diccionario generado, los atributos son los que representan las

características que posee dicho diccionario, como ejemplo de clave primaria puede ser la cédula; todos conocen que el número de cédula es único para cada ciudadano, a este diccionario que se le coloca como clave primaria el número de cédula se le asigna tres atributos como, por ejemplo: nombre, apellido y edad, con toda esta información se ha creado un diccionario. Para ofrecer al lector una mejor explicación de lo mencionado, se ha procedido generar la Figura 1.3.

**Figura 1.3**

*Representación de clave primaria y atributos*.




*Nota.* Los autores. Adaptado de Microsoft Word

El ejemplo la Figura 1.3 en código queda representado de la siguiente forma:

Código:

```
objeto1=dict(
        Cedula=120060900,
        Nombre="Enrique",
        Apellido="Veloz"

)

print(objeto1)
```

Los diccionarios al igual que las listas pueden almacenar diferentes tipos de datos como enteros, texto, listas e incluso otros diccionarios, pero muy importante es que los diccionarios manejan, el concepto **clave: valor** para cada uno de sus elementos; si se aprecia en el ejemplo, la clave sería la cédula y el valor, el número de cédula que se le asigna. Además, se debe tener presente que no necesariamente la información va a estar en forma ordenada, pero claro sería lo más óptimo, pero no necesario; a diferencia de las listas que manejan corchetes, los diccionarios se les define con las llaves.

Para poder acceder a la información generada en el diccionario se debe consultar por la clave a través del nombre o solamente el nombre de la variable a la que se le designó el diccionario, que en el ejemplo es **objeto1.**

Se pueden generar muchos diccionarios y los mismos pueden formar parte de una lista como se aprecia en el siguiente ejemplo:

Código:

```
Lista=[]

objeto1=dict(
        Nombre="Vane",
        Apellido="Valverde"

)

objeto2=dict(
        Nombre="Lore",
        Apellido="González"

)

Lista.insert(0,objeto1)
Lista.insert(0,objeto2)

print(Lista)
```

Se ha procedido a generar una lista vacía, luego se generan 2 diccionarios cada diccionario tiene su respectivo **clave=valor**; para generar un diccionario se coloca la palabra clave **dict** y se utilizan las llaves para contener el bloque de información que corresponde al diccionario; cada diccionario está almacenado a una variable con el nombre **objeto1** y **objeto2** respectivamente; como siguiente paso, se inserta cada diccionario a través de su variable a la lista a través de la función **insert**, la función **insert** tal como está expresada en el ejemplo funciona internamente igual que la función **append**, es decir, va agregando siempre al final de la lista; al proceder a imprimir se puede apreciar la lista con dos elementos donde cada elemento es un diccionario con su respectivo **clave=valor.**

En el siguiente ejemplo se va a suponer que se quiere registrar a **m** empleados, con ciertos datos importantes para la empresa, para este registro se va a generar un diccionario que contendrá esta información requerida.

Código:

```
registro=dict()

m=int(input("Cuantos registros desea ingresar"))

for i in range(m):
        nombre1=input("Ingrese su nombre")
        apellido1=input("Ingrese su apellido")
        cedula1=input("Ingrese su cedula")
        registro[i]={
                        "Nombre":nombre1,
                        "Apellido":apellido1,
                        "Cedula":cedula1
        }
for _ in registro:
    print(registro[_])
```

Se ha procedido a generar una cantidad de diccionarios, según el requerimiento del usuario, cada diccionario generado contiene como clave el Nombre, Apellido y la Cédula, a través de la variable del ciclo se ha podido generar diccionarios de forma dinámica y ya no predefinidas es decir estáticas, en cada generación de un diccionario diferente no se coloca la palabra clave **dict** porque ya fue colocada de manera inicial al momento de generarla. Para el proceso de la impresión se ha colocado un ciclo **for**, que recorre cada posición del diccionario registro para poder dar una mejor presentación de sus claves=valor.

Ejemplo del llenado de un diccionario de nombre **registro**, al cual se le podrá ingresar, mostrar, editar y eliminar la información a través de un menú generado con la sentencia condicional **match**.

Código:

```
registro=dict()
a=True
m=int(input("cuantos registros a al diccionario"))

while a==True:
    print("Seleccione una opción")
    print("1:Ingresar información")
    print("2. Mostrar información")
    print("3:Editar información")
    print("4:Eliminar información")
    print("5: Salir")
    op=input()

    match op:
        case "1":
            for i in range(m):
                nombre1=input("Ingrese su nombre")
                apellido1=input("Ingrese su apellido")
                cedula1=input("Ingrese su cedula")
                registro[i]={
                        "Nombre":nombre1,
                        "Apellido":apellido1,
                        "Cedula":cedula1
                }
        case "2":
            if registro!="":
```

```
            for _ in registro:
                print(registro[_])
        else:
            print("No se tiene registro que mostrar")
    case "3":
        num=int(input("Ingrese el número de registro que desea
modificar"))
        nombrenuevo=input("Ingrese su nuevo nombre")
        apellidonuevo=input("Ingrese su nuevo apellido")
        cedulanueva=input("Ingrese su nueva cedula")
        registro[num]={
                "Nombre":nombrenuevo,
                "Apellido":apellidonuevo,
                "Cedula":cedulanueva,
            }
    case "4":
        num=int(input("Ingrese el número del registro a eliminar"))
        del registro[num]
    case "5":
        a=False
```

En el **case 1,** se procede a solicitar la información al usuario para que a través de la sentencia cíclica **for** se genere cada diccionario trabajando con la posición de cada uno a través de la implementación de corchetes. En el **case 2**, se procede a realizar una consulta para proceder a mostrar cada diccionario generado, a través del condicional **if-else,** si el diccionario **registro** está vacío se mostrará por pantalla que no existe elementos que mostrar. En el **case 3,** a través del ingreso del número de diccionario que desea actualizar el usuario, se solicitan los nuevos

valores y se procede a solicitar el diccionario seleccionado para proceder a ingresar los nuevos datos. En el **case 4,** a través de la función **clear** y con el número del diccionario que se desea eliminar, automáticamente se elimina el diccionario indicado. En el **caso 5,** se procede a asignar a la variable **a** el valor de **False,** con esto la sentencia **while** que encierra todo el proceso del menú tendrá una condición falsa, por lo que saldrá de la sentencia para terminar el algoritmo.

Con estos ejemplos se pueden evidenciar la importancia de tener una clave que represente a cada diccionario, que en programación y en especial en los conceptos de base de datos, se lo conoce como clave primaria. Al momento de querer realizar una edición o eliminación se tiene que contar con un argumento que identifique al diccionario de forma única. Por ejemplo, se puede dar el caso de que dos técnicos encargados del mantenimiento en una empresa_x tengan el mismo nombre pero no tendrán la misma cédula, como se aprecia en la Figura 1. 4.

**Figura 1.4**

*Representación de clave primaria cédula.*


*Nota.* Los autores**.** Adaptado de Microsoft Word

Ejemplo de una aplicación que realiza un inventario para herramientas de mantenimiento.

Código:

```
herramientas=dict()

num=int(input("Cuantas herramientas desea registrar"))

for i in range(num):
    print(f"Datos de la herramienta {i+1}")
    nombre1=input("Ingrese el nombre")
    serie1=input("Ingrese la serie")
```

```
    marca1=input("Ingrese la marca")
    modelo1=input("Ingrese el modelo")
    custodio1=input("Ingrese el nombre del custodio")
    descripcion1=input("Ingrese la descripción de la herramienta")

    herramientas[i]={
                    "Nombre":nombre1,
                    "Serie":serie1,
                    "Marca":marca1,
                    "Modelo":modelo1,
                    "Custodio1":custodio1,
                    "Descripcion":descripcion1

                }

for i in range(num):
    print(herramientas[i])
```

La estructura diccionario en *Python* sirve para almacenar cualquier tipo de información, los cuales manejan la **clave: valor,** sin tener que estar ordenados, pero si se recomienda tener una **clave: valor** que lo identifique de manera única.

En la Tabla 1.12, se muestran algunas propiedades que se puede aplicar a los diccionarios.

**Tabla 1.12**

*Propiedades para diccionarios.*

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| get | obtener | **print**(objeto1.**get**("Cedula")) | Muestra el número de cédula del diccionario objeto1. |
| keys | clave | **print**(objeto1.**keys**()) | Muestra sólo las claves del diccionario objeto1. |
| values | valores | **print**(objeto1.**values**()) | Muestra sólo los valores del diccionario objeto1. |
| pop | elimina | **print**(objeto1.**pop**("Cedula")) | Muestra el valor de la clave y a su vez elimina la clave:valor indicada. |
| popitem() | elimina | **print**(objeto1.**popitem**()) | Muestra la última clave:valor que elimina del diccionario. |
| del | elimina | del objeto1["Nombre"] | Elimina la clave:valor del diccionario objeto1, pero el atributo especificado |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| clear | limpiar | objeto1.**clear**() | Elimina todas las claves:valor del diccionario. |

*Nota.* Los autores**.**

## 1.3 Archivos

En ciertos momentos lo programadores necesitan almacenar la información de forma permanente por lo que puede conseguir este objetivo a través del manejo de archivos. Los archivos son un conjunto de datos que se almacenan en forma intacta en algún dispositivo de almacenamiento.

Los manejos de archivos permiten crear, agregar, editar, actualizar y eliminar información almacenada, lo que quiere decir que son mutables.

Para crear un archivo se escribirá el siguiente comando:

Código:

```
archivo = open (“mi_primer_archivo.txt”,”w”)
archivo.write(“Archivo creado”)
archivo = open("mi_primer_archivo.txt")
print(archivo.read())
archivo.close()
```

Para conocer acerca de cada una de las funciones que se pueden trabajar con los archivos se muestran algunas de ellas en la Tabla 1. 13.

**Tabla 1.13**

*Propiedades para archivos*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| open() | Abrir/generar | archivo = **open** (“mi_primer_archivo.txt”) | Abre el archivo. |
| close() | Cerrar | archivo.**close**() | Cierra el archivo. |
| write() | Agregar | archivo.**write**(“Archivo creado”) | Agrega información al archivo. |
| read() | Leer | **print**(archivo.**read**()) | Muestra el contenido del archivo. |
| type() | Tipo | **print**(**type**(archivo.**read**())) | Muestra el tipo del archivo. |
| readline() | Leer | print(archivo.readline()) | Lee una línea del archivo. |
| readlines() | Leer | print(archivo.readlines()) | Devuelve una lista donde cada elemento de la lista es una línea del archivo. |
| remove() | borrar | **os**.**remove**("Mantenimiento.txt") | Borra el archivo. |

*Nota.* Los autores

Ejemplos de usos: para abrir un archivo desde *Python* se puede acceder a través de rutas, estas a su vez se clasifican en rutas relativas y rutas absolutas, en la Tabla 1. 14:

**Tabla 1.14**

*Rutas de archivos*.

| Ruta | Forma |
|---|---|
| Ruta relativa | archivo = **open**("mi_primer_archivo.txt") |
| Ruta absoluta | archivo = **open**("carpeta/mi_primer_archivo.txt") |

*Nota.* Los autores

Abriendo archivo, que ha sido creado en la misma ruta donde está el archivo **.py** de *Python* a través de una ruta absoluta:

Código:

```
archivo = open("mi_primer_archivo.txt")
```

Leyendo el archivo:

Código:

```
archivo = open("mi_primer_archivo.txt")
print(archivo.read())
```

Para poder leer un archivo, este primero se debe abrir como en el ejemplo con la función **open,** luego al ser leído con la función **read,** esta debe estar contenida en la función **print** para que pueda ser mostrada por pantalla.

Si lo que se desea es anexar o agregar información a un archivo se debe utilizar la función **write**.

Código:

```
archivo = open (“mi_primer_archivo.txt”,”w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(archivo.read())
```

Al archivo se le ha agregado con la función **write**, dos líneas con salto de línea con **\n**, luego se procede a volver a abrir el archivo para proceder a leerlo con la función **read**, más la función **print** para mostrarlo por pantalla.

Código:

```
archivo = open (“mi_primer_archivo.txt”,”w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(archivo.read())
archivo.close()
```

Siempre después de trabajar con archivos es necesario cerrarlos; en el ejemplo se ha procedido a añadir la línea **archivo.close** donde se le indica a *Python* que se proceda a cerrar el archivo.

Si lo que se desea es conocer el tipo de archivo con el que se está trabajando se procederá se la siguiente forma:

Código:

```
archivo = open (“mi_primer_archivo.txt”,“w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(type(archivo.read()))
archivo.close()
```

La función **type** muestra por pantalla a través de de la función **print**, el tipo de archivo que se le ha indicado.

Las estructuras cíclicas también se les puede implementar en el manejo de archivos, ya que todo forma parte de un solo código, por lo que se puede recorrer líneas en un fichero como, por ejemplo:

Código:

```
tipos_de_mantenimiento = open ("conceptos.txt",“w”)

tipos_de_mantenimiento.write(“Preventivo: Mantenimiento llevado a
cabo para evaluar y/o mitigar la degradación y reducir la probabilidad de
fallo de un elemento. (UNE EN 13306, 2018, pág. 16)\n”)
tipos_de_mantenimiento.write(“Correctivo: Mantenimiento que se realiza
después del reconocimiento de una avería y que está destinado a poner a
un elemento en un estado en que pueda realizar una función requerida.
(UNE EN 13306, 2018, pág. 16)\n”)

tipos_de_mantenimiento = open("conceptos.txt")

i=1

for fila in tipos_de_mantenimiento:
    print("**********************")
    print(f"Linea {i}: ", end=" ")
    print(fila)

    i=i+1

tipos_de_mantenimiento.close()
```

En el ejemplo se genera y se abre un archivo de nombre **"conceptos.txt",** con la función **open** y el modo **w**, el cual representa **write,** para proceder a agregar información al archivo con dos conceptos de mantenimiento, luego se procede a abrir el archivo y con la sentencia repetitiva **for** se va tomando cada línea del archivo para mostrarla en pantalla con la opción sin saltar para que el concepto vaya unido con la palabra **Línea** y el número al

que representa la misma. La variable **fila** es la que va tomando cada línea del archivo.

La salida será la siguiente:

Línea 1: Preventivo: mantenimiento llevado a cabo para evaluar y/o mitigar la degradación y reducir la probabilidad de fallo de un elemento (UNE EN 13306, 2018, p. 16).
**********

Línea 2: Correctivo: mantenimiento que se realiza después del reconocimiento de una avería y que está destinado a poner a un elemento en un estado en que pueda realizar una función requerida (UNE EN 13306, 2018, p. 16).

Si lo que desea es solo trabajar con la primera línea del archivo se puede implementar la función **readline.**

Código:

```
archivo = open (“mi_primer_archivo.txt”, “w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(archivo.readline())
archivo.close()
```

En el código se ha agregado la función **readline,** lo que hará que la función **print,** solo muestre la primera línea del archivo.

Salida **readline**:

Archivo creado 1

También la función **readline** facilita seleccionar una cantidad de caracteres de la línea.

Código:

```
archivo = open (“mi_primer_archivo.txt”, “w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(archivo.readline(5))
archivo.close()
```

En este caso solo mostrará, los primeros 5 caracteres de la línea 1.

Salida:

Archi

Muchas veces es necesario que el archivo sea pasado a una lista para poder mejor manipular cada una de sus líneas, esto es posible con la función **readlines**:

Código:

```
archivo = open (“mi_primer_archivo.txt”, “w”)
archivo.write(“Archivo creado 1\n”)
archivo.write(“Siguiente linea 2\n”)
archivo = open("mi_primer_archivo.txt")
print(archivo.readlines())
archivo.close()
```

Cada línea forma un elemento de la lista.

Salida:

['Archivo creado 1\n', 'Siguiente linea 2\n']

Para proceder a eliminar un archivo es necesario utilizar la función **remove,** pero para poder utilizarla se debe realizar la importación de la librería **os.** Las librerías se las coloca al inicio del código.

Código:

```
import os
archivo = open (“Mantenimiento.txt”,“w”)
archivo.write("\n Martillo, Tornillos")
archivo.close()
os.remove("Mantenimiento.txt")
```

Las librerías de *Python* son conjuntos de funciones que permiten realizar distintas funciones, ahorrando tiempo y esfuerzo al programador (Chaves, 2022, párr. 2).

Para el manejo de archivos también se utilizan los denominados modos, los cuales se explica en la Tabla 1. 15.

**Tabla 1.15**

*Abreviaturas para archivos*.

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| x | create | archivo = **open** (“Mantenimiento.txt”, “x”) | Crea un archivo de nombre Mantenimiento en la misma ruta del archivo de *Python*. |
| a | append | archivo = **open** (“Mantenimiento.txt”, “a”) | Abre el archivo y agrega la información al final. |

| Signo | Descripción | Uso | Interpretación |
|---|---|---|---|
| w | write | archivo = **open** (“Mantenimiento.txt”, “w”) | Elimina el contenido y lo remplaza. |
| r | read | archivo = **open** (“Mantenimiento.txt”, “r”) | Archivo solo en modo lectura. |

*Nota.* Los autores

Ejemplos de uso:
Creando el archivo desde cero o vacío:

Código:

```
archivo = open (“Mantenimiento.txt”, “a”)
```

Abriendo el archivo y agregando información:

Código:

```
archivo = open (“Mantenimiento.txt”, “a”)
archivo.write("\n Herramientas para el mantenimiento")
archivo.close()
```

Agrega al archivo una línea en blanco y en la siguiente línea el texto indicado.

