Apuntes de Python

migueluisV

Actualizadas: Octubre 2022

Índice

1 Conceptos básicos **5**
1.1 Tipos de datos . . . 5
1.2 Operadores matemáticos . . . 5
1.3 Operadores condicionales . . . 6
1.4 Operadores lógicos . . . 6
1.5 Cadenas . . . 7
1.5.1 Operaciones con Strings . . . 8
1.5.2 Funciones de cadenas . . . 8

2 Variables **9**
2.1 Entrada de datos . . . 10
2.2 Conversión de valores o tipos de datos . . . 11
2.3 Operadores de asignación o relacionales . . . 11
2.4 Asignación de un valor a múltiples variables . . . 11

3 Comentarios **12**
3.1 Docstrings . . . 12

4 Terminar la ejecución de un programa **12**

5 Condicionales **13**

6 Ciclos **14**
6.1 Ciclo While . . . 14
6.1.1 break y continue . . . 14
6.2 Ciclo For . . . 15
6.3 Ciclo Do-While . . . 15

7 Sentencia Switch **16**

8 Colecciones de datos **17**
8.1 Listas . . . 17
8.1.1 Operaciones con listas . . . 18
8.1.2 Funciones de listas . . . 19
8.1.3 Listas con reglas . . . 20
8.1.4 Ordenamiento de listas . . . 20
8.2 Pilas . . . 21
8.3 Colas . . . 22
8.4 Rangos . . . 23
8.5 Diccionarios . . . 24
8.5.1 Funciones en diccionarios . . . 24
8.6 Tuplas . . . 25
8.6.1 Desempacando tuplas . . . 25
8.7 Sets . . . 26

8.7.1 Operaciones con conjuntos 26
8.8 Seleccionando una colección de datos 27

9 Funciones **28**
9.1 Funciones que regresan valor 28
9.2 Funciones Lambda 29
9.3 Funciones map y filter 29
9.4 Generadores 30
9.5 Decoradores 30
9.6 Parámetros *args y **kwargs 31
9.7 Listas como parámetros 33
9.8 Sobrecarga de métodos 33
9.9 Números aleatorios 34
9.9.1 Números aleatorios enteros entre dos valores 34
9.9.2 Números aleatorios enteros entre dos valores con salto 35
9.9.3 Números aleatorios decimales entre dos valores 35

10 Recursividad **35**

11 Programación funcional **36**
11.1 Funciones puras e impuras 37

12 Clases **38**
12.1 Constructores 38
12.2 Destructores 38
12.3 Herencia 39
12.4 Abstraccion 40
12.5 Polimorfismo 41
12.6 Métodos mágicos 42
12.7 Ocultación de datos 43
12.8 Métodos estáticos y de clase 44
12.9 Propiedades 45

13 Excepciones **46**
13.1 finally y else 47
13.2 Lanzando excepciones 47

14 Trabajando con archivos **48**
14.1 Leyendo archivos 49
14.2 Escribiendo en archivos 50
14.3 Comprobar la existencia de un archivo 50
14.4 Obtener una lista con todos los archivos de una carpeta 51
14.5 Eliminar un archivo del sistema 51

Índice de Tablas

1 Funciones especiales para listas 5
2 Orden de operadores comunes en Python 6
3 Operadores condicionales en Python 6
4 Operadores lógicos en Python 7
5 Funciones especiales para listas 9
6 Tipos de operadores de asignación o relacionales 11
7 Operaciones con listas 19
8 Funciones especiales para listas 20
9 Operaciones para conjuntos 27

Python es uno de los lenguajes de programación más populares, funciona para desarollar proyectos web, videojuegos, ciencia de datos, machine learning, y mucho más.

1 Conceptos básicos

- **print(elemento)**: es la forma más simple y sencilla de dar salida a información en pantalla. Esta instrucción siempre despliega texto o información en una **línea nueva**.
- Podemos eliminar el salto de línea de la sentencia **print()** si dentro de sus paréntesis agregamos el parámetro **end** y le asignamos una cadena vacía: *print(end="")*.
- **Tabulación**: Python no utiliza punto y coma (;) para la finalización de cada instrucción, en cambio, utiliza tabulación de instrucciones, si se comienza un ciclo, condicional, procedimiento, función o clase, todas las instrucciones dentro de cada una de estas instrucciones tendrán más tabulación, consiguiendo así mejor legibilidad de código y cumpliendo con la regla de que, cada instrucción escrita en este lenguaje, en vez de acabar con ;, acaba con tabulaciones.

1.1 Tipos de datos

La *Tabla 1* contiene la clasificación de los datos con los que trabaja Python:

Table 1: Funciones especiales para listas

Tipo de dato	**Significado**
String	Cadena de texto, puede contenerse entre comillas dobles ("") y sencillas ('), p. e: n = "nombre", n = 'mario'
Integer	Número entero, su espacio se limita a la memoria disponible, p. e: a = 4
Float	Número decimal, tiene un espacio límitado al tipo de dato, p. e: b = 3.14
Boolean	Un estado 1 o 0, True y False, p. e: listo = False
Complex	Número real y uno imaginario, p. e: complejo = 1 + 2j

1.2 Operadores matemáticos

La *Tabla 2* contiene los operadores válidos para realizar operaciones aritméticas, o tener un orden en la ejecución de las mismas:

Table 2: Orden de operadores comunes en Python

Operador	Significado
(), [], {}	Símbolos que contienen operaciones o instrucciones a ejecutar
*, **, /, //, %	Multiplicación, exponencial, división decimal, división entera y módulo
+, -	Suma y resta

Donde:

- Las operaciones con divisiones regresan resultados decimales.
- Las operaciones con decimales regresan resultados decimales (aunque haya enteros involucrados).
- La exponenciación es la operación donde dos asteriscos eleva el número x a la potencia y. La exponenciación puede concatenarse con otras exponenciaciones (2**3**2). Puede usarse floats en la exponenciación también.
- La división entera es la operación donde dos diagonales divide el número x entre y y regresa un número entero, en vez de uno float.

1.3 Operadores condicionales

La *Tabla 3* contiene los operadores válidos para comparar expresiones en estructuras condicionales, como se verá en un futuro tema:

Table 3: Operadores condicionales en Python

Operador	Significado
>	Mayor que
>=	Mayor igual que
<	Menor que
<=	Menor igual que
==	Igual a
!=	No igual a

1.4 Operadores lógicos

La *Tabla 4* contiene los operadores válidos si buscamos que más de una condición sea evaluada en una misma instrucción if:

Table 4: Operadores lógicos en Python

Operador	Significado
and	Y, todas las condiciones deben ser True para que la expresión completa sea True
or	O, una de todas las condiciones debe ser True para que toda la expresión sea True
not	Negación, niega una condición o pasa de True a False y viceversa

1.5 Cadenas

Existe el carácter especial \, llamado **secuencia de escape**, este te permite imprimir o llevar a cabo ciertas acciones dentro de una cadena, como lo pueden ser:

- inserción de tabulaciones (\t).
- inserción de nuevas líneas (\n).
- inserción de símbolo de retorno (\r). Este posiciona el cursos al inicio de una línea.
- inserción de caracteres de cadena (\', \").

Por lo general, los sistemas operativos Windows utilizan las secuencias de escape \n\r para dar saltos de línea y posicionar el cursos al inicio de la nueva línea, mientras que los sistemas operativos Linux solamente utilizan \n.

En caso de tener grandes textos, en vez de usar \n, que pueden perderse a lo largo del texto, podemos usar las tres comillas (""") o ''' dentro de *print()*, como vemos en el siguiente ejemplo:

```
# Por cada salto de linea en print, es la cantidad de saltos de línea que se imprimen en
    pantalla.
print("""esto
es
una
multilinea""")

print('''
Aprendiendo
Python
''')

# Despliega:
# esto
# es
# una
# multilinea
# Aprendiendo
# Python
```

1.5.1 Operaciones con Strings

La **concatenación** es el proceso de unir dos o más cadenas, para tener una sola resultante, en Python, además, podemos realizar más operaciones, la siguiente lista cuenta con las distintas operaciones básicas con cadenas en este lenguaje:

- **Concatenación**: por ejemplo: print("a" + "b"), print("c" + 'd'), print('e' + 'f') *//Despliega: ab, cd, ef.*
- **Multiplicación con concatenación**: cuando se utiliza el operador de multiplicación entre un número entero y una cadena, el resultado será la cadena repetida el número entero ingresado, por ejemplo: print(3 * "a"), print(2 * '2') *//Despliega: aaa, 22.*
- No se pueden multiplicar dos cadenas.
- Podemos buscar una subcadena o carácter dentro de una cadena con la sentencia **in** (misma sentencia para buscar un elemento dentro de una colecciónd e datos): *if subcadena in cadena.*

1.5.2 Funciones de cadenas

La *Tabla 5* contiene las funciones más útiles para cadenas:

Table 5: Funciones especiales para listas

Función	Significado	Ejemplo
format(elementos)	Aplica un formato o agrega otra cadena o elemento a una cadena que tiene marcada una posición en su contenido	print("{0} {1}".format("hola", "mundo"))
join(lista)	Pasa una lista a una cadena de texto, separado por un carácter (por ejemplo, un espacio)	x = ", ".join([1, 3, 5, 7])
split(delimitador)	Separa las palabras de una cadena convirtiéndola en una lista, se puede especificar el delimitador	str = "esto es una cadena" print(str.slip(" "))
replace(cadena, reemplazo)	Cambia una subcadena de una cadena por otra cadena	str = "hola mundo" print(str.replace ("hola", "lindo"))
upper()	Formatea una cadena a mayúscula	print("hola mundo".upper())
upper()	Formatea una cadena a minúscula	print("HOLA MUNDO".upper())
find()	Busca un carácter o subcadena dentro de una cadena, si lo encuentra, regresa el índice del primer carácter donde lo encontró, si no lo encuentra, regresa -1.	cadena.find("a")

2 Variables

Para declarar variables, no es necesario declarar un tipo de dato, seguido de un nombre, operador de asignación y valor, se omite la parte de darles un tipo de dato a las variables, el valor que tenga ya le asigna un tipo de dato a la variable implícitamente. Ejemplos:

```
nombre = 'Luis'
apellido = 'Miguel'
edad = 21
pesoPersona = 64.5
```

Las variables en Python son sensitivas, así que, Nombre y nombre son variables distintas, aunque en esencia, tengan el mismo nombre. El nombre que reciba una variable puede contener mayúsculas, minúsculas, números y guiones, pero no caracteres especiales o palabras reservadas del lenguaje, ni puede comenzar el nombre con algún número.

Al no requerir de un tipo de dato en la asignación de valores a variables, tu variable al

comienzo puede tener un valor numérico, y le puedes sobre escribir un valor cadena, como se ve a continuación:

```
x = 555
print(x)

x = 'cinco'
print(x)
```

Lo anterior no es considerado una buena practica, por lo que es recomendable no realizarlo.

