APRENDIENDO PYTHON

Aprende los conceptos básicos de programación con Python

Aprendiendo Python

Aprende los conceptos básicos de programación con Python

Jonathan Pumares

Este libro está a la venta en http://leanpub.com/aprendiendo-python

Esta versión se publicó en 2020-12-13

Índice general

Introducción

Muchos piensan que aprender a programar es algo difícil y que no a todos se les da, aunque programar requiere de mucha creatividad y es todo un arte, el primer paso que hay que dar es tener el gusto de hacerlo. Programar no es más que escribir instrucciones muy específicas a una máquina tonta, pero obediente.

Sentirás que al escribir tu primera línea de código podrás hackear cualquier sistema. Es cierto, los programadores tenemos poderes, y un gran poder conlleva una gran responsabilidad, es por eso que debes usar estos poderes para crear cosas realmente útiles para ti y para las demás personas.

No quiero hacerte larga esta introducción, así que de lleno te motivo a explorar este nuevo mundo de la programación en donde ayudamos a convertir una IDEA en una GRAN empresa.

Acerca de este libro

A lo largo de mi aprendizaje, he pasado por varios profesores que me han enseñado no solamente sus conocimientos, sino su experiencia y empatía hacía mi persona, lo cuál siempre estaré agradecido con ellos por ser lo que soy profesionalmente en este momento.

Algunos de estos profesores hoy son mis amigos que me han motivado a seguir adelante y me han apoyado en todo aspecto de mi vida, tanto personal como profesionalmente.

Ahora trabajar y ser parte de una institución educativa que forma y formará profesionistas del hoy y del futuro, me permite dar lo mejor de mi para seguir compartiendo y enseñando todo lo que he aprendido. Es por ello que me motivo a escribir este libro que en este momento estas leyendo.

Si algo tengo que decirte es que nunca dejarás de aprender, siempre estarás en constante aprendizaje, leyendo, estudiando, investigando, capacitándote, aprendiendo nuevas tecnologías.

Te motivo a seguir tu propio camino como profesional dando lo mejor de ti, disfrutando y a seguir aprendiendo lo que tanto te apasiona.

¿Quién soy?

Mi nombre es Jonathan Pumares, soy Desarrollador web full-stack (Frontend, Backend y Servidores) con varios años de experiencia en el desarrollo de aplicaciones de escritorio, desarrollo web y proyectos emprendedores propios. Actualmente dedicado como Coordinador de Desarrollo de Sistemas y Asesor de Proyectos en el Instituto Tecnológico de Campeche y Docente en la Universidad Interamericana para el Desarrollo. Egresado como Ingeniero en Sistemas de Información por la Universidad Interamericana para el Desarrollo. Me gusta aprender algo nuevo cada día sobre tecnologías web y programación, tambien me apasiona enseñar y compartir conocimiento.

Si en algo puedo ayudarte, tienes dudas, sugerencias o si podemos compartir conocimiento házmelo saber en mis redes sociales, estaré igual compartiendo contenido de valor para ti:

- Sígueme en Facebook[1]
- Sígueme en Twitter[2]

[1] https://www.facebook.com/jepumares
[2] https://twitter.com/3jonapumares

Algoritmos

Definición

En algún momento de nuestra vida y estoy seguro que a diario, usamos los algoritmos, pero te preguntarás, ¿Qué es un algoritmo? Un algoritmo es un conjunto de pasos a seguir para resolver un problema o una necesidad, Y ¿Cómo es eso? A diario tienes que tomar decisiones, como que vas a comer, como lo vas a cocinar, etc.

Te pongo un ejemplo, como harías para preparar una deliciosa sopa de verduras. Primero necesitas los ingredientes, tal vez algunos ya tengas en casa y otros los tengas que comprar, pero una vez que los tengas todos, el siguiente paso es realizar el procedimiento de preparar la sopa, hervir el agua, picar todas las verduras, agregar las verduras al agua hirviendo, esperar aproximadamente unos 40 minutos hasta que las verduras estén completamente cocidas, y listo el resultado es una deliciosa sopa.

Te diste cuenta como fuimos armando un algoritmo en algo tan común que hacemos a diario, digo no es que seamos profesionales de la cocina, pero todos hemos preparado algo para comer.

Los algoritmos pueden ser aplicados a cualquier aspecto de nuestra vida, desde decidir qué ropa ponernos durante el día hasta aplicarlos a software que vayamos a construir.

Partes de un algoritmo

Todo algoritmo cuenta con una estructura básica que se compone de tres partes: Entrada, Proceso y Salida.

Vamos a explicar cada parte de un algoritmo basado en el ejemplo de la sopa y luego veremos otro ejemplo más técnico.

Entrada

Son los datos que se necesitan para luego ser procesados y generar el resultado esperado. En el ejemplo de la sopa los ingredientes son nuestros datos de entrada, los cuales necesitaremos para producir nuestra sopa.

Proceso

Son los pasos necesarios para obtener la solución a un problema o a un resultado esperado. En el ejemplo de la sopa, el procedimiento de cocinar todos nuestros ingredientes es el proceso.

Salida

Son los resultados obtenidos luego del proceso que se realizo con los datos de entrada. En el ejemplo de la sopa, el resultado es nuestra sopa ya lista.

Algoritmo suma de dos números

Ahora vayamos a un ejemplo más técnico, ¿Cómo hariamos el algoritmo para sumar dos números enteros?

Vamos a definir este algoritmo con los componentes que ya vimos: Entrada, Proceso y Salida.

¿Qué necesitamos para realizar nuestra suma?, es decir, ¿Cuales son los datos de entrada que necesitamos? Pues precisamente esos dos números o valores, así que practicamente esa sería nuestra entrada. Vamos a poner de ejemplo a los números 3 y 5.

Luego, ¿Cómo realizas el proceso de suma? Simplemente a nuestro primer valor (3) le sumamos nuestro segundo valor (5) a través del signo de suma (+).

Eso nos daría como resultado el número 8, que es la salida obtenida por el proceso.

Viste que tan simple es definir nuestros algoritmos. En aplicaciones más complejas vas a necesitar de otros componentes como tomar decisiones, repetir instrucciones, etc. Tomalo con calma que ya iremos a esos temas.

Formas de representar un algoritmo

Los algoritmos pueden ser representados principalmente de dos formas: Diagramas de flujo y Pseudocódigo. Esto te ayudará a plasmar tu algoritmo en cualquiera de estas formas y luego convertirlo a código.

Aunque te contaré algo real, al principio tal vez se te haga una buena práctica primero plasmar tu algoritmo en cualquiera de estas formas y luego pasarlo a código, pero la realidad es que con el tiempo vas agarrando experiencia y ya ves innecesario plasmarlo primero en un diagrama de flujo o pseudocódigo, pero como te repito eso te lo dará la experiencia, si estas empezando en este mundo de la programación te recomiendo tener la buena práctica de primero plasmar tu algoritmo antes de pasarlo a código, una vez que consideres que tienes la suficiente experiencia para ya no usarlos, salta ese paso.

Ahora veamos que es un diagrama de flujo.

Diagramas de flujo

Son representaciones gráficas que usamos para construir un algoritmo, principalmente son símbolos que son conectados por medio de flechas para indicar todo el flujo de instrucciones que tendrá nuestro algoritmo.

Veamos los símbolos utilizados en los diagramas de flujo.

Inicio/Fin

Inicio/Final

Se utiliza para indicar el inicio y final de un diagrama de flujo, del Inicio solamente debe salir una línea y al Final solamente debe llegar una línea. Se pone la palabra Inicio o Final dentro de la forma según sea el caso.

Línea de flujo

Se utiliza para indicar el orden en que serán ejecutadas las operaciones. Cada flecha indica el paso hacía la siguiente instrucción.

Entrada/Salida

Se utiliza para indicar entrada y salida de datos en forma general.

Entrada por teclado

Se utiliza para indicar la lectura de datos por teclado en el que la computadora espera a que el usuario teclee los datos de entrada para posteriormente guardarlos en variables.

Proceso

Se utiliza para indicar una instrucción que debe ser procesada por la computadora.

Decisión

Se analiza una situación y dependiendo del resultado se toma un camino u otro con base a los valores verdadero y falso.

Iteración

Se utiliza para indicar qué una instrucción o conjuntos de instrucciones deben ser ejecutadas varias veces dependiendo de una condición.

Salida impresa

Se utiliza para indicar la impresión de uno o varios resultados en forma de documento.

Salida en pantalla

Se utiliza para indicar la impresión de mensajes o resultados en pantalla.

Conector

Se utiliza para indicar el enlace entre dos partes del diagrama dentro de la misma página.

Conector fuera de la página

Se utiliza para indicar el enlace entre dos partes del diagrama que se encuentran en páginas diferentes.

Pseudocódigo

Es una descripción de alto nivel de un algoritmo, ¿Por qué decimos que es de alto nivel? Porque es muy similar al lenguaje natural, es decir, al lenguaje humano pero con algunas convenciones

sintácticas propias de lenguajes de programación, como asignaciones, ciclos, condicionales, etc. La diferencia es que el pseudocódigo no esta regido bajo ningún estándar.

La intención del pseudocódigo es que sea entendible por las personas al momento de leer un algoritmo, y que posteriormente esas ideas y estructuras sean trasladadas a un lenguaje de programación para su codificación.

Para escribir pseudocódigo existen algunas herramientas que podemos utilizar, una de las más famosas es PSeInt, que lo puedas encontrar disponible desde su sitio en el siguiente enlace: http://pseint.sourceforge.net/[3] lo cuál esta funciona para cualquier plataforma que utilices, ya sea Linux, Windows o Mac.

Pruebas de escritorio

Como buena práctica siempre es bueno probar el correcto funcionamiento de nuestros algoritmos, es por ello que existen las famosas pruebas de escritorio, las cuales son simulaciones del comportamiento de un algoritmo, lo que permite determinar la validez del mismo.

Y ahora, ¿Cómo se realiza una prueba de escritorio? Preparate para usar papel y lápiz, o si lo prefieres un pizarrón y plumones, ten a la mano tu algoritmo y genera una tabla en donde pondrás todas las variables que involucra tu algoritmo, ve siguiendo las instrucciones y anotando los valores que se vayan generando a lo largo de tu algoritmo.

Luego de concluir tu algoritmo y ver tus resultados te pregunto, ¿Todo salió como esperabas? La ventaja de las pruebas de escritorio es que te permiten detectar posibles errores, corregirlos o mejorar tu algoritmo.

Pruebas automatizadas

En la práctica y cuando tengas experiencia en el desarrollo de software, no harás a mano tus pruebas tal como lo vimos en el tema anterior, si no que automatizarás tus pruebas para que verifiques que tu software funciona correctamente.

Para cada lenguaje de programación existen ciertas herramientas especiales que nos ayudarán a realizar la tarea de escribir y ejecutar nuestras pruebas automáticamente y verificar que los resultados obtenidos sean los mismos que los resultados esperados.

[3] http://pseint.sourceforge.net/

¿Qué es Python?

Python es uno de los lenguajes de programación que me ha impresionado por la facilidad de uso y rápido aprendizaje, muchos lo prefieren para aprender a programar, y esto se debe a que sus declaraciones y sintaxis son explícitas y muy fáciles de aprender. Un lenguaje que no solamente podemos utilizarlo para escribir algoritmos sencillos sino es sumamente potente para crear aplicaciones web, videojuegos, aplicaciones de escritorio y una infinidad de desarrollo de software que podamos imaginar.

Es por ello que les vengo a hablar de este maravilloso y potente lenguaje. Plataformas como Instagram, Pinterest, Vine, Dropbox, Netflix, Spotify, Paypal, Uber, Platzi, entre otras, están construidas en Python.

Ahora quisiera darles una introducción al lenguaje sobre su sintaxis, empezando con los conceptos básicos de programación.

Python 2 VS Python 3

En algunas ocasiones te vas a encontrar que algunas de las librerías de terceros aún están escritos en Python 2, y esto simplemente es porque aún no han sido actualizadas para Python 3.

Y tal vez tu pregunta aquí es ¿Qué versión de Python debería aprender?

De entrada déjame decirte que desde el 1 de enero de 2020, la comunidad de Python dejo de dar soporte a la versión 2, por lo que recomiendan actualizar a Python 3 lo antes posible, esto incluye tanto aplicaciones escritas en Python como librerías de terceros.

Puedes leer más al respecto en el siguiente enlace: https://www.python.org/doc/sunset-python-2/[4]

Aún así las diferencias entre versiones son mínimas y tal vez por alguna razón te encuentres todavía con alguna librería que aún sigue escrita para Python 2, por ello es bueno conocer las diferencias entre estas dos versiones, aún así te recomiendo usar Python 3.

Sentencia print

La sentencia `print` funciona para imprimir un mensaje en pantalla, la diferencia es que en Python 2 `print` es usado como un keyword del lenguaje, y en Python 3 como una función:

```
# Python 2
print 'Hola culebrita'

# Python 3
print('Hola culebrita')
```

Eso no quiere decir que no puedas usar la función `print()` en Python 2, por el contrario en Python 3 no puedes usar la palabra reservada `print`.

Cadenas de texto

Para usar caracteres extendidos como la eñe y/o acentos en Python 2 necesitamos indicar qué utilizaremos otra codificación y esto es porque se utiliza cadenas de texto en formato ASCII. En Python 3 se utiliza cadenas de texto en formato Unicode por lo que el uso de caracteres extendidos es por defecto:

[4] https://www.python.org/doc/sunset-python-2/

```
# Python 2
'Me gusta la piña' # 'Me gusta la pi\xc3\xb1a'

# Python 3
'Me gusta la piña' # 'Me gusta la piña'
```

Como se menciono, para poder usar los caracteres extendidos en Python 2 es necesario definir la codificación a utilizar, por lo que al principio de tus archivos .py, es necesario poner la siguiente línea:

```
# coding=utf-8
print('Hola niño') # Hola niño
```

La línea `# coding=uft-8` especifica que utilizaremos una codificación utf-8 para el archivo, por lo que se mostrarán los caracteres extendidos de manera correcta al momento de imprimir el mensaje.

Para más detalles puedes ver el siguiente enlace: https://www.python.org/dev/peps/pep-0263/[5]

División entera

En Python 2 al dividir dos números enteros, resulta otro número entero, pero si al menos alguno de los dos valores a dividir es un flotante, el resultado es un número con punto decimal. Para Python 3, no importa que los dos valores a dividir sean números enteros, el resultado que devuelve es un número con punto decimal:

```
# Python 2
resultado = 3 / 2
resultado # 1

# Python 3
resultado = 3 / 2
resultado # 1.5
```

En caso de que en Python 3 queramos que el resultado de una división entre dos valores nos devuelva un número entero, debemos de usar dos veces la barra diagonal (`//`):

[5] https://www.python.org/dev/peps/pep-0263/

```
# Python 3
resultado = 7 / 4
resultado # 1.75

resultado = 7 // 4
resultado # 1
```

Levantando excepciones

En ocaciones vas a requerir lanzar un error en caso de que el usuario haga algo indebido dentro de tu aplicación o necesites validar datos introducidos por el mismo usuario, para ello existe la sentencia `raise` que permite levantar una excepción en algún punto de tu código y podamos advertirle al usuario de lo que esta pasando.

A lo largo del libro detallaremos mejor el manejo de errores en Python.

Para Python 2 se pueden utilizar dos tipos de sintaxis para levantar excepciones:

```
# Python 2
raise IOError, "file error"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: file error

raise IOError("file error")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: file error
```

Date cuenta en la pequeña diferencia entre las dos sintaxis, en la primera podemos notar qué después del tipo de excepción (`IOError`) ponemos el signo coma (`,`) y el mensaje que queremos lanzar. En la segunda sintaxis encerramos el mensaje entre paréntesis.