Remplazando y agregando nuevo contenido al archivo:

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("Martillo, Tornillos")
archivo.close()
```

El contenido del archivo **Mantenimiento.txt,** es eliminado en su totalidad y se le agrega el nuevo contenido "Martillo, Tornillos".

Existen ocasiones que en los archivos se desea realizar más de una operación en ese caso se recomienda a cada uno de los modos agregarle **+**, como se muestra en la Tabla 1. 16.

**Tabla 1.16**

*Abreviaturas de archivos con* **+**.

| Signo | Descripción | Uso |
|---|---|---|
| w+ | Leer-Escribir | archivo = **open** (“Mantenimiento.txt”, “w+”) |
| a+ | Leer- Agregar | archivo = **open** (“Mantenimiento.txt”, “a+”) |
| r+ | Leer-Escribir | archivo = **open** (“Mantenimiento.txt”, “r+”) |

*Nota.* Los autores

En el siguiente ejemplo se va a generar un archivo y se le va a agregar información ingresada por el usuario, suponiendo que se desea ingresar un informe del mantenimiento de una máquina:

Código:

```
maquina=input("Ingrese el nombre de la máquina")
fecha1=input("Ingrese la fecha que se realizó el mantenimiento")
fecha2=input("Ingrese la fecha que se será el próximo mantenimiento")

info = open ("Informe_Mtto.txt","w")
info.write("Nombre: "+maquina +"\n")
info.write("Fecha del Mtto: "+fecha1+ "\n")
info.write("Fecha proximo Mtto: "+fecha2+ "\n")

info = open ("Informe_Mtto.txt","r")
linea=info.readline()
while linea!="":
    print(linea)
    linea=info.readline()

info.close()
```

Se procede a solicitar la información al usuario, luego se genera el archivo de nombre **Informe_Mtto.txt,** en modo de escritura; para proceder a realizar los ingresos, en el caso de los archivos, se debe ayudar de la concatenación con el signo **+** para que al momento de implementar la función **write**, se pueda

ingresar el texto deseado y concatenarlo con la variable que contiene la información ingresada por el usuario y si se desea concatenar con un formato específico como en el ejemplo **\n.**

Se procede a abrir el archivo, pero en modo lectura para que a través de la sentencia repetitiva **while,** consultar si la variable **linea** que contiene la primera línea del archivo, no está vacía; si esto es verdadero, se procede a mostrar por pantalla cada línea del archivo; como proceso final, se cierra el archivo.

En algunos casos, los archivos suelen contener tablas; en el siguiente ejemplo se va a generar una tabla con una lista de máquinas y su respectiva información.

Código:

```
cantidad=int(input("Ingrese cantas máquinas desea registrar: "))

info = open ("maquinastabla.txt","w")
info.write("NOMBRE                    FECHA MATTO              FECHA PROX. MATTO \n")

for _ in range(cantidad):
    maquina=input(f"Ingrese el nombre de la máquina {_+1}: ")
    fecha1=input("Ingrese la fecha que se realizó el mantenimiento: ")
    fecha2=input("Ingrese la fecha que se será el próximo mantenimiento: ")
```

```
    info.write(maquina.ljust(30," ")+fecha1.ljust(20) +fecha2.rjust(16," ")+
"\n")

info = open ("maquinastabla.txt","r")
print(info.read())

info.close()
```

En el ejemplo se procede a ingresar a través de la función **write,** los encabezados que va tener la tabla; luego a través de un **for,** se va generando cada una de las iteraciones que se requiere para solicitar la información al usuario, para que cada ingreso quede específicamente bajo el encabezado que le corresponde; se procede a trabajar con dos funciones: **ljust** y **rjust** , las cuales son trabajadas mediante código con cada variable, para posterior pasar a concatenarlas en sí y así poder ser ingresada la información al archivo de texto generado. En la Tabla 1.17 se procede a explicar las funciones **ljust** y **rjust.**

Salida desde terminal:

Ingrese cuántas máquinas desea registrar: 2

Ingrese el nombre de la máquina 1: Motor1

Ingrese la fecha cuando se realizó el mantenimiento: 03/05/2023

Ingrese la fecha cuando será el próximo mantenimiento: 03/05/2024

Ingrese el nombre de la máquina 2: Motor2

Ingrese la fecha cuando se realizó el mantenimiento: 03/08/2023

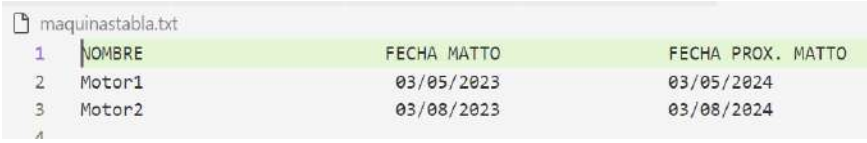
Ingrese la fecha cuando será el próximo mantenimiento: 03/08/2024

| NOMBRE | FECHA MATTO | FECHA PROX. MATTO |
|---|---|---|
| Motor1 | 03/05/2023 | 03/05/2024 |
| Motor2 | 03/08/2023 | 03/08/2024 |

Generación del archivo de texto:

**Figura 1.5**

*Archivo generado .txt.*




*Nota.* Los autores, *Python*

**Tabla 1.17**

*Funciones **ljust** y **rjust.***

| Signo | Descripción | Uso |
|---|---|---|
| **ljust** | Justifica a la izquierda el número de caracteres indicado. | fecha1.**ljust**(20) |
| **rjust** | Justifica a la derecha el número de caracteres indicado. | maquina.**ljust**(30," ") |

*Nota.* Los autores**.**

Cuando se trabaja con archivos suele existir ocasiones que estos archivos son valores y estos valores tienen que ser manipulados para ciertos cálculos, o simplemente se desea eliminar espacios en blanco innecesarios, por lo que es muy importante saber eliminar este inconveniente; en la Tabla 1. 18 se muestran las funciones que pueden ayudar a solucionarlos de forma rápida.

**Tabla 1.18**

*Funciones que eliminan espacios*.

| Signo | Descripción | Uso | |
|---|---|---|---|
| strip() | Elimina espacios en blanco y saltos de línea solo al inicio o al final del string. | **print**(texto.**strip**()) | ....Mantenimiento Industrial... Mantenimiento Industrial |
| lstrip() | Elimina espacios en blanco y saltos de línea solo al inicio del string. | **print**(texto.**lstrip**("Marti")) | ...Mantenimiento Industrial... Mantenimiento Industrial... |
| rstrip() | Elimina espacios en blanco y saltos de línea solo al final del string. | print(texto.rstrip()) | ...Mantenimiento Industrial... Mantenimiento Industrial |

*Nota.* Los autores

La función **strip** elimina al inicio o al final del archivo o *string*, como se puede apreciar en el siguiente ejemplo:

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("   Martillo,  Tornillos,  Motores   \n")
archivo.write("   Bujías,  Tornos   \n")
archivo = open (“Mantenimiento.txt”, “r”)
texto=archivo.read()
print(texto.strip())
archivo.close()
```

Salida:

Martillo,  Tornillos,  Motores

Bujias,  Tornos

Como se puede apreciar en la salida, la función eliminó el espacio en blanco que estaba al inicio de la herramienta Martillo, más no lo espacios en blanco de Tornillo y Motores, tampoco el salto de línea que se le ha especificado, pero si eliminó el salto de línea final.

Para poder implementar la función **strip,** se ha procedido a leer el archivo y almacenarlo en la variable **texto**, pues la función trabaja con *string*.

A la función también se le puede especificar un carácter específico, pero siempre buscará la coincidencia al inicio o al final del texto. Por ejemplo, se le va a indicar que elimine la letra **M**, con la siguiente línea de código.

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("   Martillo,  Tornillos,  Motores   \n")
archivo.write("   Bujías,  Tornos   \n")
archivo = open (“Mantenimiento.txt”, “r”)
texto=archivo.read()

texto=texto.strip()
print(texto.strip("M"))

archivo.close()
```

Salida:

artillo,  Tornillos,  Motores

  Bujias,  Tornos

Para el ejemplo se ha eliminado los espacios en blanco, y luego la letra indicada evidenciando que sólo elimina la coincidencia que este al inicio o final del texto.

En el caso de la función **rstrip,** sólo elimina al final del texto:

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("   Martillo,  Tornillos,  Motores   \n")
archivo.write("   Bujías,  Tornos   ")
archivo = open (“Mantenimiento.txt”, “r”)
texto=archivo.read()

print(texto.rstrip())

archivo.close()
```

Pero si se especifica alguna letra en particular eliminará, pero al final del texto.

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("   Martillo,  Tornillos,  Motores   \n")
archivo.write("   Bujías,  Tornos   ")
archivo = open (“Mantenimiento.txt”, “r”)
texto=archivo.read()

print(texto.rstrip(" so"))

archivo.close()
```

En el ejemplo se le especifica a la función **rstrip()**, que elimine los espacios en blanco que encuentre, que en este caso son tres cada eliminación; la función reinicia la condición y continuará con la siguiente condición que en este caso regresará tres veces hasta eliminar todos los espacios en blanco que existan, para continuar con el siguiente carácter especificado.

La función **lstrip(),** a diferencia de la función **rstrip,** elimina al inicio del texto.

Código:

```
archivo = open (“Mantenimiento.txt”, “w”)
archivo.write("   Martillo,  Tornillos,  Motores   \n")
archivo.write("   Bujias,  Tornos   ")
archivo = open (“Mantenimiento.txt”, “r”)
texto=archivo.read()

texto=texto.strip()
print(texto.lstrip("Marti"))

archivo.close()
```

Salida:

```
llo,  Tornillos, Motores
  Bujias,  Tornos
```

En el código se ha procedido a eliminar primero los espacios en blanco con la función **strip,** para proceder a eliminar los caracteres especificados en la función **lstrip.**

## 1.4 Data Frame

El **Data Frame** es otra estructura de datos que maneja filas y columnas, para trabajar con los **Data Frame**, se debe importar la librería **pandas**. Para generar una **Data Frame**, se ha generado las siguientes líneas de código:

Código:

```
import pandas as pd

data = {"Nombre": ["Alicate", "Martillo", "Llaves"],
        "Cantidad": [10, 10, 15]}
df = pd.DataFrame(data)

print(df)
```

En el ejemplo se procede a importar con la palabra reservada **import,** la librería Panda, procediendo a asignarla en la variable **pd.**

**Data Frame con diccionario**

A continuación, se genera una variable de nombre **data,** que recibe un diccionario con sus **claves-valor**, para a continuación, generar la variable **df**, que se le asigna el diccionario, el cual es tratado por la librería para generar la tabla.

Salida:

Nombre Cantidad
0 Alicate 10
1 Martillo 10
2 Llaves 15

Se puede observar que se ha generado una tabla con tres columnas y cuatro filas, los encabezados de las columnas son las **claves,** y los elementos de las filas son el **valor.**

**Data Frame con listas**

Código:

```
import pandas as pd

datamatto = [“Mantenimiento1”, “Mantenimiento2”, “Mantenimiento3”]
datamotores = ["Motor1", "Motor2", "Motor3"]
```

```
data = {
      "Nombre": datamatto, "Motores": datamotores}
df = pd.DataFrame(data)

print(df)
```

Salida:

```
        Nombre Motores
0  Mantenimiento 1  Motor 1
1  Mantenimiento 2  Motor 2
2   Mantenimiento3  Motor 3
```

Se ha procedido a generar dos listas para proceder a partir de estas dos listas, un diccionario que las contenga, con lo cual se genera la tabla con la librería.

Casi siempre, en la generación de código se debe realizar listados de cantidades de orden **n**, por lo que en el siguiente código se puede visualizar el manejo de **Data Frame,** con listas dinámicas.

Código:

```
import pandas as pd
datamatto=[]
datamotores=[]
```

```
num=int(input("Ingrese la cantidad de mantenimientos que desea
ingresar"))
for _ in range(num):
   datamatto.append(input("Ingrese el mantenimiento: "))
   datamotores.append(input("Ingrese el motor correspondiente"))

data = {
       "Nombre": datamatto,
       "Motores": datamotores
       }

df = pd.DataFrame(data)

print(df)
```

Trabajando con **index** y **columns**

Para mejorar la presentación de los datos al constructor del **Data Frame** se le puede agregar información al **index** que representa las filas y al **columns** que representa las columnas, por lo que al ejemplo de motores con su respectivo mantenimiento, se les va a agregar un **index**, el cual ya no será el de por defecto con ceros y uno, sino texto.

Código:

```
import pandas as pd


data=[]
```

```
datamatto=[]

num=int(input("Ingrese la cantidad de mantenimientos que desea
registrar"))

for fila in range(num):
    datamotores=[]
    datamatto.append(input("Ingrese el mantenimiento: "))
    for columna in range(1):
        datamotores.append(input("Ingrese el motor correspondiente"))
        datamotores.append(input("Ingrese la descripción  correspondiente
del motor"))
    data.append(datamotores)

encabezado=["Motores","Descripción"]



df = pd.DataFrame(data, index=datamatto, columns=encabezado)


print(df)
```

Para el ejemplo se ha procedido a generar tres listas **data, datamtto, datamotores y encabezado**, para proceder a ejecutarse de la siguiente forma:

1. Se genera la lista **data** para que contenga la lista

**datamotores**, la lista **data** se procede a transformar en una matriz.

2. La lista **datamatto**, en cada ejecución del ciclo **for** va a ir almacenando el nombre del mantenimiento que corresponde a cada motor.
3. La lista **datamotores**, almacena los motores que ingresa el usuario, cada uno de sus elementos corresponde a una columna de la matriz.
4. La lista **encabezado,** es una lista estática que contiene solo dos elementos "Motores" y "Descripción".
5. Una vez llenada las listas necesarias se procede a generar el **Data Frame,** enviándole al constructor la lista **data**, indicando en la propiedad **index** que lo que se necesita, que se coloque en cada fila del **Data Frame** son los elementos de la lista **datamtto**, además, en la propiedad **columns** se indica que en vez de una numeración ascendente que coloca el **Data Frame** por defecto, se debe colocar los elementos de la lista **encabezado.**

Salida:

| | Motores | Descripción |
|---|---|---|
| Mantenimiento1 | Motor1 | Descripción del motor1 |
| Mantenimiento2 | Motor2 | Descripción del motor2 |
| Mantenimiento3 | Motor3 | Descripción del motor3 |

## EJERCICIOS PROPUESTOS

1. Realizar un programa que muestre un menú con los diferentes tipos de mantenimiento industrial; el usuario escogerá una opción y se le mostrará toda la información necesaria acerca de este mantenimiento seleccionado.

2. Realizar un programa que muestre al usuario una lista de requerimientos; el usuario seleccionará uno de los requerimientos e ingresará el mantenimiento que aplicaría; el programa indicará si es correcto o no.

3. Realizar un programa donde el usuario ingrese la información necesaria para que el programa pueda detectar que tipo de mantenimiento se debe implementar.

4. Realizar un programa que genere una lista con los ingenieros de mantenimiento de una empresa y otra lista con los mantenimientos a los que está a cargo cada ingeniero; mostrar al ingeniero con el mantenimiento que le corresponde.

5. Realizar un programa que almacene en una lista el número de fallos de una máquina x y el fallo encontrado, mostrar al usuario el fallo con la cantidad respectiva y el total de fallos.

6. Realizar un programa que genere una matriz de orden mxn según el requerimiento del usuario, por ejemplo:

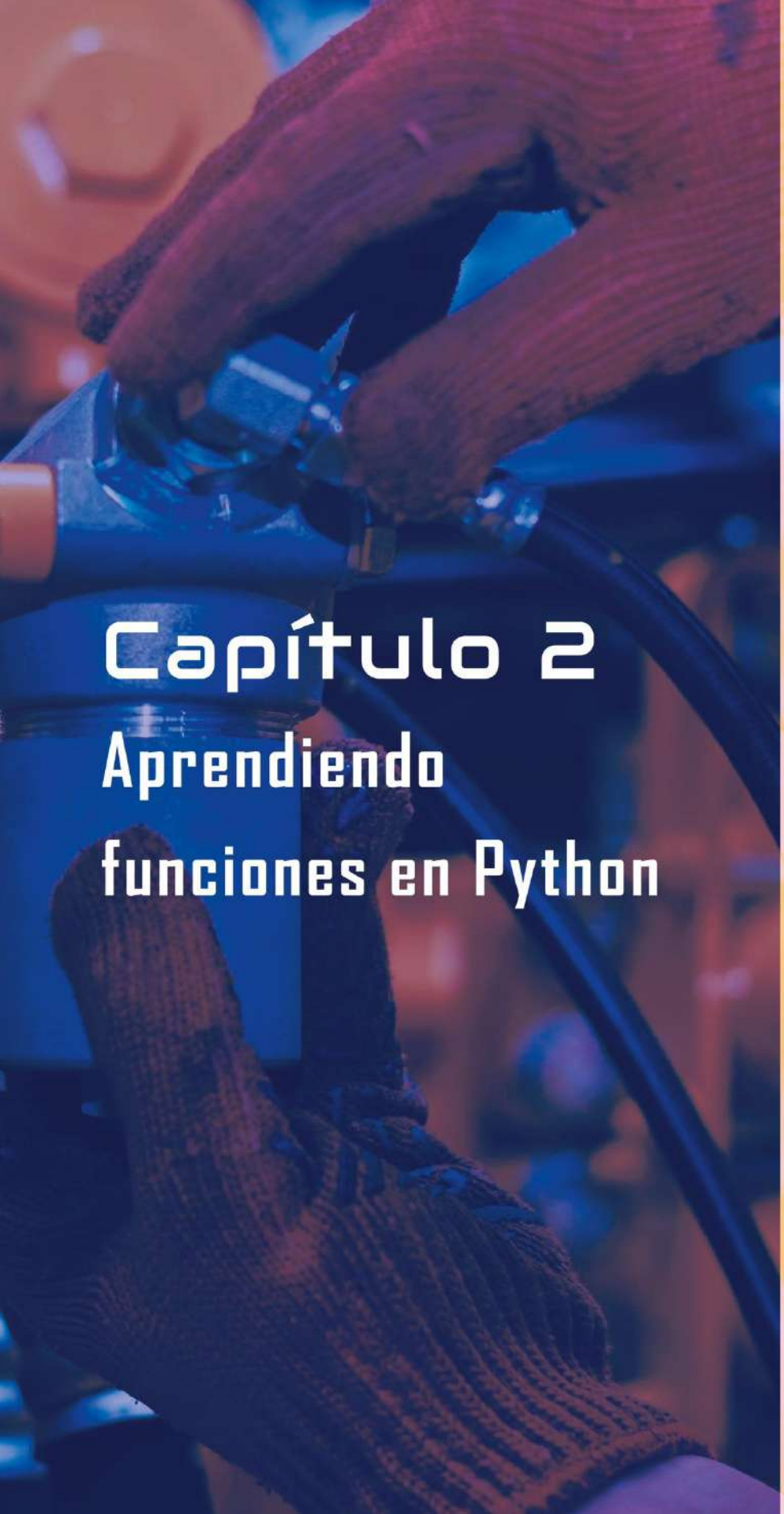
| | Máquina1 | Máquina2 | ... |
|---|---|---|---|
| Número de mantenimientos | | | |
| Número de fallas | | | |
| ... | | | |

7. Realizar el ejercicio 4, identificando a que máquina se le ha realizado el menor número de fallas.

8. Realizar un programa donde se genere un archivo de texto que contenga una tabla con por lo menos cuatro indicadores de mantenimiento para la gestión de activos.

9. Realizar un programa que modifique el programa número 8, aumentando 4 indicadores más.

10. Realizar un programa que genere a través de un Data Frame los indicadores de gestión más utilizados, como encabezado de cada columna: la categoría, el porcentaje de cada indicador y como fila, el nombre correspondiente al indicador.



2

# Capítulo 2

## Aprendiendo funciones en Python

Capítulo 2

# Aprendiendo funciones en Python

## 2.1 Introducción a las funciones en Python

### 2.1.1 ¿Qué es una función?

En el mundo de la programación, una función es un bloque de código reutilizable y autónomo que realiza una tarea específica. Las funciones permiten dividir un programa en partes más pequeñas y manejables, lo que facilita el desarrollo, la depuración y la mantenibilidad del código. En *Python*, las funciones son elementos fundamentales de la programación y se utilizan ampliamente en todo tipo de aplicaciones.

Una función en *Python* tiene las siguientes características:

- **Nombre:** toda función tiene un nombre único que la identifica. Este nombre se elige de manera significativa para describir la tarea que realiza la función.

- **Parámetros:** los parámetros son valores que una función acepta como entrada. Estos valores se utilizan dentro de la función para realizar operaciones. Una función puede aceptar cero o más parámetros.

- **Cuerpo:** el cuerpo de la función contiene el conjunto de instrucciones que se ejecutan cuando la función es llamada. Aquí es donde ocurre la tarea específica de la función.

- **Retorno:** una función puede devolver un valor como resultado de su ejecución. El valor de retorno se especifica mediante la sentencia **return**.

- **Reutilización:** una vez que se define una función, se puede llamar varias veces en diferentes partes del programa, lo que promueve la reutilización del código y evita la duplicación.

Ejemplo simple de una función en *Python*:

Código:

```
def saludar(nombre):
    """Esta función imprime un saludo personalizado."""
    print("¡Hola,", nombre, "! ¿Cómo estás?")

