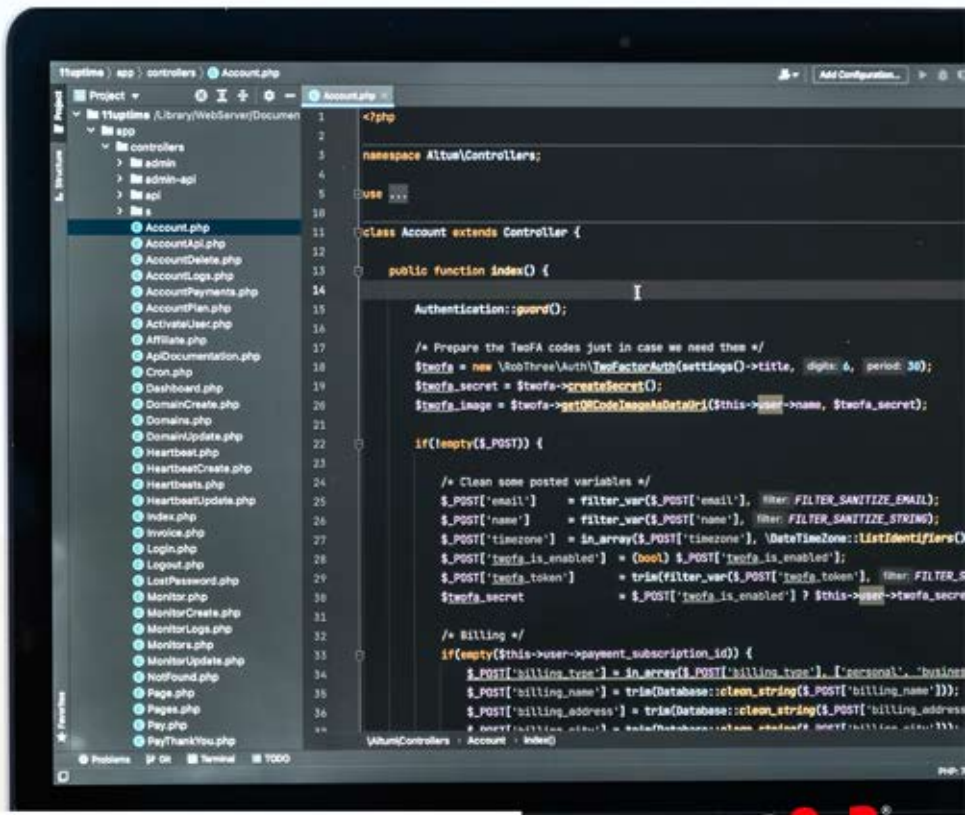
Ingeniería de sistemas

APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON

Una mirada hacia la inteligencia artificial



Carlos M. Pineda Pertuz

ediJ

Informática

APRENDIZAJE AUTOMÁTICO Y PROFUNDO EN PYTHON

Una mirada hacia la inteligencia artificial

Carlos M. Pineda Pertuz

edU®
Conocimiento a su alcance

Pineda Pertuz, Carlos M.

Aprendizaje automático y profundo en Python/ Carlos M. Pineda Pertuz -- 1a. edición. Bogotá: Ediciones de la U, 2021
342 p. ; 24 cm.
ISBN 978-958-792-316-2 e-ISBN 978-958-792-315-5
1. Informática 2. Programación 3. Aprendizaje automático 4. Python I. Tít.
621.39 cd 24 ed.

Área: Informática
Primera edición: Bogotá, Colombia, octubre de 2021
ISBN. 978-958-792-316-2

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 No. 70 D - 34, Pbx. (571) 7217756

Impreso y hecho en Colombia
Printed and made in Colombia

Agradecimientos

Por su cariño y apoyo incondicional:

A mis queridos padres Demetrio y Ruby
A mi esposa Katerine, a mi hija Luciana
y a mis hermanas: Ruby, Diana y Maira

Contenido

Prólogo

He decidido crear esta obra como una guía para que estudiantes de ingeniería o cualquier persona interesada en aprender sobre aprendizaje automático y profundo pueda adquirir las bases necesarias sobre este tema tan apasionante. Será un camino lleno de retos en donde exploraremos las técnicas y algoritmos más representativos, lo cual le permitirá al lector forjar conocimientos sólidos en la materia y ser capaz de crear sus propios modelos de aprendizaje para resolver una amplia gama de problemas presentes en el mundo real.

Este libro comenzará dando una introducción al lenguaje Python, siendo esta la herramienta de programación escogida para desarrollar los ejemplos, luego se irán explicando los conceptos teóricos y consecuentemente se abordará de manera práctica el funcionamiento de los diferentes métodos y técnicas usadas en el campo del aprendizaje automático. El lenguaje de programación Python es de los que más ha ganado fuerza en este ámbito, fundamentalmente por su facilidad y por la gran cantidad de librerías que pone a disposición de los desarrolladores y de quienes trabajan en el mundo de la ciencia de datos.