En Python 3 usamos la "nueva" sintaxis para levantar excepciones haciendo uso de paréntesis. Y decimos nueva porque es la única que se utiliza dentro de Python 3 pero realmente ya se utilizaba desde Python 2:

```
# Python 3
raise IOError("file error")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: file error
```

Si queremos utilizar el tipo de sintaxis sin paréntesis en Python 3 levantará un `SyntaxError` en su lugar:

```
# Python 3
raise IOError, "file error"
    File "<stdin>", line 1
        raise IOError, "file error"
                     ^
SyntaxError: invalid syntax
```

Manejo de excepciones

El manejo de excepciones ha cambiado ligeramente entre versiones. En Python 3 ahora tenemos que utilizar la palabra reservada “as”:

```
# Python 2
try:
    let_us_cause_a_NameError
except NameError, err:
    print(err)

name 'let_us_cause_a_NameError' is not defined

# Python 3
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err)

name 'let_us_cause_a_NameError' is not defined
```

Retornando objetos iterables en lugar de listas

Algunas funciones y métodos ahora retornan objetos iterables en Python 3, en lugar de listas como en Python 2:

```
# Python 2
print(range(3))
[0, 1, 2]

print(type(range(3)))
<type 'list'>

# Python 3
print(range(3))
range(1, 3)
```

Si vienes de otros lenguajes de programación, sabrás a lo que me refiero con algunos conceptos que ya hemos visto a lo largo de este capítulo, pero si eres nuevo en este mundo de la programación, no te preocupes si hay conceptos que todavía no entiendes, te invito a leer los próximos capítulos para entender todo esto a detalle. Pero para que me entiendas un objeto iterable es un conjunto de valores que podemos recorrer uno por uno para ser evaluados. Por otro lado una lista es igual un conjunto de valores, pero no es igual a un objeto iterable, ya verás sus diferencias más adelante.

Las funciones y métodos más utilizados que ahora no retornan listas en Python 3 son: `zip()`, `map()`, `filter()`, el método `keys()`, el método `values()`, el método `items()`.

Función input VS raw_input

Tanto la función `input()` como `raw_input()` permite a los usuarios introducir datos desde el teclado y guardarlos temporalmente en un variable. La diferencia es que en Python 2 la función `input()` retorna el tipo de dato que se introduzca, si quisiéramos retornar una cadena sin importar el tipo de dato introducido por el usuario usamos la función raw_input():

```
# Python 2
entrada = input('Ingrese un número: ')
Introduce un número: 5

type(entrada)
<type 'int'>

entrada = raw_input('Ingrese un número: ')
Ingrese un número: 5

type(entrada)
<type 'str'>
```

En Python 3 se eliminó la función raw_input() y ahora input() por defecto retorna una cadena:

```
# Python 3
entrada = input('Ingrese un número: ')
Ingrese un número: 5

type(entrada)
<class 'str'>
```

Función xrange VS range

En Python 2 la función `xrange()` retorna un objeto iterable, mientras que `range()` retorna una lista. Ahora en Python 3, se elimino la función `xrange()` y ahora se usa `range()` como si fuera `xrange()`:

```
# Python 2
print(xrange(3))
xrange(3)

print(range(3))
[0, 1, 2]

# Python 3
print(range(3))
range(0, 3)
```

Si intentas usar `xrange()` en Python 3, lanzará un error de tipo `NameError`:

```
# Python 3
print(xrange(3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xrange' is not defined
```

Variables y tipos de datos

Definición

Las variables son espacios de memoria en donde almacenamos temporalmente datos y que pueden ser utilizados en cualquier momento durante la ejecución de nuestro script. Piensen en un momento como si nuestra memoria de la computadora fuera un gran almacén y las variables como cajas en donde podemos guardar datos (números, caracteres, etc.), mientras más capacidad tenga nuestra memoria (almacén), más datos podemos almacenar.

Para definir una variable usamos la siguiente sintaxis:

```
1  nombre_de_la_variable = valor_de_la_variable
```

Cabe mencionar que existe una guía de estilo PEP 8 para escribir código Python y que nos ayudará por medio de buenas prácticas a tener un código más legible, entendible y ordenado como si una sola persona la hubiera escrito, y esto mismo ayuda al mantenimiento fácil del código.

Mencionando a nuestra guía de estilo PEP 8 se recomienda utilizar para nuestras variables nombres descriptivos y en minúsculas. Si nuestra variable se compone de dos palabras o más, separar las palabras por medio de guiones bajos, seguido de la asignación del valor debe haber espacios entre el signo = y el valor de la variable.

Constantes

Las constantes son utilizadas cuando queremos definir valores que no cambiarán durante la ejecución de nuestro script.

El mejor ejemplo que podemos dar podría ser el IVA, o inclusive el valor de PI (3.1416) son valores que no cambian.

En Python no tenemos ninguna palabra reservada para crear constantes lo cual tenemos que recurrir a las variables pero tomando en cuenta un detalle, por convención tenemos que escribirlas en mayúsculas.

Definimos una variable, pero con el detalle de que lo escribimos en mayúsculas para diferenciar y hacer entender de que se trata de una constante.

```python
# Variable que estamos indicando que es una constante escribiéndola en mayúsculas
IVA = 0.21
```

Tipos de datos

Python se considera un lenguaje de **tipado dinámico**, es decir, que no necesitamos declarar explícitamente el tipo de dato que vamos a utilizar, y esto también implica que en cualquier momento puede tomar valores de distinto tipo en distintos momentos, aunque esto no se considera una buena práctica a menos que sea muy necesario.

Cadena de texto (string)

Las variables de tipo string sirven para almacenar caracteres o palabras. Podemos usar comillas simples o dobles:

```python
cadena_de_texto = 'Python'
una_cadena_mas = "Vamos bien"
```

Manten un solo estilo

Por buena práctica se recomienda usar un solo estilo al momento de escribir variables de tipo string, ya sea usar comillas simples o usar comillas dobles, pero nunca combinar los dos, esto no quiere decir que no funcione, si no simplemente debemos mantener un orden en nuestro código, ya que eso hable bien de nosotros.

De igual manera podemos definir cadenas de texto de varias lineas usando en este caso si utilizaremos tres comillas dobles al principio de la cadena, su contenido y cerramos otra vez con tres comillas dobles:

```python
cadena_de_varias_lineas = """
Esta cadena
es ocupada por varias lineas
"""
```

Número entero (int)

Las variables de tipo int nos permiten almacenar valores numéricos enteros, es decir, que no cuentan con el punto decimal. Utilizados para las operaciones aritméticas básicas, como contadores, acumuladores, hacer cálculos y comparaciones.

```
numero = 12
```

Número flotante (float)

Al igual que las variables de tipo int nos permiten almacenar valores numéricos con la diferencia que se pueden manejar números con punto decimal.

```
precio = 7435.80
```

Booleano

En este tipo de dato solamente podemos utilizar dos valores posibles `True` o `False`. Son utilizados para indicar estados. Supongamos que tenemos la variable baja y su valor inicial es False, esta variable nos servirá para determinar a través de una serie de operaciones si un alumno se encuentra dado de baja (por el momento al indicarle el valor inicial de False estamos diciendo que se encuentra activo), detrás de escena existen una serie de criterios (validaciones) para considerar a un alumno dado de baja, si el alumno cae o cumple con alguno de estos criterios cambiará su estado a True, por lo que se consideraría como dado de baja, y es ahí donde podemos notar el cambio de estado.

```
si = True
no = False
```

La función type()

Aunque en otro capítulo veremos más a detalle el concepto de funciones, te adelanto que una función simplemente hace algo y puede o no regresar un valor como resultado de una operación.

La función `type()` pertenece a Python, aunque nosotros podemos crear nuestras propias funciones, esta función (`type()`) nos dice que tipo de dato es una variable o valor, la cual se la pasamos como argumento a la función dentro de los paréntesis:

```
type('Hola culebrita')
<class 'str'>

type(3)
<class 'int'>

type(12.7)
<class 'float'>
```

Podrás notar que en la primer llamada a la función le pasamos una cadena y de inmediato nos dice que es un `str` que viene siendo un nombre corto de `string` (**cadena**), luego le pasamos un número entero, el cuál nos devuelve un `int`, y por último le pasamos un número con punto decimal, que nos devuelve un `float`.

Más adelante veremos otros tipos de datos más complejos.

Comentarios

Los comentarios en Python, al igual que en otros lenguajes de programación, nos sirven para dar una explicación de lo que hace una línea de código de nuestro en script, alguna función en especifico o una serie de instrucciones, para que a futuro otro desarrollador sepa que se hizo y su mantenimiento sea más fácil.

Manten tus comentarios actualizados

Como una buena practica de acuerdo al PEP 8 es prioridad mantener los comentarios al día cuando el código sea actualizado.

Comentarios en línea

Un comentario en línea es aquel que se encuentra en la misma línea que una sentencia. Usualmente se utiliza para describir lo que realiza alguna instrucción en específica, como describir una variable, alguna acción, etc. Se escribe con el signo de numeral:

```
1  color = 'Verde' # En esta variable se guarda el color favorito del usuario
```

Comentarios en bloque

Muchas veces se comete el error de que los comentarios en bloque se escriben con triples comillas dobles, pero la realidad es que estas se utilizan para las cadenas de documentación que en breve se mencionará este tipo de comentario. Lo cierto es que cada línea de un comentario en bloque comienza con el signo de numeral, como los comentarios en línea, la diferencia es que este abarca varías líneas, pero no se encuentra dentro de la misma línea que una sentencia:

```
1  # Un comentario en bloque
2  # comienza con el signo de numeral
3  # y abarca varías líneas
4  # en donde cada línea vemos el numeral
5  # pero no se encuentra dentro de la misma línea que una sentencia
6  resultado = numero1 + numero2
```

Cadenas de documentación

En Python, un docstring o cadena de documentación es una literal de cadena de caracteres que se coloca como primer enunciado de un módulo, clase, método o función con el objetivo de explicar su intención. Veamos un ejemplo:

```
1  def sumar(numero1, numero2):
2      """Función para sumar dos números.
3
4      Devuelve el resultado de la suma de los dos números
5
6      Parámetros:
7      numero1 -- Número 1 a sumar
8      numero2 -- Número 2 a sumar
9
10     """
11     return numero1 + numero2
```

La primera línea de la cadena de documentación debe ser una línea de resumen terminada con un punto. Debe ser una breve descripción de la función que indica los efecto como tal. Es importante que quepa en una sola línea y que esté separada del resto del docstring por una línea en blanco.

El resto de la cadena de la documentación debe describir el comportamiento de la función, los valores que devuelve, las excepciones que arroja y cualquier otro detalle que consideremos relevante.

Al final de la cadena de documentación se recomienda dejar una línea en blanco antes del cierre de las triples comillas dobles.

Listas

Las listas son un tipo de dato complejo en Python, se le dice complejo porque son capaces de almacenar más de un valor, es decir, un conjunto de elementos o valores.

Dos características de las listas son:

- Son mutables, lo que quiere decir, que podemos actualizar sus valores en cualquier momento.
- Nos permiten almacenar cualquier tipo de valor, ya sea cadenas, números, booleanos y hasta otros tipos de datos complejos.

Definir una lista

Para definir una lista debemos tomar en cuenta lo siguiente:

1. Elegimos un nombre de variable para nuestra lista de acuerdo a las recomendaciones del **PEP 8**, es decir, que nuestra variable tenga un nombre significativo y no usar palabras reservadas del lenguaje como `if`, `else`, `for`, `continue`, `def`, por mencionar algunas.
2. Definimos el signo de igual, el cual indica que estamos realizando un asignamiento.
3. Y por último colocamos los elementos o valores dentro de corchetes, y separando cada elemento con el signo de coma.

Veamos unos ejemplos:

```
1  numeros = [1, 2, 3, 4, 5]
2  nombres = ['Diana', 'Carlos', 'Ana', 'Saul']
```

Cómo vemos en los ejemplos de arriba podemos definir listas que pueden contener el mismo tipo de dato, en el caso de la lista `numeros` vemos que definimos puros valores de tipo **int**, y en el caso de la lista `nombres` definimos puros valores de tipo **string**.

Pero lo especial de las listas es que podemos almacenar cualquier tipo de valor como cadenas, enteros, inclusive una lista dentro de otra, y agregar otros tipos de datos complejos como las tuplas y diccionarios como veremos más adelante.

```
1  lista = [3, 4.5, 'Carlos', [1, 8]]
```

Acceder a una lista

Para acceder a los elementos de una lista debemos tomar en cuenta lo siguiente:

- Debemos indicar el índice o la posición en la que se encuentra el elemento.
- El primer elemento de una lista inicia en la posición cero.

```
lista_numeros = [1, 2, 3, 4, 5]
lista[1] # 2
```

En el ejemplo de arriba estamos accediendo a la posición 1 de la lista por lo cual el valor que nos devuelve es el 2.

Acceder a una lista dentro de otra

Como mencionamos, podemos almacenar dentro de una lista cualquier tipo de valor, inclusive otra lista:

```
lista = [4, ['Hola', 'Culebrita']]
```

Y para acceder a sus valores dependen del número de listas que se definan dentro de otras. En el ejemplo de arriba, si queremos acceder al valor de **Hola**, lo hacemos de la siguiente manera:

```
lista[1][0] # Hola
```

Como vemos hacemos uso del corchete dos veces, primero definimos la posición 1 porque accedemos a la lista que está más por fuera, y luego accedemos a la posición 0, porque ahora estamos accediendo a la lista que esta anidada.

Modificar valores de una lista

Para modificar los valores de una lista tenemos que acceder a la posición del valor que queremos modificar y con el signo de asignamiento = le agregamos el nuevo valor, vamos a modificar el valor de un número por una cadena:

```
lista = [1, 2, 3, 4, 5]
lista[0] = 'Hola' # ['Hola', 2, 3, 4, 5]
```

Métodos de las listas

Como todo en Python es un objeto, en las listas no es la excepción, es por ello que las listas cuentan con muchos métodos que podemos utilizar, vamos a definir una lista base y sobre ella aplicar los métodos más importantes que podemos utilizar.

```
lista = ['Aprender', 'Python', 'es', 'genial']
```

append()

El método `append()` nos permite agregar elementos al final de una lista. Cabe mencionar que como ya hemos visto podemos almacenar cualquier tipo de valor:

```
lista.append('y divertido!') # ['Aprender', 'Python', 'es', 'genial', 'y divertido!']
lista.append(3) # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 3]
```

remove()

El método `remove()` elimina el elemento o valor que se le pase como parámetro. En caso de que el elemento no exista se lanza una excepción de tipo **ValueError**, en caso de haber elementos duplicados eliminará el primer elemento encontrado dentro de la lista:

```
lista.append(8) # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 3, 8]
lista.append(3) # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 3, 8, 3]
lista.remove(3) # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 8, 3]
lista.remove(4) # ValueError: list.remove(x): not in list
```

index()

El método `index()` devuelve la posición en la que se encuentra el elemento o valor pasado como parámetro. En caso de que el elemento no exista se lanza una excepción de tipo **ValueError**, en caso de haber elementos duplicados se devolverá el índice del primer elemento encontrado dentro de la lista:

```
lista.append(8) # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 8, 3, 8]
lista.index(8) # 5
lista.index(5) # ValueError: 5 is not in list
```

count()

El método `count()` devuelve el número de veces que un elemento se encuentra dentro de la lista, se le pasa como parámetro el elemento o valor que queremos contar. En caso de pasar como parámetro un valor que no existe dentro de la lista nos devolverá el valor **0**:

```
lista # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 8, 3, 8]
lista.count(8) # 2
lista.count(5) # 0
```

reverse()

El método `reverse()` nos permite invertir los elementos de una lista:

```
lista # ['Aprender', 'Python', 'es', 'genial', 'y divertido!', 8, 3, 8]
lista.reverse() # [8, 3, 8, 'y divertido!', 'genial', 'es', 'Python', 'Aprender']
```

pop()

El método `pop()` elimina el último elemento de la lista y nos devuelve el último valor eliminado:

```
lista # [8, 3, 8, 'y divertido!', 'genial', 'es', 'Python', 'Aprender']
lista.pop() # 'Aprender'
lista # [8, 3, 8, 'y divertido!', 'genial', 'es', 'Python']
```

sort()

El método `sort()` ordena los elementos de una lista. Para utilizar este método es necesario tener una lista que contengan el mismo tipo de dato, si intentamos ejecutar este método con una lista que tenga diferentes tipos de datos, nos devolverá un error de tipo **TypeError**:

```
lista # [8, 3, 8, 'y divertido!', 'genial', 'es', 'Python']
lista.sort() # TypeError: '<' not supported between instances of 'str' and 'int'
numeros = [2, 1, 5, 4, 3]
numeros.sort()
numeros # [1, 2, 3, 4, 5]
```

Tuplas

Las tuplas al igual que las listas son un tipo de dato complejo, porque son capaces de almacenar más de un valor, es decir, un conjunto de elementos o valores.

