DJANGO

Tabla de contenidos

Introducción

Tabla de contenidos

Django es un framework gratuito, es un marco de desarrollo de web de alto nivel y de código abierto que fomenta el desarrollo rápido y limpio en Python.

Las características claves de Django se centran en la eficiencia y simplicidad. Proporciona un conjunto robusto de herramientas y componentes que permiten a los desarrolladores construir aplicaciones web de manera rápida y con un código claro y mantenible.

El marco sigue el patrón de diseño de Modelo-Vista-Controlador(MVC), aunque en Django se le conoce más como Modelo-Vista-Plantilla(MVT). En este patrón, el modelo representa los datos y la lógica de la aplicación, la vista se encarga de la presentación y la plantilla maneja la representación de los datos. La filosofia de Django se basa en la simplicidad, la reutilización de código y el prinicpio de no te repitas.

Django está diseñado para ser rápido, seguro y escalable. Ofrece una serie de características que facilitan el desarrollo de aplicaciones complejas, como por ejemplo:

- Un sistema de plantillas potente y flexible que permite crear interfaces de usuario atractuvas y fáciles de usar.
- Un sistema de gestión de base de datos integrado que facilita el acceso a los datos.
- Una serie de librerías y herramientas que facilitan el desarrollo de tareas comunes, como la autenticacióin de usuarios, la gestión de errores y la depuración.

Django es una buena opción para el desarrollo de aplicaciones web de cualquier tipo, pero es especialmente adeacuado para aplicaciones web controladas por datos, como sitios de comercio electrónico blogs y aplicaciones de gestión.

Instalación Django

Tabla de contenidos

Para instalar Django, necesitarás cumplir con algunos requisitos previos. A continuación, se detallan los pasos básicos para instalar Django en un entorno de desarrollo.

- Requisitos para instalar Django.
    - Actualizar el sistema: para ello debemos abrir la terminal y escribir el siguiente fragmento de codigo en la terminal.

```
sudo apt update
sudo apt upgrade
```

    - Comprobamos la versión de python que tenemos.

```
pythom3 --version
```

- Si no tenemos python instalado en nuestro equipo debemos escribir en la terminal el siguiente código:

```
sudo apt install python3
```

Tenemos que tener en cuenta que la versión de python que tenemos instalada debe ser la 3.6 o superior.

- Ahora empezaremos con la isntalación de django.
    - Primero crearemos una carpeta donde vamos a establecer nuestro entorno gráfico de django. En mi caso la creare en documentos.

```
mkdir django
```

    - Nos moveremos a dicha carpeta:

```
cd django
```

    - Crear un entorno virtual llamado "django":

```
python3 -m venv django
```

Este comando crea un entorno virtual de Python llamado "django". Un entorno virtual es un ambiente aislado donde puedes instalar paquetes de Python sin afectar a otros proyectos o al sistema Python global.

    - Instalar el paquete python3.10-venv:

```
sudo apt install python3.10-venv
```

Este comando instala el módulo venv para Python 3.10 en un sistema operativo basado en Debian, como Ubuntu. venv es un módulo que viene con Python 3.3 y versiones posteriores, y se utiliza para crear entornos virtuales.

    - Crear nuevamente un entorno virtual llamado "django"

```

```

```
python3 -m venv django
```

Aquí, estás volviendo a crear el entorno virtual. Si ya creaste uno en el primer paso, este comando no es necesario. Puedes omitirlo si ya tienes un entorno virtual creado.

- Instalar Django dentro del entorno virtual:

```
pip install Django
```

El comando pip install Django se utiliza para instalar Django, que es un marco de trabajo de desarrollo web de alto nivel en Python. pip es el gestor de paquetes de Python y se utiliza para instalar y administrar paquetes de software escritos en Python.

- Comprobamos la versión de Django que tenemos

```
python3 -m django --version
```

Crear un proyecto de Django

- Crear un proyecto de Django llamado "hola_mundo":

```
django-admin startproject hola_mundo
```

Este comando crea un proyecto de Django llamado "hola_mundo". Un proyecto de Django es una colección de configuraciones y aplicaciones para un sitio web específico. Django viene con un comando de administración que se utiliza para crear proyectos de Django.

hola_mundo/ manage.py hola_mundo/ **init**.py settings.py urls.py asgi.py wsgi.py

Explicamos que consiste cada archivo que se ha creado mediante comando

- Explicamos que consiste cada archivo que se ha creado mediante comando:
  - hola_munsdo/: es el directorio raíz del proyecto. Contiene el archivo manage.py y el paquete hola_mundo.
  - manage.py: es un script de utilidad que se utiliza para administrar el proyecto. Con manage.py, puedes iniciar un servidor de desarrollo, crear y aplicar migraciones de base de datos, crear usuarios, etc.
  - hola_mundo/: es un paquete de Python que contiene los archivos de configuración del proyecto.
  - **init**.py: es un archivo vacío que indica a Python que el directorio es un paquete de Python.
  - settings.py: contiene la configuración del proyecto.

        - urls.py: contiene las URL del proyecto.
        - asgi.py: contiene la configuración para el servidor ASGI.
        - wsgi.py: contiene la configuración para el servidor WSGI.