Aprendizaje automático y profundo en Python le será un libro muy ameno con suficientes ejemplos y ejercicios para poner en práctica, y que le permitirá reforzar su aprendizaje mediante el estudio de código fuente que el autor pondrá a su alcance mediante cuadernos de Jupyter Notebook. Además, cada tema será explicado de manera sencilla y ejemplificada de manera que el lector pueda ir probando cada aspecto teórico con las herramientas y el software sugeridos por el autor. Esto y más hacen que esta obra se diferencie de otras de su tipo y sea suficientemente útil para toda aquella persona que quiera sumergirse en el mundo del aprendizaje automático de una manera organizada y fácil, pero sin dejar a un lado el fundamento teórico que está detrás de los métodos usados en esta gran disciplina.

Como complemento web al libro en www.edicionesdelau.com encontrará la carpeta Fuentes que contiene el desarrollo de ejemplos y ejercicios, para su mejor comprensión.

Introducción

Esta obra pretende ser una herramienta de apoyo y de consulta para toda aquella persona interesada en dominar los fundamentos del aprendizaje automático y profundo, a tal punto que le permita aprender lo necesario para desarrollar sus propios modelos de aprendizaje aptos para realizar predicciones con base en los datos, para ello el autor hará uso en la mayoría de los casos de explicaciones teóricas y prácticas, que permitan al lector afianzar sus ideas y fortalecer su aprendizaje. El libro arranca exponiendo algunos temas básicos y, poco a poco, irá aportando otros temas un poco más complejos necesarios para forjar un conocimiento integral y mucho más sólido sobre aprendizaje automático con la convicción de dar al lector la orientación necesaria para que sea capaz de abordar sus propios proyectos basados en las técnicas descritas.

Para completar cabalmente la lectura de este libro se requieren de unos conocimientos mínimos en cálculo, estadística y de programación, debido a que cada tema, generalmente, se aborda inicialmente con una explicación de diversos conceptos, tomando para ello aspectos del cálculo, la estadística y el algebra lineal. Posteriormente, los algoritmos se implementarán en su mayoría desde cero y en otros con el apoyo de varias librerías o APIs para facilitar su desarrollo dentro del entorno de programación.

Aprendizaje automático y profundo tiene una estructura en capítulos que inicia con explicaciones sobre el lenguaje Python, para luego abarcar los algoritmos más destacados dentro del aprendizaje de máquina. El libro se encuentra dividido en dos partes: la primera enfocada en el *machine learning* y sus diferentes algoritmos de regresión y clasificación, *clustering*, entre otros. La segunda parte comprende varias técnicas de *deep learning* donde estudiaremos diferentes arquitecturas de redes neuronales como: redes densamente conectadas, redes convolucionales y redes recurrentes.

Espero que esta obra sea del agrado y el disfrute del lector, así como lo fue para quien escribió estas palabras, que sin duda les puede decir que los temas de este libro son de los que más pasión le han conferido a lo largo de su carrera. Bienvenidos a este apasionante mundo.

CAPÍTULO 1

Conceptos básicos de programación en Python 3.9

Temas

En este capítulo se abordarán algunos fundamentos de la programación en Python que resultarán de utilidad para que aquellos lectores que nunca han manejado este lenguaje puedan entender más fácilmente los algoritmos que se ofrecen en el libro en los posteriores capítulos.

Python es un lenguaje de programación de código abierto e interpretado, esto último significa que cada línea de código que escribamos es leída por un intérprete que las va ejecutando. Esta condición le otorga más rapidez en la ejecución de los programas, puesto que estos no tienen que compilarse previamente ahorrándose el tiempo de compilación. Además, Python es un lenguaje multiplataforma, lo cual hace que cualquier programa que hagamos pueda correr sin ningún problema en diferentes plataformas o sistemas operativos. Así mismo este software de desarrollo dispone de una sintaxis muy simple lo que hace que su curva de aprendizaje no sea tan pronunciada en comparación con sus similares como Java. Y por si fuera poco cuenta con una basta cantidad de librerías actualizadas permanentemente por su gran comunidad de desarrolladores, las cuales permiten programar diver-

sas funcionalidades con un minimo esfuerzo. Estas caracteristicas y más hacen de Python el lenguaje de programación preferido por muchos programadores dedicados al desarrollo de sistemas basados en aprendizaje automático.