Algunas de las características de las tuplas son:

- Una tupla es un tipo de colección ordenada.
- Son inmutables, lo que quiere decir, que sus valores no se modifican en tiempo de ejecución.
- Son más ligeras a comparación que las listas.
- Ahorran memoria y son rápidas en su acceso a cada elemento.

Definir una tupla

Definimos una tupla en tres sencillos pasos:

1. Definimos el nombre de la variable (Según las recomendaciones del **PEP 8**).
2. Definimos el signo de igual, el cual indicará asignamiento.
3. Colocamos los elementos dentro de paréntesis y usamos el signo coma para separar los elementos.

Podemos definir tuplas con elementos del mismo tipo:

```
numeros = (20, 7, 3, 4, 5)
nombres = ('Luis', 'Ana', 'Samuel')
```

De igual manera podemos definir tuplas con elementos de distintos tipos:

```
tupla = ('Luis', 7, False, 5.72, 100)
```

Como ya hemos visto al igual que las listas, se puede tener muchos niveles de anidamiento dentro de una tupla, es decir, tener una tupla dentro de otra tupla:

```
tupla = (('A', 'B', 'C'), (1, 2, 3, 4, 5))
```

Por convención encapsulamos los elementos de una tupla entre paréntesis, pero otra manera de definir una tupla es omitiendo los paréntesis, no es muy común su uso, pero es bueno saberlo:

```
tupla = 1, 2, 3, 4
```

En caso de que una tupla solamente cuente con un elemento, es obligatorio poner una coma al final del elemento:

```
tupla = (1,)
tupla2 = 1,
```

Acceder a una tupla

Al igual que en las listas, en las tuplas se usa el mismo mecanismo para acceder a los elementos o valores, es decir, a través del índice o la posición en donde se encuentra el elemento que queremos obtener.

```
tupla = (20, 40, 60, 80, 100)
tupla[0] # 20
```

Si queremos acceder a una tupla que se encuentra dentro de otra, debemos tomar en cuenta el número de tuplas definidas, de esto dependerá cuantos corchetes se necesiten para acceder a un elemento:

```
tupla = ((1, 2, 3), (4, 5))
tupla[0][1] # 2
```

En caso de que queramos acceder a una posición que no se encuentra dentro de una tupla, nos arrojará un error de tipo **IndexError**, es decir, indicando que la posición se encuentra fuera de rango.

Métodos de las tuplas

Vamos a mencionar los métodos más importantes que son utilizados en las tuplas.

count()

El método **count()** cuenta el número de veces que se encontró un elemento dentro de una tupla, en caso de que el elemento no se encuentre dentro de la tupla nos devolverá el valor **0**:

```
tupla = (1, 2, 3, 4, 5, 2, 2, 2, 2)
tupla.count(1) # 1
tupla.count(2) # 5
tupla.count(7) # 0
```

index()

El método **index()** devuelve la posición en la que se encuentra un elemento pasado como parámetro. En caso de que el elemento no exista dentro de la tupla nos devolverá un error de tipo **ValueError** indicando de que el elemento pasado como parámetro no se encuentra dentro de la tupla. Si existen elementos duplicados dentro de la tupla, devolverá el primer elemento encontrado.

```
tupla = (1, 2, 3, 4, 5, 2, 2, 2, 2)
tupla.index(1) # 0
tupla.index(2) # 1
tupla.index(7) # ValueError: tuple.index(x): x not in tuple
```

¿Modificar valores de una tupla?

Aunque ya mencionamos que las tuplas son inmutables (que no podemos modificar sus valores en tiempo de ejecución) es bueno que conozcas que estas no cuentan con métodos para modificar sus valores como `append()`, `remove()`, `reverse()`, `pop()` y `sort()` por lo tanto si intentamos ejecutar alguna de ellas nos lanzará un error de tipo `AttributeError`, veamos un ejemplo:

```
tupla = (1, 2, 3, 4, 5, 2, 2, 2, 2)
tupla.append(5) # AttributeError: 'tuple' object has no attribute 'append
tupla.remove(5) # AttributeError: 'tuple' object has no attribute 'remove'
tupla.reverse() # AttributeError: 'tuple' object has no attribute 'reverse'
tupla.pop() # AttributeError: 'tuple' object has no attribute 'pop'
tupla.sort() # AttributeError: 'tuple' object has no attribute 'sort'
```

Diccionarios

Los diccionarios son otro tipo de dato complejo en Python, ya que nos permite almacenar cualquier tipo de valor como enteros, cadenas, y hasta otros tipos de datos complejos como ya hemos visto.

La diferencia con las listas y tuplas, es que los diccionarios ya no indexan los elementos mediante un rango numérico, si no que sus elementos son indexados mediante claves, y ya no mantiene un orden al añadir un nuevo elemento al final del diccionario.

Definir un diccionario

Definamos un diccionario en tres sencillos pasos:

1. Definimos el nombre de la variable (Según las recomendaciones del **PEP 8**).
2. Definimos el signo de igual, el cual indicará asignamiento.
3. Colocamos los elementos entre llaves. Las parejas de clave y valor se separan con comas, y la clave y el valor se separan entre dos puntos.

```
1  diccionario = {'foo': 'bar', 'baz': 'boise', 'num': 3}
```

Recomendaciones al definir un diccionario

Algunas recomendaciones a tomar en cuenta al momento de definir un diccionario son las siguientes:

- Al definir un diccionario, no puede tener claves duplicadas.
- Un valor puede contener cualquier tipo de dato, ya sea un número, cadena, lista, tupla, e inclusive otros diccionarios.
- Los valores de un diccionario son mutables, lo que quiere decir, que podemos actualizar sus valores en cualquier momento.
- Las claves de un diccionario pueden ser números, cadenas e, incluso, tuplas.
- Una clave puede ser una variable, siempre y cuando su valor sea un número o cadena.

Uso de cadenas en las claves de un diccionario

Por buena práctica, al momento de definir las claves de un diccionario se recomienda el uso de cadenas.

Acceder a un diccionario

Los diccionarios usan el mismo mecanismo de las listas y tuplas para acceder a los elementos, pero en vez de indicar el índice en forma de rango numérico, colocamos la clave de cada par de clave-valor del diccionario para obtener su valor.

```
diccionario = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
diccionario['b'] # 2
```

De igual manera que pasa con las listas y las tuplas podemos acceder a un valor de un diccionario que sea una lista, una tupla o incluso otro diccionario.

```
diccionario = {'a': [1, 2, 3, 4], 'b': (5, 6, 7, 8), 'c': {'d': 80, 'e': 100}}
diccionario['a'][2] # 3
```

Recuerda que el número de corchetes a utilizar depende de cuantas listas, tuplas o diccionarios existen para acceder a un elemento.

En caso de intentar acceder a una clave que no existe dentro del diccionario, nos arrojará un error de tipo **KeyError**.

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
diccionario['d'] # KeyError
```

Agregar nuevos elementos a un diccionario

Para agregar un nuevo elemento a un diccionario tenemos que especificar una nueva clave para ese diccionario, como la manera en la que accedemos a los elementos de un diccionario a través de corchetes, seguido del signo igual, y el valor que le queremos asignar a esa clave:

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
diccionario['d'] = 4 # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Modificar elementos de un diccionario

Para modificar elementos existentes de un diccionario, simplemente accedemos a su clave a través de los corchetes y le asignamos un nuevo valor:

```
diccionario = {'nombre': 'Carlos', 'pais': 'México', 'edad': 20}
diccionario['edad'] = 25 # {'nombre': 'Carlos', 'pais': 'México', 'edad': 25}
```

Métodos de los diccionarios

Veamos los métodos más importantes que son utlizados en los diccionarios.

clear()

El método **clear()** elimina todos los elementos de un diccionario dejándolo vació:

```
diccionario = {'a': 3, 'b': 100, 'c': 'Python'}
diccionario.clear() # {}
```

copy()

El método **copy()** retorna una copia del diccionario original:

```
diccionario = {'a': 3, 'b': 100, 'c': 'Python'}
diccionario.copy() # {'a': 3, 'b': 100, 'c': 'Python'}
```

get()

El método `get()` nos devuelve el valor de una clave pasado como parámetro, de no encontrar la clave, nos devolverá un objeto denominado `None`, el cual es considerado como vacío:

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
diccionario.get('c') # 3
diccionario.get('d') # None
```

Nota que el método `get()` es otra forma de acceder a los elementos de un diccionario.

pop()

El método `pop()` recibe como parámetro una clave del diccionario, lo elimina y devuelve su valor, en caso de no existir la clave nos devolverá un valor de tipo `KeyError`:

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
diccionario.pop('b') # 2
diccionario # {'a': 1, 'c': 3}
diccionario.pop('d') # KeyError
```

items()

El método `items()` nos devuelve una lista de tuplas, cada tupla se compone de dos elementos, el primero será la clave y el segundo, su valor, este método es muy utilizado para recorrer diccionarios dentro de un ciclo `for`, que veremos más adelante:

```
diccionario = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
elementos = diccionario.items()
elementos # [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

keys()

El método `keys()` nos devolverá una lista de elementos, las cuales son las claves del diccionario:

```
diccionario = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
claves = diccionario.keys()
claves # ['a', 'b', 'c', 'd']
```

values()

El método `values()` nos devolverá una lista de elementos, las cuales son los valores del diccionario:

```
diccionario = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
valores = diccionario.values()
valores # [1, 2, 3, 4]
```

Uso de los operadores

Ahora vamos a ver un tema muy importante en cualquier lenguaje de programación como los **operadores**, y de qué manera los podemos utilizar.

Veamos cuales son los diferentes tipos de operadores que existen en Python:

- Los Operadores Aritméticos
- Los Operadores Relacionales o de Comparación
- Los Operadores Lógicos

Los Operadores Aritméticos

Los Operadores Aritméticos son aquellos que nos permiten realizar operaciones matemáticas dentro de nuestros scripts en Python, y claro, si pensaste que en Programación no veías Matemáticas, déjame decirte que estas equivocado. Pero no te preocupes porque ahorita veremos las operaciones más básicas que son sumar, restar, multiplicar y dividir.

Antes de comenzar déjame decirte un secreto que te ayudará mucho a crecer como profesional y a tener mejores oportunidades laborales. Una habilidad que te garantizará tener seguridad financiera es **Entender matemáticas**. Eso te ayudará a resolver problemas y a construir tus propios algoritmos, desarrollar tu pensamiento deductivo y a crear modelos matemáticos para la toma de decisiones que resuelven problemas de manera automatizada. Posteriormente tendrás la oportunidad de especializarte en carreras como **Big Data y Data Science**, **Inteligencia Artificial o Desarrollo e ingeniería**. Y sí, puedes construir todo eso con Python.

Suma

La suma es representado por el signo de más (+) y nos permite sumar dos o más valores:

```
numero = 10
numero2 = 5
resultado = numero + numero2
resultado # 15
```

Resta

La resta es representado por el signo de menos (-) y nos permite restar dos o más valores:

```
numero = 10
numero2 = 5
resultado = numero - numero2
resultado # 5
```

Multiplicación

La multiplicación es representado por el signo asterisco (*) y nos permite multiplicar dos valores:

```
numero = 10
numero2 = 5
resultado = numero * numero2
resultado # 50
```

Potencia

La potencia es representado por 2 veces el signo de asterisco (*) y nos permite elevar un número a la N potencia:

```
numero = 5
potencia = 2
resultado = numero ** potencia
resultado # 25
```

Módulo

El módulo es representado por el símbolo de porcentaje (%) y nos permite indicar el residuo entre dos valores:

```
numero = 8
numero2 = 3
residuo = numero % numero2
residuo # 2
```

Si te das cuenta el operador de módulo nos trae las unidades restantes de la división, es decir, el número **3** solamente cabe dos veces dentro del número **8**, lo que quiere decir **3 * 2 = 6**, entonces el resultado del módulo sería **8 - 6 = 2**.

División entera

En ocasiones el resultado de una división nos devuelve un número flotante (float), pero si queremos que el resultado de una división entre dos valores nos devuelva un número entero, debemos de usar 2 veces la barra diagonal (//):

```
numero = 8
numero2 = 3
resultado = numero // numero2
resultado # 2
```

División normal

La división normal es representado por una barra diagonal (/), pero existe una diferencia entre versiones de Python.

Para **Python 2**, si los dos valores a dividir son números enteros, el resultado devuelve un número entero, pero si al menos uno de los dos valores es un número flotante, el resultado devuelve un número con punto decimal.

Para **Python 3**, no importa que los dos valores a dividir sean números enteros, el resultado que devuelve es un número con punto decimal.

```
# Python 2
resultado = 3 / 2
resultado # 1

# Python 3
resultado = 3 / 1
resultado # 1.5
```

Los Operadores Relacionales o de Comparación

Los operadores relacionales o de comparación son utilizados tanto en las estructuras de control condicionales, como iterativas (como veremos más adelante), y nos sirven para evaluar si dos valores cumplen con una condición, como resultado nos devuelve un valor booleano, es decir, un `True` o un `False`.

Igualdad

El operador de igualdad es representado por 2 veces el signo igual (==) y nos permite comparar si dos valores son iguales.

```
8 == 8 # True
8 == 2 # False
```

Diferente o distinto que

El operador de diferente o distinto que, es representado por el signo de admiración o exclamación, seguido del signo igual (!=) y nos permite comparar si dos valores son diferentes:

```
8 != 8 # False
8 != 2 # True
```

Menor que

El operador menor que, es representado por el signo menor (<) y nos permite comparar si un valor es menor a otro:

```
3 < 5 # True
5 < 3 # False
```

Mayor que

El operador mayor que, es representado por el signo mayor (>) y nos permite comparar si un valor es mayor a otro:

```
5 > 3 # True
3 > 5 # False
```

Menor o igual que

El operador menor o igual que, es representado por el signo menor seguido del signo igual (<=) y nos permite comparar si un valor es menor o igual a otro:

```
10 <= 15 # True
15 <= 15 # True
15 <= 10 # False
```

Mayor o igual que

El operador mayor o igual que, es representado por el signo mayor seguido del signo igual (>=) y nos permite comparar si un valor es mayor o igual a otro:

```
15 >= 10 # True
10 >= 15 # False
```

Los Operadores Lógicos

Los operadores lógicos son utilizados junto con los operadores relaciones y tienen el mismo uso que estos, es decir, trabajan sobre dos operandos y retornan un valor lógico (`True` o `False`).

and

El operador `and` siempre devolverá `True` si ambas expresiones son verdaderas. Para entender mejor el operador `and`, veamos a través de su tabla de verdad en que casos devuelve `True` o `False`:

x	y	resultado
True	True	True
True	False	False
False	True	False
False	False	False

or

El operador `or` siempre devolverá `True` si alguna expresión es verdadera. Veamos su tabla de verdad:

x	y	resultado
True	True	True
True	False	True
False	True	True
False	False	False

En el caso de los operadores `and` y `or` puedes utilizar de igual manera los signos en lugar de las palabras reservadas (`and` y `or`), los cuales son el signo `&` para el operador `and` y el signo `|` para el operador `or`.

Por buena práctica te recomienda trabajar con las palabras reservadas `and` y `or`, ya que eso mejora la legibilidad del código a la hora de leerlo y dar mantenimiento.

not

El operador `not` siempre devolverá el resultado opuesto según el valor booleano. Veamos su tabla de verdad:

x	resultado
True	False
False	True

Más adelante veremos cómo podemos hacer uso de los operadores relacionales y lógicos en las estructuras de control condicionales.

Estructuras de control condicionales

Las estructuras de control condicionales nos permiten controlar el flujo de nuestra aplicación o script dependiendo de lo que estemos comprobando. Primero quiero mostrarte un meme muy visto por Internet:

Al parecer los desarrolladores somos muy literales cuando nos dicen algo.

Si se dieron cuenta en el anterior meme, las condicionales fueron cambiando el control de flujo de las instrucciones, las cuales son:

1. Ir por una leche.
2. Preguntar si hay huevos.
3. Si hay huevos, traer 6 leches.

Vamos a traducir este algoritmo en código Python:

```
1 leche = 1
2 huevos = True
3 if huevos:
4     leche = 6
```

Al principio del algoritmo se establece la variable `leche` a 1, en caso de haber `huevos`, se establecerá `leche` en 6, en caso contrario quedaría con el mismo valor de 1.