    - Moverse al directorio "hola_mundo":

```
cd hola_mundo
```

Este comando te mueve al directorio "hola_mundo" que se creó en el paso anterior.

Ejecutar el servidor de desarrollo

```
- Ejecutar el servidor de desarrollo:
```console
python3 manage.py runserver
```

Este comando ejecuta el servidor de desarrollo de Django. El servidor de desarrollo es un servidor web ligero que se utiliza para probar tu aplicación web durante el desarrollo. Por defecto, el servidor de desarrollo se ejecuta en el puerto 8000.

- Abrir el navegador web y navegar a http://localhost:8000:
```console
http://localhost:8000
```

Este comando abre el navegador web y navega a http://localhost:8000. El servidor de desarrollo de Django se ejecuta en el puerto 8000 de forma predeterminada. Si el puerto 8000 está ocupado, puedes especificar un puerto diferente con el siguiente comando:
```console
python3 manage.py runserver 8080
```
Este comando ejecuta el servidor de desarrollo en el puerto 8080.

- Detener el servidor de desarrollo:
```console
Ctrl + C
```
Este comando detiene el servidor de desarrollo de Django.

- Podemos cambiar los puertos de escucha de nuestro servidor de desarrollo, para ello debemos escribir el siguiente comando:
```console
python3 manage.py runserver 8080
```
Este comando ejecuta el servidor de desarrollo en el puerto 8080.
```

Crear una aplicación de Django

Creando la aplicación hola mundo:

- Crear una aplicación de Django llamada "hola_mundo":
```console
python3 manage.py startapp hola_mundo
```

Este comando crea una aplicación de Django llamada "hola_mundo". Una aplicación de Django es un conjunto de código que se utiliza para realizar una tarea específica. Por ejemplo, una aplicación de Django puede ser un blog, un sitio web de comercio electrónico, un sitio web de noticias, etc.

- Los archivos que se han creado son los siguientes:
```console
hola_mundo/
    manage.py
    hola_mundo/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
    hola_mundo/
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        views.py
```

- Explicamos que consiste cada archivo que se ha creado mediante comando:

  - hola_mundo/: es el directorio raíz de la aplicación. Contiene el archivo models.py y el paquete hola_mundo.
  - __init__.py: es un archivo vacío que indica a Python que el directorio es un paquete de Python.
  - admin.py: contiene la configuración para la interfaz de administración.
  - apps.py: contiene la configuración de la aplicación.
  - models.py: contiene los modelos de la aplicación.
  - tests.py: contiene las pruebas de la aplicación.
  - views.py: contiene las vistas de la aplicación.

- Abrir el archivo hola_mundo/settings.py:
- Añadir 'hola_mundo' a la lista INSTALLED_APPS:
```console
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'hola_mundo',
]
```

Este comando añade la aplicación "hola_mundo" a la lista INSTALLED_APPS en el archivo hola_mundo/settings.py. INSTALLED_APPS es una lista de aplicaciones de Django que se utilizan en el proyecto.

- Abrir el archivo hola_mundo/urls.py:
- Añadir la siguiente línea al archivo hola_mundo/urls.py:

```console
path('', include('hola_mundo.urls')),
```

Este comando añade la ruta de la aplicación "hola_mundo" al archivo hola_mundo/urls.py. La ruta de la aplicación "hola_mundo" se añade a la lista urlpatterns en el archivo hola_mundo/urls.py. urlpatterns es una lista de rutas de URL que se utilizan en el proyecto.

- Crear un archivo llamado urls.py en el directorio hola_mundo:

```console
touch hola_mundo/urls.py
```

Este comando crea un archivo llamado urls.py en el directorio hola_mundo.

- Abrir el archivo hola_mundo/urls.py:
- Añadir el siguiente código al archivo hola_mundo/urls.py:

```console
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

Este comando añade la ruta de la aplicación "hola_mundo" al archivo hola_mundo/urls.py. La ruta de la aplicación "hola_mundo" se añade a la lista urlpatterns en el archivo hola_mundo/urls.py. urlpatterns es una lista de rutas de URL que se utilizan en el proyecto.

- Crear un archivo llamado views.py en el directorio hola_mundo:

```console
touch hola_mundo/views.py
```

```

Este comando crea un archivo llamado views.py en el directorio hola_mundo.

- Abrir el archivo hola_mundo/views.py:
- Añadir el siguiente código al archivo hola_mundo/views.py:
```console
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hola Mundo!")
```
Este comando añade una vista llamada index al archivo hola_mundo/views.py. Una vista es una función de Python que procesa una solicitud web y devuelve una respuesta web. En este caso, la vista index devuelve una respuesta web que dice "Hola Mundo!".

- Ejecutar el servidor de desarrollo:
```console
python3 manage.py runserver
```
Este comando ejecuta el servidor de desarrollo de Django. El servidor de desarrollo es un servidor web ligero que se utiliza para probar tu aplicación web durante el desarrollo. Por defecto, el servidor de desarrollo se ejecuta en el puerto 8000.