# Llamada a la función
saludar("Alice")
saludar("Bob")
```

En este ejemplo, se ha definido una función llamada **saludar** que acepta un parámetro **nombre**. Cuando la función es llamada con diferentes nombres, imprimirá un mensaje de saludo personalizado.

### 2.1.2 Ventajas de usar funciones

Las ventajas de las funciones son las siguientes:

- **Reutilización de código:** las funciones permiten escribir código una vez y reutilizarlo en múltiples partes del programa.
- **Modularidad:** dividir un programa en funciones más pequeñas hace que el código sea más claro y organizado.

- **Facilita la depuración:** las funciones aíslan problemas específicos, lo que facilita la identificación y corrección de errores.
- **Mantenibilidad:** si se necesita hacer cambios en una funcionalidad, solo es necesario modificar una función en lugar de todo el programa.
- **Colaboración:** varios programadores pueden trabajar en diferentes funciones de manera independiente.

En resumen, las funciones son bloques de construcción esenciales en *Python* y en la programación en general. Proporcionan un enfoque estructurado para resolver problemas complejos y mejorar la eficiencia del desarrollo de software.

### 2.1.3 Sintaxis básica de una función en *Python*

La sintaxis básica de una función en *Python* sigue un formato específico. Aquí se presenta cómo se define esa función:

Código:

```
def nombre_de_la_funcion(parametro1, parametro2):
    # Cuerpo de la función
    # Realizar operaciones, cálculos, etc.
    return resultado
```

Explicación de los elementos:

- **def**: es la palabra clave para definir una función.
- **nombre_de_la_funcion**: reemplaza esto con el nombre que elijas para tu función.
- **(parametro1, parametro2)**: aquí se define los parámetros que la función aceptará. Pueden ser cero o más. Estos son como variables que se utilizarán dentro del cuerpo de la función.
- **: (dos puntos):** es necesario después de los paréntesis de la definición de la función.
- **# Cuerpo de la función**: aquí es donde se escriben las instrucciones que la función ejecutará cuando se llame.
- **return resultado**: opcionalmente se puede usar la sentencia **return** para devolver un valor al llamador de la función. Esto finaliza la ejecución de la función.

Ejemplo simple de una función que suma dos números:

Código:

```
def suma(a, b):
    resultado = a + b
```

```
    return resultado

resultado_suma = suma(5, 3)
print("El resultado de la suma es:", resultado_suma)
```

En este ejemplo, la función **suma** acepta dos argumentos **a** y **b**, realiza la suma y devuelve el resultado mediante **return**.

Recuerde que una vez que has definido una función, puede llamarla en diferentes partes del programa para ejecutar su lógica con diferentes valores de entrada.

## 2.2 Parámetros y argumentos

### 2.2.1 Definición de parámetros y argumentos

En programación, los términos «parámetros» y «argumentos» están relacionados con la forma en que se pasan datos a una función o método en un lenguaje de programación. Sin embargo, se utilizan en diferentes contextos y se refieren a diferentes etapas del proceso de llamada a funciones. Aquí tiene una explicación de ambos términos:

a. **Parámetros**: los parámetros son variables declaradas en la definición de una función o método. Representan los valores que una función espera recibir cuando se llama. Los parámetros actúan como marcadores o espacios reservados para los valores que se mantienen durante la llamada a la función. En otras palabras, los parámetros definen qué tipo de datos se esperan recibir y en qué orden.

En la mayoría de los lenguajes de programación, la definición de una función incluye una lista de parámetros dentro de los paréntesis. Por ejemplo:

Código:

```
def suma(a, b):
    resultado = a + b
    return resultado
```

En este ejemplo, la función **suma** tiene dos parámetros: **a** y **b**.

b. **Argumentos**: los argumentos son los valores reales que se pasan a una función cuando se llama. Los argumentos

son provistos por el código que llama a la función y se utilizan para reemplazar los parámetros en la definición de la función. Los argumentos deben coincidir en tipo y orden con los parámetros correspondientes.

Siguiendo el ejemplo anterior, aquí está cómo se pasarían argumentos a la función **suma**:

Código:

```
resultado = suma(5, 3)
print(resultado)
```

En este caso, **5** y **3** son los argumentos que se pasan a la función **suma**, y se asignarán a los parámetros **a** y **b** respectivamente.

En resumen, los parámetros son las variables que se declaran en la definición de una función, mientras que los argumentos son los valores que se pasan a la función cuando se llama. Los argumentos reemplazan a los parámetros y permiten que la función opere con datos específicos requeridos en el momento de la llamada.

### 2.2.2 Tipos de argumentos (posicionales, keyword, default)

En programación, especialmente en lenguajes de programación como *Python*, se utilizan diferentes tipos de argumentos al llamar a funciones o métodos. Los tres tipos principales de argumentos son:

a. **Argumentos posicionales**

- Los argumentos posicionales se pasan a una función en el orden en que se definen en la firma de la función.
- Los valores se asignan a los parámetros basados en su posición.
- Si la función espera que se le pasen varios argumentos posicionales, debe asegurarse de pasarlos en el mismo orden en que se declararon los parámetros.

Código:

```
def suma(a, b):
    return a + b
resultado = suma(3, 5)  # 3 se asigna a 'a' y 5 se asigna a 'b'
```

**b. Argumentos de palabra clave (keyword arguments)**

- Los argumentos de palabra clave se pasan a una función con el nombre del parámetro al que se desea asignar un valor.
- Esto permite especificar específicamente a qué parámetros se les debe asignar un valor, independientemente del orden en que se definieron en la firma de la función.

Código:

```
def resta(a, b):
    return a - b
resultado = resta(b=5, a=3)  # Los valores se asignan a 'a' y 'b' utilizando palabras clave
```

**c. Argumentos por defecto (argumentos predeterminados)**

- Los argumentos por defecto son parámetros que tienen un valor predefinido en la firma de la función.
- Si no se proporción.

- Los argumentos por defecto son útiles para hacer que una función sea más flexible y para proporcionar valores predeterminados en caso de que el usuario no los especifique.

Código:

```
def multiplicar(a, b=2):
    return a * b
resultado1 = multiplicar(3)     # b toma el valor por defecto de 2
resultado2 = multiplicar(3, 5)  # b toma el valor especificado de 5
```

Estos son los tres tipos principales de argumentos que se utilizan en programación, y su elección depende de la flexibilidad y la claridad que se desee en sus funciones y llamadas de funciones. Puede combinar argumentos posicionales y de palabra clave en una llamada a función según sus necesidades.

«Paso por valor» y «paso por referencia» son dos conceptos relacionados con la forma en que se manejan los argumentos en las funciones, y su comportamiento varía según el lenguaje de programación. Cómo funcionan estos dos enfoques:

### 2.2.3 Paso por valor vs. paso por referencia

- **Paso por valor (Pass by Value)**

En el paso por valor, se pasa una copia del valor del argumento a la función. Esto significa que cualquier modificación realizada en el parámetro dentro de la función no afectará la variable original que se pasó como argumento.

Características del paso por valor:

a. Los cambios realizados dentro de la función no afectan la variable original.
b. Se utiliza comúnmente en lenguajes como C, C++, Java (para tipos primitivos) y algunos otros.

- **Paso por referencia (Pass by Reference)**

En el paso por referencia, se pasa una referencia o un enlace al argumento original a la función, en lugar de copiar su valor. Esto significa que cualquier modificación realizada en el parámetro dentro de la función afectará directamente a la variable original que se pasó como argumento.

Características del paso por referencia:

a. Los cambios realizados dentro de la función afectan a la variable original.
b. Se utiliza comúnmente en lenguajes como C++ (a través de punteros o referencias), *Python* (para objetos mutables) y algunos otros.

Código:

```
def duplicar(lista):
    for i in range(len(lista)):
        lista[i] = lista[i] * 2

mi_lista = [1, 2, 3]
duplicar(mi_lista)
# 'mi_lista' ahora es [2, 4, 6], ya que la función operó directamente sobre ella
```

Es importante entender cómo se maneja el paso de argumentos en el lenguaje de programación que estés utilizando, ya que puede tener un impacto significativo en el comportamiento de sus programas y en la gestión de la memoria. La mayoría de los lenguajes ofrecen un mecanismo para elegir entre paso por valor y paso por referencia, o una combinación de ambos, dependiendo del tipo de dato o de la declaración de la función.

## 2.3 Retorno de valores y sentencias return

### 2.3.1 Uso de la sentencia return

La sentencia **return** en *Python* se utiliza en una función para especificar el valor que la función debe devolver cuando se llama. En otras palabras, **return** permite que una función genere un resultado que puede ser utilizado por el programa principal o por otras partes del código. Aquí tiene algunos aspectos importantes sobre el uso de **return** *Python*:

- **Sintaxis básica**

  La sintaxis básica de **return** es la siguiente:

```
def nombre_de_la_funcion(parametros):
    # Código de la función
    return valor_a_devolver
```

- **Valor para devolver**

  El valor que se coloca después de la palabra clave **return** es lo que la función devuelve cuando se llama. Puede ser cualquier tipo de dato válido en *Python*, incluyendo números, cadenas, listas, objetos personalizados, entre otros.

Código:

```
def suma(a, b):
    resultado = a + b
    return resultado
```

- **Múltiples declaraciones de return**

  Una función puede tener múltiples declaraciones de **return**. La ejecución de la función se detendrá tan pronto como se encuentre una declaración de **return**, y el valor especificado se devolverá.

Código:

```
def es_positivo(numero):
    if numero > 0:
        return True
    else:
        return False
```

- **Valor por defecto de return**

  Si una función no tiene una declaración de **return** o si se ejecuta sin alcanzar una declaración de **return**, la función

devuelve automáticamente el valor especial **None**. **None** se utiliza para representar la ausencia de un valor.

Código:

```
def funcion_sin_return():
    pass  # Esta función devuelve automáticamente None
resultado = funcion_sin_return()
print(resultado)  # Output: None
```

- **Usos comunes**

  Las declaraciones de **return** se utilizan comúnmente para devolver resultados de cálculos, procesamientos de datos, llamadas a otras funciones, etc. También son útiles para terminar la ejecución de una función antes de que alcance su final, si es necesario.

Código:

```
def calcular_cuadrado(numero):
    return numero ** 2

def saludar(nombre):
    if nombre:
```

```
    return "Hola, " + nombre
else:
    return "Hola, desconocido"
```

En resumen, la sentencia **return** es fundamental en *Python* para devolver valores desde una función. Permite que las funciones sean más útiles al proporcionar resultados que se pueden utilizar en otros lugares de su programa.

### 2.3.2 Valores de retorno múltiple

En *Python*, es posible devolver múltiples valores desde una función utilizando una tupla, una lista u otro tipo de estructura de datos. Esto se conoce como "retorno múltiple". A continuación, se muestra cómo se puede lograr en *Python*:

Usando una tupla:

Una de las formas más comunes de devolver múltiples valores desde una función es empacar los valores en una tupla y luego devolver esa tupla. Aquí hay un ejemplo:

Código:

```
def obtener_coordenadas():
    x = 10
    y = 20
    z = 30
    return x, y, z  # Los valores se empaquetan en una tupla y se devuelven

coordenadas = obtener_coordenadas()
print(coordenadas)  # Output: (10, 20, 30)
```

Para desempaquetar los valores de la tupla, simplemente puede asignarlos a variables individuales:

Código:

```
x, y, z = obtener_coordenadas()
print(x)  # Output: 10
print(y)  # Output: 20
print(z)  # Output: 30
```

**Usando una lista o cualquier otra estructura de datos:**

No está limitado a usar solo tuplas; se puede usar cualquier estructura de datos para devolver múltiples valores, como listas, diccionarios o incluso objetos personalizados. Aquí un ejemplo con una lista:

Código:

```
def obtener_datos():
    nombre = "Alice"
    edad = 30
    ciudad = "Nueva York"
    return [nombre, edad, ciudad]  # Los valores se almacenan en una lista y se devuelven

datos = obtener_datos()
print(datos)  # Output: ['Alice', 30, 'Nueva York']
```

Para acceder a los valores en la lista, simplemente puede usar la indexación:

Código:

```
nombre = datos[0]
edad = datos[1]
ciudad = datos[2]
print(nombre)  # Output: 'Alice'
print(edad)    # Output: 30
print(ciudad)  # Output: 'Nueva York'
```

El uso de retornos múltiples es útil cuando desea devolver información relacionada desde una función sin tener que crear una estructura de datos más compleja o un objeto personalizado. Esto hace que el código sea más legible y fácil de entender en muchos casos.

### 2.3.3 Funciones sin retorno explícito

En *Python*, las funciones no siempre necesitan tener una declaración de **return** explícita. Si una función no contiene una declaración de **return** o si se ejecuta sin alcanzar una declaración de **return**, automáticamente devuelve el valor especial **None**. Esto es útil para funciones que realizan un trabajo sin necesidad de devolver un valor específico. Algunos ejemplos de funciones sin retorno explícito:

- **Funciones de impresión**

  Las funciones que simplemente imprimen información en la consola, como **print**(), no necesitan un retorno explícito, ya que su propósito principal es mostrar datos en lugar de devolverlos.

Código:

```
def saludar(nombre):
    print("Hola, " + nombre)

saludar("Alice")  # Output: Hola, Alice
```

- **Funciones de modificación de datos**

  Las funciones que modifican datos o realizan acciones, pero no necesitan devolver un valor específico tampoco requieren una declaración de **return**. Por ejemplo, una función que agrega elementos a una lista:

Código:

```
def agregar_elemento(lista, elemento):
    lista.append(elemento)

mi_lista = [1, 2, 3]
agregar_elemento(mi_lista, 4)
print(mi_lista)  # Output: [1, 2, 3, 4]
```

- **Funciones de inicialización o configuración**

  A menudo, se tienen funciones que configuran el estado inicial de algún objeto o realizan tareas de inicialización, pero no necesitan devolver un valor específico.

Código:

```
  def configurar_conexion():
  # Realizar configuración de conexión
  print("Conexión configurada correctamente")

configurar_conexion()  # Output: Conexión configurada
correctamente
```

- **Funciones de control de flujo**

  En ocasiones, las funciones se utilizan para controlar el flujo de ejecución del programa sin devolver valores específicos. Por ejemplo, una función que verifica si un usuario tiene permisos:

Código:

```
def tiene_permisos(usuario):
    if usuario == "admin":
        print("El usuario tiene permisos de administrador")
    else:
        print("El usuario no tiene permisos de administrador")

tiene_permisos("alice")  # Output: El usuario no tiene permisos
de administrador
```

En resumen, las funciones en *Python* no siempre necesitan un retorno explícito. Si no se especifica una declaración de **return**, la función automáticamente devuelve **None**. Esto es útil para funciones que realizan acciones o trabajos específicos sin necesidad de devolver un valor concreto.

## 2.4 Ámbito y variables

### 2.4.1 Variables locales y globales

En *Python*, las variables pueden tener un alcance local o global, lo que significa que pueden ser accesibles desde diferentes partes de su código en función de dónde se declaren. Aquí tiene una explicación de las variables locales y globales:

**Variables locales**

a. Las variables locales se declaran dentro de una función y solo son accesibles dentro de esa función. Esto significa que su alcance está limitado al bloque de código en el que se definen.

b. Las variables locales se crean cuando se ejecuta la función y se destruyen cuando la función termina su ejecución. No son accesibles desde fuera de la función.

c. Si intenta acceder a una variable local desde fuera de la función en la que se declaró, obtendrá un error de "NameError" o "Variable is not defined".

Código:

```
def funcion_local():
    variable_local = 10
    print(variable_local)

funcion_local()  # Output: 10
print(variable_local)  # Esto generará un error
```

**Variables globales**

a. Las variables globales se declaran fuera de cualquier función y son accesibles desde cualquier parte del código, ya sea dentro de una función o fuera de ella.

b. Las variables globales mantienen su valor a lo largo de toda la ejecución del programa, a menos que se cambien explícitamente en algún punto.

c. Puede acceder y modificar variables globales desde dentro de una función utilizando la palabra clave **global**, pero es importante tener en cuenta que si solo las accedes (sin usar **global**), *Python* considerará que estás creando una variable local con el mismo nombre en el ámbito de la función.

Código:

```
variable_global = 20

def funcion_global():
    global variable_global
    variable_global += 5
    print(variable_global)

funcion_global()  # Output: 25
print(variable_global)  # Output: 25
```

Es importante ser consciente del alcance de las variables en *Python* para evitar comportamientos inesperados o errores en su código. En general, se recomienda usar variables locales siempre que sea posible, ya que esto promueve un diseño más modular y reduce la posibilidad de efectos secundarios no deseados. Las variables globales deben usarse con precaución y solo cuando sea necesario compartir información entre diferentes partes del código.

### 2.4.2 Reglas de ámbito en *Python*

*Python* tiene reglas de ámbito (alcance) que determinan dónde una variable es accesible y cuándo se puede utilizar. Estas reglas definen cómo las variables se asignan a diferentes ámbitos dentro de un programa. Aquí están las principales reglas de ámbito en *Python*:

**Ámbito local (Local Scope)**

- Las variables declaradas dentro de una función tienen un ámbito local.
- Estas variables solo son accesibles desde dentro de esa función y no pueden ser vistas o modificadas desde fuera de ella.
- El ámbito local se crea cuando se entra en la función y se destruye cuando se sale de la función.

Código:

```
def funcion_local():
    variable_local = 10
    print(variable_local)
```

```
funcion_local()  # variable_local solo es visible dentro de la función
print(variable_local)  # Esto generará un error
```

**Ámbito encerrado (Enclosing Scope)**

- En *Python*, las funciones pueden estar anidadas dentro de otras funciones.
- Una función interna puede acceder a las variables de su función contenedora (ámbito encerrado) pero no puede modificarlas directamente.
- Si una variable no se encuentra en el ámbito local, *Python* buscará en el ámbito encerrado y luego en el ámbito global.

Código:

```
def exterior():
    variable_encerrada = 20

    def interior():
        print(variable_encerrada)

    interior()

exterior()  # Output: 20
```

**Ámbito global (Global scope)**

- Las variables declaradas fuera de cualquier función o clase tienen un ámbito global.
- Las variables globales son accesibles desde cualquier parte del código, tanto dentro como fuera de las funciones.
- Puede acceder a variables globales desde dentro de una función sin necesidad de utilizar la palabra clave **global**, pero si desea modificar una variable global desde una función, debe usar **global**.

Código:

```
variable_global = 30

def funcion_global():
    global variable_global
    variable_global += 5
    print(variable_global)

funcion_global()  # Output: 35
```

**Built-in scope (Ámbito incorporado)**

- *Python* tiene un ámbito incorporado que contiene todas las funciones y objetos incorporados en el lenguaje, como **print**(), **len**(), **str**(), entre otros.
- Puede acceder a estas funciones y objetos directamente desde cualquier parte de su programa sin necesidad de declararlos.

Código:

```
print(len([1, 2, 3]))  # Output: 3
```

En resumen, *Python* sigue un conjunto de reglas de ámbito que determinan el alcance de las variables en su programa. Es importante comprender estas reglas para evitar errores relacionados con el alcance de las variables y para escribir un código más claro y modular.

### 2.4.3 Uso de la palabra clave global

La palabra clave **global** en *Python* se utiliza para indicar que una variable que se está utilizando dentro de una función es

una variable global en lugar de una variable local. Esto permite que la función modifique la variable global en lugar de crear una nueva variable local con el mismo nombre. Aquí tiene un ejemplo de cómo se utiliza la palabra clave **global**:

Código:

```
variable_global = 10

def modificar_variable():
    global variable_global  # Indica que se esta utilizando la variable global
    variable_global = 20  # Modifica la variable global

print("Antes de llamar a la función:", variable_global)
modificar_variable()
print("Después de llamar a la función:", variable_global)
```

En este ejemplo:

a. Declara una variable global llamada **variable_global** con un valor inicial de 10.
b. Definir una función llamada **modificar_variable**, que modifica la variable global **variable_global** utilizando la palabra clave **global**.
c. Antes de llamar a la función, imprimir el valor de **variable_global**, que es 10.

d. Luego, llamar a la función **modificar_variable**, que modifica **variable_global** y le asigna el valor 20.
e. Después de llamar a la función, imprimir nuevamente el valor de **variable_global**, que ahora es 20 debido a que se modificó globalmente dentro de la función.

Es importante tener en cuenta que si no usa la palabra clave **global** dentro de la función, *Python* creará una nueva variable local con el mismo nombre en lugar de modificar la variable global. La palabra clave **global** le dice a *Python* que desea trabajar con la variable global en lugar de crear una nueva variable local con el mismo nombre.

## 2.5 Funciones como Objetos

### 2.5.1 Asignación de funciones a variables

En *Python*, puede asignar funciones a variables de la misma manera que asigna cualquier otro tipo de datos a una variable. Esto es útil cuando desea tratar una función como un objeto de primera clase, lo que permite pasarla como argumento a otras funciones, almacenarla en estructuras de datos o incluso

renombrarla. Aquí tiene un ejemplo de cómo asignar funciones a variables:

Código:

```
# Definir una función
def saludar(nombre):
    return "Hola, " + nombre

# Asignar la función a una variable
mi_funcion = saludar

# Ahora  llamar a la función a través de la variable
mensaje = mi_funcion("Alice")
print(mensaje)  # Output: Hola, Alice
```

En este ejemplo:

a. Definir una función llamada **saludar** que toma un argumento **nombre** y devuelve un mensaje de saludo.
b. Luego, asignar la función **saludar** a la variable **mi_funcion**. Ahora, **mi_funcion** es una referencia a la función **saludar**.
c. Usar la variable **mi_funcion** para llamar a la función, pasando un argumento **Alice**, y almacenar el resultado en la variable **mensaje**.
d. Imprimir el mensaje, que muestra el resultado de la llamada a la función a través de la variable.

Este enfoque es especialmente útil cuando desea pasar funciones como argumentos a otras funciones o cuando necesita almacenar un conjunto de funciones en una estructura de datos, como una lista o un diccionario. También es útil para implementar patrones de diseño como el patrón de estrategia, donde puede intercambiar diferentes implementaciones de una función en tiempo de ejecución.

Aquí hay otro ejemplo que muestra cómo almacenar funciones en una lista y luego llamarlas en un bucle:

Código:

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

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

def multiplicacion(a, b):
    return a * b

# Almacenar las funciones en una lista
operaciones = [suma, resta, multiplicacion]

for operacion in operaciones:
    resultado = operacion(5, 3)
    print(resultado)
```

Este código ejecutará cada una de las funciones almacenadas en la lista **operaciones** con los argumentos **5** y **3**, y mostrará los resultados en el bucle.

### 2.5.2 Pasar funciones como argumentos

En *Python*, puede pasar funciones como argumentos a otras funciones. Este concepto se basa en el principio de funciones de primera clase, que permite tratar las funciones como cualquier otro tipo de dato. Pasar funciones como argumentos es útil para implementar patrones de diseño como el patrón de estrategia, donde puede cambiar el comportamiento de una función llamada en tiempo de ejecución. Aquí tiene un ejemplo de cómo hacerlo:

Código:

```
# Definir funciones que realizarán diferentes operaciones
def suma(a, b):
    return a + b

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

def multiplicacion(a, b):
```

```
    return a * b

# Crear una función que toma dos números y una función como argumento
def aplicar_operacion(a, b, operacion):
    return operacion(a, b)

# Llamar a la función aplicar_operacion con diferentes operaciones
resultado_suma = aplicar_operacion(5, 3, suma)
resultado_resta = aplicar_operacion(5, 3, resta)
resultado_multiplicacion = aplicar_operacion(5, 3, multiplicacion)

print("Suma:", resultado_suma)
print("Resta:", resultado_resta)
print("Multiplicación:", resultado_multiplicacion)
```

En este ejemplo:

a. Definir tres funciones (**suma**, **resta**, **multiplicacion**) que realizan operaciones matemáticas diferentes.
b. Crear una función llamada **aplicar_operacion** que toma dos números (**a** y **b**) y una función (**operacion**) como argumentos. Esta función aplica la operación especificada a los números **a** y **b**.
c. Luego, se llama a **aplicar_operacion** tres veces con diferentes operaciones (**suma**, **resta**, **multiplicacion**)

como argumentos, lo que permite realizar diversas operaciones matemáticas.

d. Finalmente, se imprime los resultados de cada operación.

Este enfoque de pasar funciones como argumentos es muy poderoso y versátil. Puede utilizarlo para abstraer el comportamiento de las funciones y crear código más flexible y reutilizable. Además, *Python* proporciona funciones integradas como **map**(), **filter**(), **sorted**() y otras que aceptan funciones como argumentos, lo que facilita aún más el uso de esta técnica.

### 2.5.3 Funciones anónimas (lambda)

En *Python*, las funciones anónimas se crean utilizando la palabra clave **lambda**. Estas funciones se conocen como **funciones lambda** o **lambda expressions**. A diferencia de las funciones regulares definidas con **def**, las funciones **lambda** son funciones pequeñas y anónimas que pueden tener una o varias líneas de código y se utilizan generalmente en situaciones donde necesitas una función temporal y simple.

La sintaxis básica de una función **lambda** es la siguiente:

**lambda** argumentos: expresión:

- **argumentos** son los argumentos de la función, similares a los argumentos de una función definida con def.
- **expresión** es el resultado que debe devolver la función.

Aquí tiene algunos ejemplos de cómo se utilizan las funciones **lambda**:

**Función lambda simple**

Código:

```
suma = lambda a, b: a + b
resultado = suma(3, 5)
print(resultado)  # Output: 8
```

**Función lambda en una lista**

Puede utilizar funciones lambda junto con funciones como **map**(), **filter**(), y **sorted**() para aplicar operaciones a una lista.

Código:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x ** 2, numeros))
print(cuadrados)  # Output: [1, 4, 9, 16, 25]
```

**Función lambda en una expresión condicional**

Las funciones lambda son útiles cuando necesita una función pequeña en una expresión condicional.

Código:

```
es_par = lambda x: x % 2 == 0
resultado = es_par(4)
print(resultado)  # Output: True
```

**Función lambda como argumento de ordenamiento**

Puede utilizar funciones **lambda** para personalizar el ordenamiento de elementos en una lista.

Código:

```
personas = [("Alice", 25), ("Bob", 30), ("Charlie", 20)]
personas_ordenadas = sorted(personas, key=lambda x: x[1])
print(personas_ordenadas)  # Output: [('Charlie', 20), ('Alice', 25), ('Bob', 30)]
```

Las funciones **lambda** son especialmente útiles cuando necesita definir una función simple y rápida sin la necesidad de darle un nombre. Sin embargo, su uso debe ser moderado, ya que las funciones más complejas y con más lógica son más legibles y mantenibles cuando se definen con def.

## 2.6 Módulos y modularidad

### 2.6.1 Organización del código en módulos

La organización del código en módulos es una práctica esencial en programación para mantener el código limpio, modular y fácil de mantener. Los módulos son archivos de *Python* que contienen funciones, clases y variables que se pueden reutilizar en otros programas o módulos. Aquí hay algunos conceptos clave sobre cómo organizar el código en módulos en *Python*:

**Creación de módulos**

Para crear un módulo en *Python*, simplemente se crea un archivo con extensión **.py** y colocar en él, las funciones, clases y

variables que deseas organizar en ese módulo. Por ejemplo, puede tener un módulo llamado **mi_modulo.py** con el siguiente contenido:

Código:

```
# mi_modulo.py

def saludar(nombre):
    return "Hola, " + nombre

def duplicar(numero):
    return numero * 2

class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def obtener_valor(self):
        return self.valor
```

**Importación de módulos**

Para utilizar las funciones, clases o variables de un módulo en otro archivo de *Python*, debe importar ese módulo. Puede hacerlo utilizando la palabra clave **import**. Por ejemplo, para importar el módulo **mi_modulo.py**:

Código:

```
import mi_modulo

mensaje = mi_modulo.saludar("Alice")
```

También puedes importar funciones específicas o clases de un módulo en lugar de importar todo el módulo:

Código:

```
from mi_modulo import saludar

mensaje = saludar("Alice")
```

**Alias de módulos**

Puede asignar alias (nombres alternativos) a los módulos para hacer que su uso sea más conciso o evitar conflictos de nombres. Por ejemplo:

Código:

```
import mi_modulo as mm

mensaje = mm.saludar("Alice")
```

**Organización de múltiples módulos**

Cuando su proyecto se vuelve más grande, es común organizar sus módulos en directorios o paquetes. Un paquete es simplemente un directorio que contiene uno o más módulos y un archivo especial llamado **__init__.py**. *Python* reconoce estos directorios como paquetes y le permite organizar su código de manera más estructurada.

Por ejemplo:

```
mi_proyecto/
├── __init__.py
├── modulo1.py
├── modulo2.py
└── subpaquete/
    ├── __init__.py
    └── modulo3.py
```

Puede importar módulos dentro de un paquete utilizando la notación de punto, por ejemplo: **import mi_proyecto.modulo1**.

**Main y ejecución del módulo**

Puede usar la variable **__name__** para determinar si un módulo se está ejecutando como programa principal o si se está importando en otro módulo. Esto es útil para escribir código que se ejecuta solo cuando el módulo se ejecuta directamente, pero no cuando se importa como un módulo en otro programa.

Por ejemplo:

Código:

```
# mi_modulo.py

def saludar(nombre):
  return "Hola, " + nombre

if __name__ == "__main__":
  print("Este es el módulo principal")
  print(saludar("Alice"))
```

Cuando ejecuta **mi_modulo.py** como programa principal, verá el mensaje «Este es el módulo principal». Sin embargo, si importa **mi_modulo.py** en otro programa, ese código condicional no se ejecutará.

La organización del código en módulos y paquetes es una práctica fundamental para escribir código *Python* limpio y mantenible, especialmente en proyectos más grandes. Ayuda a dividir su código en partes más pequeñas y lógicas, lo que facilita la colaboración y el mantenimiento a largo plazo.

### 2.6.2 Importación de módulos y funciones

La importación de módulos y funciones es una parte fundamental de la programación en *Python*, ya que te permite reutilizar código y organizar sus programas en múltiples archivos.

**Importar un módulo completo**

Para importar un módulo completo, puede utilizar la palabra clave **import** seguida del nombre del módulo. Después, puede acceder a las funciones y objetos del módulo utilizando la notación de punto.

Código:

```
import modulo

resultado = modulo.funcion()
```

**Importar funciones específicas de un módulo**

Si solo desea importar funciones u objetos específicos de un módulo, puede usar la palabra clave **from** seguido del nombre del módulo y luego la palabra clave **import** seguida de los nombres de las funciones u objetos que desea importar.

Código:

```
from modulo import funcion1, funcion2

resultado1 = funcion1()
resultado2 = funcion2()
```

**Importar un módulo con un alias**

Puede asignar un alias (un nombre alternativo) a un módulo importado utilizando la palabra clave **as**.

Código:

```
import modulo as mod

resultado = mod.funcion()
```

**Importar todas las funciones y objetos de un módulo**

Puede importar todas las funciones y objetos de un módulo utilizando el asterisco *****. Sin embargo, este enfoque no se recomienda en general, ya que puede hacer que su código sea menos legible y propenso a conflictos de nombres.

Código:

```
from modulo import *

resultado = funcion()
```

**Importar módulos y funciones de módulos dentro de paquetes**

Si tiene un paquete con varios módulos y desea importarlos, puede utilizar la notación de punto para acceder a los módulos y funciones dentro del paquete.

Código:

```
from paquete import modulo
from paquete.subpaquete import otro_modulo

resultado1 = modulo.funcion()
resultado2 = otro_modulo.otra_funcion()
```

Tenga en cuenta que es importante que los módulos que desee importar, se encuentren en el mismo directorio que el archivo desde el cual está realizando la importación, o que estén en una ubicación que exista en la ruta de búsqueda de *Python*. En caso contrario, puede utilizar rutas relativas o absolutas para especificar la ubicación de los módulos.

### 2.6.3 Creación de módulos personalizados

La creación de módulos personalizados en *Python* le permite organizar su código en archivos separados y reutilizables.

**Crea un archivo .py para módulo:**

Comience creando un archivo **.py** en el directorio de su proyecto. Por ejemplo, si desea crear un módulo llamado **mi_modulo**, cree un archivo llamado **mi_modulo.py**.

**Defina funciones, clases o variables en el módulo**

En el archivo **mi_modulo.py**, defina las funciones, clases o variables que desea que estén disponibles en su módulo. Por ejemplo:

Código:

```
# mi_modulo.py

def saludar(nombre):
    return "Hola, " + nombre

def duplicar(numero):
    return numero * 2

class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def obtener_valor(self):
        return self.valor
```

**Usa el módulo en otros archivos**

Ahora puede importar y usar su módulo personalizado en otros archivos de *Python*. Puede hacerlo utilizando la palabra clave **import**. Por ejemplo:

Código:

```
import mi_modulo

mensaje = mi_modulo.saludar("Alice")
```

También puede importar funciones o clases específicas si no desea importar todo el módulo:

Código:

```
from mi_modulo import saludar

mensaje = saludar("Alice")
```

**Ejecución del módulo como programa principal (opcional)**

Puede agregar un bloque condicional en su módulo para que ciertas partes de código se ejecuten solo cuando el módulo se ejecute como programa principal y no cuando se importe en otro archivo. Esto se hace utilizando la variable **__name__.**

Código:

```
# mi_modulo.py

def saludar(nombre):
    return "Hola, " + nombre

if __name__ == "__main__":
    print("Este es el módulo principal")
    print(saludar("Alice"))
```

Cuando ejecuta **mi_modulo.py** directamente, el código dentro del bloque **if __name__ == "__main__":** se ejecutará, pero si importa **mi_modulo** en otro archivo, ese código no se ejecutará.

**Organización en paquetes (opcional)**

Si el proyecto se vuelve más grande y tiene varios módulos relacionados, puede organizarlos en paquetes. Para hacerlo, cree un directorio con un archivo **__init__.py** en el mismo nivel que sus módulos y coloque sus módulos dentro de ese directorio. Esto convertirá ese directorio en un paquete *Python*.

Por ejemplo:

```
mi_proyecto/
├── __init__.py
├── mi_modulo.py
└── subdirectorio/
    ├── __init__.py
    └── otro_modulo.py
```

Luego, puede importar módulos desde un paquete utilizando la notación de punto, como **from mi_proyecto.subdirectorio import otro_modulo**.

Al seguir estos pasos, puede crear y utilizar módulos personalizados en *Python* para organizar y reutilizar su código de manera efectiva. Esto es especialmente útil para proyectos más grandes y complejos.

## 2.7 Recursión

### 2.7.1 Concepto de recursión

La recursión es un concepto fundamental en programación y matemáticas que se refiere a la capacidad de una función o procedimiento de llamarse a sí mismo para resolver un problema de manera iterativa. En programación, una función recursiva es aquella que se llama a sí misma dentro de su propio cuerpo para resolver un problema más grande dividiéndolo en casos más pequeños y manejables.

**La recursión se basa en dos componentes clave**

**Caso base:** este es el punto de terminación de la recursión. Cuando se alcanza el caso base, la función recursiva deja de llamarse a sí misma y comienza a regresar valores para construir la solución final. El caso base es esencial para evitar que la recursión continúe indefinidamente.

**Caso recursivo:** en el caso recursivo, la función se llama a sí misma con argumentos diferentes para resolver un subproblema más pequeño. La idea es que cada llamada recursiva resuelve una parte más pequeña del problema, y eventualmente, todas las llamadas recursivas convergen hacia el caso base.

Un ejemplo clásico de recursión es el cálculo del factorial de un número:

Código:

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

En este ejemplo, el caso base es cuando **n** es igual a 0 o 1, y en esos casos, la función devuelve 1. En el caso recursivo, la función **factorial** se llama a sí misma con **n-1**, lo que divide el problema original en subproblemas más pequeños hasta que se alcanza el caso base. Por ejemplo, **factorial(5)** se resuelve de la siguiente manera:

factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1

Finalmente, se obtiene el resultado:

factorial(5) = 5 * 4 * 3 * 2 * 1 = 120

La recursión es una técnica poderosa que se utiliza en muchos algoritmos y problemas de programación, pero es importante tener en cuenta que debe usarse con precaución y asegurarse de que haya un caso base y que se avance hacia él en cada llamada recursiva para evitar bucles infinitos.

### 2.7.2 Implementación de funciones recursivas

La implementación de funciones recursivas implica escribir funciones que se llaman a sí mismas para resolver un problema dividido en subproblemas más pequeños. Aquí un ejemplo de cómo implementar funciones recursivas en *Python*, junto con algunas pautas para escribir funciones recursivas efectivas:

Ejemplo de una función recursiva para calcular el factorial de un número:

Código:

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

En este ejemplo, la función **factorial** se llama a sí misma con **n-1** en el caso recursivo, lo que resuelve un subproblema más pequeño. El caso base verifica si **n** es igual a 0 o 1, en cuyo caso devuelve 1.

**Pautas para escribir funciones recursivas efectivas**

**Definir un caso base:** es fundamental definir un caso base que indique cuándo debe detenerse la recursión. Sin un caso base adecuado, la función recursiva continuará llamándose indefinidamente, lo que conducirá a un desbordamiento de pila.

**Divide y conquista:** divide el problema original en subproblemas más pequeños que se pueden resolver de manera similar. Cada llamada recursiva debe acercarse al caso base.

**Llamada a sí misma:** la función debe llamarse a sí misma dentro de su propio cuerpo para resolver subproblemas.

**Avance hacia el caso base:** asegúrese de que cada llamada recursiva avance hacia el caso base. Esto significa que los argumentos pasados a la función recursiva deben acercarse al caso base en cada llamada.

**Gestión de resultados parciales:** en funciones recursivas que resuelven un problema dividiéndolo en subproblemas, debe asegurarse de que los resultados parciales se combinen adecuadamente para obtener el resultado final.

**Optimización:** en algunos casos, la recursión puede ser ineficiente debido a la sobrecarga de llamadas. En tales casos, considera la posibilidad de utilizar técnicas como la memorización o la programación dinámica para evitar llamadas innecesarias.

**Probar con casos pequeños:** antes de escribir una función recursiva, pruebe con casos pequeños y comprenda cómo funciona la recursión en ese contexto. Esto le ayudará a depurar y evitar bucles infinitos.

Recuerde que la recursión es una técnica poderosa, pero debe usarse con precaución y comprenderse completamente para evitar problemas de rendimiento y errores de programación.

### 2.7.3 Casos de uso y consideraciones

La recursión es una técnica de programación poderosa y versátil que se utiliza en diversos casos de uso.

**Casos de uso de la recursión**

a. **Cálculos matemáticos:** la recursión se utiliza para resolver problemas matemáticos como el cálculo de factoriales, números de Fibonacci y exponenciación rápida.

b. **Estructuras de datos recursivas:** las estructuras de datos como árboles y listas enlazadas pueden definirse y manipularse de manera recursiva. Por ejemplo, la búsqueda en un árbol binario es un caso típico de recursión.

c. **Divide y vencerás:** muchos algoritmos de divide y vencerás, como la ordenación rápida (*quicksort*) y la búsqueda binaria, se implementan de manera recursiva.

d. **Problemas que se dividen en subproblemas idénticos o similares:** cuando un problema se puede dividir en subproblemas idénticos o similares, la recursión es una opción natural. Ejemplos incluyen la torre de *Hanoi* y el algoritmo de búsqueda en profundidad (DFS) en grafos.

e. **Árboles de decisión y *backtracking*:** la recursión se utiliza en algoritmos de búsqueda de árboles de decisión y backtracking para resolver problemas de decisión, como el problema de las N reinas y el *Sudoku*.

**Consideraciones al usar la recursión**

a. **Casos base y convergencia:** debe asegurarse de que, la función recursiva tenga casos base correctos y que cada llamada recursiva se acerque al caso base. De lo contrario, puede enfrentar un bucle infinito.

b. **Desempeño:** la recursión puede ser ineficiente en comparación con las soluciones iterativas para algunos problemas debido a la sobrecarga de llamadas. En tales casos, considera otras técnicas como la programación dinámica o la iteración.

c. **Espacio en la pila:** cada llamada recursiva agrega una nueva entrada en la pila de llamadas, lo que puede consumir memoria y conducir a un desbordamiento de pila (*stack overflow*) si tiene demasiadas llamadas recursivas. Esto se puede mitigar ajustando la recursión o utilizando una iteración en su lugar.

d. **Memorización (caché):** en algunos casos, puede mejorar el rendimiento de la recursión utilizando técnicas de memorización, donde almacena los resultados de llamadas recursivas anteriores para evitar recalcularlos.

e. **Legibilidad y mantenibilidad:** aunque la recursión puede ser una solución elegante para ciertos problemas, a veces puede hacer que el código sea menos legible. Debe encontrar un equilibrio entre la elegancia y la claridad del código.

En resumen, la recursión es una herramienta poderosa en la programación, pero debe utilizarse con cuidado y comprenderse completamente. Es importante identificar los casos en los que la recursión es apropiada y considerar las implicaciones de rendimiento y espacio en la pila al implementarla. En algunos casos, la recursión es la elección obvia y en otros, puede ser preferible utilizar enfoques iterativos o técnicas adicionales para optimizar su uso.

## 2.8 Funciones integradas y bibliotecas

### 2.8.1 Exploración de funciones integradas útiles

*Python* ofrece una amplia gama de funciones integradas (built-in functions) que son esenciales para realizar tareas comunes en la programación. Aquí hay una exploración de algunas de las funciones integradas más útiles:

**Funciones de conversión de tipos**

- **int**(): convierte un valor a un entero.
- **float**(): convierte un valor a un número de punto flotante.
- **str**(): convierte un valor a una cadena de texto.
- **list**(): convierte un iterable en una lista.
- **tuple**(): convierte un iterable en una tupla.
- **dict**(): crea un diccionario a partir de pares clave-valor.

**Funciones matemáticas**

- **abs**(): devuelve el valor absoluto de un número.
- **pow**(): calcula una potencia (potencia(x, y) es equivalente a x**y).

- **round**(): redondea un número al número entero más cercano.
- **sum**(): calcula la suma de elementos en una secuencia.

**Funciones de cadenas de texto**

- **len**(): devuelve la longitud (número de caracteres) de una cadena o secuencia.
- **str.capitalize**(): devuelve una copia de la cadena con la primera letra en mayúscula.
- **str.upper**(): devuelve una copia de la cadena en mayúsculas.
- **str.lower**(): devuelve una copia de la cadena en minúsculas.
- **str.split**(): divide una cadena en una lista de subcadenas según un delimitador.
- **str.join**(): combina una lista de cadenas en una sola cadena usando un separador.

**Funciones de secuencias**

- **max**(): devuelve el valor máximo de una secuencia.
- **min**(): devuelve el valor mínimo de una secuencia.

- **sorted**(): devuelve una lista ordenada de una secuencia.
- **enumerate**(): devuelve pares índice-valor de una secuencia.
- **zip**(): combina varias secuencias en tuplas emparejadas.

**Funciones de entrada/salida**

- **input**(): lee una línea de entrada desde el teclado.
- **print**(): imprime un mensaje en la consola.

**Funciones de manejo de archivos**

- **open**(): abre un archivo para lectura o escritura.
- **read**(): lee el contenido de un archivo.
- **write**(): escribe contenido en un archivo.
- **close**(): cierra un archivo abierto.

**Funciones de tiempo**

- **time.time**(): devuelve la hora actual en segundos desde la época (1 de enero de 1970).
- **time.sleep**(): pausa la ejecución del programa durante un número de segundos dado.

Estas son solo algunas de las funciones integradas útiles en *Python*. Hay muchas más disponibles en la biblioteca estándar de *Python*, y también puede definir sus propias funciones personalizadas para abordar necesidades específicas en su programa. Las funciones integradas le proporcionan una base sólida para realizar operaciones comunes en *Python* de manera eficiente y efectiva.

### 2.8.2 Introducción a bibliotecas externas

Las bibliotecas externas, también conocidas como módulos o paquetes, son conjuntos de código pre escrito y funcionalidades que se pueden importar y utilizar en programas de *Python*. Estas bibliotecas se desarrollan para resolver una variedad de problemas y extender las capacidades del lenguaje *Python*. A continuación, se proporciona una introducción a las bibliotecas externas en *Python*.

**Características clave de las bibliotecas externas**

a. **Reutilización de código:** las bibliotecas externas contienen código que ha sido desarrollado previamente

por otros programadores. Puede importar estas bibliotecas en su programa y utilizar las funciones y clases que proporcionan sin tener que escribir todo el código desde cero.

b. **Ampliación de funcionalidades:** las bibliotecas externas permiten ampliar las capacidades de *Python*. Pueden abordar una amplia variedad de tareas, desde matemáticas y manipulación de datos hasta desarrollo web, gráficos, redes y mucho más.

c. **Eficiencia y calidad:** las bibliotecas externas suelen ser altamente eficientes y de alta calidad, ya que son desarrolladas y mantenidas por comunidades de programadores y sometidas a pruebas exhaustivas.

**Cómo utilizar bibliotecas externas**

Para utilizar una biblioteca externa en *Python*, debe seguir estos pasos generales:

- **Instalación:** antes de poder usar una biblioteca, debe instalarla en su sistema. Esto se hace utilizando un administrador de paquetes de *Python*, como **pip**. Por ejemplo, para instalar la biblioteca **requests**, puede ejecutar el comando:

  pip install requests

- **Importación:** una vez instalada la biblioteca, puede importarla en su programa utilizando la palabra clave **import**. Por ejemplo:

  import requests

También puede importar funciones o clases específicas desde la biblioteca:

  from biblioteca import funcion

- **Uso:** ahora que ha importado la biblioteca, puede utilizar las funciones y clases que proporciona en su programa. Por ejemplo, con la biblioteca **requests**, puede realizar solicitudes HTTP:

```
import requests
response = requests.get("https://www.ejemplo.com")
```

- **Documentación:** para utilizar una biblioteca externa de manera efectiva, es importante consultar la documentación de la biblioteca. La documentación proporciona información sobre las funciones, clases y métodos disponibles, así como ejemplos de uso.

  Ejemplos de bibliotecas externas populares en *Python* incluyen:

  - **requests**: para realizar solicitudes HTTP.
  - **numpy** y **pandas**: para manipulación de datos y análisis numérico.
  - **matplotlib** y **seaborn**: para visualización de datos.
  - **flask** y **Django**: para desarrollo web.
  - **tensorflow** y **PyTorch**: para aprendizaje profundo (*deep learning*).
  - **scikit-learn**: para aprendizaje automático (*machine learning*).

Las bibliotecas externas son una parte fundamental del ecosistema de *Python* y permiten aprovechar una amplia variedad de herramientas y capacidades para sus proyectos de programación.

### 2.8.3 Instalación y uso de bibliotecas con pip

La instalación y el uso de bibliotecas externas en *Python* se realizan comúnmente mediante la herramienta **pip**, que es el administrador de paquetes estándar de *Python*. A continuación, se detallan los pasos para instalar y usar bibliotecas con **pip**:

**Instalación de bibliotecas con pip**

- **Abrir una terminal o línea de comandos:** para instalar bibliotecas con **pip**, debe abrir una terminal o línea de comandos en su sistema operativo. Asegúrese de tener acceso a internet, ya que **pip** descargará las bibliotecas desde el repositorio **PyPI** (*Python Package Index*).

- **Instalar la biblioteca:** utilizar el siguiente comando para instalar una biblioteca específica. Reemplazar

**nombre_de_la_biblioteca** con el nombre de la biblioteca que deseas instalar.

bash
pip install nombre_de_la_biblioteca

Por ejemplo, para instalar la biblioteca **requests**, puede ejecutar:

```
bash
pip install requests
```

Esto descargará e instalará la biblioteca y todas las dependencias necesarias en su sistema.

- **Verificar la instalación:** puede verificar si la biblioteca se instaló correctamente ejecutando el siguiente comando para listar todas las bibliotecas instaladas:

  bash
  pip list

Esto mostrará una lista de todas las bibliotecas instaladas, incluida la que acaba de instalar.

**Uso de bibliotecas en su código**

Una vez que haya instalado una biblioteca con **pip**, puede importarla y utilizarla en sus programas de *Python* de la siguiente manera:

Código:

```
import nombre_de_la_biblioteca

# Usa funciones y clases de la biblioteca
resultado = nombre_de_la_biblioteca.funcion(parametro)
```

Por ejemplo, si ha instalado la biblioteca **requests**, puede importarla y utilizarla para realizar una solicitud HTTP:

Código:

```
import requests

response = requests.get("https://www.ejemplo.com")
print(response.text)
```

Recuerde que la forma exacta de usar una biblioteca dependerá de la documentación de esa biblioteca específica. Consulte la documentación de la biblioteca para obtener ejemplos y detalles sobre cómo utilizar sus funciones y clases.

**Actualización de bibliotecas con pip**

Puede mantener sus bibliotecas actualizadas utilizando **pip**. Para actualizar una biblioteca a su versión más reciente, ejecute el siguiente comando:

```
bash
pip install --upgrade nombre_de_la_biblioteca
```

Esto actualizará la biblioteca a la última versión disponible en **PyPI**.

En resumen, **pip** es una herramienta esencial para instalar y gestionar bibliotecas externas en *Python*. Permite acceder a una amplia gama de bibliotecas desarrolladas por la comunidad de *Python* y ampliar las capacidades de sus programas de manera eficiente.

## 2.9 Aplicaciones prácticas de funciones

### 2.9.1 Creación de calculadoras y herramientas útiles

Crear calculadoras y herramientas útiles en *Python* es un excelente ejercicio para mejorar sus habilidades de programación y para construir aplicaciones que puedan ser útiles en su vida cotidiana o en proyectos específicos. Aquí se presentan algunos ejemplos de calculadoras y herramientas útiles que se puede desarrollar en *Python*:

**1. Calculadora de propinas:** una calculadora que le permite ingresar el monto de la factura y calcular automáticamente la propina basada en un porcentaje que el usuario elija.

**2. Calculadora de conversión de unidades:** una herramienta que permite convertir entre diferentes unidades de medida, como longitud, peso, temperatura, etc.

**3. Calculadora de hipotecas:** una aplicación que ayuda a calcular los pagos mensuales de una hipoteca, teniendo en cuenta el monto del préstamo, la tasa de interés y el plazo.

**4. Calculadora de tiempo y fecha**: una herramienta que permite calcular la diferencia entre dos fechas y horas, o que sume o reste un número de días o semanas a una fecha determinada.

**5. Calculadora de IMC (Índice de Masa Corporal):** una aplicación que calcula el IMC a partir del peso y la altura ingresados y proporciona una categoría de peso (bajo peso, peso normal, sobrepeso, etc.).

**6. Herramienta de generación de contraseñas:** una herramienta que genera contraseñas seguras de forma aleatoria y permite especificar la longitud y los caracteres permitidos.

**7. Calculadora de tiempos de viaje:** una aplicación que calcula el tiempo estimado de viaje entre dos ubicaciones, teniendo en cuenta la distancia y la velocidad de viaje.

**8. Calculadora de interés compuesto:** una calculadora que ayuda a calcular el crecimiento de una inversión con interés compuesto, dada una tasa de interés anual y un período de tiempo.

**9. Herramienta de conversión de monedas:** una aplicación que permite convertir entre diferentes monedas extranjeras utilizando tasas de cambio en tiempo real o tasas predefinidas.

**10. Calculadora de calificaciones o promedios:** una calculadora que permite calcular el promedio de calificaciones, ya sea con diferentes pesos para las asignaturas o con diferentes sistemas de calificación.

Para desarrollar estas calculadoras y herramientas, puede utilizar las bibliotecas y módulos de *Python* que sean apropiados para cada caso. Por ejemplo, para interfaces gráficas de usuario, puede utilizar bibliotecas como **Tkinter** o **PyQt**. Para cálculos matemáticos, puede aprovechar bibliotecas como **math** o **NumPy**. Además, tenga en cuenta la usabilidad y la interfaz de usuario al diseñar estas herramientas para que sean fáciles de usar y comprender.

Estos son solo algunos ejemplos, y las posibilidades son casi ilimitadas. Piense en las necesidades que tiene o en los problemas que puede resolver con una calculadora o herramienta útil y ponga en práctica sus habilidades de programación para crearlas.

### 2.9.2 Procesamiento de listas y datos

El procesamiento de listas y datos es una parte fundamental de la programación en *Python*. *Python* ofrece una amplia gama de herramientas y técnicas para trabajar con listas, tuplas, diccionarios y otros tipos de datos. Aquí hay una introducción a algunas de las operaciones de procesamiento de datos más comunes en *Python*:

**1. Iteración:** la iteración es un concepto clave para procesar listas y datos. Puede utilizar bucles **for** para recorrer elementos en una lista, tupla o diccionario. Por ejemplo, para imprimir todos los elementos de una lista:

Código:

```
mi_lista = [1, 2, 3, 4, 5]
for elemento in mi_lista:
    print(elemento)
```

**2. Filtrado de datos:** puede utilizar bucles **for** con condicionales para filtrar datos basados en ciertos criterios. Por ejemplo, para imprimir solo los números pares de una lista:

Código:

```
mi_lista = [1, 2, 3, 4, 5]
for elemento in mi_lista:
    print(elemento)
```

**3. Comprensiones de listas:** las comprensiones de listas son una forma concisa y poderosa de crear listas nuevas basadas en una iteración y una expresión condicional. Por ejemplo, para crear una lista de los cuadrados de los números del 1 al 5:

Código:

```
cuadrados = [x ** 2 for x in range(1, 6)]
```

**4. Funciones de agregación:** *Python* proporciona funciones incorporadas como **sum**(), **max**(), **min**(), y **len**() que permiten realizar cálculos en listas y datos, como sumar todos los elementos de una lista o encontrar el valor máximo.

**5. Ordenación de datos:** puede ordenar listas de datos utilizando la función **sorted**(). Por ejemplo:

Código:

```
mi_lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
mi_lista_ordenada = sorted(mi_lista)
```

**6. Eliminación y modificación de elementos:** puede agregar, eliminar o modificar elementos en listas y diccionarios utilizando métodos como **append**(), **remove**(), **pop**(), **insert**(), y operaciones de asignación.

**7. Filtrado y mapeo con funciones:** puede utilizar funciones como **filter**() y **map**() para filtrar y transformar datos de manera eficiente en función de criterios específicos.

**8. Agrupación de datos:** la función **groupby**() de la biblioteca **itertools** permite agrupar elementos de una secuencia basándote en un criterio específico.

**9. Manejo de errores:** utilizar bloques **try...except** para manejar excepciones que puedan ocurrir durante el procesamiento de datos, como divisiones por cero o índices fuera de rango.

**10. Comprensiones de diccionarios:** al igual que con las comprensiones de listas, puede utilizar comprensiones de diccionarios para crear diccionarios nuevos o modificar existentes en función de una iteración y una expresión.

**11. Funciones de búsqueda:** *Python* ofrece funciones como **in**, **index**(), y **count**() para buscar elementos en listas y datos.

El procesamiento de listas y datos es una habilidad esencial en *Python* y se utiliza en una amplia variedad de aplicaciones, desde análisis de datos hasta desarrollo web y automatización de tareas. A medida que adquiera experiencia en programación, explorará aún más técnicas y bibliotecas especializadas para trabajar con datos de manera más eficiente y efectiva.

### 2.9.3 Ejemplos en el ámbito científico y de análisis de datos

El uso de *Python* en el ámbito científico y de análisis de datos es muy común debido a las numerosas bibliotecas y herramientas disponibles.

**1. Análisis de datos con Pandas**

La biblioteca Pandas es ampliamente utilizada para el análisis y manipulación de datos. Puede cargar conjuntos de datos, filtrar, agrupar y realizar operaciones estadísticas de manera eficiente. Por ejemplo, para calcular estadísticas básicas en un conjunto de datos CSV:

Código:

```
import pandas as pd

datos = pd.read_csv('datos.csv')
estadisticas = datos.describe()
```

**2. Visualización de datos con Matplotlib y Seaborn**

**Matplotlib** y **Seaborn** son bibliotecas populares para crear gráficos y visualizaciones de datos. Puede generar gráficos de barras, gráficos de dispersión, histogramas y más. Por ejemplo, para trazar un histograma de datos:

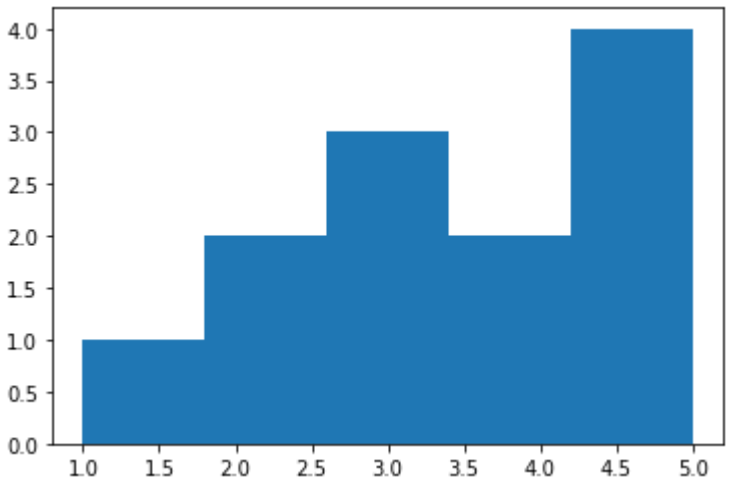
Código:

```
import matplotlib.pyplot as plt

datos = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]
plt.hist(datos, bins=5)
plt.show()
```

Salida:

**Figura 2.1**

*Diagramade barras*.




*Nota.* Los autores

## 3. Aprendizaje automático con Scikit-Learn

**Scikit-Learn** es una biblioteca popular para el aprendizaje automático en *Python*. Puede crear modelos de regresión, clasificación y agrupación, así como evaluar su rendimiento. Por ejemplo, para entrenar un modelo de regresión lineal:

Código:

```
from sklearn.linear_model import LinearRegression

modelo = LinearRegression()
modelo.fit(X, y)
```

### 4. Procesamiento de imágenes con OpenCV

**OpenCV** es una biblioteca especializada en procesamiento de imágenes. Se utiliza en aplicaciones de visión por computadora para detectar objetos, realizar seguimiento de objetos, procesar imágenes médicas, entre otros. Por ejemplo, para cargar y mostrar una imagen:

Código:

```
import cv2

imagen = cv2.imread('imagen.jpg')
cv2.imshow('Imagen', imagen)
cv2.waitKey(0)
```

### 5. Simulaciones numéricas con NumPy

**NumPy** es una biblioteca fundamental para computación científica y numérica. Se utiliza para realizar cálculos numéricos eficientes, como álgebra lineal, cálculos estadísticos y simulaciones. Por ejemplo, para crear una matriz y calcular su determinante:

Código:

```
import numpy as np

matriz = np.array([[1, 2], [3, 4]])
determinante = np.linalg.det(matriz)
```

6. Análisis de Redes Sociales con **NetworkX**

**NetworkX** es una biblioteca para el análisis de redes y grafos. Puede modelar y analizar redes sociales, redes de transporte, redes de comunicación y más. Por ejemplo, para calcular medidas de centralidad en una red:

Código:

```
import networkx as nx

G = nx.Graph()
G.add_edges_from([(1, 2), (2, 3), (3, 1)])
centralidad = nx.betweenness_centrality(G)
```

Estos son solo algunos ejemplos de cómo *Python* se utiliza en el ámbito científico y de análisis de datos. *Python* ofrece una amplia gama de bibliotecas y herramientas que permiten a los

científicos de datos y analistas realizar análisis sofisticados y resolver problemas en diversas disciplinas, desde la biología y la física hasta la economía y la ciencia social.

## EJERCICIOS PROPUESTOS

1. **Registro de mantenimiento**. Crear un programa que permita a un usuario registrar detalles de tareas de mantenimiento. Debe permitir al usuario ingresar información como el tipo de mantenimiento (preventivo o correctivo), la descripción de la tarea, la fecha y el tiempo dedicado a la tarea. Luego, almacena esta información en un archivo de registro o una base de datos.

2. **Programación de mantenimiento preventivo**. Desarrollar un programa que genere automáticamente un plan de mantenimiento preventivo. El usuario debe ingresar información sobre los equipos, sus ciclos de mantenimiento y las fechas de inicio. El programa debe calcular las fechas futuras en las que se deben realizar las tareas de mantenimiento preventivo.

3. **Gestión de inventario de piezas de repuesto**. Crear una aplicación que permita gestionar un inventario de piezas de repuesto. Debe permitir al usuario agregar, eliminar, buscar y actualizar información sobre las piezas de repuesto, incluyendo su nombre, número de parte, cantidad en stock y proveedor.

4. **Cálculo de índices de mantenimiento.** Escriba un programa que calcule índices de mantenimiento, como el MTBF (Mean Time Between Failures) y el MTTR (Mean Time To Repair). El usuario debe ingresar datos sobre el tiempo entre fallas y el tiempo de reparación para calcular estos índices.

5. **Gestión de órdenes de trabajo**. Desarrollar una aplicación que permita gestionar órdenes de trabajo. El usuario debe poder crear, asignar, actualizar y cerrar órdenes de trabajo. Cada orden de trabajo debe incluir detalles como el equipo afectado, la descripción del problema y el estado de la orden.

6. **Análisis de fallas**. Escriba un programa que analice los datos de fallas de equipos a lo largo del tiempo y muestre estadísticas como la frecuencia de fallas, los tipos de fallas más comunes y la duración promedio de las reparaciones.

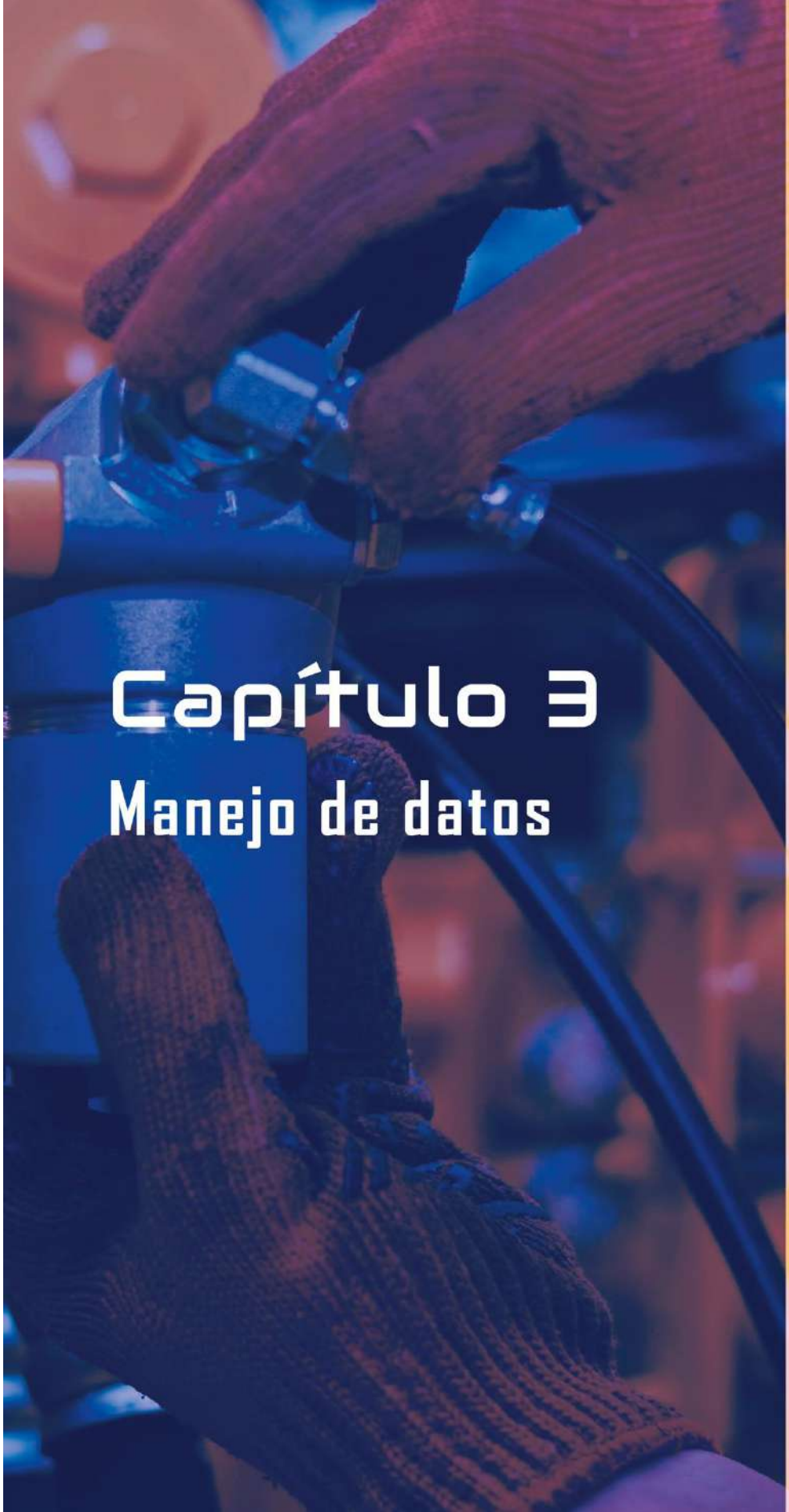
7. **Programación de paradas de mantenimiento**. Crear una herramienta que permita programar paradas de mantenimiento en una planta industrial. El usuario debe ingresar detalles como la fecha y la duración planificada de la parada, así como los equipos afectados. El programa debe ayudar a evitar conflictos de programación.

8. **Evaluación de riesgos de mantenimiento**. Desarrollar un programa que evalúe los riesgos asociados a las tareas de mantenimiento. El usuario debe ingresar información sobre la tarea y sus posibles riesgos. El programa debe calcular un índice de riesgo y proporcionar recomendaciones.

9. **Gestión de contratos de mantenimiento**. Crear una aplicación que permita gestionar contratos de mantenimiento con proveedores externos. Debe incluir

detalles como los servicios cubiertos, las fechas de vencimiento y los costos asociados.

10. **Informes de mantenimiento**. Desarrollar un generador de informes de mantenimiento que recopile información de múltiples fuentes, como registros de mantenimiento y órdenes de trabajo, y genere informes detallados sobre el estado y el rendimiento de los equipos.



# Capítulo 3

## Manejo de datos

Capítulo 3

# Manejo de datos

## 3 Análisis descriptivo de datos con Python

El análisis descriptivo de los datos con *Python* desempeña un papel esencial en el mantenimiento industrial, tanto para el análisis de indicadores de gestión, como para el análisis de las variables de las máquinas e instalaciones en el monitoreo de la condición, diagnóstico y pronóstico.

Estos análisis proporcionan información valiosa para la toma de decisiones relacionadas con las actividades de mantenimiento, de tal manera que permiten mejorar la eficacia de la planificación, optimizando recursos, reduciendo costos y garantizando una disponibilidad y fiabilidad aceptable.

Para la mejor comprensión sobre las funciones de *Python* aplicadas en el análisis descriptivo de los datos en el mantenimiento, se desarrollará un caso práctico sobre

mantenimiento basado en la condición, bajo las siguientes condiciones:

El espesor inicial de la carcasa tipo involuta de una trituradora de madera esquematizada en la Figura 3.1, es de 8 mm. Durante el trabajo normal presenta desgaste abrasivo lo que con el tiempo provocará que el espesor de este elemento disminuya hasta alcanzar la medida crítica de 2 mm. Cuando esto ocurra, la trituradora perderá su función (falla funcional).

Para que esta falla funcional no se presente de manera imprevista (mantenimiento correctivo) y ocasione como consecuencia operacional, pérdidas de lucro cesante, es medido en estos 2 últimos años cada 6 meses, el espesor en mm en la zona de impacto mediante la técnica de ultrasonido activo, mismos que se indican en la Tabla 3.1.

Con las medias de estas medidas de espesor, se debe realizar una regresión lineal con el propósito de hacer un pronóstico o estimación del tiempo hasta la falla (TTF, del inglés *Time To Failure*) (UNE ES 13306, 2018, p. 24) para poder planificar la reparación con anticipación (mantenimiento preventivo).

**Figura 3.1**

*Trituradora de madera.*


*Nota.* Adaptado de AGRIEURO (2023).

**Tabla 3.1**

*Datos del espesor de una carcasa***.**

| Espesor_1 | Espesor_2 | Espesor_3 | Espesor_4 |
|---|---|---|---|
| 8,39 | 7,69 | 6,64 | 6,41 |
| 6,86 | 7,24 | 7,54 | 6,32 |
| 7,98 | 8,32 | 7,10 | 5,92 |
| 9,02 | 7,27 | 6,73 | 6,33 |
| 8,69 | 7,86 | 6,77 | 6,66 |
| 8,24 | 6,88 | 6,85 | 5,92 |
| 7,90 | 7,23 | 7,43 | 6,15 |
| 7,90 | 6,98 | 7,32 | 6,63 |

| Espesor_1 | Espesor_2 | Espesor_3 | Espesor_4 |
|---|---|---|---|
| 8,56 | 7,79 | 7,27 | 6,52 |
| 7,58 | 6,29 | 7,79 | 6,38 |
| 8,08 | 7,53 | 6,07 | 6,87 |
| 6,65 | 7,81 | 6,66 | 6,75 |
| 7,63 | 8,00 | 6,93 | 6,61 |
| 8,00 | 7,94 | 6,57 | 6,62 |
| 8,39 | 6,46 | 7,32 | 6,32 |
| 7,02 | 6,01 | 7,04 | 6,25 |
| 7,46 | 7,70 | 6,61 | 6,77 |
| 7,74 | 7,55 | 6,55 | 6,27 |
| 7,55 | 7,39 | 7,26 | 6,86 |
| 7,41 | 7,49 | 6,60 | 6,40 |
| 8,38 | 7,38 | 6,78 | 6,57 |
| 8,43 | 7,27 | 7,25 | 6,45 |
| 8,33 | 7,53 | 5,86 | 6,45 |
| 8,44 | 7,33 | 6,81 | 7,08 |
| 8,08 | 7,00 | 7,27 | 6,45 |
| 8,09 | 7,75 | 6,19 | 6,52 |
| 8,64 | 7,76 | 6,96 | 6,98 |
| 8,05 | 8,34 | 7,10 | 6,17 |
| 8,33 | 8,23 | 6,67 | 6,84 |
| 8,52 | 7,18 | 7,56 | 6,89 |

*Nota.* Los autores.

Las funciones de *Python* que se van a utilizar requieren de la importación la las librarías **numpy**, **pandas**, **scipy**, **matplotlib** y **seaborn**, de la siguiente manera:

Código:

```
# Importación de librerías
import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import seaborn as sns
```

## 3.1 Importación y exploración de datos

Antes que nada, se debe copiar sin formato la Tabla 3.1 en la Hoja1 de un archivo de *Excel* y guardarlo con el nombre «Espesores de carcasa.xlsx» en el mismo directorio donde se guardará el archivo de *Python* o *Jupyter Notebook*.

### 3.1.1 Importación de datos

Existen varias maneras y funciones para importar información a *Python*. La que se plantea a continuación es una de las opciones en el caso de que los datos estén almacenados en un archivo de *Excel* con decimales separados por una coma ",". Una vez abierto el archivo de *Excel*, se debe consultar las hojas que contiene.

Código:

```
excel = pd.ExcelFile('Espesores de carcasa.xlsx')
print(excel.sheet_names)
```

Salida:

```
['Hoja1']
```

A continuación, toda la información de la "Hoja1" se la asigna a la variable "espesores" en formato *DataFrame* y se cierra el archivo de *Excel*. Ahora la tabla con las 4 muestras de 30 mediciones de espesores se llama **espesores**.

Código:

```
espesores = excel.parse('Hoja1')
# Cierre del archivo de Excel
excel.close()
```

### 3.1.2 Exploración de datos

El número de los primeros registros que se desee observar, se coloca dentro del paréntesis de la función **head**(); se procede de manera análoga con función **tail()** para indicar los últimos

registros. Si dentro del paréntesis de las funciones **head()** o **tail()** no se coloca ningún número, se observarán por defecto 5 registros.

Código:

```
print(espesores.head())
print(espesores.tail())
```

Salida:

```
   Espesor_1  Espesor_2  Espesor_3  Espesor_4 0
         8.39       7.69       6.64       6.41
1        6.76        NaN       7.54       6.32
2        7.98       8.32       7.10       5.92
3        9.02       7.27       6.73       6.33
4        8.69       7.86       6.77       6.66
    Espesor_1  Espesor_2  Espesor_3  Espesor_4
25       8.09       7.75       6.19       6.52
26       8.64       7.76       6.96       6.98
27       8.05       8.34       7.10       6.17
28       8.33        NaN       6.67       6.46
29       8.52       7.18       7.56       6.89
```

## 3.2 Identificación y tratamiento de datos faltantes y atípicos.

### 3.2.1 Identificación de datos faltantes

Para observar la existencia de datos faltantes con facilidad, se puede utilizar la función **info(),** que indica a la salida el tipo de variable (**pandas.core.frame.DataFrame**), el número de columnas (total 4 *columns*), el número de datos no nulos de cada columna (0 Espesor_1 30 *non-null*) y el tipo de dato de cada columna (*float64*). La segunda columna, correspondiente a «Espesor_2» tiene 28 datos no nulos (1 Espesor_2 28 *non-null*), por lo que 2 datos son nulos.

Código:

```
informacion = espesores.info()
print(informacion)
```

Salida:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Espesor_1  30 non-null     float64
 1   Espesor_2  28 non-null     float64
 2   Espesor_3  30 non-null     float64
 3   Espesor_4  30 non-null     float64
dtypes: float64(4)
memory usage: 1.1 KB
None
```

### 3.2.2 Tratamiento de datos faltantes

Una práctica recurrente consiste en rellenar los valores por la mediana, mediante la función ***fillna(mediana, inplace=True)***. En lugar de la mediana se puede reemplazar por cualquier otro valor que se desee. Para cubrir las 4 columnas se emplea la función de recurrencia **for**.