Ahora veamos cuales son las estructuras de control condicionales que existen en Python:

Sentencia if

La sentencia `if` permite evaluar si una condición es verdadera, y si es verdadera se ejecuta un conjunto de instrucciones. Veamos su sintaxis:

```
1 if <condición_a_evaluar>:
2     instrucciones_a_ejecutar
```

Uso de la indentación

Toma en cuenta que las instrucciones a ejecutar hacen uso de la indentación, el cual es definida con 4 espacios en blanco. Python hace uso de la indentación para agrupar instrucciones de código que se encuentra dentro de una sentencia anterior, lo que mejora la legibilidad del código. En otros lenguajes de programación como PHP, Java o C# (Por mencionar algunos) hacen uso del signo de llaves (`{}`) para agrupar instrucciones de código que se encuentra dentro de una sentencia.

¿De qué manera determinamos si una persona es mayor de edad? En muchas países, y en mi caso en México, la mayoría de edad se lograr a partir de los 18 años de edad, veamos como podemos plasmar esto con el uso de las condicionales:

```
1 edad = 25
2 if edad >= 18:
3     print('Eres mayor de edad')
```

El ejemplo anterior es muy claro, definimos una variable `edad` con un valor de `25`, esto puede simular la edad que tiene una persona en tiempo de ejecución, luego viene la condicional, que pregunta si `edad` que en este caso vale `25` es mayor o igual a `18`, en caso verdadero nos imprime el mensaje `Eres mayor de edad`.

Uso de los operadores relacionales o de comparación

Si te diste cuenta, en las estructuras de control condicionales se hacen uso de los operadores relacionales o de comparación, los cuales si te acuerdas nos sirven para evaluar si dos valores cumplen con una condición.

Sentencia if/else

Aquí se hace uso de la palabra reservada `else`, la cual ejecutará un conjunto de instrucciones en caso de que la condición que se evaluó en el `if` sea falsa. Veamos su sintaxis:

```
1  if <condición_a_evaluar>:
2      instrucciones_a_ejecutar
3  else:
4      instrucciones_a_ejecutar
```

No puedes usar else sin if

El `else` es un complemento del `if`, es decir, no puedes hacer uso de la palabra reservada `else`, si antes no evalúas una condición con el `if`.

Siguiendo el ejemplo de la mayoría de edad, veamos que mensaje podemos mandarle al usuario, en caso de que no tenga la mayoría de edad:

```
1  edad = 5
2  if edad >= 18:
3      print('Eres mayor de edad')
4  else:
5      print('Eres un niño(a)')
```

Puedes darte cuenta que en el `else` ya no se evalúa una condición, si no simplemente se ejecuta en caso de que la condición del `if` sea falsa.

if anidados

Es posible tener un `if` dentro de otro `if` o dentro de un `else`, a eso se le conoce como **if anidados** y podemos utilizarlo tanto como la lógica de nuestra aplicación lo requiera.

¿De qué manera podemos determinar cuándo un número es cero, positivo o negativo? Vamos a plasmar esto haciendo uso de los **if anidados**:

```
numero = 9
if numero == 0:
    print('El número es cero')
else:
    if numero > 0:
        print('El número es positivo')
    else:
        print('El número es negativo')
```

Si se dieron cuenta la primera condición a evaluar es si `numero` es igual a cero, si la condición es verdadera, se imprime el mensaje `El número es cero`, si la condición es falsa, se hace uso de un **if anidado** dentro del `else`, y ahora se evalúa si `numero` es mayor a cero, si la condición es verdadera, se imprime `El número es positivo`, si es falsa, se imprime `El número es negativo`.

Usa correctamente los if anidados

Por buena práctica no se recomienda hacer uso de los **if anidados** cuando se tiene muchas sentencias condicionales dentro de otras, ya que eso no ayuda a la legibilidad del código, y a la hora de que alguien más o inclusive tú, requiera leer el código generaría confusión. Es por ello que se recomienda que las sentencias deberían estar en el mismo nivel y de manera ordenada.

Sentencia if/elif

En caso de que se requiere hacer varias comprobaciones, podemos hacer uso de la palabra reservada `elif`, la cual nos permite realizar **N** cantidad de comprobaciones adicionales en caso de que la condición que se evaluó en el `if` sea falsa. Veamos su sintaxis:

```
if <condición_a_evaluar>:
    instrucciones_a_ejecutar
elif <condición_a_evaluar>:
    instrucciones_a_ejecutar
```

Usa tantos elif como se requieran

Podemos hacer uso de tantos `elif` como se requieran realizar comprobaciones adicionales. Inclusive el uso de sentencias condicionales puede terminar en un `elif` o en un `else`.

Veamos ahora como podemos mejorar el código que comprobaba si un número es cero, positivo o negativo, haciendo uso del `elif`:

```
numero = 9
if numero > 0:
    print('El número es positivo')
elif numero == 0:
    print('El número es cero')
else:
    print('El número es negativo')
```

Vean cómo mejora la legibilidad del código haciendo uso de las sentencias condicionales en el mismo nivel y de manera ordenada.

Conversión de objetos a booleano

Cuando realices aplicaciones más completas en Python, vas a necesitar comprobar si una lista de usuarios (elementos) esta vacía, si el usuario no dejo vacío un campo del formulario, etc.

Es por ello que en nuestras sentencias condicionales podemos pasar simplemente variables o valores sin hacer uso de los operadores.

Los siguientes valores son evaluados como `False`:

- Una lista vacía (`[]`)
- Una tupla vacía (`()`)
- Un diccionario vacío (`{}`)
- Una cadena vacía (`''`)
- El número cero (`0`)
- El objeto `None`
- El objeto `False`
- Todo lo demás es evaluado como `True`.

Ahora veamos un ejemplo en el que comprobaremos si una lista tiene elementos y mostrarlos mediante un ciclo, en caso de no tener elementos mostrar un mensaje:

```
usuarios = ['Juan', 'Ana', 'Veronica', 'Luis']
if usuarios:
    for usuario in usuarios:
        print(usuario)
else:
    print('No hay elementos para mostrar')
```

No te preocupes si en este momento no entiendes la sintaxis de la sentencia `for`, en el siguiente capítulo veremos más a detalle las estructuras de control iterativas.

Uso de los operadores lógicos en las condicionales

En ocasiones vas a realizar más de una comprobación dentro de una misma sentencia condicional, para ello hacemos uso de los operadores lógicos como ya hemos visto.

Vamos a mostrar con un ejemplo para entender mejor. ¿De qué manera comprobarías si un número es par y si es múltiplo de 4?

En el ejemplo que acabamos de mencionar vemos que tenemos que realizar dos comprobaciones. Una manera de realizarlo sin usar los operadores lógicos es con el uso de **if anidados**:

```
numero = 12
if numero % 2 == 0:
    if numero % 4 == 0:
        print('El número es par y es múltiplo de 4')
```

Para comprobar si un número es par lo tienes que dividir entre 2 y su residuo tiene que dar 0, es lo que hacemos en la primera comprobación, ahora con otro if anidado realizamos la segunda comprobación, para saber si un número es múltiplo de 4 dividiendo el número entre 4 y su residuo debe dar 0.

Realizamos nuestra prueba de escritorio con el número 12 y obtenemos los siguientes resultados:

Comprobación	Resultado
`12 % 2 == 0`	`True`
`12 % 4 == 0`	`True`

Vemos que las dos comprobaciones pasan la prueba y se imprime el mensaje en pantalla `El número es par y es múltiplo de 4`.

El operador and

Ahora veamos de que manera podemos hacer las mismas comprobaciones pero dentro de una misma sentencia condicional usando los operadores lógicos:

```
1  numero = 12
2  if numero % 2 == 0 and numero % 4 == 0:
3      print('El número es par y es múltiplo de 4')
```

Te acuerdas de la tabla de verdad del operador `and`, en donde `True` y `True` da como resultado `True`, es decir, ambas expresiones son verdaderas.

Date cuenta como ahora las dos comprobaciones se encuentran dentro de una misma línea sin hacer uso de los if anidados, esa es la ventaja de usar los operadores lógicos cuando se requieren.

El operador or

Veamos otro ejemplo pero ahora usando el operador `or`, vamos a comprobar si un número es par o es múltiplo de 5:

```
1  numero = 15
2  if numero % 2 == 0 or numero % 5 == 0:
3      print('El número es par o es múltiplo de 5')
```

Realizamos nuestra prueba de escritorio con el número 15 y obtenemos los siguientes resultados:

Comprobación	Resultado
`15 % 2 == 0`	`False`
`15 % 5 == 0`	`True`

Vemos que al menos una de las dos comprobaciones pasan la prueba y se imprime en pantalla `El número es par o es múltiplo de 5`.

Recordemos que la tabla de verdad del operador or nos dice que si algunas de las expresiones es verdadera resultará en `True`.

El operador not

Por último veamos el uso del operador `not`, en donde siempre devolverá el resultado opuesto según el valor booleano:

```
usuarios = ['Juan', 'Ana', 'Veronica', 'Luis']
if not usuarios:
    print('No hay elementos para mostrar')
else:
    for usuario in usuarios:
        print(usuario)
```

Ahora que hemos visto que una lista vacía es considerada como `False` en un contexto booleano, podemos realizar una comprobación para mostrar un mensaje cuando no hay usuarios dentro de la lista, en caso contrario que muestre los usuarios uno por uno mediante un ciclo.

Condicional ternario

Cuando queremos definir el valor de una variable dependiendo de una condición podemos hacer uso del condicional ternario, el cual tiene la siguiente sintaxis:

```
variable = <valor_si_condición_es_verdadera> if <condición_a_evaluar> else <valor_si\
_condición_es_falsa>
```

Veamos esto con un ejemplo en el que vamos a determinar si un número es mayor o menor sobre otro:

```
numero = 3
numero2 = 8

mensaje = 'Es mayor' if (numero > numero2) else 'Es menor'
print(mensaje) # Es menor
```

Hay otra forma de definir un condicional ternario, veamos cual es su sintaxis:

```
variable = expresion1 expresion2
```

En donde `expresion1` puede ser una lista o una tupla con dos elementos, el primer elemento representará al valor cuando la condición sea **falsa**, y el segundo elemento representará al valor cuando la condición sea **verdadera**. La `expresión2` representará la condición mediante corchetes.

Si ya te diste cuenta, el mecanismo de este condicional ternario funciona muy parecido al acceso a una lista o una tupla, en donde la `expresion1` representa a una colección ordenada, es decir, una lista o una tupla, y la `expresion2` es la posición a la que queremos acceder a través de la condición, si la condición es `True` se accede a la posición **1**, y si la condición es `False` se accede a la posición **0**. Veamos el mismo ejemplo para determinar si un número es mayor o menos sobre otro:

```
numero = 3
numero2 = 8

mensaje = ['Es menor', 'Es mayor'][numero > numero2]
print(mensaje) # Es menor
```

Estructuras de control iterativas

Las estructuras de control iterativas, más conocidas, como ciclos, nos permiten ejecutar varias veces un conjunto de instrucciones mientras una condición se cumpla.

Veamos cuales son las estructuras de control iterativas que existen en Python:

while

El `while` nos permite repetir un bloque de código mientras que la condición sea verdadera. Veamos cual es su sintaxis:

```
while <condición_a_evaluar>:
    instrucciones_a_ejecutar
```

¿De qué manera podemos imprimir los primeros 100 números empezando con el 1?

Si recuerdas la función `print()` nos permite imprimir en pantalla un mensaje que se le pase como parámetro, tal vez muchos pensarán hacer algo como esto:

```
print(1)
print(2)
print(3)
...
```

Y así hasta llegar al número 100, pero hacer esto con los ciclos es realmente sencillo. Veamos como podemos resolver esto usando los ciclos:

```
i = 1 # Valor inicial
while i <= 100:
    print(i) # Se imprime el valor
    i = i + 1 # Se incrementa a uno
```

Primero definimos un **valor inicial** con el que queremos empezar, en este caso empezamos con el **número 1**. Luego evaluamos si el valor inicial es menor o igual que 100, en este caso la primera vez que se ejecuta la condición es verdadera, por lo tanto entra dentro del conjunto de instrucciones a ejecutar. Se imprime el valor inicial e incrementa su valor en uno. Y así se repite hasta que la condición sea falsa, es decir, cuando el valor inicial haya llegado a los 100.

Nota que es importante incrementar el valor de la variable `i` en 1, ya que al no hacerlo, estaríamos creando un ciclo infinito, ya que la condición siempre sería verdadera.

Recorrer objetos iterables con while

Y si estabas pensando que para acceder a todos los elementos de una lista o tupla tenias que hacerlo uno por uno. Déjame decirte que gracias a los ciclos esta es una tarea más sencilla. Como podemos hacerlo, veamos un ejemplo:

```
1  i = 0
2  nombres = ['Carlos', 'Agustín', 'Saul']
3  while i < len(nombres):
4      print(nombres[i])
5      i = i + 1
```

La función len()

La función `len()` es nativa de Python y devuelve el número de elementos que contiene una lista, tupla o diccionario.

Recuerda que la primera posición de un objeto iterable (lista, tupla o cadena) es la posición 0, por lo tanto definimos un valor inicial con el número 0, luego definimos una lista de nombres con tres elementos, nos ayudamos con la función `len()`, la cual devuelve el número de elementos de la lista y evaluamos si el valor inicial es menor que la longitud de nuestra lista. La primera vez que se ejecuta la condición es verdadera, por lo tanto entra dentro del conjunto de instrucciones a ejecutar. Se imprime el valor del elemento de acuerdo a la variable `i`, ya que es la que accede a la posición del elemento e incrementa su valor a uno. Y así sucesivamente hasta imprimir todos los elementos de la lista.

¿Qué es un objeto iterable?

Un objeto iterable es un tipo de dato que se puede secuenciar, es decir, que se pueden acceder a sus elementos a través de un índice. Estos pueden ser una lista, tupla, diccionario, cadena, etc.

for

El ciclo `for` es utilizado sobre un objeto iterable, o como ya vimos una secuencia, es decir, sobre una lista, tupla, diccionario, cadena, etc. Veamos su sintaxis:

```
for <variable> in <iterable>:
    instrucciones_a_ejecutar
```

Vamos a visualizar un ejemplo:

```
nombres = ['Carlos', 'Agustín', 'Saul']
for nombre in nombres:
    print(nombre)
```

Si te das cuenta aquí ya no hacemos uso de un valor inicial para indicar la primera posición de la lista, si no que ya el ciclo `for` permite iterar sobre esos elementos a través de la variable `nombre`, donde este toma un valor diferente en cada iteración del ciclo.

La función range()

Aunque en el siguiente capítulo veremos más a detalle el concepto de funciones y si leíste el tema Función xrange VS range[6] del capítulo Python 2 VS Python 3[7], sabrás que la función `range()` nos retorna una secuencia de números enteros al pasarle como argumento un número:

```
list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Pues bien la función `range()` es muy utilizado en ciclos `for`, ¿Cómo haríamos para imprimir los números del 1 al 10? Si recuerdas la secuencia de números empieza desde el cero (`0`), por lo que una solución sería de la siguiente forma:

[6] python-2-vs-python-3.md#funcion-xrange-vs-range
[7] python-2-vs-python-3.md

```
for i in range(10):
    print(i + 1)

1
2
3
4
5
6
7
8
9
10
```

Date cuenta que hemos sumado un `1` para que al momento de imprimir, la secuencia comience desde el uno (`1`).

Si pasamos dos argumentos a la función `range()`, el primero indicará desde donde empezará la secuencia de números y el segundo en donde terminará, pero sumando uno (`1`) a ese número (`n + 1`):

```
for i in range(1, 11):
    print(i)

1
2
3
4
5
6
7
8
9
10
```

Las palabras reservadas break, continue y pass

Las palabras reservadas `break`, `continue` y `pass` nos permiten modificar el comportamiento de un ciclo.

break

La instrucción `break` como lo dice su nombre, rompe la ejecución de un ciclo por completo:

```
while True:
    numero = int(input('Ingrese un número: '))
    if numero % 2 == 0:
        print('El número es par')
    else:
        print('El número es impar')
        break
```

En el código anterior, le pedimos al usuario a través de la función `input()`, la cual permite obtener un texto escrito por teclado, que se ingrese un número como una entrada, y le pasamos la función `int()` para que convierta ese texto en un número entero. Luego a través de una condicional determinamos si un número es par o impar, es decir, con ayuda del operador de módulo (`%`), sacamos el residuo del número introducido por teclado, y si el residuo es `0`, quiere decir que es un número par, en caso contrario el número es impar. Este proceso se repite **N** cantidades de veces hasta que el usuario ingrese un número impar, aquí es donde hacemos uso de la sentencia `break`.

continue

La instrucción `continue` salta la iteración actual sin romper el ciclo por completo:

Vamos a realizar un ejemplo bastante sencillo. Se requiere contar los números del 1 al 5 y mostrarlos en pantalla, pero no queremos mostrar el número 4. Veamos cómo queda:

```
cont = 0
while cont <= 5:
    cont = cont + 1
    if cont == 4:
        continue
    print(cont)
```

Nota que es importante el orden de los factores, ya que si ponemos el incremento de nuestro contador al final de las instrucciones a ejecutar, se puede generar un ciclo infinito, ya que al llegar al número `4` entraría la condición donde se ejecuta la sentencia `continue`, y nunca incrementaría nuestro contador.

pass

En ocasiones tu código todavía estará en construcción, por lo que aún no tienes las instrucciones definidas a ejecutar dentro de un bloque de código. Es por ello que puedes hacer uso de la sentencia `pass`, para indicar que por el momento no se ejecutará ninguna instrucción, es decir, no se ejecutará nada. La sentencia pass puede ser utilizada en las estructuras de control (tanto condicionales como iterativas), en funciones y clases (como veremos más adelante).