- Abrir el navegador web y navegar a http://localhost:8000:
```console
http://localhost:8000
```

Este comando abre el navegador web y navega a http://localhost:8000. El servidor de desarrollo de Django se ejecuta en el puerto 8000 de forma predeterminada. Si el puerto 8000 está ocupado, puedes especificar un puerto diferente con el siguiente comando:
```console

python3 manage.py runserver 8080
```

Este comando ejecuta el servidor de desarrollo en el puerto 8080.

- Detener el servidor de desarrollo:
```console
Ctrl + C
```
Este comando detiene el servidor de desarrollo de Django.

Instalación PyCharm
```

Tabla de contenidos

- Introducción sobre PyCharm:

  Es un entorno de desarrollo integrado (IDE) creado por JetBrains específicamente para el desarrollo de proyectos en Python.

  - Características:

    - Editor de código avanzado: ofrece un editor de código robusto con resaltado de sintaxis, completado automático, refactorización de código y navegación inteligente.
    - Gestión de Proyectos Eficiente: Facilita la creación y gestión de proyectos Python. Ofrece herramientas para la instalación y gestión de paquetes, y permite trabajar con entornos virtuales para aislar las dependencias de cada proyecto.
    - Depuración Integrada: permite establecer puntos de interrupción, inspeccionar variables.
    - Integración con Herramientas Externas: en este caso podríamos incluir un sistema de versión de controles como es git y base de datos como mysql.

- Instalación de PycCharm:

  - Para la instalación necesitamos acceder al siguiente sitio web https://www.jetbrains.com/es-es/pycharm/ y ahi descargar pycharm, una vez descargado escribiremos en consola la siguiente línea de comando:

  Descomprimir el Archivo de PyCharm:

```
tar -zxf pyCharm-professional-2023.3.1.tar.gz
```

  Ahora crearemos un alias para poder acceder a pyCharm más rapido

```
nano source ~/.bashrc
```

  Una vez dentro al final del documento debemos escribir la siguiente línea para crear un alias:

```
alias pyCharm="la ruta donde se encuentra este archivo"pyCharm-
2023.3.1/bin/pycharm.sh
```

  Actualizar la Configuración del Shell:

```
source ~/.bashrc
```

Sintaxis de Django

Tabla de contenidos

Python utiliza la sangría para indicar bloques de código. Django utiliza llaves, corchetes y paréntesis para indicar bloques de código. Por ejemplo, las llaves se utilizan para indicar bloques de código en las plantillas de Django. Los corchetes se utilizan para indicar bloques de código en los archivos de configuración de Django. Los paréntesis se utilizan para indicar bloques de código en las vistas de Django.

Un ejemplo de de sangría en python:

```
if 5 > 2:
  print("Cinco es mayor que dos!")
```

Comentarios

- Comentarios:
  - Comentarios de una línea:

```
{# Este es un comentario de una línea #}
```

  - Comentarios de varias líneas:

```
{% comment %}
  Este es un comentario de varias líneas.
{% endcomment %}
```

Variables

- Variables: Una variable se crea en el momento en que se le asigna un valor. Las variables se utilizan para almacenar datos, como cadenas de texto, números enteros, números de punto flotante, etc. En Django, las variables se utilizan para almacenar datos que se utilizan en las plantillas. Las variables de Django se crean utilizando la sintaxis {{variable}}. Por ejemplo, la variable {{nombre}} se utiliza para almacenar el nombre de un usuario.

```
x = 5
y = "Hola Mundo!"
print(x)
print(y)
```

  print se utiliza para imprimir en pantalla el valor de una variable en la consola.

- Obtener el tipo de dato de una variable: Python tiene varios tipos de datos estándar, pero también podemos definir nuestros propios tipos de datos personalizados. Para obtener el tipo de dato de una variable, podemos usar la función type():

```
x = 5
y = "Hola Mundo!"
print(type(x))
print(type(y))
```

- Las variables de cadena de texto se pueden declarar de varias formas:

```
x = "Hola Mundo!"
# es lo mismo que
x = 'Hola Mundo!'
```

- Las variables numéricas se declaran de la siguiente forma:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

- Python diferencia sus variables entre mayúsculas y minúsculas, por lo que las variables x y X son diferentes. Por ejemplo:

```
a = 4
A = "Sally"
#A es una variable diferente a a
```

- Podemos nombrar las variables de diferente forma:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

- Las variables no pueden comenzar con un número:

```
2myvar = "John"
```

- Las variables no pueden tener espacios:

```
my var = "John"
```

- Booleanos: Las variables booleanas solo pueden tener dos valores: Verdadero o Falso.