La palabra reservada **del** puede ser utilizada para eliminar la memoria utilizada por una variable utilizada a lo largo de una función o procedimiento, como vemos enseguida:

```
x = 555
print(x)
del x
```

Esta palabra reservada puede ser utilizada acabando de usar una función, sin embargo, en el código sigue presente la declaración de la variable, por lo que, si se borra de la memoria en la línea 5 de la función, se puede volver a utilizar la variable en futuras líneas si es requerido, no olvide que, si usa una variable ya eliminada y sin volver a inicializar, ocurrirá un error en el código, como vemos en el siguiente ejemplo:

```
x = 5 # Declara variable entera que recibe el numero 5.

print(x) # Imprime 5.
del x # Es eliminada de la memoria la variable.

print(x + 5)
print(x) # Las dos anteriores lineas generan un error en la ejecucion del programa.

x = "hola mundo"

print(x) # Imprime hola mundo.
```

2.1 Entrada de datos

Se utiliza la función **input()** para recibir valores del usuario, esta función recibe una entrada de datos y la regresa en forma de **string** (sin importar si lo que se recibe es un número), dentro de los paréntesis de *input*, podemos ingresar una cadena que será lo que le estamos solicitando al usuario que ingrese, como vemos en el siguiente código:

```
nombre = input() # Variable que recibe un nombre por parte del usuario.

print(nombre) # Despliega el nombre ingresado por el usuario.

apellido = input("Digita tu apellido: ") # Variable que recibe un apellido por parte del
    usuario, el programa lanza mensaje donde solicita apellido.

print("El nombre del usuario es: " + nombre + apellido) #Despliega nombre completo del
    usuario.
```

2.2 Conversión de valores o tipos de datos

Siguiendo con el tema anterior, en caso de que ocupemos la edad de un usuario en su forma numérica, podemos convertir la cadena que nos regresa *input()* con la función **int()**, como vemos enseguida:

```
x = int(input("Ingresa tu edad: ")) # Variable recibe edad de usuario, el valor es
    convertido a entero.
print("Tu edad más 20 es: " + (x + 20)) # Despliega la edad del usuario más 20.
```

Las funciones **float()** y **str()** funcionan para convertir valores de un determinado tipo a decimal y cadena respectivamente, solamente tome en cuenta que, si tiene un decimal y lo quiere pasar a entero, la función *int()* le devolverá el valor decimal redondeado a su elemento entero únicamente (de 9.99 se le es regresado 9).

2.3 Operadores de asignación o relacionales

Los **operadores de asignación** son permitidos en Python, estos permiten pasar de $x = x + 1$ a $x += 1$, permitiendo un código más corto. Los operadores de asignación realizan una acción dependiendo del operador que se le quiera usar, los encontramos en la *Tabla 6*:

Table 6: Tipos de operadores de asignación o relacionales

Operador	Significado
+=	x = x + n, x es igual a x más n
-=	x = x - n, x es igual a x menos n
=	x = x * n, x es igual a x por n
/=	x = x / n, x es igual a x entre n
%=	x = x % n, x es igual a x entre n y regresa el residuo de la división
*=	x = x ** n, x es igual a x a la n
//=	x = x // n, x es igual a x entre n y regresa un valor entero

2.4 Asignación de un valor a múltiples variables

Si buscamos asignarle un valor (p. e: 1) a varias variables (p. e: tres variables), podemos seguir el siguiente formato:

```
a = b = c = 1
```

Si tenemos varias variables y les queremos asignar distintos valores, podemos optar por el siguiente formato como una alternativa a asignar valores a variables en distintas líneas de texto:

```
a, b, c = 1, 2, 3
```

3 Comentarios

No es más que una línea de código que no se ejecuta, son usados para explicar una función o instrucción, se utiliza el símbolo de gato (#) para hacer que una línea se convierta en un comentario:

```
# Esto es un comentario.
# print("hola mundo").
print("hola mundo").
```

Si se desea comentar múltiples líneas de código sin tener que poner demasiados símbolos #, puede poner tres dobles comillas (""") al inicio del bloque de código que desea comentar y otras tres comillas final del bloque:

```
"""
print("a")
print("a")
print("a")
print("Estos son varios comentarios")
print("a")
print("a")
"""
print("a")

# Imprime: a.
```

3.1 Docstrings

Este tipo de comentario especial es utilizado para dar una explicación más completa al código, se mantiene durante el tiempo de ejecución, para que otros programadores puedan examinar dichos comentarios, la estructura es la siguiente:

```
def nombre():
"""
Esta es una funcion que despliega que es un robot!
"""
print("hola, soy un robot")

nombre()
```

4 Terminar la ejecución de un programa

Para finalizar la ejecución de un programa, se utiliza la instrucción **exit()**, no requiere de ningún parámetro.

5 Condicionales

La estructura **if-else** permite ejecutar un bloque de código en caso de que su expresión se cumpla, en caso contrario, puede ejecutarse otro bloque de código. La estructura de declaración es:

```
if (condición):
    # Código.
else:
    # Código.
```

La estructura *if-else* puede ser simple, es decir, que solo incluya **if** y no **else**. Recordemos que Python no usa puntos y comas (;) ni llaves ({}) para encerrar ciertos bloques bajo ciertas funciones, procedimientos, condiciones, ciclos o clases, usa la **indentación**, y las expresiones a evaluar de *if* pueden contener variables tipo booleano, operaciones con operaciones aritméticas o comparación con operadores relacionales. No olvide los **dos puntos** (:) al final de la instrucción if y else, y que las sentencias ifs pueden ser anidadas (un if dentro de otro), como vemos a continuación:

```
x = 20 # Declara e inicializa variable.

if (x == 0): # Primer condición.
    print("Cero") # Despliega mensaje.
else: # Si no se cumplio la condición.
    if (x > 0): # Condición anidada.
        print("Mayor a cero")
    else:
            if (x >= 10): # Condición anidada.
                    print("Mayor a 10")
            else:
                    print("Número no válido")
```

Vemos que anidar de esa manera ifs se ve algo mal, por cierto, existe la estructura **elif**, la cual es una abreviación de **else if**, estructura conocido en otros lenguajes de programación, repetimos el ejemplo pero con elifs:

```
x = 20 # Declara e inicializa variable.

if (x == 0): # Primer condición.
    print("Cero") # Despliega mensaje.
elif (x > 0): # Segunda condición con elif.
    print("Mayor a cero")
elif (x >= 10): # Tercera condición con elif.
    print("Mayor a 10")
else: # Si no se cumplio ninguna condición.
    print("Número no válido")
```

No olvide que puede utilizar **operadores lógicos** para juntar más de una condición en una instrucción a evaluar:

```
x = 20 # Declara e inicializa variable.

if (x > 0 and x <= 10): # Condición con operadores relacionales y logicos.
    print("Número válido") # Imprime mensaje.
```

```
else: # Si no se cumple la condición.
    print("Número no válido") # Imprime mensaje.
```

La condición del ejemplo anterior puede ser simplificado de la siguiente manera:

```
x = 20 # Declara e inicializa variable.

if (0 < x < 10): # Condición simplificada.
    print("Número válido") # Imprime mensaje.
else: # Si no se cumple la condición.
    print("Número no válido") # Imprime mensaje.
```

Si leemos la condición, vemos que 0 debe ser menor a x, y x debe ser menor a 10, así lo interpreta el lenguaje y la evalúa, con este pensamiento, podemos simplificar otras condiciones numéricas.

6 Ciclos

Al igual que las estructuras condicionales, los **ciclos** deben tener **dos puntos** (:) al final de su declaración e **indentación** para su correcto funcionamiento.

6.1 Ciclo While

Ciclo que se repite mientras una condición se cumpla, debe ser cuidadoso, ya que debe tener algún tipo de variable que haga que el ciclo pare, sino, el ciclo continuará indefinidamente, causando problemas tanto en el programa, como en la máquina donde se está ejecutando. Estructura:

```
while (condición):
    # Código.


x = 0 # Declara e inicializa variable.

while (x < 9): # Ciclo que se repite 10 veces.
    print(x) # Imprime x.
    x -= 1 # Variable que hace que el ciclo termine.

print("Terminado") # Mensaje de fin de ciclo.
```

Al igual que las estructuras if, la condición del ciclo while puede contener distintas condiciones que conforman una expresión a evaluar, puede usar operadores relacionales y lógicos y contener estructuras if dentro del ciclo.

6.1.1 break y continue

Estas son palabras reservadas para manipular el flujo de ciclos while, en caso de que queramos terminar abruptamente un ciclo while, podemos usar la instrucción **break**, por ejemplo, si queremos que el usuario ingrese diez números, pero si uno de ellos es menor a cero, se termina el ciclo; la instrucción **continue** puede decirse que "salta" la iteración actual del ciclo y pasa

al siguiente, no para como tal al ciclo, si se ejecuta *continue*, toma en cuenta las siguientes instrucciones a continue y pasa a la siguiente iteración. Ejemplos de estas dos instrucciones:

```
x = 0 # Declara e inicializa variable.

while (x < 9): # Ciclo que se repite 10 veces.
    num = int(input("Ingrese un numero entre 0 y 9: ")) # Entrada de un número.

    if (num < 0): # Si el número es menor a cero, termina el ciclo con break.
           print("Numero no valido.")
           break

    if (num > 9) # Si el número es mayor a 9, no se toma en cuenta el numero y pasa a la
        siguiente iteración.
           print("Número muy elevado")
           x += 1
           continue

    print("El número que ingresaste es: " + num)
    x += 1
print("Terminado") # Mensaje de fin de ciclo.
```

6.2 Ciclo For

Ciclo que se repite una cantidad fija de veces, se utiliza para recorrer listas o cadena de texto, se pueden utilizar las instrucciones **break** y **continue** para salir o saltar en el ciclo, como con el ciclo while. Sintaxis:

```
# Declara lista y cadena de ejemplo.
lista = [1, 2, 3, 4]
nombre = "esto es una prueba con ciclo for"

for x in lista: # Declara una variable que represente los elementos de la lista.
    print(x) # Imprime los elementos de la lista.

for y in nombre # Declara una variable que represente los caracteres de la cadena.
    print(y) # Imprime caracter a caracter de la cadena de texto.
```

6.3 Ciclo Do-While

Como tal, no existe este tipo de ciclo en Python, sin embargo, podemos crearlo o simularlo. Para lograr esto, utilizamos un ciclo While, donde la condición que evalúe simplemente sea un valor booleano True, con esto, el ciclo entra automáticamente y cumple con la característica de que el ciclo al menos da una vuelta. Es necesario que haya una condición dentro del ciclo que le de fin al mismo, para evitar un ciclo indeterminado, veamos el siguiente ejemplo:

```
# Variable contador.
x = 1

# Condición cierta, entra al ciclo.
while True:
```

```
        # Imprime el valor de la variable.
        print(x)
        # Incrementa el valor de la variable.
        x = x + 1
        # Si el valor de la variable es 10, rompe el ciclo.
        if x == 10
                break
```

Al tener un valor boleando True, la condición es cierta y entra al ciclo, si la variable contador llega a cierto valor o condición, se rompe el ciclo con la sentencia **break**, con esto, aseguramos que se de una vuelta al menos dentro del ciclo y aseguramos que el ciclo tenga un fin.