```python
# Es utilizado en las estructuras de control
if True:
    pass

while True:
    pass

# De igual manera puede ser utilizado en funciones
def saludo():
    pass

# Y en las clases
class Persona(object):
    pass
```

Funciones

A lo largo del libro hemos estado utilizando algunas funciones como `len()`, `int()`, `input()`, `print()`, las cuales permiten realizar una tarea en específica, y que nos han ayudado en nuestros scripts. Pero esas funciones pertenecen a Python, y en ocasiones tenemos que escribir el mismo bloque de código en varias partes de nuestra aplicación para realizar la misma operación varias veces, la ventaja con las funciones es que escribimos una sola vez ese mismo código y lo reutilizamos en cualquier parte de nuestra aplicación.

Ahora veamos cómo crear nuestras propias funciones.

¿Qué es una función?

Las funciones son bloques de código que realizan una tarea concreta y que podemos reutilizar en cualquier parte de nuestra aplicación o script.

Definir una función

Para definir una función usamos la palabra reservada `def` seguido del nombre de la función, que debe ser un nombre descriptivo, luego abrimos y cerramos paréntesis y de manera opcional podemos definir dentro de los paréntesis uno o más parámetros que serán pasados a la función. Para finalizar colocamos dos puntos. Y de manera indentada escribimos el bloque de código que ejecutará la función. Veamos su sintaxis:

```
1  def <nombre_función>(<parámetro1>, <parámetro2>, ...):
2      instrucciones_a_ejecutar
```

Así como en las estructuras de control, de igual manera en las funciones debes tomar en cuenta la indentación de tu código, es decir, el conjunto de instrucciones que se encuentran dentro de la función.

Por defecto una función devuelve un valor `None`, pero en caso de que queramos que la función retorne un valor distinto, tenemos que hacer uso de la sentencia `return` seguido del objeto a retornar:

```
def <nombre_función>(<parámetro1>, <parámetro2>, ...):
    instrucciones_a_ejecutar
    return objeto
```

Tipos de funciones

De manera general podemos clasificar o identificar 4 tipos de funciones que existen, veamos cuales son:

Funciones que no reciben parámetros y no retornan un valor

Este tipo de función es el más sencillo, ya que no debemos especificar parámetros de entrada a la función, y por defecto al no utilizar la sentencia `return` dentro de la función devuelve `None`. Veamos un ejemplo muy sencillo, vamos a saludar a alguien:

```
def saludo():
    print('Hola')
```

Hasta el momento hemos solamente definido nuestra función, pero por si solo no se ejecuta ninguna acción hasta que mandemos a llamar a nuestra función. Para llamar a nuestra función simplemente escribimos el nombre de nuestra función seguido de paréntesis.

```
saludo() # Hola
```

Funciones que reciben parámetros y no retornan un valor

En este tipo de función, debemos de especificar los parámetros de entrada que tendrá la función, estos pueden ser **N** cantidad de parámetros, tanto como se necesiten utilizar, y deben ser separados con comas. Veamos un ejemplo:

```
def saludo(nombre):
    print('Hola', nombre)

saludo('Python') # Hola Python
```

En los parámetros de una función podemos especificar cualquier tipo de objeto, ya sea una cadena, un número, un valor booleano, una colección e inclusive otra función. Y no necesitamos especificar el tipo de parámetro que tendrá, Python lo detectará automáticamente en tiempo de ejecución.

Funciones que no reciben parámetros y que retornan un valor

En este tipo de función hacemos uso de la sentencia `return`, que como ya hemos visto, puede devolver un valor distinto de `None`. Veamos un ejemplo:

```
def saludo():
    return 'Hola Python'

mensaje = saludo()
print(mensaje) # Hola Python
```

Si nos damos cuenta en el ejemplo anterior, no estamos pasando parámetros a la función, pero si estamos devolviendo un valor, en este caso una cadena, que luego con la variable `mensaje` estamos capturando el valor que regresa de la función, para luego imprimirlo en pantalla.

En estos ejemplos hemos estamos trabajando con funciones que solamente tienen una línea de código, pero lo interesante de las funciones es que pueden ser tan complejas como sean necesarias. El objetivo de definir funciones es que podamos realizar una tarea en concreta y reutilizar ese código, es decir, usar la función cuántas veces queramos, escribiendo una sola vez su funcionalidad.

Funciones que reciben parámetros y que retornan un valor

En este tipo de funciones debemos especificar los parámetros de entrada que tendrá la función. De igual manera hacemos uso de la sentencia `return`, el cual devolverá un valor desde la función.

Vamos a definir una función que permita sumar dos valores pasados como parámetros y devuelva el resultado:

```
def suma(num1, num2):
    return num1 + num2

resultado = suma(10, 5)
print(resultado) # 15
```

En el ejemplo anterior ahora pasamos dos parámetros a la función, los cuales se usan para realizar el proceso de suma, a la vez de que estamos retornando el resultado de la suma.

Diferencia entre parámetros y argumentos

Algo que me confundía mucho era la diferencia entre lo que es un parámetro y lo que es un argumento. Al principio pensaba que se trataba de lo mismo, y aunque esto en cierto punto puede parecer algo irrelevante en conocer, déjame decirte que te servirá mucho al momento de diferenciar cuál es uno y otro.

Bueno pues los parámetros son las propiedades o variables que pasamos a un función al momento de definirla, y que podemos utilizar para realizar las operaciones necesarias que realizará dicha función:

```
def nombre_funcion(<parámetro1>, <parámetro2>, ...):
    instrucciones_a_ejecutar
```

Veamos ahora cuales son los argumentos. Cuando una función es llamada, y en la definición de la función tiene parámetros, los valores que son pasados a dicha función se le conocen como argumentos:

```
nombre_funcion(argumento1, argumento2, ...) # Se ejecuta la función
```

Argumentos claves VS argumentos posicionales

A una función en Python se le puede llamar usando argumentos claves o argumentos posicionales, y en algunos casos, los dos al mismo tiempo.

Pero, ¿A qué se refiere eso de los argumentos claves y argumentos posicionales? Cuando definimos una función y está tiene parámetros, los cuales tienen un nombre de variable asociado, pues en una llamada a una función por argumentos claves, se especifican los nombres de los argumentos junto con sus valores:

```
def suma(num1, num2):
    return num1 + num2

print(suma(num1=10, num2=5))
print(suma(num2=8, num1=3))
```

Nota que mandamos a llamar dos veces a nuestra función suma, y en esta ocasión le estamos especificando los nombres de los argumentos, junto con sus valores. De igual manera date cuenta que no importa el orden en el que pasemos los argumentos a la función, sino con el nombre del argumento estamos especificando a qué variable se le va a asignar dicho valor.

En una llamada a una función con argumentos posicionales solamente pasas los valores a esa función sin especificar los nombres de los argumentos, tomando en cuenta que tienes que respetar el orden en el que fueron listados al momento de definir la función:

```python
def saludo(nombre, edad, pais):
    print('Hola %s tienes %s años y vives en %s' % (nombre, edad, pais))

saludo('José',  25, 'México') # Hola José tienes 25 años y vives en México
```

Date cuenta que cada argumento pasado a la función corresponde con el orden listado en la definición de la función. Para el mismo ejemplo podemos hacer uso de los dos tipos de argumentos al mismo tiempo, tomando en cuenta de que siempre los argumentos posicionales estén listados antes que los argumentos claves:

```python
saludo('José', 25, pais='México')
saludo('José', edad=25, pais='México')
saludo('José', pais='México', edad=25)
```

Parámetros con valores por defecto

En ocasiones van a requerir que uno o más parámetros de una función tengan un valor por defecto, es decir, un valor alternativo, esto permite crear funciones más flexibles y emplearlo para distintas circunstancias.

¿Cómo se definen los parámetros con valores por defecto?

Estos son definidos desde la definición de la función en donde se les asigna un valor, y este valor es utilizado en caso de que en la llamada a la función no sea pasado dicho argumento.

```python
def carrito(producto, precio, cantidad = 1):
    print('Producto:', producto, 'Precio:', precio, 'Cantidad:', cantidad)

carrito('Computadora', 20000, 3) # Producto: Computadora Precio: 20000 Cantidad: 3
carrito('Teclado', 500) # Producto: Teclado Precio: 500 Cantidad: 1
```

En caso de utilizar parámetros con valores por defecto, siempre debemos definirlas al final de todos los parámetros que no tengan valores por defecto.

Si te das cuenta en el ejemplo anterior, en la primera llamada a la función si estamos pasando la **cantidad**, es por ello que en la salida en pantalla nos aparece una **cantidad** de **3**, pero en la segunda llamada a la función no estamos pasando la **cantidad**, al no pasar la **cantidad**, automáticamente se asigna a la variable `cantidad` el valor por defecto definido en la función y sobre ese opera dentro de la misma.

Argumentos variables: *args

Una característica muy interesante de Python es que nos permite usar argumentos variables, veamos con un ejemplo a que me refiero con esa característica:

```
def sumatoria(*args)
    print(sum(args))

sumatoria(1, 2, 3, 4) # 10
sumatoria(10, 5) # 15
```

En Python existe un parámetro especial llamado `*args` (date cuenta del signo `*`) que permite a las funciones pasar de forma dinámica y arbitraria un número de argumentos desconocidos, estos argumentos son posicionales, lo que quiere decir que se comporta como una tupla, veamos una vez más otro ejemplo:

```
def imprimir(*args):
    print('Los argumentos posicionales son:', args)

imprimir(1, 2, 3, 4) # Los argumentos posicionales son: (1, 2, 3, 4)
```

Por convención, cuando utilizas argumentos posicionales variables se le llama al parámetro `args`, pero en realidad puedes ponerle el nombre que tú quieras, siempre y cuando en la definición del parámetro no olvides el signo asterisco (`*`) al principio. Un detalle más a tomar en cuenta es que cuando ya utilizas tu variable `args` dentro de la función no tienes que ponerle el signo asterisco (`*`).

En caso de que tengas más parámetros recuerda tomar en cuenta el siguiente orden:

1. Parámetros sin valores por defecto.
2. Parámetros con valores por defecto.
3. Parámetro especial `*args`.

```
def mi_funcion(parametro1, parametro2 = 'valor', *args):
    pass
```

Argumentos variables: **kwargs

Como ya vimos podemos hacer uso de los argumentos posicionales variables a través del parámetro especial `*args`, pero igual podemos hacer uso de los argumentos claves variables a través del parámetro especial `**kwargs`, este nos permite de igual manera pasar un número desconocido de argumentos clave que se comportará como un diccionario. Veamos un ejemplo:

```python
def imprimir(**kwargs):
    for clave in kwargs.keys():
        print(clave)

imprimir(foo='bar', baz='boise')
# foo
# baz
```

Si recuerdas en el tema **Argumentos claves VS argumentos posicionales**, los argumentos claves son aquellos que en la llamada a una función necesitamos especificar los nombres de los argumentos junto con sus valores.

En el ejemplo anterior cada nombre de argumento representa a la clave del diccionario `kwargs`.

En el siguiente capítulo te adentrarás en el mundo de la Programación Orientada a Objetos, un paradigma de programación muy utilizado en este maravilloso mundo de la programación.

Programación Orientada a Objetos (POO)

Definición

La Programación Orientada a Objetos es un paradigma de programación en el que los objetos del mundo real son modelados en clases y objetos.

Pero antes de avanzar, vamos a ir por paso a paso, para definir primero que es un paradigma de programación.

¿Qué es un paradigma de programación?

En pocas palabras un paradigma de programación es una forma de solucionar uno o varios problemas. Cada forma de solucionar ese problema es un paradigma de programación diferente.

Les pongo un ejemplo para entenderlo mejor, imagínate que quieres llegar a un lugar, y tienes diferentes formas y maneras de hacerlo, una de ellas es elegir el transporte público, que ya manejan rutas definidas, otra opción si quieres llegar más rápido al lugar sería elegir un taxi, en este caso el taxi encontrará la mejor ruta para llegar al lugar. Una opción más económica sería en bicicleta, claro, donde tú mismo escogerás tu ruta. Por último pudieras elegir ir caminando, en todo caso la manera que elijas te llevará al mismo lugar, unas formas más rápidas que otras, pero todo dependerá de la situación en la que te encuentras, tal vez tengas una entrevista de trabajo y necesitas llegar rápido al lugar, en este caso escogerías el taxi.

Con este ejemplo quiero que comprendes que para cada situación deberás escoger la mejor manera de solucionarlo, para eso son los paradigmas de programación, que ya a modo técnico es una propuesta tecnológica adoptada por una comunidad de programadores para resolver un problema claramente delimitado.

Paradigmas de programación que soporta Python

Ahora si te preguntaste que paradigma de programación maneja Python, déjame decirte que Python es un lenguaje de programación multiparadigma, esto quiere decir, que no estamos forzados a escoger un estilo particular de programación, sino que Python nos da la libertad de optar por varios estilos.

Entre los paradigmas de programación que soporta Python son: **programación orientada a objetos**, **programación imperativa**, **programación funcional**, y otros paradigmas son soportados mediante el uso de extensiones.

La Programación Orientada a Objetos en Python

Retomando lo que vimos al principio del capítulo en el que mencionamos que la Programación Orientada a Objetos intenta tomar los objetos del mundo real y llevarlos al mundo del código por medio de clases y objetos para solucionar un problema claramente definido y delimitado, ahora piensa por un momento todas las cosas que ves a tu alrededor, ahorita a lo mejor estes viendo una silla, una mesa, o inclusive tu computadora o smartphone, dependiendo en donde estes leyendo este libro, todos esas cosas que logramos percibir son considerados objetos y cada una de ellas (objetos) pertenecen a una clase. No te preocupes si por el momento no entiendes muy bien, a lo largo de este capítulo iremos aclarando con más detalle cada uno de estos conceptos.

Ventajas de la Programación Orientada a Objetos

Es importante conocer que ventajas nos trae escoger este paradigma de programación:

- Permite la reutilización del código: lo primero que pienso en cuando mencionamos esta primer ventaja es en el principio **DRY**, que significa **no te repitas** (don´t repeat yourself) en el que la idea consiste en evitar la duplicación de código tanto como sea posible, eso nos permite evitar inconsistencias y a la larga facilita el mantenimiento del código.
- Crear sistemas más complejos: al utilizar todos los elementos y características de la POO, nos permite separar en cada componente todas las funcionalidades que tendrá el sistema a construir y poder escalarlo.
- Relaciona el sistema al mundo real: cómo ya mencionamos los objetos del mundo real podemos llevarlos al mundo del código, un ejemplo muy rápido es el de un usuario de una red social que tiene características como usuario, nombre, fecha de nacimiento. Y funciones cómo iniciar sesión, publicar un post, comentar, etc.
- Ayudar a mantenener el software actualizado: Todas las ventajas anteriores hacen de alguna manera más fácil manejar y mantener un sistema.

Elementos y Características de la POO

Llegamos a la parte que tanto estabas esperando en el que mencionaremos los elementos y características de la POO para que empieces a entender cómo es que funciona este paradigma de programación.

Una analogía que me encanta para entender sobre los elementos y características de la POO es del libro **Curso: Python para Principiantes** la cuál es la siguiente: "Los elementos de la POO, pueden entenderse como los "**materiales**" que necesitamos para diseñar y programar un sistema, mientras que las características podrían asumirse como las "**herramientas**" de las cuáles disponemos para construir el sistema con esos materiales.", (Bahit, 2012, p. 53)

Elementos principales de la POO

Entre los elementos principales de la POO, podemos encontrar a:

Clases

Una clase es una plantilla o definición de algo. Para que me entiendas mejor, piensa en una persona como un objeto, esa persona tiene características como un nombre, un apellido, un color de ojos, etc. A todo ese conjunto de propiedades que mencionamos se le denominan atributos. Ahora esa persona realiza ciertas acciones como caminar, comer, leer, etc. A todo ese conjunto de acciones o funciones que realiza un objeto se le llaman métodos.

Si te diste cuenta solamente mencionamos a un conjunto de propiedades y acciones que tiene una persona, pero la realidad es que cada sistema va a abstraer la información necesaria de cada objeto, todo dependerá de las necesidades a satisfacer del sistema que se este construyendo.

Como ejemplo imagina una red social, que es el sistema a construir, el sistema tiene entidades como usuarios, publicaciones, comentarios, etc. Cada uno de estos objetos mencionados tiene sus propias características y acciones. Por mencionar las de un usuario son su nombre de usuario, su correo electrónico, su número telefónico, etc. Por otra parte sus acciones serían iniciar sesión, publicar un post, comentar una publicación, ¿Te diste cuenta de algo?, Si te diste cuenta habrás notado que igual existen relaciones entre objetos. Viste que maravilloso es la POO.

Retomando el ejemplo del objeto persona, observa que en ningún momento mencionamos a una persona específica ni tampoco mencionamos cuales son sus propiedades de esa persona, es decir, si la persona se llama Juan, si se apellida Pérez, si tiene ojos cafés, ojos azules, etc. Sino solamente mencionamos la definición de ese objeto, es decir, definimos una plantilla, una base, de la cual podemos crear objetos con características particulares, eso es una clase.

Ahora vamos a ver como se define una clase, su sintaxis es la siguiente:

```
class <NombreClase>(object):
    <atributos>
    <métodos>
```

Primero definimos la palabra reservada `class`, para indicar que se trata de una clase, seguido del nombre de la clase en upper camelcase, luego entre paréntesis y dentro de ellos ponemos la palabra `object`, para finalizar colocamos dos puntos, y de manera indentada definimos los atributos y métodos que tendrá la clase. Veamos un ejemplo:

```
class Persona(object):
    pass
```

Usa upper camelcase

El nombre de una clase por convención debe estar en upper camelcase, la cual es una recomendación del PEP8 en el que indica que la primera letra de cada palabra debe comenzar con mayúscula.

Por el momento no definimos los atributos y métodos de la clase en el ejemplo anterior, porque primero quiero que avancemos paso por paso, hasta que lleguemos a esos conceptos.

Atributos

Son aquellas cualidades o características que describen a nuestros objetos y son representadas como variables. Los atributos diferencian a un objeto de otro y pueden tomar valores que pueden ser cambiados o no en tiempo de ejecución. Veamos como definimos los atributos dentro de una clase:

```
1 class Persona(object):
2     nombre = ''
3     apellido = ''
4     color_ojos = ''
```

Como dijimos, cada atributo es representado como variables, por lo tanto, en nuestros atributos podemos hacer uso de cadenas, enteros, listas, tuplas, diccionarios, inclusive otros objetos, que como ya vimos, se puede tener relación entre objetos:

```
1 class Usuario(object):
2     usuario = ''
3     correo_electronico = ''
4     numero_telefonico = ''
5 
6 class Post(object):
7     usuario = Usuario()
8     contenido = ''
9     fecha_publicacion = ''
```

Métodos

Los métodos son la acción o función que realiza un objeto y que pueden cambiar su estado sobre sí mismo, es decir, sobre sus atributos. Los métodos son representados como funciones:

```
class Persona(object):
    nombre = ''
    apellido = ''
    color_ojos = ''

    def mostrar_nombre(self):
        print(self.nombre)
```

El parámetro self

Como observas en el método `mostrar_nombre()` definido en la clase `Persona` tiene un parámetro llamado `self`, este parámetro se refiere a un nombre convencional para el primer parámetro de un método en Python, la cual su tarea es la de realizar una referencia hacia el mismo objeto permitiendo acceder a su atributos y métodos dentro de la clase. Por lo tanto si defines un método `accion(self, a, b, c)` debe ser llamado como `x.accion(a, b, c)`, por la instancia x.

Objeto

Cuando ya hacemos uso de la plantilla o clase, es decir, que definimos sus características particulares, es cuando hacemos uso de los objetos.

Un objeto no es más que la instancia de una clase, y para que un objeto pueda existir primero necesitamos crearlo, es decir, cuando se le asigna un espacio de memoria de la computadora a ese objeto.

¿Cómo creamos un objeto? Necesitamos definir una variable y asignar la clase como un valor:

```
persona = Persona()
```

> Primero que nada debe existir la clase para poder crear un objeto de dicha clase

Ahora veamos algo que quedo pendiente por ver, cómo poder acceder a los atributos y métodos de un objeto. Veamos un ejemplo y luego su explicación:

```
persona = Persona()

persona.nombre # ''
persona.apellido # ''
```

Primero creamos un objeto de la clase Persona, luego desde el objeto creado con la notación de punto (.) accedemos a los atributos de la clase. En caso de querer cambiar los valores de los atributos, asignamos un valor de la siguiente forma:

```
persona.nombre = 'José'
persona.apellido = 'Pérez'
```

Ahora veamos cómo podemos acceder a los métodos de una clase. Al igual que en las funciones, para llamar a un método simplemente escribimos desde el objeto creado luego de la notación de punto (.) el nombre de nuestro método seguido de paréntesis:

```
persona.mostrar_nombre() # José
```

Características de la POO

Entre las características de la POO, podemos encontrar a:

Herencia

Es un mecanismo que permite construir una clase a partir de otra heredando u obteniendo todos sus atributos y métodos. No podemos negar que nos han dicho que nos parecemos a nuestros padres. Y eso es cierto ya que heredamos de ellos ciertos rasgos físicos, que podemos compararlos con los atributos o propiedades de una clase, y eso no es todo, a veces (casi siempre) aprendemos comportamientos que hacen nuestros padres (métodos).

La ventaja de la herencia es que la clase hija (la clase que hereda de otra), no solamente obtiene sus atributos y métodos de su clase padre, sino que puede definir los suyos propios, lo que la hace diferente. Lo mismo pasa con nosotros, aunque heredamos ciertos rasgos físicos e imitamos ciertos comportamientos de nuestros padres, nosotros tenemos los propios, lo que nos hacen diferentes, inclusive de nuestros hermanos.

Vamos a poner un poquito más divertido esto, y veamos un ejemplo basado en un juego de estrategia para que conozcas como se define la herencia en Python:

```
class Unidad(object):
    nombre = ''

    def mover(self, direccion):
        print('Mover hacia', direccion)

class Soldado(Unidad):
    def atacar(self):
        print('Ataque')

class Arquero(Unidad):
    def disparar(self):
        print('Disparo')
```

La clase object

Cuando una clase no hereda de ninguna, por defecto heredara de la clase `object`, que es la clase principal de Python que define un objeto.

Como te habrás dado cuenta definimos una clase `Unidad`, esta clase es considerada como la clase base o superclase ya que de ahí heredamos sus atributos y métodos para construir la clase `Soldado` y la clase `Arquero`. Y cada clase hija a su vez define sus propios métodos.

Herencia múltiple

Una de las sorprendentes características de Python y que no todos los lenguajes de programación manejan, es la herencia múltiple, que consiste en no solamente heredar de una clase sino heredar de varias clases a la vez.

El más claro ejemplo que tenemos de la herencia múltiple es de nosotros mismos, ya que heredamos tanto características de nuestro padre como de nuestra madre:

```
1  class Padre(object):
2      pass
3
4  class Madre(object):
5      pass
6
7  class Hijo(Padre, Madre):
8      pass
```

Atributos y métodos con el mismo nombre

En el caso de que las clases padres tuvieran atributos y métodos con el mismo nombre, la clase que este más a la izquierda es la que sobreescribirá los atributos y métodos iguales.

Polimorfismo

El polimorfismo se refiere a que un objeto puede comportarse de varias formas, es decir, realizamos las acciones de diferente manera, pero la esencia es la misma.

Pongamos un ejemplo para aclarar esta característica de la POO. Una persona, un perro, un pato y un pez pueden realizar la acción de nadar, pero todos lo hacen de diferente forma.

Este concepto podemos aplicarlo de diferentes maneras dependiendo del lenguaje de programación, en Python la manera de aplicarlo es simplemente tomar el nombre del método que queremos modificar de la superclase e implementarlo de manera distinta en la subclase:

```
class Unidad(object):
    def nadar(self):
        print('Nadar')

class Persona(Unidad):
    def nadar(self):
        print('Nado con mis brazos y piernas')

class Perro(Unidad):
    def nadar(self):
        print('Nado con mis 4 patitas')

class Pato(Unidad):
    def nadar(self):
        print('Nado con mis 2 patitas')

class Pez(Unidad):
    def nadar(self):
        print('Nado con mis aletas')
```

En el ejemplo anterior definimos nuestra superclase `Unidad`, que podemos entenderlo como algo muy abstracto en la que puede significar algo para ti o inclusive nada, pero la idea es que para nuestro ejemplo lo tomamos simplemente como una unidad de algo, eso es todo, luego definimos las subclases `Persona`, `Perro`, `Pato` y `Pez` que implementan el mismo método `nadar()` de la superclase `Unidad` pero con instrucciones distintas, a este comportamiento se le conoce como sobreescritura de métodos.

Encapsulamiento

El encapsulamiento o encapsulación es el nivel de visibilidad que permite controlar el acceso a los atributos y métodos de una clase.

En lenguajes de programación como PHP, Java o C# hacen uso de los modificadores de acceso como `public`, `protected` y `private` para aplicar este concepto de la POO. En Python simplemente ponemos dos guiones bajos (__) antes de los nombres de aquellos atributos y métodos que queremos que sean privados, es decir, que no puedan ser accesibles desde fuera de la clase:

```
class Persona(object):
    __nombre = 'Juan'

    def mostrar_nombre(self):
        print(self.__nombre)

persona = Persona()
persona.__nombre # AttributeError: 'Persona' object has no attribute '__nombre'
persona.mostrar_nombre() # Juan
```

En el ejemplo anterior intentamos acceder directamente al atributo `__nombre` desde el objeto `persona`, pero nos marca un error de tipo `AttributeError` indicando que el objeto de la clase `Persona` no tiene un atributo `__nombre`, pero si intentamos acceder al atributo `__nombre` mandando a llamar al método `mostrar_nombre()`, todo funciona como esperamos, ya que el método es de acceso público, es decir, que puede ser accesible desde fuera de la clase.

¿Qué pasaría si ahora ponemos el método mostrar_nombre como privado?:

```
class Persona(object):
    __nombre = 'Juan'

    def __mostrar_nombre(self):
        print(self.__nombre)

persona = Persona()
persona.__nombre # AttributeError: 'Persona' object has no attribute '__nombre'
persona.__mostrar_nombre() # AttributeError: 'Persona' object has no attribute '__mo\
strar_nombre'
```

Como esperabas, el método no es accesible desde el objeto `persona` y ahora tanto el atributo `__nombre` como el método `__mostrar_nombre()` solamente podrán ser usados dentro de la clase.

El método __init__()

En ocasiones vas a requerir inicializar los valores de los atributos de un objeto al momento de crearlo. El objetivo del método `__init__()` es precisamente realizar esa tarea.

Para entenderlo mejor vamos a retomar el ejemplo del juego de estrategia en donde creamos las clases `Unidad`, `Soldado` y `Arquero`, como habrás notado, la clase `Unidad` tiene un atributo `nombre`, que al mismo tiempo lo tiene la clase `Soldado` y `Arquero` ya que heredan de la misma clase `Unidad`, sería muy buena idea poder inicializar ese nombre, ¿Por qué? Si has jugado videojuegos seguro habrás notado que en la mayoría de ellos te piden iniciar con un nombre, es decir, en cierta forma se esta

construyendo el objeto para poder iniciar el juego, ese es el objetivo del método `__init__()`, ser el constructor de la clase para poder inicializar los valores de un objeto.

La ventaja de utilizar el método `__init__()` es que es llamado automáticamente al momento de crear un objeto, por lo que no tenemos que olvidarnos de llamarlo. Veamos su sintaxis:

```
class <NombreClase>(object):
    def __init__(<parámetro1>, <parámetro2>, ...):
        instrucciones_a_ejecutar
```

Definimos un método llamado `__init__()`, con dos guiones bajos al principio y al final de la palabra **init**, y opcionalmente le pasamos los parámetros a usar dentro del método. Ya que vimos como se define su sintaxis ahora veamos un ejemplo con el mismo juego de estrategia:

```
class Unidad(object):
    nombre = ''

    def __init__(self, nombre):
        self.nombre = nombre
```

Los parámetros en el método __init__()

Al constructor `__init__()` podemos pasarle los parámetros que sean necesarios con la finalidad de poder inicializar los atributos de un objeto.

Ahora que tenemos definido nuestro constructor de la clase `Unidad`, es el momento de crear un objeto en donde debemos pasarle el nombre:

```
unidad = Unidad('SmoothieMX')
```

Y a la vez podemos acceder al atributo `nombre`:

```
unidad.nombre # SmoothieMX
```

La función super()

Ya hemos visto que Python incorpora muchas funciones para realizar una tarea muy concreta. En esta ocasión quiero mencionar la función `super()`, que prácticamente permite llamar un método de una clase padre desde una clase hija. Veamos un ejemplo práctico basado en el juego de estrategia:

```
class Unidad(object):
    nombre = ''

    def __init__(self, nombre):
        self.nombre = nombre

    def mover(self, direccion):
        print('Mover hacia', direccion)

class Soldado(Unidad):
    def __init__(self, nombre, armadura = False):
        super(Soldado, self).__init__(nombre)
        self.armadura = armadura

soldado = Soldado('Buzz', True)
```

Ahora hemos incorporado el atributo `armadura` en la clase `Soldado`, y estoy seguro que te diste cuenta de un detalle, vemos que el atributo `armadura` no se encuentra definido al comienzo de la clase, sino que se encuentra directamente dentro del método `__init__()`, pues bien, no te preocupes ya que desde cualquier método podemos crear más atributos utilizando `self`.

De igual manera vemos que desde el método `__init__()` mandamos a llamar a la función `super()` con dos argumentos, el primero hace referencia a la subclase `Soldado`, y el segundo argumento se refiere a un objeto, que en este caso es `self`. Luego con la notación de punto (`.`) se manda a llamar al método constructor `__init__()` y le pasamos cómo argumento el atributo `nombre`. En pocas palabras, esto quiere decir, manda a llamar al método `__init__()` de la clase padre de `Soldado`, es decir, de la clase `Unidad`. Y esto ejecuta el constructor de la clase `Unidad` que vemos solamente recibe el parámetro nombre.

La ventaja con la función `super()` es que se pueden llamar otras clases, no directamente tiene que ser la propia clase hija. Veamos un ejemplo aplicado a las figuras geométricas. Primero vamos a construir nuestra clase base `Rectangulo`:

```
class Rectangulo(object):
    def __init__(self, ancho, altura):
        self.ancho = ancho
        self.altura = altura

    def area(self):
        return self.ancho * self.altura
```

De manera rápida vemos que definimos dos atributos (`ancho` y `altura`) y definimos un método para calcular el área. Ahora vamos a construir la clase `Cuadrado` a partir de la clase `Rectangulo`:

```
class Cuadrado(Rectangulo):
    def __init__(self, longitud):
        super(Cuadrado, self).__init__(longitud, longitud)
```

Lo único que modificamos fue el constructor de la clase `Cuadrado` ya que el área se calcula de la misma manera que un rectángulo con la diferencia que el cuadrado es una figura que tiene todos sus lados iguales, por lo tanto el ancho y la altura serán la misma.

Ahora por último vamos a construir un cubo, que prácticamente es una figura de 6 caras y cada cara esta formada de un cuadrado:

```
class Cubo(Cuadrado):
    def area_total(self):
        area_cara = super(Cuadrado, self).area()
        return area_cara * 6
```

Por lo tanto para calcular el área de un cubo tenemos que sacar primero el área de una cara y luego a ese resultado multiplicarlo por 6, ya que es el número de caras totales que tiene un cubo.

Quiero que te des cuenta que en el método `area_total()` hacemos uso de la función `super()`, solamente que en esta ocasión no pasamos como argumento directamente la clase `Cubo`, sino que usamos la clase `Cuadrado`, es decir, esto hace que `super()` comience a buscar el método `area()` en un nivel superior a la clase `Cuadrado`, entonces, ¿Cuál es ese nivel superior de `Cuadrado`? Pues precisamente la clase `Rectangulo` que es padre de la clase `Cuadrado`.

Módulos y paquetes

A medida que va creciendo tu código es importante mantener una estructura y un orden que te ayudará a encontrar de forma más rápida un error, crear una nueva funcionalidad concreta, refactorizar tu código, etc.

En otras palabras es poner cada cosa en su lugar. Si bien sabes dentro de los archivos de Python (`.py`) podemos escribir dentro variables, funciones, clases, etc.

Ahora te hago la siguiente pregunta, ¿Qué pasaría si tienes un archivo de 1000 líneas de código y por alguna razón te sale un error en la línea 900? Sería algo tedioso y cansado revisar y tener que leer todas esas líneas de código para ver en dónde se encuentra el error. Ves a dónde quiero llegar.

En el mundo laboral la mayor parte del tiempo te la pasarás leyendo código y un porcentaje muy pequeño escribiendo código.

En Python podemos organizar nuestro código en módulos y paquetes. Bien ahora veamos su definición.

Módulos

Cada archivo con extensión `.py` se considera un módulo, el cual puede contener variables, funciones y/o clases. De esa manera pasamos de un archivo con muchas líneas de código a crear varios módulos para organizar las funcionalidades concretas de nuestra aplicación.

Ahora veamos un ejemplo, tenemos un sistema de notificaciones en donde tenemos los siguientes módulos:

- `correo.py`
- `mensaje.py`
- `whatsapp.py`

Si te diste cuenta, cada módulo representa la manera en que se puede enviar notificaciones, ya sea por correo electrónico, por mensaje de texto o por Whatsapp.

Definir un punto de entrada

Ahora que trataremos con varios archivos, es decir, con varios módulos, debemos definir un punto de entrada a nuestra aplicación, qué es el lugar donde empezará la ejecución de instrucciones de nuestro código Python.

En otros lenguajes de programación como Java, C++ y/o C# definen un método `main()` como punto de entrada a la aplicación. Cuando ejecutamos un archivo con el intérprete de Python, este archivo será leído de arriba hacia abajo ejecutando todas las instrucciones que se encuentren a su paso.

Cada módulo (archivo de Python) tiene una variable especial llamada `__name__` que su valor dependerá desde dónde es ejecutado dicho módulo.

Si el módulo es ejecutado como el principal, es decir, si ejecutamos directamente el archivo con el intérprete de Python desde la consola la variable `__name__` tendrá el valor de `'__main__'`.

Vamos a comprobarlo, crea un archivo llamado `modulo.py` en cualquier ubicación de tu sistema de archivos, en donde tendrá el siguiente código:

```
1  def main():
2      print(__name__)
3
4  if __name__ == '__main__':
5      main()
```

Manten organizado tus carpetas

Asegurate de tener organizado tus carpetas en los proyectos que estas trabajando, es importante mantener un orden a la hora de trabajar.

Ahora vamos a ejecutarlo desde el intérprete de Python de la siguiente manera (solamente asegurate de estar ubicado en el mismo lugar en donde creaste tu archivo de Python):

```
1  python modulo.py
```

Verás el siguiente resultado en pantalla:

```
1  __main__
```

Eso significa como habíamos dicho anteriormente, que estamos ejecutando el archivo `modulo.py` como el principal por lo tanto la variable `__name__` toma el valor de `'__main__'`.

Ahora te preguntarás que significa las siguientes líneas:

```
1 if __name__ == '__main__':
2     main()
```

Prácticamente la línea `if __name__ == '__main__':` significa que estamos ejecutando el módulo como el principal lo cual ejecutará todas las líneas de código que están dentro del condicional. De igual manera nos sirve como un punto de entrada a nuestra aplicación. Ahora veamos cómo podemos utilizar un módulo dentro de otro.

Formas de importar módulos

La ventaja de separar nuestro código en módulos es que podemos reutilizar la funcionalidad de un módulo dentro de otro. Veamos de qué manera podemos importar un módulo dentro de otro.

import

La palabra reservada `import` nos permite importar un módulo dentro de otro, es decir, usar el contenido que se encuentra dentro de un archivo de Python (variables, funciones y/o clases) y utilizarlo en otro archivo.

Veamos el siguiente ejemplo, tenemos un módulo llamado `principal.py` que será nuestro punto de entrada en nuestra aplicación y tenemos otro módulo llamado `modulo.py` que tendrá las funciones y clases necesarias que utilizaremos en el `principal.py`.

Entonces para importar `modulo.py` dentro de `principal.py` lo hacemos de la siguiente manera:

```
1 import modulo
```

De esa manera estamos importando todo el contenido que se encuentra dentro de `modulo.py`, es decir, que ahora podemos usar todas las variables, funciones y/o clases que se encuentran dentro de dicho módulo.

Debes tomar en cuenta que los dos módulos se deben encontrar al mismo nivel de tu estructura de archivos, es decir, que se ubiquen dentro de la misma carpeta.

Al momento de importar un módulo no se tiene que especificar la extensión `.py`

Ahora para usar el contenido dentro del módulo `principal.py` lo hacemos de la siguiente manera:

```
modulo.funcion() # Si queremos usar una función

modulo.Clase() # Si queremos crear un objeto de dicha clase
```

Ves ahora la potencia que tienen los módulos al organizar tu código de mejor manera. Cada cosa debe ir en su lugar.

De igual podemos importar varios módulos al mismo tiempo separándolos con el signo coma (,) de la siguiente manera:

```
import modulo1, modulo2, modulo3
```

from

La segunda forma de importar un módulo dentro de otro es haciendo uso de la palabra reservada `from`, a diferencia de `import`, `from` solamente carga lo necesario, es decir, no carga todo el contenido del módulo sino solamente lo que le indiquemos. Veamos su uso:

```
from modulo import funcion, Clase
```

De esa forma solamente estaremos cargando el contenido `funcion` y `Clase`.

Paquetes

Ya resolvimos el detalle de tener todo nuestro código dentro de un mismo archivo, con ayuda de los módulos ahora podemos organizarlos de mejor manera. Pero ahora te imaginas tener muchos módulos dentro de una sola carpeta. No sería para nada organizado y sería tedioso tener que buscar un módulo en donde existen muchos de ellos.

Es por ello que podemos organizar nuestros módulos en carpetas que son llamadas paquetes, que nos permite agrupar los módulos que guardan una relación. Veamos un ejemplo, tenemos el sistema de notificaciones que mencionamos anteriormente, el cuál tenia como módulos: `correo.py`, `mensaje.py` y `whatsapp.py`, pues bien podemos organizarlos dentro de un paquete (carpeta) llamada `notificaciones`, teniendo en cuenta la relación que tienen entre módulos.

¿Cómo crear un paquete?

La utilidad de los paquetes tiene una enorme ventaja al tener mejor organizado un conjunto de carpetas de acuerdo a las funcionalidades de una aplicación. Para crear un paquete lo hacemos de la siguiente forma:

- Crea una carpeta con un nombre que indique la relación que tienen entre módulos.
- Coloca dentro de la carpeta todos los módulos que guarden una relación.
- Crea un archivo llamado `__init__.py`, por el momento no importa si su contenido se encuentra vacío.

¿Cómo acceder a un paquete?

De la misma forma en la que accedemos a un módulo por medio de las palabras reservadas `import` y `from`, podemos hacer lo siguiente:

Importar un paquete completo

Al importar un paquete completo tenemos acceso a todos los módulos de dicho paquete:

```
1 import paquete
```

Importar un módulo de un paquete

Podemos importar un módulo específico de un paquete:

```
1 import paquete.modulo
```

Importar varios módulos de un paquete

De igual manera podemos ser más específicos de que módulos queremos importar:

```
1 from paquete import modulo1, modulo2, modulo3
```

Importar funciones y/o clases de un módulo dentro de un paquete

Si no nos interesa importar el módulo completo y solamente queremos importar ciertas funciones y/o clases específicas lo hacemos de la siguiente manera:

```
1 from paquete.modulo import funcion, Clase
```

Manejo de errores

Durante la ejecución de una aplicación pueden ir surgiendo errores de diferentes tipos, los dos más comunes son los errores de sintaxis y las excepciones.

Errores de sintaxis

A medida que vamos escribiendo código podemos cometer un error de dedo y escribir algo que para Python no es entendible, es ahí donde surgen los errores de sintaxis.

El intérprete de Python detecta que algo no esta escrito correctamente y de inmediato nos lanza el error indicando el nombre del archivo, la línea en donde se encuentra el error y el tipo de excepción.

Este tipo de errores de sintaxis se resuelven simplemente verificando la sintaxis y escribiendo de manera correcta en donde nos haya indicado el error.

Las excepciones

Aunque tu código no tenga errores de sintaxis, se pueden generar errores intentando ejecutar alguna operación dentro de tu aplicación, como intentar conectarte a una base de datos, división por cero, intentar convertir un tipo de dato a otro, etc.

Esos tipos de errores son conocidos como excepciones y a menos que sean manejados por nosotros, Python simplemente nos lanzará un mensaje de error y se interrumpe la ejecución de nuestro script.

Manejo de excepciones

En una aplicación es importante manejar este tipo de errores mencionados anteriormente ya que la hace más segura. Como así mismo podemos ejecutar ciertas acciones adicionales antes de mostrar un mensaje de error al usuario para informarle de lo que esta pasando, lo más común es guardar en un log todo lo que esta sucediendo dentro de nuestra aplicación.

Los logs son registros de eventos que suceden dentro de nuestra aplicación. Por lo general estos registros son guardados en un archivo que posteriormente podemos consultar para rastrear si se produce algún error.

Veamos un ejemplo, imagina que hay un policía en un retén verificando los permisos o licencias de conducir, ese (el policía) sería nuestro manejador de excepciones, ya que estamos verificando si el

conductor cuenta con sus papeles vigentes para poder circular por las vías públicas. Si cumple con toda su documentación vigente puede seguir su camino, en caso de no contar con su documentación o que no se encuentre vigente, se le pondría una multa y se llenaría un registro del suceso (log).

Lo mismo sucede con el manejo de excepciones, se pone dentro de un bloque especial aquello que puede causar una excepción, si nada dentro del bloque causo una excepción sigue con la ejecución normal del script, en caso de lanzar una excepción, es atrapada y se ejecutan un par de instrucciones para manejarla.

Para manejar una excepción en Python tenemos la siguiente sintaxis:

```
try:
    pass
except:
    pass
```

Definimos la palabra reservada `try` seguido de dos puntos (`:`), luego de forma indentada ira el bloque de código que puede causar una excepción. Luego se define la palabra reservada `except` que permite tratar la excepción si esta es producida, despues de colocar los dos puntos (`:`) y de forma indentada puede ir un mensaje más amigable para el usuario o registrar los errores producidos.

Veamos un ejemplo, al intentar dividir un número entre cero se le considera en aritmética o álgebra como algo indefinido, por lo tanto en Python nos marcaría un error. Por lo tanto tenemos que validar esa operación:

```
def division(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        print(resultado)
    except:
        print('Error!')

division(15, 3) # 5.0
division(3, 0) # Error!
division(4, 'Hola') # Error!
```

Al usar la función `division` en distintos casos vemos que nos arroja distintos resultados, en el primer llamado se hace la operación `15 / 3` lo cual todo sale de manera correcta y nos arroja el resultado de la división. En los otros dos casos nos arroja un error, en el segundo llamado es porque estamos dividiendo un número entre cero la cuál se considera una excepción de tipo `ZeroDivisionError`, y en el tercer llamado estamos pasando como segundo argumento una cadena por lo tanto se considera una excepción de tipo `TypeError` lo que quiere decir que no son datos compatibles para realizar la división.

Si te has dado cuenta en los últimos dos llamados a la función `division` simplemente nos arroja el mensaje `Error!`, pero podemos ser más específicos al indicarle a Python el tipo de excepción que vamos a manejar y por lo tanto arrojarle al usuario mensajes más amigables sobre el error sucedido:

```
def division(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        print(resultado)
    except ZeroDivisionError:
        print('El divisor debe ser diferente de cero')
    except TypeError:
        print('Los argumentos deben ser números')
```

La ventaja del manejo de excepciones es que podemos validar los datos introducidos por el usuario o simplemente validar operaciones que sabemos que pueden fallar como ejemplo conectarse a una base de datos.

De igual manera podemos agrupar tipos de excepciones similares de la siguiente manera:

```
try:
    pass
except (TipoExcepcion1, TipoExcepcion2, TipoExcepcion3):
    pass
```

Sentencia finally

En ocasiones vas a requerir realizar acciones de limpieza al utilizar algún recurso en nuestra aplicación de Python, como al intentar abrir y leer un archivo de texto. La finalidad de la sentencia `finally` es ejecutarse siempre, haya surgido una excepción o no. Por lo tanto en el ejemplo en el que intentamos abrir y leer un archivo, al final tenemos que cerrar dicho archivo, haya surgido una excepción o no.

Veamos simplemente la estructura usando la sentencia finally, ya que más adelante en el capítulo de manejo de archivos quiero enseñarte esto más a detalle.

```
try:
    # Sentencias o instrucciones que pueden arrojar una excepción
    pass
except:
    # Se arroja la excepción
    pass
finally:
    # Siempre se ejecutará, haya surgido una excepción o no
    pass
```

Manejo de archivos

¿Qué es un archivo en Python?

Un archivo es un conjunto de datos que ocupa un determinado espacio en nuestro disco duro.

Existen dos formas de acceder a un archivo en Python, la primera es utilizarlo como un archivo de texto, que puede ser procesado línea por línea. La otra forma es tratarlo como un archivo binario, que será procesado byte por byte.

Cuando accedemos a un archivo en Python 2 se crea una instancia de la clase `file`, en el caso de Python 3 se crea una instancia de la clase `TextIOWrapper` que se encuentra dentro del módulo `io` nativo de Python.

Lectura y escritura de archivos en Python

Para leer y escribir un archivo usamos la función `open()`, que recibirá dos parámetros:

1. El primer parámetro es una cadena que indicará la ruta y el nombre del archivo.
2. El segundo parámetro es una cadena que indicará el modo de apertura del archivo. Este parámetro es opcional y si no le pasamos ningún valor por defecto abrirá el archivo en modo lectura y en modo texto.

Al intentar leer y escribir en un archivo se puede generar una excepción que debemos manejarla dentro de un bloque `try` y `except`, posteriormente se haya generado una excepción o no, debemos de cerrar el archivo dentro de un bloque `finally`. Veamos un ejemplo:

```
try:
    f = open('file.txt')
except:
    print('Hubo un error al abrir el archivo')
finally:
    if not f.closed:
        f.close()
```

Existe una mejor forma de manejar archivos y es usando la sentencia `with`, que es un manejador de contexto y que se utiliza cuando ciertas operaciones siempre van en par, como cuando abrimos un archivo queremos cerrarlo al final de ser procesado.

Modos de apertura de un archivo

La finalidad de acceder a un archivo es hacer algo con ese archivo, se puede querer abrirlo para leerlo, escribir, para crear uno nuevo, etc.

El modo que le indiquemos determinará las acciones que podremos realizar sobre dicho archivo. Veamos los modos de apertura posibles que existen:

Indicador	Modo de apertura
r	Solo lectura
w	Solo escritura. Sobreescribe el archivo si existe. Crea el archivo si no existe
a	Añadido (Agregar contenido). Crea el archivo si éste no existe
r+	Lectura y escritura

Lectura de archivos

Como habíamos dicho la mejor forma de manejar archivos es usando la sentencia `with`. Veamos como podemos leer un archivo y lo procesaremos línea por línea:

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

Usando la sentencia `with` nos aseguramos que abrimos y cerramos el archivo sin especificar el método `close()` del objeto `File`. Luego con la función `open()` abrimos el archivo con nombre `file.txt`, y en el segundo argumento estamos indicando que abrimos el archivo en modo lectura. Para finalizar esa línea recibimos el archivo en la variable `f`, para luego ser procesado línea por línea a través de un ciclo `for`.

La ventaja de leer archivos es que podemos realizar ciertas operaciones con ellos como la búsqueda de texto.

Escritura de archivos

Para escribir en un archivo debemos de tener mucho cuidado, ya que si el archivo existe hay riesgo de perder toda la información que ya tenía dicho archivo. Ahora si queremos escribir en un archivo que no existe, Python lo va a crear. Veamos un ejemplo en el que escribiremos en un archivo los números del 1 al 100:

```
with open('numeros.txt', 'w') as f:
    for i in range(100):
        f.write(str(i + 1))
```

Añadir texto al final de un archivo

Para evitar sobreescribir un archivo que ya existe podemos usar el modo de apertura `a` que permite agregar contenido al final del archivo sin necesidad de borrar el contenido anterior. En caso de no existir, Python lo va a crear. Veamos un ejemplo de ello:

```
with open('numeros.txt', 'a') as f:
    f.write('Se añadieron 100 números al archivo numeros.txt')
```

Procesar archivos CSV

Muchas veces tu aplicación va a requerir importar datos a través de un archivo para alimentar tu base de datos y en otras ocasiones necesitarás exportar datos para generar un archivo como reporte que pueda ser leído por una aplicación de hojas de cálculo.

Python cuenta con módulo nativos para procesar archivos CSV, es decir, cómo leer y escribir datos a un archivo CSV, pero antes de seguir avanzando veamos que es un archivo CSV.

¿Qué es un archivo CSV?

Un archivo CSV (valores separados por comas) tiene organizado sus datos en forma tabular (muy parecido a una hoja de cálculo de Excel), es decir, en filas y columnas, y tienen la extensión `.csv`. Como ya hemos mencionado es muy utilizado en aplicaciones para importar y exportar datos.

Módulos para procesamiento de archivos CSV

El módulo `csv` nativo de Python tiene varios métodos y clases para leer y escribir archivos CSV.

Método reader

Para leer los datos de un archivo CSV usamos el método `reader()` del módulo `csv` de la siguiente manera:

```
import csv

with open('datos.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Fíjate que el argumento que recibe el método reader() es el archivo que contiene las líneas de texto que queremos leer.