```
x = True
y = False
```

- Las variables booleanas a menudo se utilizan para comparar valores:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

- Las variables booleanas se devuelven como resultado de comparaciones:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Casting en variables

- Casting: Si queremos especificar el tipo de dato de una variable, podemos usar la función de casting:

```
x = str(3)    # x será '3'
y = int(3)    # y será 3
z = float(3)  # z será 3.0
```

Obtención de tipo de datos de variablesQ

- Obtener el tipo de dato de una variable: Python tiene varios tipos de datos estándar, pero también podemos definir nuestros propios tipos de datos personalizados. Para obtener el tipo de dato de una variable, podemos usar la función type():

```
x = 5
y = "Hola Mundo!"
print(type(x))
print(type(y))
```

Asignación de multiples variables

- Asignar múltiples valores a múltiples variables: Python le permite asignar valores a múltiples variables en una línea:

```
x, y, z = "Naranja", "Plátano", "Cereza"
print(x)
print(y)
print(z)
```

- Asignar el mismo valor a múltiples variables: Python le permite asignar el mismo valor a múltiples variables en una línea:

```
x = y = z = "Naranja"
print(x)
print(y)
print(z)
```

Concatenación de variables

- Podemos utilizar la función + para combinar variables de cadena de texto:

```
x = "Python es "
y = "increíble"
z =  x + y
print(z)
```

También podemos utilizar la funcion + en el print para concatenar variables de cadena de texto:

```
x = "Python es "
y = "increíble"
print(x + y)
```

- Para los números, el operador + funciona como una operación matemática:

```
x = 5
y = 10
print(x + y)
```

- Si intentamos combinar una cadena de texto y un número, Python nos dará un error:

```
x = 5
y = "John"
print(x + y)
```

Funciones globales y variables globales

- Podemos utilizar la función global() para crear una variable global, incluso si se crea dentro de una función:

```
x = "awesome"

def myfunc():
print("Python is " + x)

myfunc()
```

- Si se crea una variable con el mismo nombre dentro de una función, esta será local y solo se podrá utilizar dentro de la función. La variable global con el mismo nombre seguirá existiendo como variable global.

```
x = "awesome"

def myfunc():
x = "fantastic"
print("Python is " + x)

myfunc()

print("Python is " + x)
```

- Para crear una variable global podemos utilizar la palabra clave global:

```
def myfunc():
global x
x = "fantastic"

myfunc()

print("Python is " + x)
```

- También podemos cambiar el valor de una variable global dentro de una función:

```
x = "awesome"
```

```
def myfunc():
global x
x = "fantastic"

myfunc()

print("Python is " + x)
```

Tipos de datos

- Tipos de datos en python:
  - Texto: str
  - Numérico: int, float, complex
  - Secuencia: list, tuple, range
  - Mapeo: dict
  - Set: set, frozenset
  - Booleano: bool
  - Binario: bytes, bytearray, memoryview
  - Tipos de datos:

Tipo de Dato	**Ejemplo**	**Ejemplo con Constructor**
str	`x = "Hello World"`	`x = str("Hello World")`
int	`x = 20`	`x = int(20)`
float	`x = 20.5`	`x = float(20.5)`
complex	`x = 1j`	`x = complex(1j)`
list	`x = ["apple", "banana", "cherry"]`	`x = list(("apple", "banana", "cherry"))`
tuple	`x = ("apple", "banana", "cherry")`	`x = tuple(("apple", "banana", "cherry"))`
range	`x = range(6)`	`x = range(6)`
dict	`x = {"name" : "John", "age" : 36}`	`x = dict(name="John", age=36)`
set	`x = {"apple", "banana", "cherry"}`	`x = set(("apple", "banana", "cherry"))`

Tipo de Dato	Ejemplo	Ejemplo con Constructor
frozenset	`x = frozenset({"apple", "banana", "cherry"})`	`x = frozenset(("apple", "banana", "cherry"))`
bool	`x = True`	`x = bool(5)`
bytes	`x = b"Hello"`	`x = bytes(5)`
bytearray	`x = bytearray(5)`	`x = bytearray(5)`
memoryview	`x = memoryview(bytes(5))`	`x = memoryview(bytes(5))`

Operadores

- Tipos de operadores aritméticos en python:

Operador	Nombre	Ejemplo
+	Addition	`x + y`
-	Subtraction	`x - y`
*	Multiplication	`x * y`
/	Division	`x / y`
%	Modulus	`x % y`
**	Exponentiation	`x ** y`
//	Floor division	`x // y`

- Tipos de operadores de asignación en python:

Operador	Ejemplo	Equivalente
=	`x = 5`	`x = 5`
+=	`x += 3`	`x = x + 3`
-=	`x -= 3`	`x = x - 3`
*=	`x *= 3`	`x = x * 3`
/=	`x /= 3`	`x = x / 3`
%=	`x %= 3`	`x = x % 3`
//=	`x //= 3`	`x = x // 3`
**=	`x **= 3`	`x = x ** 3`
&=	`x &= 3`	`x = x & 3`
	=	`x |= 3`

Operador	Ejemplo	Equivalente
^=	`x ^= 3`	`x = x ^ 3`
>>=	`x >>= 3`	`x = x >> 3`
<<=	`x <<= 3`	`x = x << 3`

- Tipos de operadores de comparación en python:

Operador	Nombre	Ejemplo
==	Equal	`x == y`
!=	Not equal	`x != y`
>	Greater than	`x > y`
<	Less than	`x < y`
>=	Greater than or equal to	`x >= y`
<=	Less than or equal to	`x <= y`

- Tipos de operadores lógicos en python:

Operador	Descripción	Ejemplo
and	Devuelve True si ambos son ciertos	`x < 5 and x < 10`
or	Devuelve True si uno de los dos es cierto	`x < 5 or x < 4`
not	Invierte el resultado, devuelve False si el resultado es cierto	`not(x < 5 and x < 10)`

- Tipos de operadores de identidad en python:

Operador	Descripción	Ejemplo
is	Devuelve True si ambas variables son el mismo objeto	`x is y`
is not	Devuelve True si ambas variables no son el mismo objeto	`x is not y`

- Tipos de operadores de bit a bit en Python:

Operador	Nombre	Descripción
&	AND	Conjunto de bits
|	OR	Conjunto de bits
^	XOR	Conjunto de bits
~	NOT	Conjunto de bits
<<	Zero fill left shift	Desplazamiento de bits a la izquierda

Operador	Nombre	Descripción
>>	Signed right shift	Desplazamiento de bits a la derecha

Diferencias entre listas, tuplas, sets y diccionarios

Las listas, tuplas, sets y diccionarios son estructuras de datos en Python que se utilizan para almacenar colecciones de datos. Aquí están las principales diferencias entre ellos:

Listas: Son colecciones ordenadas y modificables que permiten duplicados. Se definen con corchetes []. Ejemplo: mi_lista = [1, 2, 3, 4, 5]

Tuplas: Son colecciones ordenadas e inmutables que permiten duplicados. Se definen con paréntesis (). Ejemplo: mi_tupla = (1, 2, 3, 4, 5)

Sets: Son colecciones no ordenadas, no indexadas y no permiten duplicados. Se definen con llaves {} o con la función set(). Ejemplo: mi_set = {1, 2, 3, 4, 5}

Diccionarios: Son colecciones no ordenadas, modificables e indexadas. No permiten duplicados y se definen con llaves {}. Ejemplo: mi_diccionario = {"nombre": "Juan", "edad": 30}

En resumen, las listas y tuplas son muy similares, pero las tuplas son inmutables. Los sets son útiles cuando quieres una colección que no permita duplicados. Los diccionarios son útiles cuando necesitas asociar valores con claves.

Listas

- Listas: Las listas se utilizan para almacenar varios elementos en una sola variable. Las listas se crean utilizando corchetes:

```
lista = ["manzana", "plátano", "cereza"]
print(lista)
```

- Podemos acceder a los elementos de una lista haciendo referencia al número de índice:

```
lista = ["manzana", "plátano", "cereza"]
print(lista[1])
```

- Podemos saber la longitud de la lista utilizando la función len():

```
lista = ["manzana", "plátano", "cereza"]
print(len(lista))
```

- La lista puede contener diferentes tipos de datos:

```
lista = ["manzana", "plátano", "cereza", 1, 2, 3]
print(lista)
```

- Para saber el tipo de una lista se utiliza la función type():

```
lista = ["manzana", "plátano", "cereza"]
print(type(lista))
```

- Puedes comprobar si un valor existe en una lista utilizando la palabra clave in:

```
lista = ["manzana", "plátano", "cereza"]
if "manzana" in lista:
  print("Sí, 'manzana' está en la lista de frutas")
```

- Insertar y añadir valores a una lista:

```
lista = ["manzana", "plátano", "cereza"]
lista.append("naranja")
lista.insert(1, "limón")
print(lista)
```

- Eliminar valores de una lista:

```
lista = ["manzana", "plátano", "cereza"]
lista.remove("plátano")
lista.pop()
del lista[0]
print(lista)
```

- Eliminar la lista por completo:

```
lista = ["manzana", "plátano", "cereza"]
del lista
print(lista) #this will cause an error because "thislist" no longer exists.
```

- Ampliar la lista con otra lista:

```
lista1 = ["a", "b" , "c"]
lista2 = [1, 2, 3]
```

```
lista1.extend(lista2)
print(lista1)
```

- Como recorer una lista:

```
lista = ["manzana", "plátano", "cereza"]
for x in lista:
  print(x)
```

- Ordenar una lista:

```
lista = ["manzana", "plátano", "cereza"]
lista.sort()
print(lista)
```

Tuplas

- Tuplas: Las tuplas se utilizan para almacenar varios elementos en una sola variable. Una tupla es una colección ordenada y no cambiable. Las tuplas se crean utilizando paréntesis:

```
tupla = ("manzana", "plátano", "cereza")
print(tupla)
```

Sets

- Sets: Los sets se utilizan para almacenar varios elementos en una sola variable. Un set es una colección que no está ordenada ni indexada. En Python, los sets se escriben con llaves.

```
set = {"manzana", "plátano", "cereza"}
print(set)
```

Diccionarios

- Diccionarios: Los diccionarios se utilizan para almacenar varios elementos en una sola variable. Un diccionario es una colección desordenada, modificable e indexada. En Python, los diccionarios se escriben con llaves y tienen claves y valores.

```
diccionario = {
  "marca": "Ford",
  "modelo": "Mustang",
  "año": 1964
```

```
}
print(diccionario)
```

Estrucuturas de control

If

- If: La declaración if se utiliza para tomar decisiones basadas en diferentes condiciones.