7 Sentencia Switch

En Python no existe esta estructura condicional, la forma de simularlo es un conjunto de *if-elif's*, también se pueden utilizar diccionarios, donde su llave es el índice que se evalúa y su valor es una función que contiene el valor que queremos mostrar, pero para no complicarlos la vida, nos quedaremos con la serie de *if-elif's*.

8 Colecciones de datos

8.1 Listas

Las **listas** son los **vectores**, **matrices** o **arreglos** de otros lenguajes de programación, son tipos que almacenan una cantidad finita de elementos de un mismo tipo, pero, la peculiaridad en este lenguaje, es que en una lista puedes almacenar elementos de distintos tipos de datos. Estructura y ejemplos:

```
<nombre lista> = [<elemento 1>, <elemento 2>, ..., <elemento n> ] # Estructura de
    declaración de una lista.

nombres = ["mario", 'kevin', "jose", 'enrique] # Ejemplo de declaración de una lista
    string.
nums = [0, 9, 1, 8, 7, 6] # Ejemplo de declaración de una lista int.
variado = [99, "cien", 9, "kkk"] # Ejemplo de declaración de una lista con varios tipos de
     datos.
```

Como podemos ver, los ejemplos son declarados de un tipo e inicializados con n cantidad de elementos, para acceder a ellos, trabajarlos o desplegarlos, utilizamos el **operador**[], cada elemento posee un **índice**, que va de 0 a n, entonces, desplegaremos como ejemplo un elemento de una de las listas anteriores en el siguiente ejemplo:

```
print(nombres[1]) # Despliega "kevin".
print(numbs[0]) # Despliega 0.
print(variado[3]) # Despliega "kkk".
```

Podemos hacer que los elementos de una lista sean listas también, agregando una dimensión adicional a la lista (como los arreglos multidimensionales), como se ve a continuación:

```
nums = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
n = [0, [1, 2, 3], 9, [2, 2, 2], 1, 3, 5, [6, 7, 8]] # Ejemplo de declaración de una lista
     int.

print(nums[1][1]) # Imprime 5.
print(n[0]) # Imprime 0.
print(n[1][0]) # Imprime 1.
```

Apreciamos que, para acceder a un elemento multidimensional de una lista, existen n pares de llaves cuadradas ([]), donde van ingresados los índices del elemento al que queremos acceder. Cuando son multidimensional las listas, podemos ver las listas de dos dimensiones como tablas, con filas y columnas, donde Python primero recorre las filas y luego las columnas para posicionarse en el elemento con índice solicitado; cuando la lista tiene elementos listas y elementos que no son listas, podemos ver que al elemento que no es lista solo se le aplica el uso de un par de llaves cuadradas, mientras a los elementos lista si se les aplica el uso de varios pares de llaves cuadradas.

Podemos aplicar el uso del *operador[]* a las cadenas. Si tenemos una variable cadena y utilizamos el par de llaves cuadradas con un número entero como índice, accedemos al carácter en dicho índice de la cadena, como vemos en el siguiente ejemplo:

```
nombre = 'Luis' # Declara e inicializa cadena con nombre.
```

```
print(nombre[1]) # Despliega "u".
```

También podemos acceder a ciertos elementos de la lista, utilizando los **dos puntos (:)** dentro de los paréntesis cuadrados del nombre de la lista:

```
nums = [1, 3, 5, 7, 9, 11]

print(nums[0:1]) # Imprime 1.
print(n[1:4]) # Imprime 3, 5, 7.
print(n[4:5]) # Imprime 11.
```

Puede pasar que olvidemos escribir un número en alguno de los dos lados de los dos puntos, sin embargo, esto no es un error, podemos obtener los elementos partiendo de uno de los dos números escritos del lado izquierdo o derecho de los dos puntos, además, podemos obtenerlos de uno en uno, dos en dos, tres en tres, etc:

```
nums = [1, 3, 5, 7, 9, 11]

print(n[:3]) # Imprime 1, 3, 5.
print(n[1:]) # Imprime 3, 5, 7, 9, 11.
print(n[1:5:2]) # Imprime 3, 7.
print(n[1::2]) # Igual que la instrucción anterior.
```

A su vez, podemos obtener los elementos de una lista de manera negativa, es decir, de n elemento hacía atrás:

```
nums = [1, 3, 5, 7, 9, 11]

print(n[:4:-1]) # Imprime 7, 5, 3, 1.
print(n[::-1]) # Imprime 11, 9, 7, 5, 3, 1.
```

8.1.1 Operaciones con listas

Puedes realizar distintas operaciones similares a las que se pueden hacer con cadenas y con otras expresiones, como vemos en la *Tabla 7*:

Table 7: Operaciones con listas

Operación	Significado	Ejemplo
Asignación de valor a un elemento en concreto	Puedes acceder a un elemento de la lista y cambiar su valor	n = [1, 3, 5] n[0] = 33 print(n)
Sumas una lista a otra	Puedes agregarle una lista a otra con el operador +	n = [1, 3, 5] n += [7, 9, 11] print(n)
Operaciones aritméticas entre elementos	Puedes sumar, multiplicar, restar, dividir u obtener el residuo de una división entre elementos de una misma lista o distintas	n = [1, 3, 5] m = [7, 9, 11] print(n[1] // m[1])
Multiplicación de listas	Repite todos los elementos de una lista n cantidad de veces	n = [1, 3, 5] print(n * 4)
Búsqueda de un elemento	Se busca un elemento en una lista utilizando la palabra reservada **in**, regresa True si lo halló, en caso contrario, regresa False	n = [1, 3, 5] if (7 in n): print("Encontrado") else: print("No encontrado") print(1 in n)
Comprobar si un elemento no está en la lista	Se comprueba si un elemento no existe en una lista con la palabra reservada **not**, regresa True si no existe, en caso contrario, regresa False	n = [1, 3, 5] if (7 not n): print("No existe") else: print("Existe") print(1 not n)

8.1.2 Funciones de listas

La *Tabla 8* contiene algunas de las funciones más útiles de las listas son:

Table 8: Funciones especiales para listas

Función	Significado	Ejemplo
len(lista)	Regresa la cantidad de elementos	len(lista)
append(elemento)	Agrega un elemento al final de la lista	lista.append(4)
insert(índice, elemento)	Agrega un elemento en el índice deseado	lista.insert(0, 10)
index(elemento)	Regresa el índice de la primer aparición de un elemento en la lista	letras.index("r")
max(lista)	Regresa el valor más alto de la lista	max(numeros)
min(lista)	Regresa el valor más pequeño de la lista	min(numeros)
count(elemento)	Regresa la cantidad de veces que se repite un elemento en la lista	letras.count("r")
remove(elemento)	Remueve un elemento existente de la lista	letras.remove("r")
reverse()	Invierte los elementos de la lista	numeros.reverse()

8.1.3 Listas con reglas

Podemos declarar elementos de una lista siguiendo una condicional o un ciclo, por ejemplo, declarar los elementos de una lista que estén elevados a la tercera potencia, o que sean números pares, etc. Esto se consigue metiendo ciclos for o condicionales dentro de los paréntesis cuadrados [], como vemos ahora:

```
# Declara lista con cinco elementos a la tercera potencia.
cubos = [i ++ 3 for i in range(5)]

print(cubos) # Imprime [0, 1, 8, 27, 64].

# Declara lista con números al cuadrado y que sean pares.
pares = [i ** 2 for i in range(10) if i ** 2 % 2 == 0]

print(pares) # Imprime [0, 4, 16, 36, 64]
```

8.1.4 Ordenamiento de listas

La forma nativa para ordenar una lista en Python es por medio de la función de listas **sort()**:

```
# Declara lista con números.
numeros = [16, 4, 9, 1, 3, 20, 8]
# Llamada a la función que ordena ascendentemente una lista.
```

```
numeros.sort()
# Imprime la lista.
print(numeros)

# Imprime: [1, 3, 4, 8, 9, 16, 20]
```

Esta función cambia a la lista propiamente, si queremos crear una lista ordenada a partir de una desordenada, podemos utilizar la función **sorted()**:

```
# Declara lista con números.
numeros = [16, 4, 9, 1, 3, 20, 8]
# Asigna a una nueva lista la llamada a la función que ordena ascendentemente una lista.
nums = numeros.sorted()
# Imprime las lista.
print(numeros)
print(nums)

# Imprime:
# [16, 4, 9, 1, 3, 20, 8]
# [1, 3, 4, 8, 9, 16, 20]
```

Por defecto en ambas funciones, las listas son ordenadas de manera ascendente, sin embargo, los dos métodos poseen un parámetro llamado **reverse**, el cual está inicializado en *false*, si cambiamos su valor a *true*, la lista será ordenada descendentemente:

```
# Declara lista con números.
numeros = [16, 4, 9, 1, 3, 20, 8]
# Llamada a la función que ordena descendentemente una lista.
numeros.sort(reverse=true)
# Imprime la lista.
print(numeros)

# Imprime: [20, 16, 9, 8, 4, 3, 1]
```

Si se quiere ordenar una lista irregular (que contenta cadenas, caracteres, números o decimales), las función sort y sorted lanzarán un error.

8.2 Pilas

Las **pilas** son una estructura de datos que sigue la filosofía de Último en llegar, Primer en irse (LIFO, Last In, First Out), estas pueden ser estáticas (no cambia tope de elementos) o dinámicas (cambia su tope de elementos durante ejecución).

Python no suele ser estático, por lo cual, las pilas tampoco, se les puede establecer un número tope de elementos a almacenar, pero se mantiene su esencia dinámica; en otros lenguajes, existen librerías o clases que deben ser importadas o adjuntadas a nuestro proyecto para trabajar con métodos especializados para pilas, sin embargo, este lenguaje no posee algo similar, las **listas** son pilas en sí, así que nos limitaremos simplemente a crear una clase con las funciones básicas de una pila:

```
class Pila:
    def __init__(self):
            self.pila = []
```

```
    def agregar_elemento(self, elemento):
           self.pila.append(elemento)

    def eliminar_elemento(self. elemento):
           self.pila.pop()

    def verificar_vacia(self):
           return self.pila == []

    def buscar_elemento(self, elemento):
           if elemento in self.pila:
                  return true
           else
                  return false

    def desplegar_elementos(self):
           print(self.pila)

    def vaciar_pila(self):
           self.pila.clear()
```

La clase posee un método para agregar, eliminar, recorrer, buscar y vaciar la pila, además de verificar si está vacía o no, esto gracias a las funciones de las listas. La lista actúa como una pila debido a que la función **append()** agrega los elementos al final de la lista (se agrega el primero, se agrega el segundo después y el primero se recorre, entonces el segundo ahora es el primero y así sucesivamente), y la función **pop()** elimina el último elemento de la lista, respetando así que el último elemento entrar es el primero en salir o ser eliminado.