1.1 Entorno de desarrollo y primeros pasos

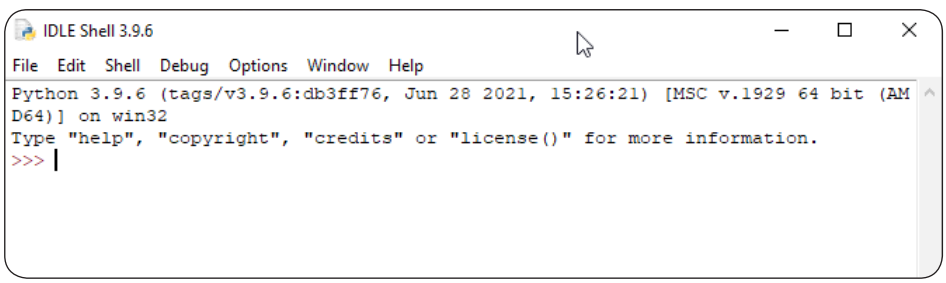
Para comenzar a programar en Python se requiere descargar el instalador de la página oficial en internet: https://www.python.org/

Dentro de esta página se puede escoger para que sistema operativo lo vamos a instalar y la versión del lenguaje Python. Para esto último se puede optar por la última versión, que al momento de escribir este libro es la 3.9.6.

La instalación es como la de cualquier otro software, donde a menos que se desee establecer una configuración personalizada se resumiría solo a pulsar el botón "Siguiente" para avanzar a través de las ventanas del asistente de instalación.

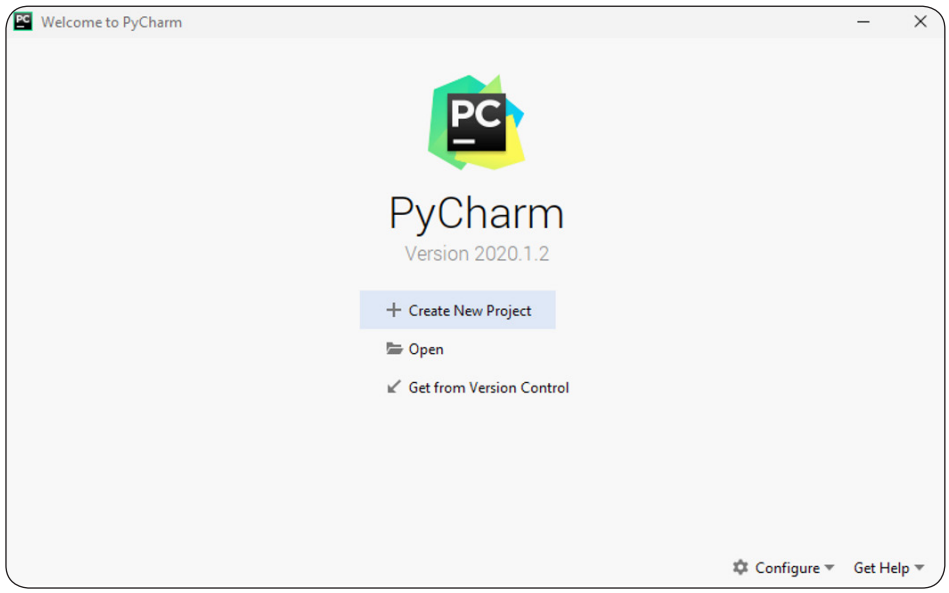
Una vez instalado, ya podemos probar la herramienta y escribir nuestros primeros programas, para ello necesitamos un editor donde coloquemos las líneas de código, en ese sentido Python pone en nuestras manos un programa llamado IDLE que viene junto con la instalación realizada en el punto anterior, y aunque no es el más sofisticado para desarrollar aplicaciones, si puede ser de utilidad para realizar algunas pruebas o ejemplos de poca envergadura.

Al abrir IDLE observamos una interfaz como se muestra en la siguiente imagen:



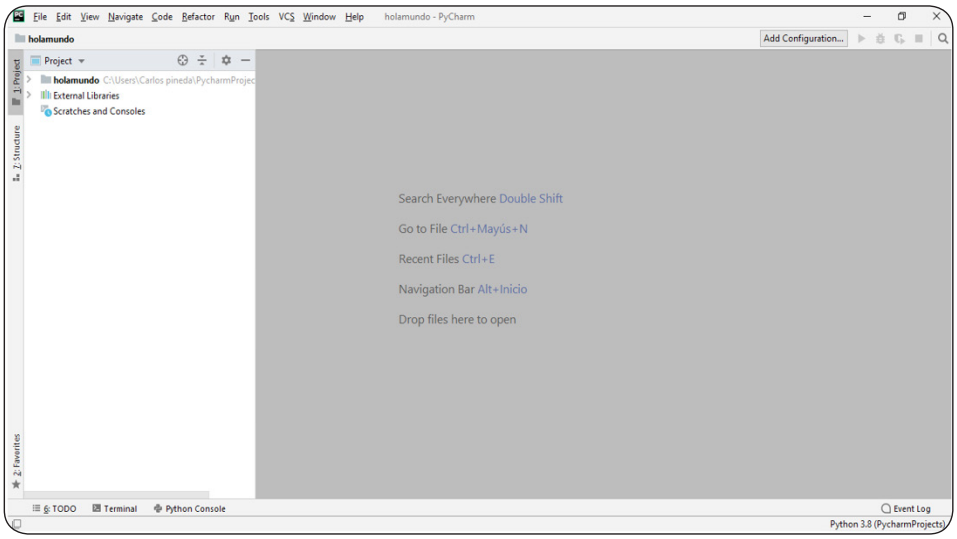
Sin embargo, es de anotar que existen IDEs o entornos de desarrollo integrados: como PyCharm, VIM, Wing, entre otros que ofrecen un entorno más completo y mucho más adecuado para el desarrollo de aplicaciones. En nuestro caso usaremos PyCharm por ser a consideración del autor muy liviano, intuitivo y con una gran cantidad de opciones en su versión community (open source). Este software es desarrollado por la empresa Jetbrains y lo podemos encontrar en la siguiente dirección de internet: https://www.jetbrains.com/es-es/pycharm/