```
a = 33
b = 200
if b > a:
  print("b es mayor que a")
```

- Elif: La declaración elif se utiliza para evitar que se ejecute el bloque if y el bloque else. Si la declaración elif es verdadera, se ejecutará el bloque elif, y el bloque else se omitirá.

```
a = 33
b = 33
if b > a:
  print("b es mayor que a")
elif a == b:
  print("a y b son iguales")
```

- Else: La declaración else se utiliza para ejecutar un bloque de código si ninguna de las condiciones es verdadera.

```
a = 200
b = 33
if b > a:
  print("b es mayor que a")
elif a == b:
  print("a y b son iguales")
else:
  print("a es mayor que b")
```

- Short Hand If: Si solo hay una declaración a ejecutar, puede ponerla en la misma línea que la declaración if.

```
if a > b: print("a es mayor que b")
```

- Short Hand If ... Else: Si solo hay una declaración a ejecutar para ambas condiciones, puede ponerlas en la misma línea:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

- And: La palabra clave and es un operador lógico, y se utiliza para combinar condiciones lógicas.

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Ambas condiciones son verdaderas")
```

- Or: La palabra clave or es un operador lógico, y se utiliza para combinar condiciones lógicas.

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("Al menos una de las condiciones es verdadera")
```

- Not: La palabra clave not es un operador lógico, y se utiliza para invertir el valor de una condición lógica.

```
a = 200
b = 33
if not b > a:
  print("b no es mayor que a")
```

- Nested If: Una declaración if dentro de otra declaración if.

```
x = 41

if x > 10:
  print("Por encima de diez,")
  if x > 20:
    print("y también por encima de 20!")
  else:
    print("pero no por encima de 20.")
```

While

- While: Con la instrucción while podemos ejecutar un conjunto de declaraciones siempre que se cumpla una condición especificada.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

- Break: Con la instrucción break podemos detener el bucle aunque la condición while sea verdadera.

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

- Continue: Con la instrucción continue podemos detener la iteración actual del bucle y continuar con la siguiente.

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

- Else: Con la instrucción else podemos ejecutar un bloque de código una vez cuando la condición ya no es verdadera.

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i ya no es menor que 6")
```

For

- For: Un bucle for se utiliza para iterar sobre una secuencia (que es una lista, una tupla, un diccionario, un conjunto o una cadena).

```
lista = ["manzana", "plátano", "cereza"]
for x in lista:
  print(x)
```

- Looping Through a String: Podemos recorrer una cadena de texto utilizando un bucle for.

```
for x in "banana":
  print(x)
```

- Break: Con la instrucción break podemos detener el bucle aunque la condición while sea verdadera.

```
lista = ["manzana", "plátano", "cereza"]
for x in lista:
  print(x)
  if x == "plátano":
    break
```

- Continue: Con la instrucción continue podemos detener la iteración actual del bucle y continuar con la siguiente.

```
lista = ["manzana", "plátano", "cereza"]
for x in lista:
  if x == "plátano":
    continue
  print(x)
```

- Range: La función range() devuelve una secuencia de números, comenzando desde 0 de forma predeterminada, y se incrementa en 1 (de forma predeterminada), y termina en un número especificado.

```
for x in range(6):
  print(x)
```

- Else: Con la instrucción else podemos ejecutar un bloque de código una vez cuando la condición ya no es verdadera.

```
for x in range(6):
  print(x)
else:
  print("Finalmente terminó!")
```

- Nested Loops: Un bucle anidado es un bucle dentro de un bucle. El "bucle interno" se ejecutará una vez por cada iteración del "bucle externo":

```
adj = ["rojo", "grande", "delicioso"]
frutas = ["manzana", "plátano", "cereza"]

for x in adj:
  for y in frutas:
    print(x, y)
```

Funciones

- Funciones: Una función es un bloque de código que se ejecuta cuando se llama a la función. Puedes pasar datos, conocidos como parámetros, a una función. Una función puede devolver datos como resultado.

```
def my_function():
  print("Hola desde una función")
```

- Llamar a una función: Para llamar a una función, utiliza el nombre de la función seguido de paréntesis:

```
def my_function():
  print("Hola desde una función")

my_function()
```

- Argumentos: La información se puede pasar a las funciones como argumentos. Los argumentos se especifican después del nombre de la función, entre paréntesis. Puedes agregar tantos argumentos como desees, solo sepáralos con una coma.

```
def my_function(nombre):
  print("Hola " + nombre)

my_function("Juan")
my_function("Carlos")
my_function("Pedro")
```

- Número de argumentos: Al definir una función, se puede especificar el número de argumentos que se espera que tenga la función. Esto se hace agregando un número al nombre de argumento.

```
def my_function(fname):
  print(fname + " Refsnes")
```

```
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

- Argumentos arbitrarios: Si no sabes cuántos argumentos se pasarán a tu función, agrega un * antes del nombre del parámetro en la definición de la función.