8.3 Colas

Las **pilas** son una estructura de datos que sigue la filosofía de Primero en llegar, Primer en irse (FIFO, First In, First Out), estas pueden ser estáticas (no cambia tope de elementos) o dinámicas (cambia su tope de elementos durante ejecución).

A diferencia de las pilas, esta estructura si cuenta con un módulo que podemos importar a Python, la cual es llamada **deque**:

```
from collections import deque
```

Dejaremos las operaciones básicas que se pueden realizar con las colas y su módulo importado en una clase:

```
from collections import deque

class Cola:
    def __init__(self, cola):
           self.cola = []

    def agregar_elemento(self, elemento):
           self.cola.append(elemento)

    def eliminar_elemento(self. elemento):
```

```
            self.cola.popleft()

    def verificar_vacia(self):
            return self.cola == []

    def buscar_elemento(self, elemento):
            return elemento in self.cola

    def desplegar_elementos(self):
            print(self.cola)

    def vaciar_cola(self):
            self.cola.clear()
```

Nota: fíjese que el método para eliminar un elemento de la cola es llamado **popleft()**, esto es así debido a que el módulo **deque** tiene dos métodos para eliminar elementos: el anterior mencionado y **popright()**.

Para poder trabajar con esta clase y el módulo *deque*, primero debemos crear un objeto del módulo anterior y, posterior a ello, crear un objeto de la clase que nosotros creamos, y pasarle como parámetro el objeto *deque*; podríamos trabajar tranquilamente solamente con el objeto deque y sus métodos mostrados en el ejemplo anterior, pero para mayor comodidad de lectura, decidimos implementarlo dentro de una clase:

```
# Si se imprime la siguiente línea, resulta en una lista vacía.
nueva_cola = deque()

clase_cola = Cola(nueva_cola)
```

8.4 Rangos

Los **rangos** son una lista con una secuencia de números dada, si requerimos una lista con los números del 1 al 100, podemos utilizar esta herramienta, **no regresa el último número** (si queremos un rango de 1 al 100, obtenemos el rango del 0 al 99, ya sabemos que muchos lenguajes de programación utilizan el 0 como primer valor de iteración). Un ejemplo de los rangos con tres tipos de casos.

```
numeros = range(100) # Genera un rango de 100 numeros (0 - 99).
x = range(1, 20) # Genera un rango de uno a 19 (1 - 19).
y = range(1, 30, 3) # Genera un rango de uno a 30 de tres en tres (1, 4, 7, 10, ..., 28).
z = range(100, 1, -5) # Genera un rango negativo de 100 a uno de menos cinco en menos
    cinco (100, 95, 90, 85, ..., 5).

for a in range(5):
    print("hola mundo") # Imprime cinco veces el mensaje "hola mundo".
```

Como vimos, la función range puede crear rangos negativos, que vaya de n en n números, y pueden ser utilizados para repetir un ciclo for n cantidad de veces.

También podemos convertir los rangos a listas, con la función **list()**.

8.5 Diccionarios

Los **diccionarios** son un tipo de colección de datos donde cada **llave** (identificador) tiene un **valor** (contenido), algo así como una dupla, pero los diccionarios son colecciones de duplas; podemos acceder a los valores de los diccionarios utilizando el nombre del diccionario y su llave.

```
Nombres = { # Declara diccionario.
    # Inicializa diccionario con tres llaves y valores.
    "David" : 19
    "Maria" : 26
    "Luis" : 21
}

print(Nombres["David"]) # Despliega el valor de la llave (19).
```

Vemos que se utilizan tipos de datos básicos para las llaves y valores, estos tipos son **mutables**, quiere decir que puede cambiar de valor o ser modificado, a la par de que puede cambiar de tipo de dato (recordemos que esto no es una buena practica en el diseño de programas, es posible, pero no recomendable); las *listas* también son mutables, por lo que podemos utilizar listas como valores de una llave, no como una llave en sí, si utilizamos una lista como una llave, causaría un error.

8.5.1 Funciones en diccionarios

Podemos utilizar las palabras reservadas **in** o **not in** para determinar si una llave existe o no en un diccionario; usamos **get** para desplegar el valor de una llave, algo como el operador [] para acceder al valor, sin embargo, si no se encuentra la llave en el diccionario, despliega otro valor como resultado. Veamos el siguiente ejemplo:

```
# Declara diccionario.
Numeros = {
    # Inicializa diccionario con llaves y valores.
    1 : "UNO"
    2 : "DOS"
    3 : "TRES"
    4 : "CUATRO"
    5 : "CINCO"
    6 : "SEIS"
    7 : "SIETE"
}

# Si no existe la llave en numeros, despliega mensaje.
if (8 not in Numeros):
    print("No existe esa llave.")
    # Si 1 existe en mensaje, despliega su valor.
elif (1 in Numeros):
    print(Numeros[1])
    # Busca la llave 2 y despliega su valor, sino, despliega "Ese es un dos".
    print(Numeros.get(2, "Ese es un dos"))
```

8.6 Tuplas

Las **tuplas** son como las listas, pero **inmutables** (no puedes cambiar el valor de uno de sus elementos, si lo intentas, ocurrirá un error), se inicializa con paréntesis (o sin los paréntesis) y se pueden acceder a sus elementos por medio del operador [], como vemos en el siguiente ejemplo:

```
# Declara diccionario.
Nombres = {
    "David" : 19
    "Maria" : 26
    "Luis" : 21
}

# Declara lista.
Numeros = [1, 2, 3, 4, 5]

# Declara tupla con parentesis.
Municipios = ("Tijuana", "Mexicali", "Tecate", "Ensenada")

# Declara tupla sin parentesis.
Estados = "Baja California", "Baja California Sur", "Sonora", "Chihuahua"

print(Municipio[2]) # Despliega Tecate.

Estados[0] = "Quintana Roo" # Error.
```

Vemos que las listas son declaradas con paréntesis cuadrados [], los diccionarios con llaves {}, y las tuplas con paréntesis ().

Pueden ser utilizadas las tuplas en las listas, si recordamos, mencionamos previamente que los diccionarios pueden contener tipos de datos mutables, no inmutables para las llaves, sin embargo, podemos crear un diccionario y que sus elementos sean tuplas, esto da como resultado un diccionario con valores que no pueden ser modificados, logrando así mantener la integridad de los datos.

8.6.1 Desempacando tuplas

Si tenemos un conjunto de variables y una tupla con la misma cantidad de elementos dentro de ella, podemos asignar cada uno de los elementos a cada una de las variables, a esta acción se le llama **desempacar tuplas**, además, podemos asignar todos los elementos a la derecha de una tupla a una sola variable, con el operador *, como vemos a continuación:

```
# Declara tupla.
Municipios = ("Tijuana", "Mexicali", "Tecate", "Ensenada")

# Declara tupla.
Estados = "Baja California", "Baja California Sur", "Sonora", "Chihuahua"

# A a se le asigna Tijuana, a b se le asigna Mexicali,
# a c se le asigna Tecate, y a d se le asigna Ensenada.
a, b, c, d = Municipios
```

```
# A e se le asigna Baja California, y a f se le asigna todos los elementos
# restantes de la tupla.
e, *f = Estados

# Imprime las variables.
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
```

8.7 Sets

Los **conjuntos** (o **sets**, como son conocidos), son algo así como una lista o diccionario, pero, a diferencia de las listas, los elementos de un conjunto no deben estar duplicados, a la vez que están desordenados, por lo que no se pueden acceder por medio del operador [], logrando con esto que la búsqueda de los elementos sea más rápida que con las listas.

Si un elemento del conjunto está duplicado, a la hora de imprimirlo, solo se muestra una vez, si se elimina uno de los elementos duplicados, se eliminan el resto de las copias, y si se imprime la cantidad de elementos, se descartan todos los elementos duplicados (si se repite cuatro veces 1, solo se imprime una vez 1).

Se declaran igual que una lista, podemos utilizar los métodos **add(elemento)**, **remove(elemento)** y **len()** para agregar, eliminar un elemento respectivamente y desplegar la cantidad de items del conjunto, como vemos a continuación:

```
# Declara conjunto con elementos duplicados.
numeros = {1, 1, 3, 4, 5, 7, 8, 8, 9}

print(numeros) # Imprime 1, 3, 4, 5, 6, 7, 8, 9.
print(len(numeros)) # Imprime 7.

numeros.add(0) # Agrega 0 al conjunto.
numeros.remove(1) # Elimina todos los 1 del conjunto.

print(numeros) # Imprime 0, 3, 4, 5, 6, 7, 8, 9.
print(len(numeros)) # Imprime 7.
```

8.7.1 Operaciones con conjuntos

Como en las matemáticas, podemos aplicar las operaciones de conjuntos a esta colección de datos (*Tabla 9*):

Table 9: Operaciones para conjuntos

Operación	Significado	Ejemplo
union (—)	Regresa la suma de todos los elementos de los conjuntos	primero — segundo
intersection (&)	Regresa solamente los elementos que coinciden en ambos conjuntos	primero & segundo
difference (-)	Regresa los elementos de un conjunto, pero no del otro, ni los que coinciden	primero - segundo
symmetric difference ()̂	Regresa los elementos que no coinciden en ambos conjuntos	primero ŝegundo

El resultado de los ejemplos anteriores se puede ver a continuación:

```
# Declara dos conjuntos.
first = {1, 2, 3, 4, 5, 6}
second = {4, 5, 6, 7, 8, 9}

print(first | second) # Imprime {1, 2, 3, 4, 5, 6, 7, 8, 9}
print(first & second) # Imprime {4, 5, 6}
print(first - second) # Imprime {1, 2, 3}
print(second - first) # Imprime {8, 9, 7}
print(first ^ second) # Imprime {1, 2, 3, 7, 8, 9}
```

8.8 Seleccionando una colección de datos

La siguiente lista da algunos tips sobre como escoger una colección de datos que más se adecue a nuestras necesidades:

- Si necesitas dos valores que vayan ligados o tengan una relación directa, velocidad a la hora de buscar valores y, que a su vez, estos valores puedan cambiar, utiliza un **diccionario**.
- Si necesitas un conjunto de datos sin orden y sin relación con otros datos y, que a su vez, sea bastante sencillo e iterable, utiliza una **lista**.
- Si requieres de un conjunto de datos que no estén repetidos, utiliza un **conjunto**.
- Si requieres un conjunto de datos que no puedan ser modificados, utiliza una **tupla**.
- Las *tuplas* pueden ser combinadas con el resto de colecciones de datos.