La siguiente figura muestra la interfaz principal de PyCharm, donde básicamente nos indica que podemos crear un nuevo proyecto (Create New Project), abrir uno existente (Open) u obtener un proyecto de un repositorio de control de versiones (Get from Version Control).



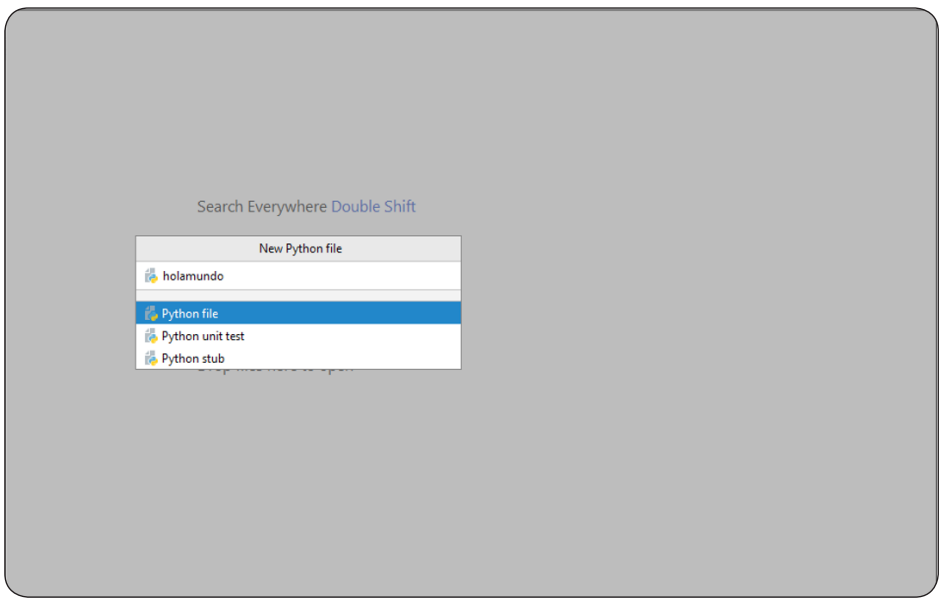
Al escoger la opción "Create New Project" seguidamente especificamos un nombre para el proyecto, por ejemplo, llamémosle "holamundo" y una ruta para el mismo, pero podemos dejar esta última y las demás opciones por defecto. Tenga presente el lector que solo haremos una breve introducción a la herramienta, así que se omitirán los detalles referentes a las diferentes configuraciones.

En la siguiente imagen vemos que se creó el proyecto llamado "HolaMundo".



Ahora dentro del proyecto creado procedemos a adicionar un primer programa de prueba. Para hacer esto, hacemos clic derecho sobre la carpeta "holamundo" y en el menú contextual que aparece escogemos la opción new Python File, mediante la cual creamos un archivo cuyo nombre en nuestro caso será nuevamente "holamundo", sin necesidad de preocuparnos por la extensión ya que el IDE le agrega la extensión .py automáticamente, siendo esta la utilizada para los scripts o programas escritos en Python.

Luego presionamos la tecla ENTER y finalmente se habrá creado el archivo deseado.