Para procesar cada línea que contiene el archivo usamos el ciclo `for`, que por cada iteración que hace cada registro o fila de datos se convierte en una lista de cadenas de texto.

Método writer

Para escribir datos en un archivo CSV usamos el método writer() del módulo csv de la siguiente manera:

```
import csv

with open('datos.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerow(('Título 1', 'Título 2', 'Título 3', 'Título 4'))
```

El método writer() devuelve un objeto para empezar a escribir en el archivo CSV, y con el método writerow de ese mismo objeto escribimos los valores de cada fila pasando como argumento una tupla, donde cada elemento de la tupla representa una columna.

Ahora imaginate las posibilidades que puedes hacer para empezar a construir archivos CSV para crear reportes. Puedes consultar tu base de datos y crear reportes personalizados según se requiera.

En el siguiente capítulo veremos el manejo de las bases de datos en el que nos conectaremos a una base de datos y haremos ciertas operaciones para insertar, seleccionar, actualizar y eliminar datos.

Manejo de base de datos

¿Qué es una base de datos?

Las bases de datos es una colección masiva de datos que se encuentran de forma organizada para su posterior uso.

Si eres nuevo en este mundo de las bases de datos te voy a explicar los componentes que conforman una base de datos.

Componentes de una base de datos

Tablas

Las tablas son un componente de las bases de datos que contienen todos sus datos. Se organizan en forma de filas y columnas (Muy similar a una hoja de cálculo de Excel).

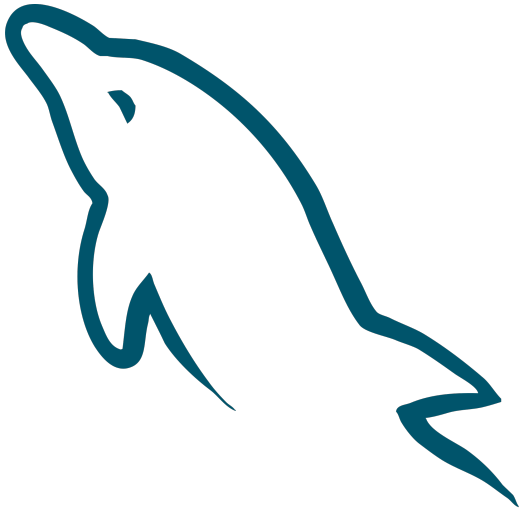
Cada fila representa a un registro único dentro de la tabla y cada columna un dato de ese registro. Veamos un ejemplo para que quede más claro. Tenemos una tabla que contiene los datos de los empleados de una empresa, por cada fila representa a un empleado de la empresa y las distintas columnas representan los detalles de ese empleado, como su nombre, apellido, edad, país y email.

Sistemas Gestores de Bases de Datos

Un sistema gestor de bases de datos es un software que se encarga de administrar nuestras bases de datos, en este software podemos crear la estructura de nuestra base de datos, insertar datos a nuestras tablas, consultar esos datos para generar reportes que produzcan información detallada y crear usuarios definiendo sus roles y permisos para acceder a cierta información asignada. Veamos los sistemas gestores de bases de datos más utilizados en la industria tecnológica.

MySQL

Es el SGBD más utilizado en sitios y aplicaciones web. Actualmente cuenta con dos licencias, la GPL que es muy utilizada en aplicaciones de software libre y la licencia de Uso comercial (con la compra de Oracle a Sun Microsystems, adquirió a Java y MySQL) que es para empresas con productos privativos.



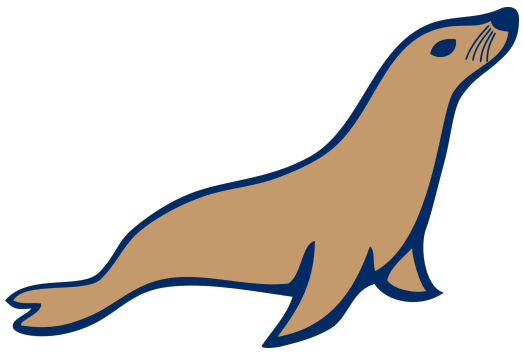
Algunas de las ventajas de MySQL son:

- Su facilidad de instalación y configuración.
- Su facilidad de uso.
- Soporte multiplataforma.

La principal desventaja de MySQL es la escalabilidad, lo que no permite trabajar con bases de datos muy grandes.

MariaDB

Es un fork o copia de MySQL y cuenta con muchas de la características de este. Cuenta con una licencia GNU General Public License lo cuál sigue con la filosofía de Open Source que tenía MySQL antes de la adquisición por parte de Oracle.



Algunas de la ventajas de MariaDB son:

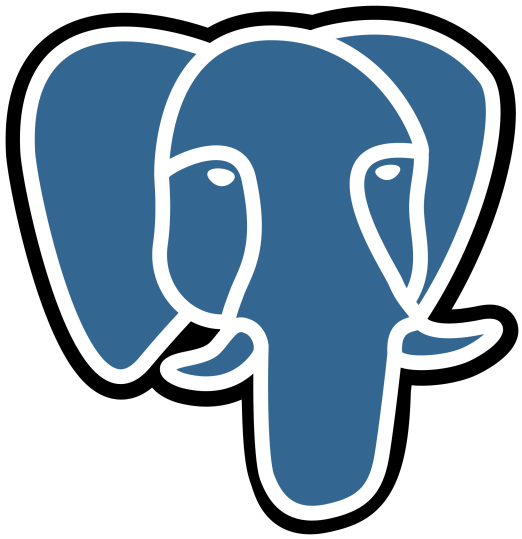
- Es compatible con MySQL.

- Mejoras de velocidad en consultas complejas.
- Seguridad y rapidez en transacciones.

La desventaja más notable en MariaDB es algunas pequeñas incompatibilidad de migración con MySQL.

PostgreSQL

Es el sistema gestor más avanzado del mundo, como bien menciona el mismo PostgreSQL en su sitio oficial ya que permite trabajar con bases de datos relacionales orientadas a objetos y es libre.



Algunas de las ventajas de PostgreSQL son:

- Es multiplataforma.
- Tiene su propia herramienta llamada pgAdmin para su administración de las bases de datos.
- Es robusto, eficiente y puede manejar una gran cantidad de datos.

La principal desventaja de PostgreSQL es la lentitud en operaciones como la inserción o actualización en bases de datos pequeñas, ya que PostgreSQL esta optimizado para manejar grandes volúmenes de datos.

Microsoft SQL Server

Es un sistema propietario de Microsoft y es capaz de atender a una gran cantidad de usuarios de manera simultánea.

Algunas de las ventajas de Microsoft SQL Server son:

- Soporte exclusivo por parte Microsoft.
- Escalabilidad, estabilidad y seguridad.
- Tiene un entorno gráfico bastante potente para administrar las bases de datos.
- Su principal uso es en Windows pero puede ser utilizado en Linux o Docker.
- Tiene soporte en la nube con Microsoft Azure.

Su principal desventaja es el precio, pero existe una versión gratuita (Express) que tiene ciertas limitaciones pero muy bueno para empezar.

SQLite

No es considerado un SGBD, pero es muy utilizado para entornos de desarrollo, la ventaja es que no tienes que realizar ningún tipo de instalación y configuración como lo harías con un servidor de base de datos.

Algunas de las ventajas de SQLite son:

- No ocupa mucho espacio en disco al tratarse de una biblioteca.
- Permite el uso de transacciones.
- Gran portabilidad y rendimiento.

La gran desventaja de SQLite es la escalabilidad ya no soporta grandes cantidades de datos, por lo que no es recomendado para entornos de producción.

Clientes de bases de datos

Para utilizar un sistema gestor de bases de datos necesitamos de un cliente, ya que los primeros responden al modelo cliente/servidor, en donde nuestro SGBD ya sea MySQL, MariaDB, PostgreSQL, Microsoft SQL Server es nuestro servidor y para conectarse a ellos y poder administrarlos necesitamos de un cliente. Existen distintos tipos de clientes para poder conectarnos a nuestro SGBD.

Clientes gráficos

Una manera de trabajar con nuestras bases de datos es a través de una interfaz gráfica, la ventaja de utilizar este tipo de cliente es que tenemos toda una serie de componentes como campos de texto, botones, listas, etc. que hacen la administración de una forma más intuitiva y fácil de manejar. Algunos clientes gráficos son:

- phpMyAdmin (MySQL)
- HeidiSQL (MySQL, PostgreSQL, Microsoft SQL Server)
- pgAdmin (PostgreSQL)

Clientes de consola

Es una alternativa a los clientes gráficos, todas las operaciones es a través de comandos. Es ideal y muy utilizado cuando trabajamos con servidores que no cuentan con una interfaz gráfica. Prácticamente puedes acceder a un SGBD desde cualquier consola de tu sistema operativo. Algunas de las consolas utilizados son:

- Consola de Linux
- El Símbolo del Sistema de Windows
- PowerShell de Windows
- iTerm para MacOS

Lenguajes de programación

Es el momento que estabas esperando, y así como pensaste, otra manera de conectarnos a nuestra base de datos es través de un lenguaje de programación, es nuestro caso utilizaremos Python, para ello es necesario contar con el driver correspondiente al SGBD para poder trabajar con nuestra base de datos.

El único driver que ya viene instalado junto con Python es para SQLite por lo que simplemente debemos importar el módulo a nuestro script. Para los otros drivers tenemos que instalarlos.

Trabajando con SQLite en Python

Como ya hemos mencionado que el driver para SQLite ya viene por defecto con Python, por lo que estaremos utilizandolo para manejar nuestra base de datos.

Crear una conexión

Lo primero que debemos de hacer para conectarnos a nuestra base de datos es importar el módulo de SQLite y luego establecer una conexión que nos permitirá ejecutar sentencias SQL:

```
1 import sqlite3
2
3 conexion = sqlite3.connect('database.db')
4 conexion.close()
```

Luego de ejecutar el método `connect()` del módulo `sqlite3` establecerá una conexión a nuestra base de datos creando un nuevo archivo `database.db` si este no existe.

Algo importante a tomar en cuenta como pasa con el manejo de archivos es de no olvidarnos de cerrar nuestra conexión a la base de datos al terminar de utilizarla, eso lo hacemos con el método `close()` del objeto `conexion`.

Creando un cursor

Un cursor nos permitirá ejecutar sentencias SQL, para ello a través del objeto `conexion` mandamos a llamar al método `cursor`:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()
conexion.close()
```

Crear una tabla

Para crear una tabla tenemos que ejecutar el método `execute()` del objeto `cursor` en donde le pasaremos como argumento la consulta CREATE TABLE:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()
cursor.execute("""
    CREATE TABLE empleados(
        id int primary key not null,
        nombre text not null,
        apellido text not null,
        edad int not null,
        pais text null,
        email text null
    );
""")
conexion.commit()

conexion.close()
```

Cuando se involucren cambios en nuestra base de datos siempre debemos confirmar esos cambios con el método `commit()` del objeto `conexion`. Utilizamos las tres comillas dobles para indicar una cadena que ocupará varias líneas.

Insertar datos en una tabla

Para insertar datos en una tabla utilizaremos el mismo método `execute()` del objeto `cursor` en donde le pasaremos como argumento la consulta `INSERT INTO`:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()
cursor.execute("""
    INSERT INTO empleados(id, nombre, apellido, edad, pais, email)
        VALUES(1, 'Isaac', 'Cantún', 40, 'México', 'isaac@example.com')
""")
cursor.execute("""
    INSERT INTO empleados(id, nombre, apellido, edad, pais, email)
        VALUES(2, 'Carolina', 'Centurión', 24, 'México', 'carolina@example.com')
""")
cursor.execute("""
    INSERT INTO empleados(id, nombre, apellido, edad, pais, email)
        VALUES(3, 'Alejandro', 'Martínez', 25, 'México', 'alejandro@example.com')
""")
conexion.commit()

conexion.close()
```

Seleccionar datos de una tabla

Ya que tenemos datos en nuestra tabla, ahora podemos consultar esos datos a través de la sentencia SQL SELECT que se utiliza para seleccionar datos de una tabla. En esta ocasión tenemos que crear una variable que tomará los datos consultados a través del método `execute()` del objeto `cursor`, y recorreremos esos datos a través de un ciclo `for`:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()
consulta = cursor.execute("""
    SELECT id, nombre, apellido, edad, pais, email
    FROM empleados
""")

for item in consulta:
    print(item[0], item[1], item[2], item[3], item[4], item[5])

conexion.close()
```

Actualizar datos de una tabla

Para actualizar los datos de una tabla tenemos que utilizar la sentencia SQL UPDATE dentro del método `execute()` del objeto `cursor`:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()

cursor.execute("""
    UPDATE empleados
    SET edad = '47'
    WHERE id = 1
""")

conexion.commit()
conexion.close()
```

Eliminar datos de una tabla

Para eliminar los datos de una tabla tenemos que utilizar la sentencia SQL DELETE FROM dentro del método execute() del objeto cursor:

```
import sqlite3

conexion = sqlite3.connect('database.db')
cursor = conexion.cursor()

cursor.execute("""
    DELETE FROM empleados
    WHERE id = 1
""")

conexion.commit()
conexion.close()
```

Recapitulación

Si te gustó el libro, no dudes compartirlo para que más personas aprendan programación en este mundo maravilloso de Python.

Comparte el libro en Facebook haciendo clic aquí[8]

Y en Twitter con el hashtag #AprendiendoPython[9]

Espero que a tráves de este libro puedas inspirarte, para seguir aprendiendo, descubras tu poder, alcances tus metas profesionales, y #NuncaTeDesPorVencido.

¿Te gustaría aprender más sobre tecnologías web y programación?

- Síguenos en Facebook como Guardianes del Código[10]
- Síguenos en Twitter[11]
- Suscríbete a nuestro canal en YouTube[12]

[8] https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fleanpub.com%2Faprendiendo-python

[9] https://twitter.com/intent/tweet?text=%23AprendiendoPython%20con%20el%20ebook%20de%20@3jonapumares%20encu%C3%A9ntralo%20en%20@leanpub%20https://leanpub.com/aprendiendo-python%20&source=clicktotweet&related=clicktotweet

[10] https://www.facebook.com/guardianesdelcodigo

[11] https://twitter.com/GdelCodigo

[12] https://www.youtube.com/channel/UCEHOhZVd3_jrhAkllOmU6qA