9 Funciones

Una **función** es un bloque de código que puede ser llamado dentro de otra sección del código para realizar una tarea en específico. En todo lenguaje de programación, las funciones están constituidas de un **nombre** y **argumentos**, como hemos visto a lo largo de estos apuntes. Podemos crear nuestras propias funciones con la palabra reservada **def**, seguido del nombre y los argumentos, como vemos en la siguiente estructura:

```
def nombre_funcion(argumentos):
    # Código.
```

Como vemos, sigue la misma estructura que la declaración de un ciclo o una condicional, dentro de alguna otra función, mandamos a llamar a nuestra función creada escribiendo su nombre y sus argumentos (en caso de que tenga, sino, solo escribimos el par de paréntesis), debe recordar que la función debe ser declarada previamente a su llamada:

```
# Función que recibe el nombre del usuario y lo despliega.
def Nombre():
    nombre = str(input("Escribe tu nombre"))
    apellidos = str(input("Escribe tus apellidos: "))
    print(nombre + " " + apellidos)

# Función que despliega un mensaje y un nombre.
def principal():
    print("Hola, mi nombre es ")
    Nombre() # Llamada a la función creada.
```

9.1 Funciones que regresan valor

Las funciones que no regresan valor (las que fueron definidas en la sección anterior) les podemos llamar **procedimientos**, que son funciones que no regresan valor, aquellos bloques de código repetibles y reusables que regresan algún tipo de valor les denominaremos funciones **funciones**, para tener clara una diferencia entre el tipo que regresa y el que no un valor. Para que una función regrese un valor, se declara como ya vimos, y utilizamos la palabra reservada **return**, seguido del valor que queremos que se regrese, este puede ser entero, cadena, decimal o boleano.

```
def Potencia(x, y): # Función que regresa un número elevado a una potencia.
    return x * y # Regresa la potenciación.

# Ingreso de valores a las variables.
a = int(input("Ingresa un número: "))
b = int(input("Ingresa otro número: "))

# Imprime el resultado.
print("El número " + a + " elevado a la " + b + " es: "Cuadrado(a, b))
```

Como vemos, la función declarada utiliza dos argumentos en su declaración, además de que la llamamos dentro de print, por lo que puede actuar como si fuera una variable, podemos utilizar esta función para asignarle un valor a una variable si tuviéramos la necesidad.

9.2 Funciones Lambda

Existen dos formas de crear funciones en Python: declarándolas con la palabra **def**, con un nombre, argumentos y conjunto de instrucciones, y creando la función con la palabra reservada **lambda**. Podemos decir que las funciones lambda son hechas en el camino, para un uso rápido, y que contiene solo una instrucción, su estructura consiste en la palabra lambda, los parámetros que se van a utilizar, dos puntos, la instrucción a realizar, y el valor de o los parámetros, como vemos enseguida:

```
# Función con def.
def polynomial(x):
    return x**2 + 5*x + 4
print(polynomial(-4))

# Función con lambda.
print((lambda x: x**2 + 5*x + 4) (-4))
```

Entonces, vemos que solo se puede realizar una instrucción con este tipo de instrucciones, que también son llamadas **funciones anónimas**.

9.3 Funciones map y filter

La función **map** regresa un objeto iterable, y recibe como parámetros una función y otro objeto iterable (que puede ser un conjunto de datos o ciclos), podemos decir que la función map repite, en un objeto iterable, una función que interactúe con un objeto iterable, es como decir que a cada elemento de una lista se le realice una acción, como sumar cinco. Veamos como se utiliza:

```
# Declara una lista con números.
numeros = [3, 4, 5, 6, 7, 8]

# Se utiliza map para multiplica cada uno de los elementos de la lista por dos con una
# función lambda, el resultado se convierte en lista.
lista = list(map(lambda x: x * 2, numeros))

print(lista) # Imprime [6, 8, 10, 12, 14, 16]
```

La función **filter** se encarga de tomar únicamente ciertos elementos de un objeto iterable que cumplan con una condición (llamada **predicado**):

```
# Declara una lista con números.
numeros = [3, 4, 5, 6, 7, 8]

# Se utiliza filter para obtener los elementos mayores a cinco con una
# función lambda, el resultado se convierte en lista.
lista = list(filter(lambda x: x>5, numeros))

print(lista) # Imprime [6, 7, 8].
```

9.4 Generadores

Los **generadores** son un tipo iterable, como listas o tuplas, que no poseen alguna forma de acceder a los elementos que lo componen, pero que pueden ser recorridos con ciclos. Este tipo de iterable se utiliza dentro de funciones que serán repetidas en ciclos, las variables o acciones que se puedan borrar entre iteración e iteración se mantendrán intactas gracias a palabra reservada **yield**, que es como el return de la función, esta palabra es lo que define a una función como generadora.

```
def numeros():
    x = 5
    while x > 0:
        yield x
        x -= 1

for x in numeros():
    print(x)
```

Debemos recordar poner alguna condición que acabe con el ciclo o el generador, para que no se repitan las instrucciones de manera infinita.

9.5 Decoradores

Los **decoradores** son funciones que pueden modificar otras funciones, de tal forma que la decoración o modificación sea algo leve, pero despliega de otra forma los resultados de la función a decorar, para lograr este cometido, debemos declarar una función dentro de otra función, como vemos a continuación:

```
# Declara función para decorar otra función como parámetro.
def decor(fun):
    # Función que realiza la modificación a la función parámetro.
    def partir():
        # Agrega iguales al despliegue de resultados.
        print("================")
        # Llamada a la función parámetro.
        fun()
        print("================")
    # Regresa la función que realizó la modificación.
    return partir

# Función que será modificada.
def imprimir_texto():
    print("Hola mundo")

# Variable que recibe la función modificada.
decoracion = decor(imprimir_texto)
# Llamada a la función modificada.
decoracion()

#Despliega:
#================
#Hola mundo
#================
```

La función *decor* recibe una función como parámetro (llamada *fun*), dentro, se declara la función *partir* (declaración de función dentro de otra), dentro de la función partir, se realiza la decoración o modificación a la función que se quiere decorar, vemos que en el ejemplo anterior se agregan iguales encima y debajo de la función parámetro *fun*, finalmente, se regresa a la propia función *partir*, vemos que tenemos una función a decorar llamada *imprimir_ texto*, que simplemente imprime "Hola mundo", la variable *decoracion* recibe el resultado de llamar a *decor* con *imprimir_ texto* como parámetro, ya para terminar, la variable se convierte en un tipo de función, por lo que le agregamos paréntesis al final para poder ver el resultado.

Utilizamos una variable para almacenar la función decorada, sin embargo, podemos hacer que la función a modificar reciba a ella misma, pero con la modificación ya hecha, además, podemos utilizar el arroba () antes de la función a modificar, seguido del nombre de la función decoradora (*decor* en el ejemplo anterior). Veamos el ejemplo anterior pero con lo que acabamos de mencionar:

```
# Declara función para decorar otra función como parámetro.
def decor(fun):
    # Función que realiza la modificación a la función parámetro.
    def partir():
        # Agrega iguales al despliegue de resultados.
        print("================")
        # Llamada a la función parámetro.
        fun()
        print("================")
    # Regresa la función que realizó la modificación.
    return partir

# Funcion que sera modificada.
@decor
def imprimir_texto():
    print("Hola mundo")

#La función a modificar será modificada en el momento.
imprimir_texto = decor(imprimir_texto)
# Llamada a la función modificada.
imprimir_texto()

# Despliega:
#================
#Hola mundo
#================
```

9.6 Parámetros *args y **kwargs

Podemos hacer que una función reciba una cantidad indeterminada de parámetros con ***args**, es decir, en una llamada puede recibir dos parámetros, en otra llamada, puede recibir cuatro, en otra, solamente tres, la forma en la que se reciben estos parámetros es por medio de **tuplas**. Veamos el siguiente ejemplo:

```
# Define función que recibe una cantidad variable de parámetros con *args.
```

```
def funcion(*args):
    # Imprime el/los parámetros recibidos.
    print(args)

# Llamada a la función y se le mandan distintos numeros de parámetros.
funcion(1, 2, 3, 4, 5) # Imprime (1, 2, 3, 4 5).
funcion(1, 2)   # Imprime (1, 2).
funcion(2, 3, 4)  # Imprime (2, 3, 4).
funcion(4)    # Imprime (4, ).
```

Como vemos, si llamamos a la misma función con distinta cantidad de parámetros y los imprimimos, se reciben e imprimen correctamente, como este tipo de parámetro es una tupla, podemos acceder a los elementos de la misma con el operador [], podemos hacer que la función tenga, por ejemplo, tres parámetros, donde dos de ellos no posean el asterisco y uno si, como vemos enseguida.

```
# Define función que recibe dos parámetros y tupla variable de parámetros con *args.
def funcion(par1, par2, *args):
    # Imprime el/los parámetros recibidos.
    print(args)
    print(args[0])
    print(par1)
    print(par2)

# Llamada a la función y se le mandan distintos numeros de parámetros.
funcion(1, 2, 3, 4, 5) # Imprime (3, 4, 5) 3 1 2.
funcion(1, 2)   # Imprime () 1 2.
funcion(2, 3, 4)  # Imprime (4, ) 2 3.
```

Entonces, podemos tener tuplas variables de parámetros y parámetros fijos, en caso de que la función tenga tupla de parámetros y no se reciban en la llamada a la función, se despliegan una tupla vacía. El parámetro ***kwargs** (por las palabras *keywords arguments*) permite recibir parámetros como si fueran variables, es decir, una variable con su nombre y su tipo de dato, la función trata a estos parámetros como listas, donde el nombre de la variable es la llave, y el valor de la variable es el valor de la llave; veamos un ejemplo para que quede más claro:

```
# Define función que recibe dos parámetro fijos, una tupla de parámetros y un diccionario
    de parámetros.
def my_func(x, y=7, *args, **kwargs):
    print(x)  # Imprime el parámetro x.
    print(y)  # Imprime el parámetro y, que inicialmente vale 7.
    print(args) # Imprime una tupla de parámetros.
    print(kwargs) # Imprime un diccionario de parámetros.

# Llamada a la función con distintos tipos de parámetros.
my_func(2, 3, 4, 5, t = 1, a=7, b=8)
# Imprime:
# 2
# 3
# (4, 5)
# { 't':1, 'a':7, 'b':8 }
```

Al igual que con las tuplas de parámetros, para acceder a los elementos de los diccionarios parámetros utilizamos el nombre del parámetro y el operador [], donde aparece el nombre de la llave a buscar o utilizar.