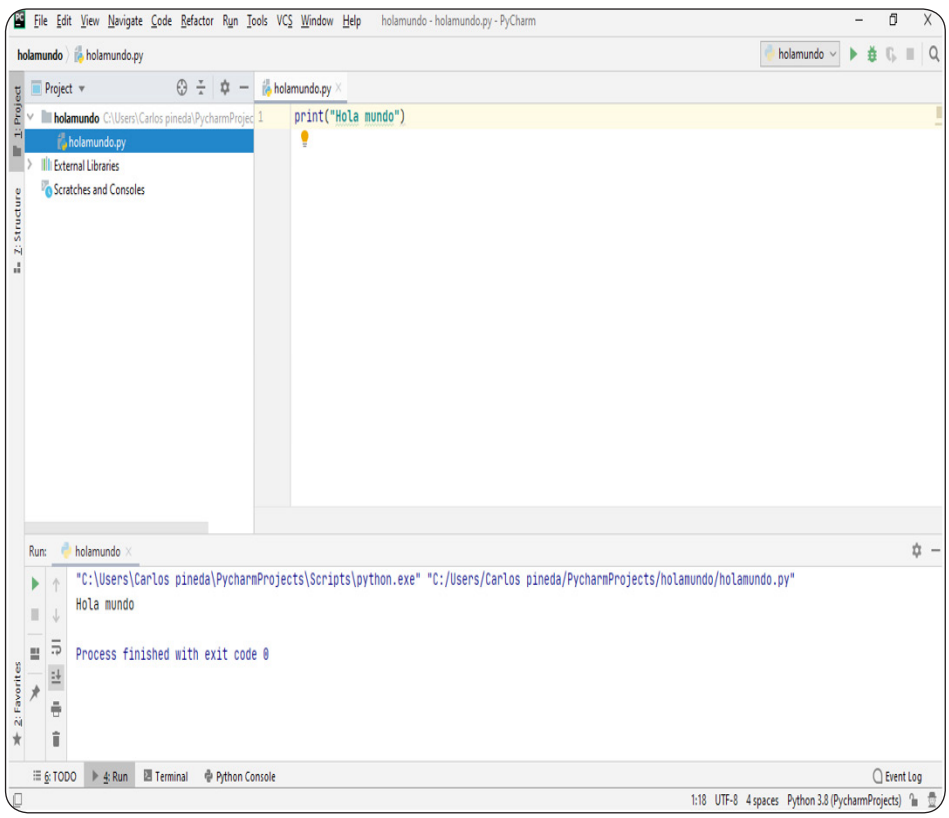
Una vez realizado los pasos anteriores y teniendo desplegado nuestro archivo "holamundo.py" en el editor de código de PyCharm, escribimos la siguiente línea:

```
print("Hola mundo")
```

Esta simple instrucción imprime un mensaje en la consola, en ella actúa la función print que se encarga de mostrar en la salida estándar lo que especifiquemos dentro de los paréntesis, siendo en esta oportunidad la cadena "Hola Mundo" lo que se mostraría.

Para ejecutar el programa simplemente nos dirigimos al menú run y de allí seleccionamos la opción run. Si es la primera vez que ejecutamos nuestro programa se nos mostrará una ventana donde nos pide seleccionar el archivo (holamundo.py). Si no hay errores de sintaxis la ejecución nos mostrará el texto "Hola mundo" y un mensaje al final, que indica que el proceso finalizó con éxito:

Process finished with exit code 0



Hasta aquí hemos creado nuestro primer "hola mundo" en Python y antes de entrar a revisar otros elementos de la programación en este lenguaje es necesario mencionar la importancia que para este significa la indentación. La indentación consiste básicamente en aplicar sangrías a las instrucciones según el nivel que tengan dentro de un bloque de código. Las declaraciones que requieren un primer nivel de sangría terminan con dos puntos como en el caso de las definiciones de clase, funciones o sentencias control de flujo y las que están dentro del bloque se les aumenta la sangría y se colocan alineadas. Por ejemplo, el fragmento de código siguiente tiene un condicional *if* en un primer nivel y más abajo aparecen dos sentencias con la función print() con sangría o desplazadas hacia la derecha demostrando que están en un nivel inferior y por tanto dependen de dicho condicional.

```
if a > 0:
   print("El número es positivo")
   print("Se trata del número " , a)
```

Por supuesto dentro de un script más grande puede haber muchos más niveles o mucha mayor anidación, pero las sentencias que hacen parte de un mismo nivel como en el ejemplo mostrado deben estar alineadas, si esto no se tiene en cuenta se corre el riesgo de recibir el error:

IndentationError: unindent does not match any outer indentation level

1.2 Variables, Tipos de datos y operadores

Variables

Una variable es un espacio en memoria destinado para el almacenamiento de un valor el cual va a ser usado en un programa. Siendo posible modificar el dato que en algún momento almacena por otro en cualquier otra parte del programa, de esta característica se deriva el nombre de variable.

En cuanto a la declaración de variables, en Python no es necesario especificar los tipos de datos ya que estos son definidos y asumidos por el lenguaje a partir del valor que se le asigne a la variable. Por esta razón, decimos que es un lenguaje de tipado dinámico. Este modo de tratar los tipos de datos permite que a una variable *n* que se le asigne por ejemplo el 10, siendo en ese momento de tipo int (entero), se le pueda posteriormente asignar una cadena como "Hola" pasando a ser de tipo string (cadena de caracteres).

Tipos de datos

Los tipos de datos manejados en Python son básicamente: números (enteros, reales y complejos), strings(cadenas) y booleanos. Los enteros o *int* manejan un rango infinito de números incluyendo los números positivos, negativos y el cero. Ejemplos: -5, 0, 10, por su parte los reales o *float* también manejan un rango infinito de números y son aquellos que tienen decimales. Ejemplos: -5.0, 3.5, 0.9.