```
def my_function(*kids):
  print("El niño más joven es " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

- Argumentos con palabras clave: También puedes enviar argumentos con la sintaxis clave = valor. De esta forma, el orden de los argumentos no importa.

```
def my_function(child3, child2, child1):
  print("El niño más joven es " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

- Argumentos arbitrarios, palabras clave: Si no sabes cuántos argumentos con palabras clave se pasarán a tu función, agrega dos asteriscos: ** antes del nombre del parámetro en la definición de la función.

```
def my_function(**kid):
  print("Su apellido es " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

- Valor de parámetro predeterminado: Si llamamos a la función sin argumentos, utiliza el valor predeterminado:

```
def my_function(pais = "Noruega"):
  print("Yo soy de " + pais)

my_function("Suecia")
my_function("India")
my_function()
my_function("Brasil")
```

- Pasar una lista como argumento: Puedes enviar cualquier tipo de datos como argumento a una función (cadena, número, lista, diccionario, etc.), y se tratará como el mismo tipo de datos dentro de la función.

```
def my_function(comida):
  for x in comida:
    print(x)

frutas = ["manzana", "plátano", "cereza"]

my_function(frutas)
```

- Devolver valores: Para permitir que una función devuelva un valor, utiliza la palabra clave return:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Lambda

- Lambda: Una función lambda es una pequeña función anónima. Una función lambda puede tomar cualquier número de argumentos, pero solo puede tener una expresión.

```
x = lambda a : a + 10
print(x(5))
```

- Lambda con varios argumentos: Una función lambda puede tomar cualquier número de argumentos:

```
x = lambda a, b : a * b
print(x(5, 6))
```

- Lambda dentro de otra función: Una función lambda puede tomar cualquier número de argumentos, pero solo puede tener una expresión.

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Arrays

- Arrays: Los arrays se utilizan para almacenar varios valores en una sola variable, a diferencia de las variables normales. Si tienes una lista de elementos (una lista de nombres, por ejemplo), almacenar los nombres en variables podría parecerse a esto:

```
nombre1 = "Juan"
nombre2 = "Carlos"
nombre3 = "Pedro"
```

- Pero, ¿qué pasa si quieres recorrer los nombres y encontrar un nombre en particular? Y ¿qué pasa si no sabes cuántos nombres hay? La solución es un array. Un array puede contener muchos valores bajo un solo nombre, y puede acceder a los valores haciendo referencia a un número de índice.

```
nombres = ["Juan", "Carlos", "Pedro"]
print(nombres[1])
```

- Longitud de un array: Utiliza la función len() para determinar la longitud de un array.

```
nombres = ["Juan", "Carlos", "Pedro"]
print(len(nombres))
```

- Un array con datos de diferentes tipos: Un array puede contener diferentes tipos de datos:

```
array = ["Juan", 1, True]
print(array)
```

Iteradores

- Iteradores: Un iterador es un objeto que contiene un número contable de valores. Un iterador es un objeto que se puede iterar sobre, lo que significa que puede atravesar todos los valores. Los iteradores se pueden crear utilizando la función iter(). Para crear un objeto iterable, debe implementar el método **iter**() en la clase.

```
class MiClase:
  def __iter__(self):
    self.a = 1
    return self

mi_clase = MiClase()
mi_iterador = iter(mi_clase)

print(next(mi_i
```

Polimorfismo

- Polimorfismo: Polimorfismo significa "muchas formas", y ocurre cuando tenemos muchas clases que están relacionadas entre sí por herencia.

```
class Perro:
  def sonido(self):
    print("Guau")

class Gato:
  def sonido(self):
    print("Miau")

def hacer_sonar(animal_type):
  animal_type.sonido()

perro = Perro()
gato = Gato()

hacer_sonar(perro)
hacer_sonar(gato)
```

Módulos

- Módulos: Considere un módulo como una biblioteca de funciones para incluir en su aplicación.

```
import mi_modulo

mi_modulo.saludar("Juan")
```

- Variables en módulos: El módulo puede contener funciones, así como variables de todo tipo (listas, diccionarios, objetos, etc.):

```
import mi_modulo

a = mi_modulo.personas1["nombre"]
print(a)
```

- Renombrar un módulo: Puede crear un alias cuando importa un módulo, al usar la palabra clave as:

```
import mi_modulo as mx

a = mx.personas1["nombre"]
print(a)
```

- Importar solo partes de un módulo: Puede elegir importar solo partes de un módulo, al usar la palabra clave from.

```
from mi_modulo import personas1

print(personas1["nombre"])
```

- Variables integradas: Hay varios módulos integrados en Python, que se pueden importar en cualquier momento y en cualquier lugar, sin importar si se ha instalado el módulo:

```
import platform

x = platform.system()
print(x)
```

POO en Python

Clases y objetos

- Clases y objetos: Python es un lenguaje orientado a objetos. Casi todo en Python es un objeto, con sus propiedades y métodos. Una clase es como un constructor de objetos, o un "plano" para crear objetos.

```
class MiClase:
  x = 5

p1 = MiClase()
print(p1.x)
```