9.7 Listas como parámetros

No es más que agregar un nombre de variable a los parámetros de una función, y tratar a este nombre como una lista dentro de la función, como vemos en el siguiente ejemplo:

```
# Función que regresa la suma de todos los elementos de una lista.
mi_funcion(lista):
    # Declaración de variable.
    suma=0
    # Ciclo que recorre los elementos de la lista parámetro.
    for x in range(len(lista)):
        suma=suma+lista[x]
    return suma
```

9.8 Sobrecarga de métodos

En otros lenguajes, como en CSharp, Java o C++, podemos crear una función o método con un nombre y cierto número de parámetros, posterior a esto, podemos crear nuevamente la función, pero con una cantidad distinta de parámetros, y no necesariamente la función debe regresar el mismo tipo de dato, esto es la **sobrecarga de métodos**; la llamada de la función será dependiendo de la cantidad de parámetros que le pasemos.

Sin embargo, esto no existe directamente en Python, se puede simular utilizando los parámetros *args, veamos el siguiente ejemplo:

```
# Regresa el perímetro de un triángulo.
def perimetro(*args):
    if args[0] != 0:
        # Perímetro de un triángulo equilátero.
        return args[0] * 3
    elif args[1] != 0:
        # Perímetro de un triángulo isósceles.
        return (args[1] * 2) + args[2]
    elif args[3] != 0:
        # Perímetro de un triángulo escaleno.
        return args[3] + args[4] + args[5]
```

Vemos que la función tiene el tipo de parámetro especial donde podemos mandarle una cantidad de argumentos en una llamada y otra cantidad distinta en futuras llamadas; la función evalúa si el primer elemento de la tupla del parámetro es distinto de 0, si es así, regresa un valor, esto aplica para la primer función sobrecargada con n cantidad de parámetros (en este caso: 1); si es falta esta evaluación, evalúa si el segundo elemento de la tupla es distinto de 0, si es así, regresa el perímetro de un triángulo isósceles, el cual requiere de dos valores (altura y base) para ser calculado, lo cual correspondería con otra versión de la función sobrecargada donde se reciben dos parámetros en lugar de uno; por último, si la segunda evaluación es falsa, verifica si el cuarto elemento en la tupla es distinto de 0, si es cierto,

regresa el perímetro de un triángulo escaleno, el cual requiere de tres valores (lado 1, lado 2 y lado 3) para ser calculado, coincidiendo así con una tercera versión de la función sobrecargada.

Lo que describimos anteriormente sería lo mismo al siguiente ejemplo en CSharp:

```
// Método Perimetro.
public float Perimetro(float lado)
{
    return 3 * lado;
}

// Método sobrecargado una vez.
public float Perimetro(float altura, float base)
{
    return (2 * altura) + base;
}

// Método sobrecargado dos veces.
public float Perimetro(float lado1, float lado2, float lado3)
{
    return lado1 + lado2 + lado3;
}
```

Como vemos, CSharp requiere de tres declaraciones y de la misma función con distinta cantidad de parámetros, Python simula esto con los parámetros *args pero, al poder recibir en una llamada un argumento, en la siguiente dos, en la siguiente tres, debemos especificar dentro de la función qué comportamiento debe adoptar en cualquier de los casos que les acabamos de describir, es por ello que la función evalúa si el primer, segundo y cuarto elemento de la tupla es distinto de 0.

9.9 Números aleatorios

Es necesario importar el módulo **random** para poder generar números aleatorios:

```
import random
```

La función **random()** te genera un número decimal entre 0.0 y 1.0 (excluye a estos dos):

```
import random

for i in range(5):
    print(random.random())

# Imprime:
# 0.6355590910913725.
# 0.38870490605141683.
# 0.3757381647176976.
# 0.38770694501889935.
# 0.22472067642268556.
```

9.9.1 Números aleatorios enteros entre dos valores

La función **randint(x, y)** te genera un número entero entre dos valores (incluidos):

```
import random

for i in range(5):
    print(random.randint(1, 10))

# Imprime:
# 6.
# 3.
# 7.
# 3.
# 2.
```

9.9.2 Números aleatorios enteros entre dos valores con salto

La función **randrange(x, y, salto)** te genera un número entero entre dos valores (incluidos) dando un salto a partir del valor mínimo ingresado:

```
import random

for i in range(5):
    print(random.randrange(5, 27, 4))

# Imprime:
# 17.
# 13.
# 5.
# 17.
# 13.
```

9.9.3 Números aleatorios decimales entre dos valores

La función **uniform(x, y)** te genera un número decimal entre dos valores (incluidos):

```
import random

for i in range(5):
    print(random.uniform(100, 200))

# Imprime:
# 170.3543065193162.
# 103.47025653056637.
# 126.52588283656675.
# 169.60671144065486.
# 145.21872629322894.
```

10 Recursividad

La **recursividad** es que una función se llame a sí misma una cantidad finita de veces, teniendo que poseer un **caso** o **condición** para que las llamadas a la función acabe, sino,

sería indeterminada; la recursividad es una alternativa al uso de ciclos, el ejemplo más clásico es el factorial:

```
# Declara función recursiva.
def factorial(x):
    # Condición de salida de la recursividad.
    if x == 1:
        return 1
    else:
        # Regresa el parámetro multiplicado por la misma función con el parámetro alterado.
        return x * factorial(x-1)

# Despliega 120.
print(factorial(5))
```

El ejemplo anterior corresponde con la **recursividad directa**, la **recursividad indirecta** corresponde con la llamada de una función 1 a la función 2, y luego la función 2 llama a la función 1, así durante n cantidad de veces.

11 Programación funcional

Este es un estilo de programación que se enfoca en las **funciones**, es decir, programaremos alrededor de una función, que reciba un valor, y regrese otro, como las funciones matemáticas (f(x)). Las funciones que reciben otra función como parámetro o argumento y regresa un valor o función son llamadas **funciones de orden mayor**. Este tipo de programación no conlleva palabras reservadas, simplemente es un estilo de programar. Veamos el siguiente ejemplo para ver de qué estamos hablando:

```
# Declara función que se llama a sí misma.
def doble(fun, para):
    return fun(fun(para))

# Declara función que suma 5 a x.
def cinco_mas(x):
    return x + 5

# Imprime la función doble, le pasa como parámetro la función
# cinco_mas, y el valor a sumar.
print(doble(cinco_mas, 10)) #Imprime 20
```

El sentido aquí es que se llama a la función de orden mayor, esta recibe como parámetro otra función a ejecutar, más uno o más parámetros que serán manipulados por la función a ejecutar, dentro de la función de orden mayor, se regresa a ella misma, que a su vez se llama a ella misma, dándole sentido al nombre de la función, ejecuta la función que fue pasada originalmente como parámetro, junto con el parámetro que igual fue pasado al inicio, el resultado, si ejecuta este código en algún compilador de Python podrán analizar y comprender un poco más el funcionamiento de la programación funcional.

11.1 Funciones puras e impuras

Existen tipos de funciones en este estilo, nos encontramos con las **funciones puras**, las cuales reciben parámetros y los utilizan para regresar un valor independiente a estos parámetros, los cuales nunca son modificados; por otro lado, tenemos las **funciones impuras**, las cuales si modifican los parámetros que se reciben, podemos ver los siguientes ejemplos:

```
# Declara lista vacía.
numeros = []

# Declara función que recibe dos parámetros, asigna un calculo con los
# parámetros a una variable y la regresa.
def pura(x, y):
    temp = x * y
    return temp

# Declara función que agrega el parámetro recibido a la lista vacía.
def impura(num):
    numeros.append(num)
```

Así pues, la función "pura" calcula un valor para 'temp' con los parámetros recibidos, y se regresa la variable 'temp', convirtiéndola en pura por el simple hecho de no modificar los valores de los parámetros, por otro lado, la función "impura" si bien no modifica el valor del parámetro recibido, modifica una lista declarada anteriormente, volviéndola impura.

Las **ventajas** de las funciones puras es que son fáciles de entender, probar y son eficientes en rendimiento, la **desventaja** es que esta fácilidad para entender lo vuelve difícil de escribir en algunas situaciones.

12 Clases

Creamos una clase con la palabra reservada **class**, seguido de su nombre y dos puntos (:).

Para crear **métodos** de una clase, sabiendo que siguen la sintaxis de una función, agregamos indentación a la clase, y le damos un nombre y parámetros. Todos los métodos creados dentro de una clase deben tener un parámetro llamado **self**, refiriéndose a algún atributo de la clase.

Para crear **atributos** de una clase no es necesario crear una sección con encapsulamiento, ni tipos de datos, ni nombres, como dijimos previamente, basta con utilizar la palabra reservada *self* en cada método creado (ya sea el constructor u otro alguno) para que la clase sepa que dicho parámetro es un atributo de la clase, podemos acceder a él desde un objeto de clase por medio del operador ., simplemente recuerde qué atributos tendrá la clase y puede apuntarlos en comentarios, para recordarlos más fácilmente.

12.1 Constructores

Este es el método llamado __ **init**__, el cual es llamado cuando un objeto de una clase es llamado, si se desea inicializar algún atributo de la clase por medio del constructor, debe contener al inicio el parámetro *self*, como vemos enseguida:

```
# Declara clase.
class Perro:
    # Constructor.
    def __init__(self, nombre, raza):
        # Atributos: nombre y raza.
        self.nombre = nombre
        self.raza = raza

    # Método que despliega sonido del perro.
    def sonido(self):
        print("Woof")

# Declara objeto de la clase Perro.
perrito = Perro("chuchis", "amarillo")
# Despliega el nombre del perro.
print(perrito.name)
# Llamada al método sonido de la clase Perro.
perrito.sonido()

# Despliega:
# chuchis
# Woof
```

12.2 Destructores

Este es el método llamado __ **del**__, el cual es llamado cuando un objeto de una clase deja de ser utilizado, es eliminado o la ejecución del programa ha finalizado.

```
def __del__(self):
```

12.3 Herencia

Para heredar los atributos y métodos de una clase, primero debemos escribir el nombre de la clase heredera, seguido de paréntesis, en su contenido, escribimos el nombre de la clase a heredar y dos puntos:

```
nombreClase(claseHeredera):
```

Con esto, pasamos el contenido de una clase a otra, además, sabemos que la clase heredera es una **superclase** o **clase padre**, mientras que una heredada es una **sub clase** o **clase hija**, en la clase hija podremos crear métodos independientes a la clase padre. Veamos el siguiente ejemplo.

```
# Declara clase base.
class Animal:
    # Constructor.
    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza

    # Método de clase base.
    def despliegue(self):
        print("Mi nombre es" ++ self.nombre ++ " y mi raza es " ++ self.raza)

# Declara clase hija de Animal.
class Perro(Animal):
    # Método de clase hija.
    def sonido(self):
        print("Woof")

# Declara objeto de clase hija.
perrito = Perro("chuchis", "chihuahua")
# Llamada a métodos de clase hija,
perrito.despliegue()
perrito.sonido()
```

Para acceder a los métodos de una superclase utilizamos la palabra reservada **super()**, y para acceder a los atributos de la superclase, utilizamos la palabra reservada **self**.

```
# Clase base.
class A:
    # Constructor con atributo de prueba.
    def __init__(self, prueba):
        self.prueba = prueba
    # Método propio.
    def spam(self):
        print(1)

# Clase derivada de A.
class B(A):
    # Método que utiliza el método y atributo de su clase base.
    def spam(self):
        print(2)
        super().spam()
```

```
        print(self.prueba)

# Declara objeto de la clase B y llama directamente a su método.
B("hola mundo").spam()
```

12.4 Abstraccion

De manera rápida y directa, Python no posee palabras reservadas para indicar que una clase y sus métodos son abstractos, para acabar con esta problemática, debemos importar la **clase ABC** de la **librería abc** y su **decorador abstractmethod**:

```
from abc import ABC, abstractmethod
```

La *clase abc* debe ser la clase padre de la clase que deseamos que sea abstracta, esto permite, más que heredemos atributos o métodos, que utilicemos el decorador que recién importamos en los métodos de la clase derivada, podemos pensar en la clase abc meramente como una etiqueta para la clase a convertir en abstracta.