Algunas sentencias de declaración de variables con todos estos tipos de datos serían:

```
n1 = -5
n2 = 0
n3 = 10
n4 = -5
n5 = 3.5
n6 = 0.9
n1 = 0.5
```

Note el lector que a la variable n1 que inicialmente tenía el valor -5, en la última instrucción se le ha asignado el número 0.5, lo cual permite cambiar el tipo de dato de la variable de entero (int) a real (float), lo cual reafirma el concepto de tipado dinámico explicado anteriormente.

Después de ejecutar el código anterior el lector puede emplear la función type(variable) para conocer el tipo de dato de la variable pasada como parámetro:

```
print(type(n1))
```

En cuyo caso el resultado obtenido será <class 'float'>, lo cual quiere decir que el tipo de dato de la variable *n1* es número de coma flotante.

Por otro lado, existe otro tipo de dato conocido como *boolean,* que solo puede tomar dos valores posibles: True (verdadero) o False (falso).

```
es_adulto = False
esta_trabajando = True
```

Y por último se encuentran los *str* o cadenas de caracteres, que son conjuntos de caracteres, es decir, combinaciones de números, letras y caracteres especiales, como por ejemplo la dirección de una residencia:

```
direccion = "cll 27a # 9i 10"
```

Como consideración importante, se deben colocar comillas simples o dobles cuando se trabaja con cadenas de caracteres. De igual manera es importante resaltar la concatenación, operación que consiste en unir dos o más cadenas por medio del operador (+). En este nuevo ejemplo, unimos la palabra "Hola", más una cadena con un espacio (" ") y la cadena "Mundo":

```
cad1 = "Hola"
cad2 = "Mundo"
cad3 = cad1 + " " + cad2
print(cad3)
```

Otro elemento importante en programación son los comentarios, usados principalmente para hacer explicaciones en el código lo cual facilita enormemente las labores de mantenimiento de los programas, permitiendo que otras personas pueden entender lo que hace cada instrucción con una mayor facilidad. En Python los comentarios de una línea se crean anteponiendo el símbolo # antes del escrito, mientras que en los de varias líneas se colocan tres comillas dobles al principio y al final del texto. Miremos como se pueden aplicar con un ejemplo:

```
#Esto es un comentario de una linea

""" Esto es un comentario de
   varias líneas
"""
```

Operadores

Los operadores son un conjunto de símbolos usados para realizar operaciones sobre números, variables, etc. La ejecución de dichas operaciones trae consigo la obtención de un valor determinado.

En Python se usa el operador igual (=) para realizar asignaciones, esto es, colocar datos o expresiones dentro de variables. Además, entre los operadores más usados encontramos principalmente: aritméticos, relacionales y lógicos.

Operadores aritméticos

Son aquellos que operan sobre números, llevando a cabo operaciones aritméticas básicas y devuelven un valor numérico. El resultado exacto depende de los tipos de operandos involucrados.

Operador	Descripción	Ejemplo (a = 5)
+	Suma	b = a + 5
-	Resta	b = a – 5
*	Multiplicación	b = a * 2
/	División real	c = b / a
//	División entera	c = b // a
%	Residuo	c = b % 2
**	Exponenciación	c = b**2

Operadores relacionales

Realizan comparaciones entre datos que sean de tipo numérico, carácter y boolean, devolviendo siempre un resultado booleano.

Operador	Descripción
= =	Igual que
!=	Diferente que
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Operadores lógicos

Permiten conectar expresiones a las cuales se les aplican las tablas de verdad para obtener un valor booleano (True o False).

Operador	Descripción
not	Negación
and (y)	Y lógico
or (o)	O lógico

Un aspecto a considerar cuando se está programando es que las expresiones son evaluadas de la misma manera como se evalúan las expresiones aritméticas. Esto quiere decir, que se pueden utilizar las mismas reglas matemáticas para evaluar una expresión en Python. Las expresiones se evaluarán de izquierda a derecha teniendo en cuenta la prioridad de los operadores. La siguiente tabla muestra el orden de evaluación de algunos operadores, de mayor a menor nivel de prioridad.

Operador	Descripción
()	Paréntesis
**	Exponenciación
*	Multiplicación
/	División
Suma	+
Resta	-

1.3 Estructuras de datos: Tuplas, listas y diccionarios

Tuplas

Son estructuras que permiten almacenar deferentes tipos de datos, además se consideran inmutables, es decir, no permiten modificar o agregar elementos una vez se haya definido la tupla. Las tuplas usan paréntesis para especificar los datos. Ejemplo:

```
tupla = (1, "hola", 3.5)
# mostramos todos los datos de la tupla
print(tupla)
# mostramos el primer elemento
print(tupla[0])
# Esta operación no está permitida
tupla[0] = 2
```

Del anterior ejemplo, se tiene que, para mostrar un elemento en concreto de la tupla se utilizan corchetes y dentro de estos un índice que denota la posición del dato. Los índices en Python comienzan desde 0. Por ejemplo, si deseamos mostrar el primer elemento el índice es 0, si deseamos mostrar el segundo elemento el índice sería 1 y así sucesivamente. Además, observe como la última asignación genera error por lo ya comentado acerca de la inmutabilidad de las tuplas.