Código:

```
for columna in espesores.columns:
    mediana_columna = espesores[columna].median()
    espesores[columna].fillna(mediana_columna, inplace=True)
print(espesores.info())
```

Salida:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Espesor_1  30 non-null     float64
 1   Espesor_2  30 non-null     float64
 2   Espesor_3  30 non-null     float64
 3   Espesor_4  30 non-null     float64
dtypes: float64(4)
memory usage: 1.1 KB
None
```

Ahora, la segunda columna tiene 30 valores no nulos (1 Espesor_2  30 non-null), tal cual las otras columnas.

### 3.2.3 Identificación de datos atípicos

El diagrama de cajas es utilizado principalmente para conocer la distribución de los datos (Mendenhall et al., 2015, p. 77; Triola, 2018, p. 119); sin embargo, también resulta práctico para identificar los valores atípicos que podrían conservarse o imputarse de acuerdo con los criterios y requerimientos del análisis.

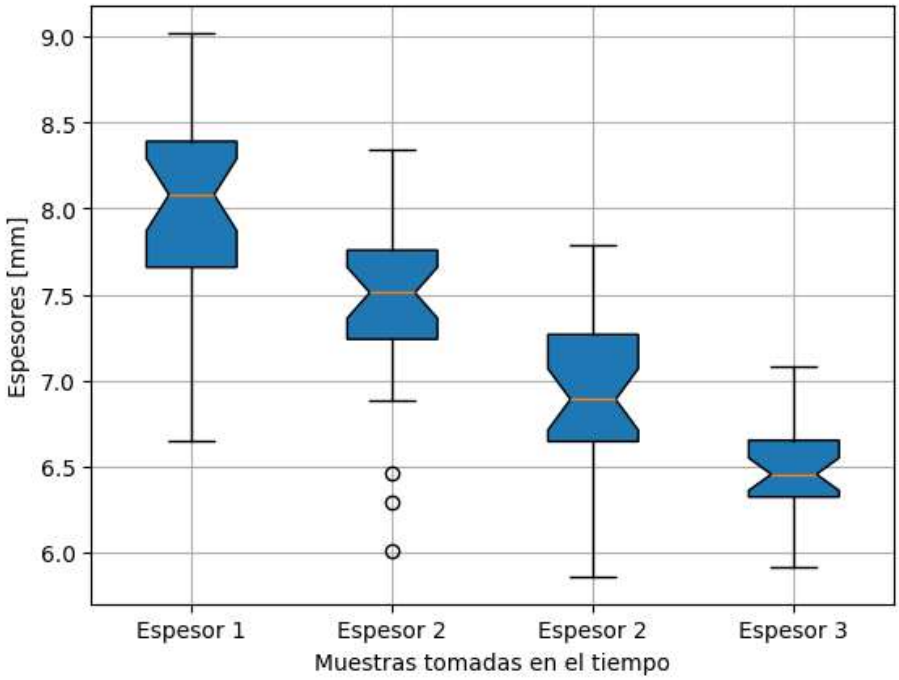
Código:

```
plt.boxplot(espesores, labels=['Espesor 1', 'Espesor 2', 'Espesor 2',
        'Espesor 3'], vert=True, patch_artist=True, notch ='False')
plt.xlabel('Muestras tomadas en el tiempo')
plt.ylabel('Espesores [mm]')
plt.grid(True)
plt.show()
```

Salida:

**Figura 3.2**

*Diagrama de cajas con datos atípicos.*




*Nota.* Los autores**.**

El diagrama de cajas mostrado en la Figura 3.2, se pueden observar 3 valores atípicos en la segunda muestra (Espesor 2), que están representados por los círculos pequeños ubicados debajo de la caja.

### 3.2.4 Tratamiento de datos atípicos

Se observa datos atípicos únicamente en la segunda columna; se plantea utilizar un algoritmo que identifique e impute los datos atípicos por la mediana en todas las columnas de una base de datos.

Código:

```
for columna in espesores.columns:
    mediana = espesores[columna].median()
    Q1 = np.percentile(espesores[columna], 25)
    Q3 = np.percentile(espesores[columna], 75)
    umbral_inf = Q1 - 1.5 * (Q3 - Q1)
    umbral_sup = Q3 + 1.5 * (Q3 - Q1)
    valores_atipicos = (espesores[columna] > umbral_sup) |
    (espesores[columna] < umbral_inf)
    espesores[columna][valores_atipicos] = mediana
```

Una vez imputados los datos atípicos, se verifica mediante los diagramas de cajas.

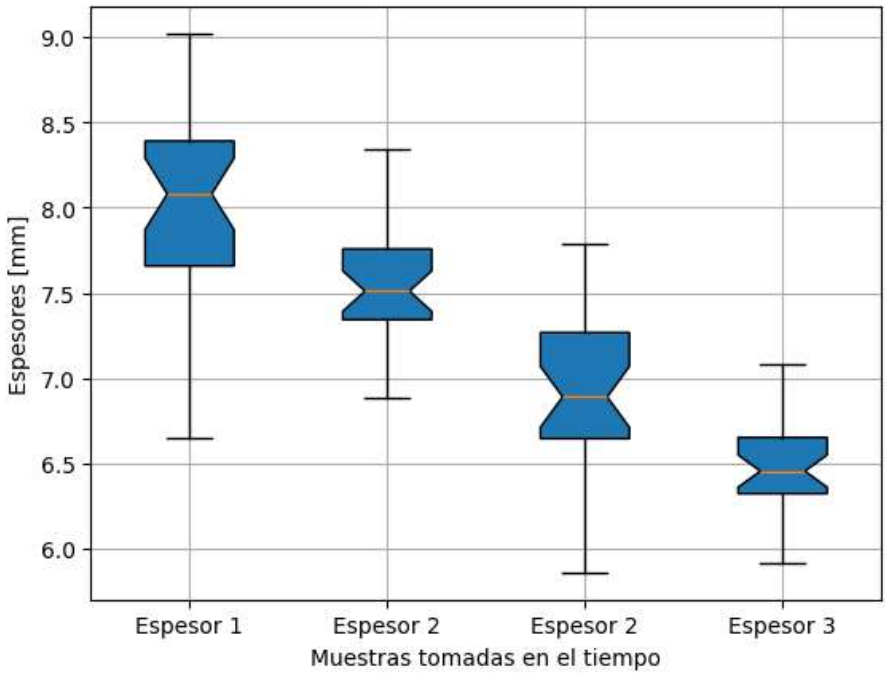
Código:

```
plt.boxplot(espesores, labels=['Año 0', 'Año 1', 'Año 2', 'Año 3'],
            vert=True, patch_artist=True, notch ='True')
plt.xlabel('Muestras en el tiempo')
plt.ylabel('Espesores [mm]')
plt.grid(True)

plt.show()
```

Salida:

**Figura 3.3**

*Diagrama de cajas sin datos atípicos.*




*Nota.* Los autores**.**

Los diagramas de caja de la Figura 3.3 indican que ya no existen valores atípicos; por lo tanto, el tratamiento aplicado fue eficaz.

## 3.3 Distribución de los datos

Existen métodos analíticos y gráficos para conocer la distribución analítica de los datos; en donde, de manera general, se prueba si se asemejan a la distribución normal por ser la más utilizada y en muchos casos se convierte en un requisito para aplicar ciertos procedimientos estadísticos. La función de la distribución normal forma una curva similar a una campana invertida por lo que se ha popularizado llamarla «campana de Gauss» (Pértegas & Pita, 2001).

### 3.3.1 Prueba de normalidad

El método analítico para probar si los datos se asemejan con la distribución normal es el de «Shapiro – Wilk» que se puede ejecutar utilizando la función ***shapiro****(x)* de la librería ***scipy.stats*.**

Código:

```
p_valor = espesores.apply(lambda x: stats.shapiro(x))
print('P valor:\n', p_valor.iloc[1])
```

Salida:

```
P valor:
Espesor_1    0.682914
Espesor_2    0.366067
Espesor_3    0.814627
Espesor_4    0.240202
Name: 1, dtype: float64
```

Como el **p** valor de las 4 muestras son mayores a 0,05 se puede concluir que los datos se distribuyen normalmente con el 95% de confianza.

### 3.3.2 Histograma de frecuencias relativas

Primeramente, se debe calcular como punto de inicio el número de barras que va a tener el histograma, conocido técnicamente como el número de clases, proveniente de la cantidad de filas que tiene la tabla de frecuencias, misma que en la actualidad ya no es práctico elaborar debido a las facilidades que presentan los *softwares* estadísticos para explorar y analizar grandes cantidades de datos directamente sobre los valores individuales de las muestras.

Uno de los métodos empleados para el cálculo del número de clases es el de la ley de *Sturges* (Triola, 2018, p. 51), que se aplica a continuación:

Código:

```
q = round(np.log2(len(espesores)))
print('Número de clases (q):',q)
```

Salida:

```
Número de clases (q): 5
```

El histograma de frecuencias relativas es un diagrama de barras anchas sin espacios sobre el cual se traza la campana de Gauss, y si esta se ajusta con las cimas de las barras, se puede concluir que los datos se distribuyen normalmente (Johnson, 2012, pp. 17-20).

Código:

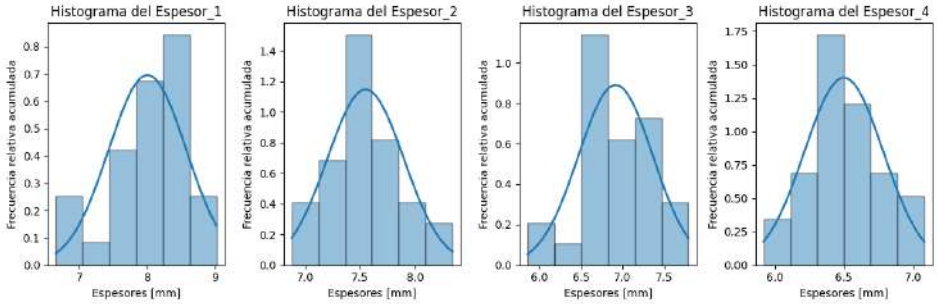
```
fig, axs = plt.subplots(1, 4, figsize=(12, 4))  # 1 fila, 4 columnas

# Elaboración de los histogramas de frecuencia relativas
for i, columna in enumerate(espesores.columns):
    caudal = espesores[columna]
    x = np.linspace(caudal.min(), caudal.max(), num=100)
    y = stats.norm.pdf(x, caudal.mean(), caudal.std())
    axs[i].plot(x, y, linewidth=2)
    axs[i].hist(caudal, density=True, bins=q, color='#3182bd', alpha=0.5,
            ec='black')
```

```
    axs[i].set_title(f'Histograma del {columna}')
    axs[i].set_xlabel('Espesores [mm]')
    axs[i].set_ylabel('Frecuencia relativa acumulada')

plt.tight_layout()  # Ajustar el espaciado entre subplots
 plt.show()
```

Salida:

**Figura 3.4**

*Histograma de frecuencias relativas.*




*Nota.* Los autores**.**

### 3.3.3 Diagrama Q-Q

Otro método gráfico para probar la normalidad de los datos es el diagrama **Q-Q**, donde se verifica que los puntos se ajusten a la recta del gráfico cuando los datos son normales.

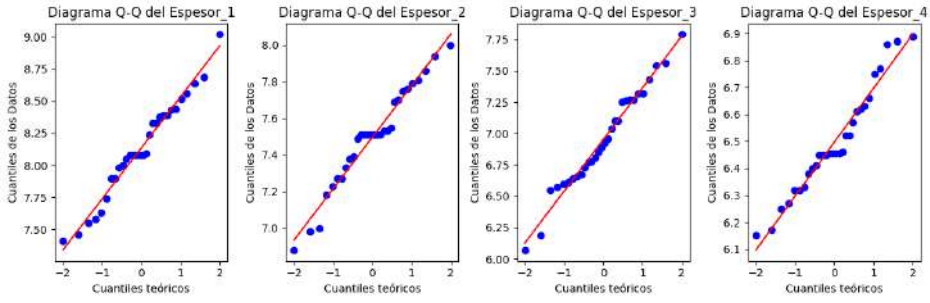
Código:

```
fig, axs = plt.subplots(1, 4, figsize=(12, 4))  # 1 fila, 4 columnas

# Elaboración de los diagramas Q-Q
for i, columna in enumerate(espesores.columns):
    caudal = espesores[columna]
    stats.probplot(caudal, dist='norm', plot=axs[i])
    axs[i].set_title(f'Diagrama Q-Q del {columna}')
    axs[i].set_xlabel('Cuantiles teóricos')
    axs[i].set_ylabel('Cuantiles de los Datos')

plt.tight_layout()  # Ajustar el espaciado entre subplots
plt.show()
```

Salida:

***Figura 3.1***

*Diagrama Q-Q.*




*Nota.* Los autores**.**

## 3.4 Medidas de tendencia central y dispersión

### 3.4.1 Medidas de tendencia central

Las principales medidas de tendencia central son la media, la moda y la mediana, que tienen ecuaciones específicas de cálculo, tanto para los valores individuales a la muestra, como para los valores agrupados en una tabla de frecuencias (Spiegel & Stephens, 2009, p. 66; Triola, 2018, p. 96), sin embargo, dado que se está utilizando es un *software*, se prefiere utilizar las funciones disponibles para este fin.

Código:

```
media = espesores.mean()
print('Media aritmética:\n', media)
```

Salida:

```
Media aritmética:
Espesor_1    8.135000
Espesor_2    7.496000
Espesor_3    6.951000
Espesor_4    6.496667
dtype: float64
```

Código:

```
moda = espesores.mode()
print('Moda:\n', moda)
```

Salida:

```
Moda:
    Espesor_1  Espesor_2  Espesor_3  Espesor_4
0        8.08       7.51       7.10      6.455
1         NaN        NaN       7.27        NaN
2         NaN        NaN       7.32        NaN
```

Como en la tercera columna (Espesor_3) existen 3 modas, la salida de la función **mode()** proporciona una tabla.

Código:

```
mediana = espesores.median()
print('Mediana:\n', mediana)
```

Salida:

```
Mediana:
Espesor_1    8.080
Espesor_2    7.510
Espesor_3    6.910
Espesor_4    6.455
dtype: float64
```

### 3.4.2 Medidas de dispersión

Para el cálculo de las medidas de dispersión, también existen ecuaciones específicas para los valores individuales de las muestras, de la población y para los valores agrupados (Spiegel & Stephens, 2009, p. 66; Triola, 2018, p. 96); sin embargo, *Python* permite utilizar funciones directas de las que se plantea las utilizadas con los valores individuales de las muestras por ser los comunes en aplicaciones del mantenimiento industrial.

Código:

```
varianza = espesores.var()
print('Varianza:\n', varianza)
```

Salida:

```
Varianza:
Espesor_1    0.150067
Espesor_2    0.076052
Espesor_3    0.161051
Espesor_4    0.038571
dtype: float64
```

Código:

```
desviacion = espesores.std()
print('Desviación estándar:\n', desviacion)
```

Salida:

```
Desviación estándar:
Espesor_1    0.387385
Espesor_2    0.275776
Espesor_3    0.401311
Espesor_4    0.196396
dtype: float64
```

### 3.4.3 Descripción de los datos

De preferencia, en lugar de calcular por separado las medidas de tendencia central y las de dispersión como se hizo en los 2 subtemas anteriores, en *Python* se recomienda utilizar la función **describe()**, que permite obtener ordenadamente el tamaño de la muestra (**count**), la media aritmética (**mean**), la desviación estándar (**std**), el mínimo (**min**), el primer cuartil Q1 (*25%*), el segundo cuartil Q2 o mediana (50%), el tercer cuartil (75%) y el máximo (*max*).

Código:

```
descripcion = espesores.describe()
print('Descripción de los datos:\n', descripcion)
```

Salida:

Descripción de los datos:

| | Espesor_1 | Espesor_2 | Espesor_3 | Espesor_4 |
|---|---|---|---|---|
| count | 30.000000 | 30.000000 | 30.000000 | 30.000000 |
| mean | 8.135000 | 7.496000 | 6.951000 | 6.496667 |
| std | 0.387385 | 0.275776 | 0.401311 | 0.196396 |
| min | 7.410000 | 6.880000 | 6.070000 | 6.150000 |
| 25% | 7.920000 | 7.342500 | 6.662500 | 6.385000 |
| 50% | 8.080000 | 7.510000 | 6.910000 | 6.455000 |
| 75% | 8.390000 | 7.697500 | 7.267500 | 6.617500 |
| max | 9.020000 | 8.000000 | 7.790000 | 6.890000 |

## 3.5 Regresión lineal

Para poder hacer la regresión lineal, se debe crear una nuevo *DataFrame* con las dos variables: la dependiente que corresponde a las medias de las medidas de los espesores en milímetros y, la independiente, conformada por el tiempo transcurrido entre cada medición del espesor de la involuta en meses.

Código:

```
t = [0, 6, 12, 18] #
e = [espesores[columna].mean() for columna in espesores.columns]
espesores_m = pd.DataFrame({'t':t, 'e':e })
print(espesores_m)
```

Salida:

| | t | e |
|---|---|---|
| 0 | 0 | 8.135000 |
| 1 | 6 | 7.496000 |
| 2 | 12 | 6.951000 |
| 3 | 18 | 6.496667 |

### 3.5.1 Diagrama de dispersión

Para la elaboración del diagrama de dispersión se ha elegido la función **lmplot()** de la librería **seaborn**, debido a que, además que permite obtener gráficos de mejor presencia, integra los límites de confianza a los lados de la recta, cuyo porcentaje puede regularse con el parámetro **ci**, que tiene por defecto el valor del 95%.

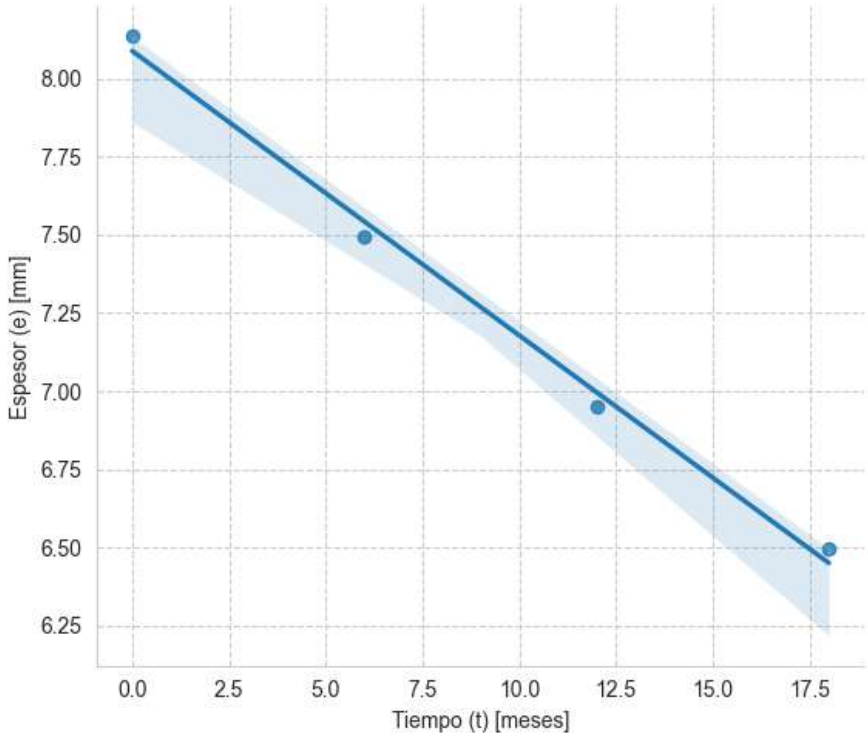
Código:

```
sns.set_style('whitegrid', {'grid.linestyle': '--'}) # Configurar el estilo de Seaborn
g = sns.lmplot(data=espesores_m, x='t', y='e', ci=95)
g.fig.set_size_inches(6, 5)
plt.xlabel('Tiempo (t) [meses]')
plt.ylabel('Espesor (e) [mm]')
```

Salida:

**Figura 3.5**

*Diagrama de dispersión.*




*Nota.* Los autores**.**

### 3.5.2 Obtención de parámetros

La proximidad de los puntos sobre la recta del diagrama de dispersión de la Figura 3.5, permite intuir que los datos tienen

una tendencia lineal inversa, por lo que se considera factible la regresión lineal de la siguiente ecuación:

$$e = b_0 + b_1 t\,, \tag{1}$$

donde, $\boldsymbol{e}$ es la variable dependiente correspondiente al espesor en milímetros, $\boldsymbol{t}$ es la variable independiente correspondiente al tiempo en meses, $\boldsymbol{b_0}$ es la intersección de la recta con el eje de las ordenadas y $\boldsymbol{b_1}$ es la pendiente de la recta.

Para la obtención de los parámetros $\boldsymbol{b_0}$ y $\boldsymbol{b_1}$, se emplea la función **polyfit***(x,y,1)*, y encontrar el coeficiente de correlación $\boldsymbol{r}$ de *Pearson*, se utiliza la función **corrcoef***(x,y)*; los 2 pertenecientes a la librería ***numpy***.

Código:

```
b1, b0 = np.polyfit(espesores_m['t'], espesores_m['e'], 1)
r_val = np.corrcoef(espesores_m['t'], espesores_m['e'])[0, 1]
R2 = r_val**2
print('r: %.4f' % r_val)
print('R2: %.4f' % R2)
print('Intersección: %.4f' % b0)
print('Pendiente:   %.4f' % b1)
```

Salida:

r: -0.9972
R2: 0.9943
Intersección: 8.0887
Pendiente: -0.091

Como el coeficiente de correlación **r** de *Pearson* es muy próximo a **-1**, se concluye que las variables de espesor y tiempo tienen una buena correlación lineal inversa.

### 3.5.3 Pronóstico de la falla

Para hacer una aproximación del tiempo donde la carcasa de la trituradora de madera perderá su función ($TTF$), se despeja el tiempo $\boldsymbol{t}$ de la Ecuación (1), obteniendo la siguiente expresión:

$$t = \frac{(e - b_0)}{b_1}. \qquad (2)$$

Donde, reemplazando los valores de los parámetros $\boldsymbol{b_0}$ y $\boldsymbol{b_1}$, y del espesor crítico $\boldsymbol{e} = 4\,mm$, se llega a la Ecuación (3); en la que, adicionalmente, se realiza la conversión de meses a años y se resta 1,5 debido al tiempo que ya transcurrió para obtener las 4 muestras de espesores:

$$TTF = \frac{(4\ mm - 8{,}089\ mm)}{-0{,}091\ mm/mes}\ \frac{1\ año}{12\ meses} - 1{,}5\ años. \qquad (3)$$

Código:

```
TTF = (4-b0)/b1/12 – 1.5
año = int(TTF)
mes = int(TTF%1*12)
dia = int((TTF%1*12)%1*30)
print('La carcasa fallará aproximadamente en %.0f años, %.0f meses y
%.0f días' % (año, mes, dia))
```

Salida:

La carcasa fallará aproximadamente en 2 años, 2 meses y 27 días.

Este resultado no hay que tomarlo con exactitud debido a que es solo una estimación y en todo caso se debe continuar con las mediciones de espesor cada 6 meses.

## EJERCICIOS PROPUESTOS

**1.** Guarde en la "Hoja2" de un archivo de Excel la siguiente tabla e importe a *Python* indicando los 3 primeros y 3 últimos registros (coloque el código empleado y la salida).

| Temperatura_1 | Temperatura_2 | Temperatura_3 |
|---|---|---|
| 24,1 | 26,1 | |
| 23,9 | 25,9 | 28,1 |
| 25,1 | 28,7 | 25,7 |
| 24,0 | 25,9 | 28,2 |
| 23,1 | 26,4 | 25,1 |
| 23,7 | 26,8 | 28,0 |
| 24,5 | 27,9 | 28,8 |
| 23,7 | 27,3 | 29,0 |
| 23,1 | 26,4 | 27,4 |
| 24,1 | 25,8 | 28,5 |

**2.** Identifique y haga el tratamiento de los datos faltantes únicamente a la columna 3 (Temperatura_3) de los datos del Ejercicio 1 (coloque el código empleado y la salida).

**3.** Identifique y haga la imputación de los datos atípicos con la mediana de las 3 columnas de los datos del Ejercicio 1 (coloque el código empleado y la salida).

**4.** Haga en un solo gráfico, los diagramas de cajas de antes y después de la identificación e imputación de los datos atípicos del Ejercicio anterior (coloque el código empleado y la salida).

**5.** Determine el **p** valor de la prueba de Shapiro-Wilk de las columnas de los datos obtenidos en el Ejercicio 3; mediante una función de recurrencia (coloque el código empleado y la salida).

**6.** Elabore en un solo gráfico, los histogramas de las 3 columnas de los datos del Ejercicio 3 y haga un comentario sobre lo observado (coloque el código empleado y la salida).

**7.** Elabore en un solo gráfico, los diagramas Q-Q de las 3 columnas de los datos del Ejercicio 3 y haga un comentario sobre lo observado (coloque el código Haga (coloque el código empleado y la salida).

**8.** Haga una función para calcular consecutivamente la media aritmética, la moda y la mediana de los datos del Ejercicio 3 (coloque el código empleado y la salida).

**9.** Haga una función para calcular consecutivamente la varianza y la desviación estándar de los datos del Ejercicio 3 (coloque el código empleado y la salida).

**10.** Haga una función para calcular consecutivamente el primero, segundo y tercer cuartil de los datos del Ejercicio 3 (coloque el código empleado y la salida).

# Conclusiones

- El manejo de estructuras, sirven para el almacenamiento de información en forma organizada y eficiente.

- La implementación de Listas permite una manipulación dinámica de la colección de información que en estas se almacena, aunque con un desempeño menor al de una tupla.

- Las tuplas al ser una estructura inmutable, resulta útil al requerir que la información almacenada no sea alterada lo que permite integridad de los datos.

- Se ha explorado a fondo el mundo de las funciones en *Python* y aprender que las funciones son una parte esencial de la programación en *Python*. Permiten encapsular fragmentos de código para que puedan ser reutilizados fácilmente en múltiples lugares dentro de un programa.

- Las ventajas de usar funciones son numerosas. Desde una mejor organización del código hasta la reducción de duplicación de código y la mejora de la legibilidad; las funciones son una herramienta poderosa para los programadores.

- Se comprendió la sintaxis básica de la creación de funciones en *Python*, incluyendo cómo definir parámetros y cómo usar la sentencia **return** para devolver valores desde una función.

- Explorar los diferentes tipos de argumentos que pueden ser utilizados en funciones, desde argumentos posicionales hasta argumentos con palabras clave y argumentos por defecto.

- Se ha profundizado en el concepto de ámbito de las variables, comprendiendo la diferencia entre variables locales y globales, así como el uso de la palabra clave **global** para modificar variables globales dentro de funciones.

- En el campo de la ingeniería no es práctico crear directamente las estructuras de datos, debido a la gran cantidad de información que se requiere; por este motivo *Python* cobra

relevancia al tener funciones que con facilidad permiten importar grandes cantidades de datos, desde archivos como **txt**, **xls**, **csv**, entre otros.

- Cuando se trabaja con muchos datos es importante emplear métodos automáticos para identificar y dar tratamiento a los valores faltantes y atípicos. Para este fin, *Python* cuenta con funciones que simplifican y reducen notablemente los recursos, el esfuerzo y el tiempo.

- *Python* brinda prestaciones efectivas y rápidas para visualizar la distribución de los datos pequeños o masivos, a través de métodos analíticos y gráficos.

- El cálculo de las medidas de tendencia central, de dispersión y de posición, pueden ser obtenidos con funciones directas, por lo que se requiere únicamente de una línea de código para efectuar cada cálculo.

- Para conocer las principales medidas de tendencia central, de dispersión y de posición, *Python* cuenta con la función **describe()**, que proporciona estos indicadores con una sola

línea de código, a más que se puede aplicar a varias muestras al mismo tiempo, mediante la utilización de un *DataFrame*.

- La función **polyfit()** dela librería **numpy**, permite calcular fácilmente los parámetros de un polinomio de cualquier grado, incluyendo el de grado 1 que corresponde a la línea recta.

- En el caso práctico aplicado al pronóstico de vida de un elemento o aproximación del tiempo hasta la falla, se destaca la flexibilidad de *Python* para el análisis y tratamiento de datos.

# Glosario

- ***Algoritmo:*** secuencia de pasos que da solución a un problema.

- ***Break:*** clausura una instrucción.

- ***Código:*** conjunto de instrucciones propias de un programa para indicar a la máquina secuencias de ejecución.

- ***Constructor:*** inicializa valores del objeto.

- ***Data frame:*** para manejo de datos, están compuestos por filas y columnas.

- ***Lenguaje de programación:*** programa con reglas y gramáticas propias del programa, para generar nuevos programas.
- $\mu$***:*** media poblacional.

- ***Me:*** mediana.

- ***Mo:*** moda.

- ***MTBF:*** tiempo medio entre fallos, del inglés *Mean Time Between Failure*.

- ***n:*** tamaño de la muestra.

- ***N:*** tamaño de la población.

- ***Python:*** lenguaje de programación.

- ***q:*** número de clases.

- ***Q:*** cuartil.

- ***RIC:*** rango intercuartílico.

- ***s:*** desviación estándar muestral.

- $\sigma$***:*** desviación estándar poblacional.

- ***Sets:*** estructura de datos para almacenar datos.

- ***Software:*** Programas para ejecutar tareas en un dispositivo.

- ***String:*** cadena de caracteres.

- ***TBF:*** tiempo entre fallas, del inglés *Time Between Failure*.

- ***TTR:*** tiempo hasta la recuperación, del inglés *Time To Restoration*.
- $x$**:** marca de clase
- $\bar{x}$**:** media muestral.

# Bibliografía


AGRIEURO, (2023). Recuperado el 16 de septiembre de 2023 de: https://www.agrieuro.es/share/images/categorie/biotrituratror/biotrituratore-ruoll.png

Chaves, S. (2022). *El lenguaje Python y sus principales librerías*. https://formadoresit.es/el-lenguaje-python-y-sus-principaleslibrerias/#:~:text=Las%20librer%C3%ADas%20de%20Python%20son,y%20potentes%20que%20pueden%20utilizarse.

Coser, R. (2023). *Python Charts*. Recuperado el 9 de septiembre de 2023 de: https://python-charts.com/es/.

En GMAO, Reglamentos de Instalaciones (2020). *¿Qué diferentes tipos de mantenimiento existen en una empresa?* https://www.eurofins-environment.es/es/diferentes-tipo-de-mantenimiento-existen-empresa/

Johnson, R. (2012). *Probabilidad y estadística para Ingenieros* (8va ed). Pearson Educación de México.

Kelmansky, D. (2009). *Estadística para todos. Estrategias de pensamiento y herramientas para la solución de problemas* (1ra ed). Artes gráficas Rioplatense.

Lifeder (2020). *Población estadística: concepto, tipos, ejemplos*. Recuperado el 26 de agosto de 2023 de: https://www.lifeder.com/poblacion-estadistica/

Lind, D., Marchal, W., & Wathen, S. (2012). *Estadística aplicada a los negocios y la economía* (15ta ed). McGraw-Hill Companies, Inc.

Mendenhall, W., Beaver, R., & Beaver, B. (2015). *Introducción a la probabilidad y estadística* (14a ed). Cengage Learning Editores.

Nube de datos (2015). *Introducción al diagrama de caja (box plot) en R*. Recuperado el 27 de agosto de 2023 de: https://nubededatos.blogspot.com/2015/02/introduccion-al-diagrama-de-caja-box.html.

Pértegas, S. y Pita, S. (2001). *La distribución normal*. Elsevier Recuperado el 7 de septiembre de 2023 de: https://www.fisterra.com/formacion/metodologia-investigacion/la-distribucion-normal/.

Python.org. (2023). *El tutorial de Python*. https://docs.python.org/3/tutorial/index.html.

Rebollar, J. (2021). *Herramientas básicas de trabajo de mantenimiento*. https://revistamedica.com/herramientas-basicas-trabajo-mantenimiento/

Spiegel, M., & Stephens, L. (2009). *Estadística - Schaum* (4ta ed.). McGraw-Hill Companies, Inc.

Trejos Buriticá, O. y Muñoz Guerrero, L. (Il.) (2021). *Introducción a la programación con Python*. 1. RA-MA Editorial. https://elibro.net/es/ereader/espoch/230298?page=14.

Triola, M. (2018). *Estadística*. (12da ed). Pearson Educación.

UNE EN 13306 (2018). *Mantenimiento. Terminología del mantenimiento*. AENOR INTERNACIONAL S.A.U.


CIDE
EDITORIAL

Python aplicado al Mantenimiento Industrial es el título y el objetivo principal de este libro es ayudar al lector a través de la práctica la generación de código para la manipulación de datos; al aprender a programar se estimula al lector a tener un razonamiento crítico y poder solucionar un problema por medio del lenguaje de programación Python. El Capítulo 1 inicia con conceptos básicos y todo el manejo de estructuras, para pasar al Capítulo 2 con el manejo de funciones y así finalizar con el Capítulo 3 con una introducción en el manejo de datos estadísticos.

**Vanessa Lorena Valverde González.** – Magister en Informática Educativa, Ingeniera en Sistemas Informáticos, Analista en Sistemas Informáticos por la Escuela Superior Politécnica de Chimborazo; Master Universitario en Ingeniería de Software y Sistemas Informáticos por la Universidad de la Rioja. Ha ejercido la docencia por 12 años, forma parte del Grupo de Investigación Ciencia del Mantenimiento CIMANT.

**Félix Antonio García Mora.** – Ingeniero de mantenimiento con una destacada carrera en la ingeniería y la gestión del mantenimiento industrial. Obtuvo su título en la Escuela Superior Politécnica de Chimborazo y una Maestría en Gerencia e Ingeniería de Mantenimiento en la Universidad Nacional de Ingeniería en Lima, Perú. Ha ocupado cargos importantes en la dirección del mantenimiento en diversas industrias, además de dedicarse a la enseñanza durante 3 años. Es miembro del Grupo de Investigación Ciencia del Mantenimiento (CIMANT), lo que refleja su interés en la investigación y el avance de las mejores prácticas en el campo de la ingeniería y la gestión de mantenimiento.

**Eduardo Segundo Hernández Dávila.**– Ingeniero de Mantenimiento, por la Escuela Superior Politécnica de Chimborazo; Maestría en Gestión del Mantenimiento Industrial. Docente de la Escuela Superior Politécnica de Chimborazo. Ha ejercido cargos en la dirección del mantenimiento en empresas manufactureras. Ha ejercido la docencia por 16 años, forma parte del Grupo de Investigación Ciencia del Mantenimiento CIMANT.

ISBN: 978-9942-636-33-1

9789942636331