Si recordamos, una clase abstracta no puede tener objetos, y todos sus métodos deben ser utilizados en las clases hijas, veamos la creación de una clase abstracta en el siguiente ejemplo:

```
# Importa clase ABC y decorador abstractmethod de la librería abc.
from abc import ABC, abstractmethod

# Declara clase derivada abstracta de clase base ABC.
class Figuras(ABC):
    # Declara métodos abstractos sin inicializar.
    @abstractmethod
    # Regresa el volumen de la figura.
    def calcular_volumen(self):
        pass

    @abstractmethod
    # Despliega el volumen y medidas para desplegar el volumen.
    def desplegar_volumen(self):
        pass
```

El cuerpo del método abstracto no debe quedar vacío, por lo que se utiliza la palabra reservada **pass**. Ahora crearemos una clase derivada de la clase abstracta:

```
class Cilindro(Figuras):
    def __init__(self, radio, altura):
        self.radio = radio
        self.altura = altura

    def calcular_volumen(self):
        return math.pi * math.pow(self.radio, 2) * self.altura

    def desplegar_volumen(self):
        print("""
        ---Datos del cilindro---
        La medida del radio del cilindro es de {}u.
```

```
        La medida de la altura del cilindro es de {}u.
        El volumen del cilindro es de {}u^3.
        """.format(self.radio, self.altura,          round(self.calcular_volumen(), 3)))
```

Si quisiéramos crear más clases derivadas que tuvieran un comportamiento distinto lo podríamos hacer (un cono, un cilindro, un cubo, etc), siempre y cuando declaremos los métodos abstractos en cada clase derivada.

Declaremos un objeto de la clase Cilindro:

```
# Declara objeto Cilindro y se le pasan valores para el radio
# y la altura.
cilindro = Cilindro(5.5, 10.66)
# Llamada al método que despliega el volumen del cilindro.
cilindro.desplegar_volumen()
```

En caso de querer declara un objeto de la clase Figuras, el lenguaje lanzará un error.

12.5 Polimorfismo

No es más que el pensamiento de que una función puede adoptar un comportamiento distinto en las distintas clases derivadas de una base, como lo que se acaba de ver en el punto anterior, pero sin ser abstracto, pondremos un breve ejemplo sobre lo que es polimorfismo:

```
# Clase base.
class Animal:
    # Destructor.
    def __del__(self):
        print("Objeto Animal Eliminado.")

    # Método que adoptará distinto comportamiento en sus clases derivadas (polimorfismo).
    def desplazarse(self):
        pass

# Clase derivada de clase base Animal.
class Ave(Animal):
    # Constructor.
    def __init__(self, nombre):
        self.nombre = nombre

    # Destructor.
    def __del__(self):
        print("Objeto Ave Eliminado.")

    # Método que adopta su propio comportamiento al de la clase base.
    def desplazarse(self):
        print("Hola, mi nombre es {} y me desplazo volando.".format(self.nombre))

class Mamifero(Animal):
    def __init__(self, nombre):
        self.nombre = nombre

    def __del__(self):
        print("Objeto Mamifero Eliminado.")
```

```
    def desplazarse(self):
        print("Hola, mi nombre es {} y me desplazo caminando.".format(self.nombre))

# Declara objetos de clase Ave y Mamifero.
ave = Ave("Mauricio")
mamifero = Mamifero("Gonzalo")
# Llama a los métodos de desplazarse.
ave.desplazarse()
mamifero.desplazarse()

# Imprime:
# Hola, mi nombre es Mauricio y me desplazo volando.
# Hola, mi nombre es Gonzalo y me desplazo caminando.
```

La clase base posee un método que tendrá un comportamiento distinto en sus clases derivadas, vemos que aquí también aparece la palabra reservada **pass** para no dejar completamente vacío el cuerpo de esta función. A diferencia de la abstracción, si declaramos un objeto de la clase base, el lenguaje no lanzará un error, si lo vemos como jerarquía, primero está el polimorfismo y luego la abstracción, pero ambos son dos conceptos muy ligados a la programación orientada a objetos.

12.6 Métodos mágicos

Son aquellos que tienen dos guiones bajos al inicio y final del nombre, funcionan para representar funcionalidades que no pueden ser creadas con un método regular, por ejemplo, poder realizar operaciones aritméticas con objetos de clases, de hecho, los constructores (__ init__) es un método mágico. La lista de métodos mágicos más comunes es:

- **__init__** para constructores.
- **__add__** para sumas.
- **__sub__** para restas.
- **__mul__** para multiplicaciones.
- **__truediv__** para divisiones regulares.
- **__floordiv__** para divisiones enteras.
- **__mod__** para residuos.
- **__pow__** para potencias.
- **__and__** para comparaciones y.
- **__xor__** para comparaciones xor.
- **__or__** para comparaciones o.
- **__lt__** para comparación menor.

- __**le**__ para comparación menor igual.
- __**eq**__ para comparación igual.
- __**ne**__ para comparación distinto.
- __**gt**__ para comparación mayor.
- __**ge**__ para comparación mayor igual.

12.7 Ocultación de datos

La *encapsulación* de atributos y métodos de una clase es algo normal en la POO y otros lenguajes de programación, sin embargo, este concepto no existe como tal en Python, podemos utilizar la **ocultación de datos** para evitar que secciones de código fuera de la clase accedan a los estos miembros privados, para complementar, solamente existen los "métodos privados", no existen los métodos protegidos. Un método puede hacerse privado poniendo un guión bajo antes del nombre, veamos este ejemplo.

```
# Declara clase.
class Cola:
    #Construtor con lista "escondido" privada.
    def __init__(self, contenido):
        self._escondido = list(contenido)

# Método que inserta valor en la cola.
    def push(self, valor):
        self._escondido.insert(0, valor)

# Declara objeto Cola.
cola = Cola([1, 2, 3])
print(cola)
cola.push(0)
print(cola)
print(cola.escondido)

# Imprime:
# Queue[1, 2, 3].
# Queue[0, 1, 2, 3].
# [0, 1, 2, 3].
```

Al no existir la encapsulación en Python, el guión bajo en métodos es solo una convención, ya que si podemos acceder a estos métodos por fuera de la clase. Si seguimos buscando utilizar miembros privados, agregamos un guión más al inicio del nombre del miembro, con esto logramos un efecto mayor de privacidad, pero aún podemos acceder a este método por fuera, solamente tenemos que llamarlo de una forma particular.

```
class Ejemplo:
    # Declara atributo privado.
    __huevo = 7
    # Método que imprime el valor del atributo privado.
    def imprimir_Huevo(self)
```

```
        print(self.__huevo)

# Declara objeto Ejemplo.
e = Ejemplo()
# Llama al método de la clase.
s.imprimir_Huevo()
# Imprime el valor del atributo privado.
print(e._Ejemplo__huevo)
```

Es entonces que, para acceder al miembro privado fuera de la clase, debemos escribir el nombre del objeto, un punto y guión bajo, el nombre de la clase y enseguida el nombre del atributo:

```
e._Ejemplo__ Variable
```

12.8 Métodos estáticos y de clase

Los **métodos de clase** son como una clase dentro de otra clase, o un tipo especial de método dentro de clase que no puede ser llamado por medio de un objeto de clase, sino que son llamados directamente desde el nombre de la clase, como las clases no pueden ser utilizadas directamente sin un objeto de por medio, para lograr utilizar los métodos de clases se debe utilizar una variable de por medio, que termina actuando como un objeto del método de clase.

Para diferenciar estos métodos de los otros, utilizamos el decorador **classmethod**, como en el siguiente ejemplo.

```
# Declara clase.
class Rectangulo:
    # Constructor
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    # Método que calcula área del rectángulo.
    def area(self):
        return self.ancho * self.alto

    # Método de clase que declara un cuadrado.
    @classmethod
    # Así como self, el método de clase debe tener su palabra reservada,
    # en este caso, utilizamos la palabra "cls".
    def cuadrado(cls, lado)
        return cuadrado(lado, lado)

# Variable que recibe el método de la clase Rectangulo.
sqr = Rectangulo.cuadrado(5)
# Llama al método que calcula el área del rectángulo, pero
# aplicado a un cuadrado.
print(sqr.area())
```

Los métodos de clase deben tener su propia palabra reservada, así como las clases tienen la palabra **self**, estas dos palabras pueden tener el nombre que desees, sin embargo, **self** y

cls son conocidas en muchos sitios y utilizadas en muchas partes, por lo que es recomendable utilizarlas para no confundir a quien lea tu código.