Listas

Así como las tuplas, las listas se usan para almacenar diferentes tipos de datos, pero con la diferencia que, a estas, si se les pueden modificar sus elementos o agregar elementos nuevos. Se definen con corchetes y al igual que con las tuplas los elementos se acceden mediante un índice entero que inicia desde cero. Veamos un ejemplo donde creamos dos listas y mostramos su contenido:

```
# creamos un lista vacia
lista1 = []
# adicionamos un elemento tipo String
lista1.append("Aprendizaje automático")
print(lista1)
# Creamos otra lista con algunos elementos
lista2 = ["Python","Java","C++"]
# mostramos el primer elemento (Python)
print(lista2[0])
```

Observemos en el código el uso de la función append() para adicionar un elemento nuevo a la lista *list1*, también se puede ver como para acceder a un valor de una lista se usa un índice entero dentro de corchetes. Sin embargo, Python ofrece una manera fácil y a la vez muy útil de acceder a varios elementos de una lista, mediante el operador (:), observemos cómo los podemos usar:

```
edades = [18, 22, 33, 15, 25, 26, 35, 40, 13, 20]
# muestra todas las edades
print("Todas: " , edades[:])
# Muestra de la 1 hasta el último
print("1-ultimo: " , edades[1:])
# Muestra de la 1 a la 3 sin incluir la 3
print("1-3: " , edades[1:3])
# Muestra del 0 hasta el penultimo
print("0-penultimo: " , edades[0:-1])
```

Por otro lado, las listas cuentan con una serie de funciones útiles que el lector debe conocer, como, por ejemplo:

- len(lista): Permite obtener la longitud de una lista pasada como parámetro.
- append(valor): Adiciona un nuevo valor a la lista.
- insert(posición, valor): Permite agregar el elemento *valor* en la posición *posición*.

- del(elemento): Permite eliminar un elemento de la lista pasado como parámetro.
- remove(elemento): Al igual que la anterior función, elimina de la lista el elemento pasado como parámetro.
- sort([reverse=True]): Permite ordenar una lista en orden ascendente, este es el comportamiento por defecto, si se indica el parámetro reverse=True el ordenamiento se realizará descendentemente.

Probemos el siguiente ejemplo para observar el trabajo de las anteriores funciones:

```
# mostramos la longitud de lista2
print("Longitud:" , len(lista2))
# Adicionamos un elemento en la posición 3
lista2.insert(3, "PHP")
print(lista2)
# Elimamos el elemento C++
del(lista2[2])
# Eliminamos e elemento PHP
lista2.remove("PHP")
print(lista2)
# Ordenamos la lista en orden ascendente
lista2.sort()
print(lista2)
```

```
Longitud: 3
['Python', 'Java', 'C++', 'PHP']
['Python', 'Java']
['Java', 'Python']
```

Diccionarios

Son una colección de datos mutable muy similar a las listas en cuanto a su función de almacenamiento, pero con una diferencia en el modo de acceso a los elementos, realizado en este caso por medio del uso de una clave (key) la cual debe expresarse entre corchetes y puede ser de tipo entero o cadena. A los diccionarios también se les conoce como arrays asociativos y se construyen empleando conjuntos de pares clave-valor.

```
registro = {"ID": 12345 , "Nombre": "Carlos" , "Apellido": "Pineda"}
# Imprimimos el diccionario
print(registro)
# Accedemos al valor "Carlos" a través de la clave "Nombre"
print(registro["Nombre"])
# Agregamos un nuevo elemento
registro["Profesion"] = "Profesor"
print(registro)
```

```
{'ID': 12345, 'Nombre': 'Carlos', 'Apellido': 'Pineda'}
Carlos
{'ID': 12345, 'Nombre': 'Carlos', 'Apellido': 'Pineda', 'Profesion': 'Profesor'}
```

El anterior diccionario contiene tres claves "ID", "Nombre" y "Apellido" y tres valores 12345, "Carlos" y "Pineda". El acceso a un valor se hace escribiendo dentro del corchete la clave asociada al mismo.

1.4 Estructuras selectivas

Las estructuras de selección como el nombre lo indica permite escoger o seleccionar que parte del código de un programa deberá ejecutarse. Esto se logra a partir de la evaluación de una condición. Esta última es una expresión de tipo lógica que al evaluarse tomará como posibles valores **True** o **False**. Por esto también se conocen como estructuras de control de flujo, porque permiten controlar o modificar el flujo (secuencia) de ejecución de un programa. En Python como en otros lenguajes se admiten básicamente tres tipos de estructuras selectivas: Simple, doble y múltiple.