- Objetos: Una vez que tenga una clase creada, puede utilizar el objeto para acceder a sus atributos.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

p1 = Persona("Juan", 36)

print(p1.nombre)
print(p1.edad)
```

- Modificar propiedades de un objeto: Puede modificar las propiedades de un objeto de la siguiente manera:

```
p1.edad = 40
```

- Eliminar propiedades de un objeto: Puede eliminar propiedades de objetos de la siguiente manera:

```
del p1.edad
```

- Eliminar objetos: Puede eliminar objetos utilizando la palabra clave del:

```
del p1
```

- Añadir propiedades: Puede añadir propiedades a clases y objetos existentes de la siguiente manera:

```
class Persona:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Estudiante(Persona):

  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def bienvenida(self):
    print("Bienvenido", self.firstname, self.lastname, "a la clase de",
self.graduationyear)

x = Estudiante("Juan", "Perez", 2019)
x.bienvenida()
```

Métodos

- Métodos: Los métodos son funciones que pertenecen a los objetos.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def mi_funcion(self):
    print("Hola mi nombre es " + self.nombre)
```

```
p1 = Persona("Juan", 36)
p1.mi_funcion()
```

- El self parámetro: El self parámetro es una referencia a la instancia actual de la clase, y se utiliza para acceder a las variables que pertenecen a la clase.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def mi_funcion(self):
    print("Hola mi nombre es " + self.nombre)

p1 = Persona("Juan", 36)
p1.mi_funcion()
```

- Función super(): Python también tiene una función super() que hará que la clase hija herede todos los métodos y propiedades de su clase padre:

```
class Persona:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Estudiante(Persona):

  def __init__(self, fname, lname):
    super().__init__(fname, lname)

x = Estudiante("Juan", "Perez")
x.printname()
```

Getter y Setter

- Getter y Setter: Los getters y setters se utilizan para proteger los datos, controlar el acceso a ellos, obtenerlos y establecerlos.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad
```

```
  def mi_funcion(self):
    print("Hola mi nombre es " + self.nombre)

p1 = Persona("Juan", 36)
p1.mi_funcion()
```

Encapsulación

- Encapsulación: La encapsulación significa que los atributos y métodos internos de una clase no se pueden acceder desde el exterior de la clase.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def mi_funcion(self):
    print("Hola mi nombre es " + self.nombre)

p1 = Persona("Juan", 36)
p1.mi_funcion()
```

Herencia

- Herencia: La herencia nos permite definir una clase que hereda todos los métodos y propiedades de otra clase.

```
class Persona:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Estudiante(Persona):

    def __init__(self, fname, lname, year):
      super().__init__(fname, lname)
      self.graduationyear = year

    def bienvenida(self):
      print("Bienvenido", self.firstname, self.lastname, "a la clase de",
self.graduationyear)

x = Estudiante("Juan", "Perez", 2019)
x.bienvenida()
```

Métodos especiales de Python

- Métodos especiales de Python: Los métodos especiales de Python son métodos especiales que puede utilizar dentro de su clase para realizar tareas específicas, como la inicialización y la creación de objetos, etc. Los métodos especiales se definen con guiones bajos dobles **init**(), **del**(), **repr**(), etc.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def __str__(self):
    return "Mi nombre es " + self.nombre

p1 = Persona("Juan", 36)
print(p1)
```

Métodos estáticos

- Métodos estáticos: Los métodos estáticos en Python son métodos que se pueden llamar sin crear una instancia de una clase. Los métodos estáticos se definen con la palabra clave @staticmethod.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  @staticmethod
  def mi_funcion():
    print("Hola")

Persona.mi_funcion()
```

Métodos de clase

- Métodos de clase: Los métodos de clase en Python son métodos que se llaman con la clase como primer argumento. Los métodos de clase se definen con la palabra clave @classmethod.

```
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  @classmethod
  def mi_funcion(cls):
    print("Hola")
```

```
Persona.mi_funcion()
```

Herencia múltiple

- Herencia múltiple: La herencia múltiple significa que una clase puede heredar de múltiples clases. Python admite una forma limitada de herencia múltiple en clases. Las clases derivadas pueden heredar de una o más clases base.

```
class Persona:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Estudiante(Persona):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def bienvenida(self):
    print("Bienvenido", self.firstname, self.lastname, "a la clase de",
self.graduationyear)

class Estudiante2(Persona):

  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def bienvenida(self):
    print("Bienvenido", self.firstname, self.lastname, "a la clase de",
self.graduationyear)

x = Estudiante("Juan", "Perez", 2019)
x.bienvenida()
```

Clases abstractas

- Clases abstractas: Las clases abstractas, que nunca están destinadas a ser instanciadas, pueden ser útiles para definir interfaces que deben ser implementadas por clases concretas.

```
from abc import ABC, abstractmethod

class Persona(ABC):
  def __init__(self, nombre, edad):
    self.nombre = nombre
```

```
    self.edad = edad
    super().__init__()

  @abstractmethod
  def mi_funcion(self):
    pass

class Estudiante(Persona):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def mi_funcion(self):
    print("Bienvenido", self.firstname, self.lastname, "a la clase de",
self.graduationyear)

x = Estudiante("Juan", "Perez", 2019)
x.mi_funcion()
```