Los **métodos estáticos** son iguales a los métodos regulares de clases o fuera de clases, pero no requieren de la palabra reservada *self* dentro de sus parámetros, y puede llamarse a estos métodos por medio de la clase sin necesidad de un objeto, utilizamos el decorador **staticmethod**, veamos el ejemplo.

```
# Declara clase.
class Shape:
    # Constructor.
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

# Método estático.
@staticmethod
    def area(ancho, alto):
        return ancho * alto

# Variables reciben valores
w = int(input())
h = int(input())
# Imprime el área de la figura, usando el metodo estático sin necesidad
# de crear un objeto de la clase.
print(Shape.area(w, h))
```

12.9 Propiedades

El decorador **property** ayuda a poder acceder a los métodos de una clase como si fueran atributos, o acceder a atributos de la clase por medio de un método, el uso más común de este decorador es volver atributos a solo-lectura. Otro tipo de decorador propiedad son los **Setters** y **Getters**, el primero asigna un valor a un atributo, y el segundo obtiene dicho valor, por ejemplo.

```
# Declara clase.
class Trabajador:
    # Constructor
    def __init__(self, primero, segundo):
        self.primero = primero
        self.segundo = segundo

    # Método que define un correo electrónico, es tratado como atributo.
    @property
    def correo(self):
        return "{}.{}@email.com".format(self.primero, self.segundo)

    # Método que define el nombre completo, es tratado como atributo.
    @property
    def nombre_completo(self):
        return "{} {}".format(self.primero, self.segundo)

    # Método con decorador Setter que define el nombre completo
```

```
        # del trabajador.
        @nombre_completo.setter
        def nombre_completo(self, nombre):
            primero, ultimo = nombre.split(" ")
            self.primero = primero
            self.segundo = segundo

    # Declara objeto Trabajador.
    emp1 = Trabajador("Luis", "Miguel")

    # Asigna nombre completo al objeto, cambiando los atributos de dicho objeto.
    emp1 = nombre_completo("Ernesto Casas")

    # Imprime el primer nombre.
    print(emp1.primero)
    # Imprime el segundo nombre.
    print(emp1.segundo)
    # Llama al metodo nombre_completo, pero al ser una propiedad, no requiere de parentesis.
    print(emp1.nombre_completo)
```

Vemos que se crea un objeto de la clase Trabajador con ciertos valores, después se cambia el nombre del trabajador por medio de un Setter, logrando cambiar los valores de los atributos del objeto, el método *nombre_ completo* regresa los atributos *primero* y *segundo* como una sola cadena, con el decorador property este método es tratado como un atributo más, por lo que descartamos el uso de paréntesis al final del nombre, lo mismo aplica con el método *correo*.

13 Excepciones

El manejo de excepciones se logra utilizando el bloque **try:-except:**, donde dentro del bloque try se escribe el código que puede fallar, y el bloque exception contiene el código en caso de que el error se de. Algunas de las excepciones por defecto de Python son:

- **ImportError**: cuando una importación de biblioteca falla.
- **IndexError**: cuando se intenta acceder a un índice inexistente de una colección de datos.
- **NameError**: cuando se trata de usar una variable desconocida.
- **SyntaxError**: cuando el código no se puede analizar correctamente.
- **TypeError**: cuando un tipo de dato no corresponde con la función o variable.
- **ValueError**: cuando un valor no corresponde con el tipo de dato de la función o variable.
- En caso de que no se asigne un tipo de excepción a bloque except, quiere decir que except va a atrapar todos los tipos de excepciones existentes.

Un bloque except puede contener más de una excepción, contenida en paréntesis, o tener múltiples bloques except para cada tipo de excepción.

```
# Bloque try-except.
try:
    variable = 10
    # Suma de un entero y una cadena.
    print(variable + "hello")
    print(variable / 2)
# Excepción por si se divide entre 0.
except ZeroDivisionError:
    print("Divided by zero")
# Excepción por si un valor o tipo no coincide.
except (ValueError, TypeError):
    print("Error occurred")
```

13.1 finally y else

El bloque **finally** se ejecuta después del bloque try, sin importar si el bloque except se ejecutó o no, además, podemos utilizar el bloque *else*, este se ejecuta únicamente si el bloque try se ejecutó con normalidad, es decir, sin error alguno.

```
# Bloque try-except.
try:
    # Variable que recibe un entero.
    val = int(input())
    print(val)
# Excepción por si el tipo ingresado no es válido.
except TypeError:
    print("Ese no es un numero")
# Si no hubo error, se despliega mensaje.
else:
    print("Todo salio bien")
# Bloque con mensaje final.
finally:
    print("Hemos terminado")
```

13.2 Lanzando excepciones

Podemos lanzar o llamar excepciones si alguna condición o se cumple, con la palabra reservada **raise**, seguido del nombre de la excepción a llamar.

```
# Asignación de valor a variable.
tweet = input()

# Bloque try-except-else.
try:
    #Si el largo de la variable es mayor a 42, lanza excepción.
    if len(tweet) > 42:
        raise ValueError
except:
    print("Error")
```

```
    else:
        print("Posted")
```

14 Trabajando con archivos

Para abrir un archivo en Python, utilizamos la función **open**, y dentro de sus paréntesis, va la ruta del archivo (si el archivo está en la carpeta actual donde se ubica la solución, solamente es necesario escribir el nombre, si está en otra carpeta, se indica la ruta completa del archivo).

```
miArchivo = open("prueba.txt")
```

Podemos especificarle el **como** se trabajará el archivo, es decir, si va a ser sobreescrito, solo leído, o escribir un archivo binario.

```
# Modo Escritura.
open("prueba.txt", "w")
# Modo Lectura.
open("prueba.txt", "r")
# Modo Escritura Binaria.
open("prueba.txt", "wb")
# Modo Lectura Binaria.
open("prueba.txt", "rb")
```

Una vez abrimos y trabajamos el archivo, debemos cerrarlo, esto lo conseguimos con la función **close()**.

```
miArchivo = open("prueba.txt")
miArchivo.close()
```

Es una buena practica siempre abrir, trabajar el archivo, y cerrarlo, para no desperdiciar recursos, podemos evitar esto utilizando un bloque try-finally o la instrucción **with**, donde se crea una variable temporal y se utiliza únicamente dentro del bloque identado, después de eso, la variable es borrada.

```
# Opción 1:
# Declara variable f con instrucción with abriendo un archivo.
with open("archivo.txt") as f:
    # Imprime el contenido del archivo.
    print(f.read())

# Se sale del bloque identado, la variable f es borrada.
# Opción 2:
# Bloque try-finally para abrir y cerrar un archivo.
try:
    # Variable que declara un archivo.
    f = open("archivo.txt")
    # Imprime el contenido del archivo.
    print(f.read())
finally:
    # Después del bloque try, se ejecuta finally que cierra el archivo.
    f.close()
```

14.1 Leyendo archivos

Con la función **read()** del archivo que hemos abierto podemos desplegar el contenido del mismo.

```
# Declara variable con un archivo.
archivo = open("libros.txt")
# Asigna a una variable el contenido del archivo.
aux = archivo.read()
# Imprime el contenido del archivo.
print(aux)
# Cierra el archivo.
archivo.close()
```

En caso de que queramos leer únicamente cierta cantidad de caracteres del archivo, podemos pasarle como parámetro a la función *read()* la cantidad de caracteres deseados, la función regresará la cantidad deseada.

```
# Declara variable con un archivo.
archivo = open("libros.txt")
# Asigna a una variable cierta cantidad de caracteres del archivo.
aux = archivo.read(7)
aux2 = archivo.read(5)
# Imprime 7 y 5 caracteres del archivo.
print(aux)
print(aux2)
# Cierra el archivo.
archivo.close()
```

Otra forma de imprimir el contenido de un archivo es por medio de la función **readlines()**, el cual regresa una lista con las línea por línea el contenido de archivo, debemos utilizar un ciclo *for* para poder imprimir línea a línea. A su vez, si no queremos utilizar las dos funciones anteriormente mencionadas, podemos simplemente juntar un ciclo for y el puro nombre del archivo, como vemos enseguida.

```
# Declara variable con un archivo.
file = open("filename.txt", "r")
# Ciclo for que lee línea a línea el contenido de un archivo.
for line in file.readlines():
    #Imprime la linea.
    print(line)
# Cierra el archivo.
file.close()

# Declara variable con un archivo.
file = open("filename.txt", "r")
# Ciclo for que imprime el contenido del archivo.
for line in file:
    # Imprime la línea.
    print(line)
# Cierra el archivo.
file.close()
```

La diferencia entre *read()*, *readlines()* y *in file*, es que estos dos últimos agregan un salto de línea adicional al que da *print()* cuando despliega algo.

14.2 Escribiendo en archivos

Con la función **write()** escribimos contenido en el archivo, si el **modo** del archivo es *write* y el archivo ya existe, el archivo es sobreescrito, si no existe, Python lo crea en la ruta establecida.

Recordemos que el *modo a* permite agregar contenido a un archivo ya existente, así que podemos abrir un archivo con dicho modo y Python te posicionará en la última línea del archivo para que puedas agregar nuevo contenido.

```
# Declara variable con un archivo en modo write.
file1 = open("filename.txt", "w")
# Sobre escribe el contenido del archivo si ya existe, o lo crea y escribe el contenido en
    el.
file.write("Borron y cuenta nueva")
# Cierra el archivo.
file.close()

# Declara variable con un archivo en modo append.
file2 = open("filename.txt", "r")
# Sobre escribe el contenido del archivo en la última línea.
file.write("\nAgregado al final")
# Cierra el archivo.
file.close()
```

La función write regresa el número de caracteres escritos en el archivo, si es que se pudieron escribir correctamente.

```
# Declara variable con un archivo en modo write.
file1 = open("filename.txt", "w")
# Asigna a una variable el número de caracteres escritos en el archivo.
num_caracteres = file1.write("abcde")
# Despliega 5.
print(num_caracteres)
# Cierra el archivo.
file.close()
```

14.3 Comprobar la existencia de un archivo

Para saber si existe un archivo o directorio dentro de una carpeta en Windows con Python requerimos importar el módulo **os** para llamadas al sistema:

```
import os
```

Sea cual sea la situación o problema que estemos programando o resolviendo, podemos utilizar la siguiente función que comprueba la existencia de un archivo o directorio en nuestro sistema de archivos:

```
import os

def verificar_archivo():
    if os.path.exist(ruta):
        # Instrucciones si la ruta o archivo existe.
    else:
```

```
        # Instrucciones si la ruta o archivo no existe.
```

14.4 Obtener una lista con todos los archivos de una carpeta

La siguiente función resulta útil si queremos seleccionar un archivo de una carpeta, pero primero queremos ver todo el contenido de la carpeta donde se encuentra almacenado:

```
import os

def lista_archivos_en_directorios():
    return [arch.name for arch in os.scandir(ruta) if arch.is_file()]
```

Nota: esta función también requiere del módulo **os** para funcionar.

14.5 Eliminar un archivo del sistema

Requiere importar la clase **remove** del módulo **os** para funcionar:

```
from os import remove
```

Entonces, si en la carpeta "trabajos" tengo tres archivos: "final.docx", "ahora si final.docx" y "final final final en serio.docx" y quiero borrar el último, basta con utilizar la siguiente instrucción para eliminarlo:

```
from os import remove

remove("trabajos/final final final en serio.docx")
```

Tenga en cuenta cómo funcionan las rutas en el sistema operativo donde esté programando.