Simple

Esta sentencia permite ejecutar una acción si la condición es verdadera o también podríamos decir que evita la acción si la condición es falsa. La sintaxis es como sigue:

```
if (expresión_condicional):
    sentencias
```

En la anterior definición notamos que la sentencia if debe ser escrita siempre en minúsculas, seguida de una *expresión_condicional* que es una condición evaluada de forma lógica y va seguida del operador (:). Si esta condición se evalúa como *True* se ejecutan las sentencias localizadas posteriormente. En el ejemplo siguiente se

cumple la condición *num > 0,* dando con ello paso a la sentencia del la función print():

```
num = 5
if(num > 0):
   print("El número " , num , "Es positivo")
```

El número 5 es positivo

Tenga presente que la mencionada condición no se restringe a una sola subexpresión, es posible definir varias dependiendo de lo que se quiera verificar o controlar, pero eso sí cumpliendo con el requisito que cada una deba ser evaluada logicamente. Consideremos el siguiente pseudocódigo de dos estructuras *if* con expresiones A y B conectadas con los operadores lógicos and y or.

```
if (A and B){
   sentencias
}
if (A or B){
   sentencias
}
```

En situaciones como estas la evaluación se llevará a cabo teniendo en cuenta las tablas de verdad. Observe como en la siguiente tabla se evalúan las expresiones booleanas: A y B.

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Doble

Realiza una acción si la condición es verdadera, o realiza otra acción distinta si la condición es falsa. La sintaxis es como sigue:

```
if (expresión_condicional):
   sentencias1
else:
   sentencias2
```

Ampliemos el ejemplo anterior ahora con una estructura selectiva doble:

```
num = 5
if(num > 0):
   print("El número " , num , "Es positivo")
else:
   print("El número " , num , "Es negativo")
```

El número 5 es positivo

Múltiple

Esta estructura consiste en definir varias opciones o alternativas de las cuales solo una será ejecutada. O dicho de otra manera permite establecer diferentes caminos por donde puede dirigirse el flujo del programa, pero que solo se puede tomar uno en un momento determinado. La sintaxis sería la siguiente:

```
if (expresión_condicional):
    sentencias
elif:
    sentencias
…
else:
   sentencias
```

Miremos un ejemplo de estructura de selección múltiple donde a partir de una nota numérica mostraremos su representación cualitativa equivalente:

```
nota = 3
if (nota > 0 and nota <= 1):
   print("Nota Insuficiente")
elif (nota > 1 and nota <= 2):
   print("Nota Regular")
elif (nota > 2 and nota <= 3):
   print("Nota Buena")
elif (nota > 3 and nota <= 4):
   print("Nota Sobresaliente")
elif (nota > 4 and nota <=5 ):
   print("Nota Excelente")
else:
   print("Nota no valida")
```

Como se aprecia cada una de las condiciones es verificada, si una de ellas se cumple se ejecuta las sentencias de ese bloque, o sea aquellas asociadas a esa condición, si ninguna de las condiciones tras su comprobación resulta verdadera se ejecutará lo establecido en el *else,* que bien podría considerarse la opción por defecto.

1.5 Estructuras repetitivas

Las estructuras repetitivas son usadas para ordenarle a un programa repetir un conjunto de sentencias un número determinado de veces.

Suponga que necesita repetir el mensaje "Bienvenido a Python" 100 veces. Puede por supuesto resultar tedioso escribir dichas sentencias en 100 ocasiones una detrás de la otra, y más aún si son 1.000 o 1.000.000 las veces que debe hacerse dicha repetición. Para evitar tener que hacer este trabajo desgastante, en programación existe lo que se conoce como estructuras repetitivas, ciclos o bucles, las cuales nos facilitan las cosas en este sentido.

Las sentencias provistas por Python para apoyarnos en esta labor son: *for* y *while*. Pero, además, es bastante común cuando se trabaja con ciclos emplear dos tipos especiales de variables: **contadores** y **acumuladores**. Un contador es una variable cuyo valor se aumenta o disminuye en un valor fijo. Como su nombre lo indica son usados para contar. Mientras que un acumulador es una variable cuyo valor aumenta o disminuye en un valor que puede variar. Es usado típicamente para ir sumando cantidades dentro un ciclo. Las líneas que siguen representan dos contadores y un acumulador:

```
# estos son contadores
contador = contador + 1
i = i + 2
# esto es un acumulador
suma = suma + edad
```

En este ejemplo, las variables *i* y *contador* siempre se incrementaría en 1 y en 2 unidades respectivamente, por su parte *suma* iría adicionando diferentes valores de la variable *edad* por cada iteración dentro de un ciclo.

El lenguaje Python maneja dos ciclos fundamentalmente que son el *for* y el *while*.

for

El ciclo *for* es una estructura de control que en Python se usa mayoritariamente para recorrer los elementos de una lista.