1. MÓDULOS

Un módulo en Python es un archivo que contiene código reutilizable. Estos módulos se utilizan para organizar y estructurar el código de manera más efectiva, facilitando la reutilización y el mantenimiento del mismo. Algunas ventajas de usar módulos en Python son:

- Organización del código: Los módulos permiten organizar el código en unidades lógicas y separadas. Puedes agrupar funciones, clases y variables relacionadas en un solo archivo, lo que hace que el código sea más legible y fácil de entender.
- Reutilización de código: Al dividir el código en módulos, puedes reutilizar funciones, clases y variables en diferentes programas. Esto evita la duplicación de código y facilita la implementación de la misma funcionalidad en diferentes partes de tu proyecto.
- Mantenibilidad: Los módulos permiten separar diferentes componentes de un programa, lo que facilita la identificación y corrección de errores. Además, si necesitas realizar cambios en una parte específica del código, solo tienes que modificar el módulo correspondiente, sin afectar el resto del programa.
- Modularidad: Los módulos promueven la modularidad en el diseño de programas. Puedes pensar en cada módulo como una "caja negra" que encapsula su funcionalidad y proporciona una interfaz clara y definida para interactuar con él. Esto facilita la colaboración en equipos de desarrollo y permite dividir proyectos grandes en módulos más pequeños y manejables.
- Espacio de nombres (namespace) separado: Cada módulo en Python tiene su propio espacio de nombres, lo que significa que las variables, funciones y clases definidas en un módulo no entran en conflicto con los nombres definidos en otros módulos. Esto ayuda a evitar colisiones de nombres y proporciona un mejor control y organización del código.
- Módulos estándar y externos: Python ofrece una amplia biblioteca estándar que incluye una variedad de módulos para realizar tareas comunes, como manipulación de archivos, acceso a bases de datos, generación de números aleatorios, entre otros. Además, la comunidad de Python ha creado una gran cantidad de módulos externos que puedes instalar y utilizar para ampliar las capacidades de Python en diferentes áreas.

Al utilizar módulos, puedes aprovechar el código existente, modularizar tu programa en partes más pequeñas y fáciles de mantener, y acceder a

funcionalidades adicionales proporcionadas por módulos externos. Esto ayuda a mejorar la eficiencia y la productividad en el desarrollo de software.

2. CREACIÓN y USO DE MÓDULOS

Para crear un módulo necesitamos abrir un nuevo archivo en un editor de texto o entorno de desarrollo integrado (IDE) y guardarlo con una extensión .py. Por ejemplo, llamarlo mi_modulo.py.

Dentro del archivo mi_modulo.py, definir las funciones, clases, variables y cualquier otro código que desees incluir en tu módulo. Por ejemplo:

```
# mi_modulo.py

def saludar(nombre):

    print("Hola,", nombre)

def despedir(nombre):

    print("Adiós,", nombre)

pi = 3.14159
```

En otro archivo o en la consola interactiva de Python, utilizar el módulo mi_modulo importándolo. Por ejemplo:

```
# Importar el módulo completo

import mi_modulo


mi_modulo.saludar("Juan")  # Salida: Hola, Juan

mi_modulo.despedir("Juan")  # Salida: Adiós, Juan

print(mi_modulo.pi)  # Salida: 3.14159
```

En este ejemplo, hemos importado el módulo mi_modulo utilizando la declaración import. Luego, hemos utilizado las funciones saludar() y despedir() del módulo, así como la variable pi.

También puedes importar funciones o variables específicas del módulo utilizando la forma de importación from. Por ejemplo:

```
# Importar funciones/variables específicas del módulo

from mi_modulo import saludar, pi

saludar("María")  # Salida: Hola, María
```

```
print(pi)  # Salida: 3.14159
```

Recordá que el archivo mi_modulo.py debe estar en el mismo directorio o en una ruta de búsqueda especificada por la variable de entorno PYTHONPATH para que pueda ser encontrado y utilizado correctamente.

3. EJECUCIÓN DE MÓDULOS

Para ejecutar un módulo como script, tenemos que añadir un bloque condicional utilizando la variable global __name__ para comprobar si el módulo se está ejecutando como un script. Por ejemplo:

```
# mi_modulo.py

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

def despedir(nombre):
    print("Adiós,", nombre)

pi = 3.14159

if __name__ == "__main__":
    nombre = input("Ingresa tu nombre: ")
    saludar(nombre)
    despedir(nombre)
```

Luego debemos guardar el archivo.

Para ejecutar el módulo como un script:

- Tenemos que abrir una terminal o línea de comandos.

- Navegar al directorio donde se encuentra el archivo mi_modulo.py.
- Ejecutar el siguiente comando:

```
python mi_modulo.py
```

Esto ejecutará el módulo como un script independiente y se mostrará el mensaje "Ingresa tu nombre:". Puedes ingresar un nombre y presionar Enter para ver los saludos correspondientes.

Al utilizar el bloque condicional if __name__ == "__main__":, te aseguras de que el código dentro de ese bloque solo se ejecute cuando el módulo se ejecuta como un script independiente. Si el módulo se importa desde otro archivo, el bloque no se ejecutará automáticamente.

4. BIBLIOTECAS DE MÓDULOS

Python cuenta con una amplia biblioteca estándar que proporciona una serie de módulos listos para usar en tus programas. Algunas de las bibliotecas de módulos estándar más comunes y útiles de Python incluyen:

- os: Proporciona funciones para interactuar con el sistema operativo, como manipular archivos y directorios, acceder a variables de entorno y ejecutar comandos del sistema.
- sys: Contiene funciones y variables que interactúan con el intérprete de Python. Puede utilizarse para acceder a argumentos de línea de comandos, manipular la ruta de búsqueda de módulos y otras operaciones relacionadas con el intérprete.
- math: Ofrece funciones matemáticas y constantes, como operaciones trigonométricas, exponenciales, logarítmicas y funciones especiales.
- datetime: Proporciona clases y funciones para trabajar con fechas, horas y manipulación de tiempo. Permite realizar operaciones como cálculos de diferencia de tiempo, formato de fecha y hora, y conversiones entre diferentes formatos.
- random: Permite generar números aleatorios y realizar operaciones relacionadas con aleatoriedad, como mezclar listas, seleccionar elementos aleatorios y generar secuencias aleatorias.
- json: Proporciona funciones para trabajar con el formato de datos JSON (JavaScript Object Notation), permitiendo la serialización y deserialización de datos en formato JSON.
- re: Ofrece funciones y clases para trabajar con expresiones regulares, lo que permite buscar y manipular patrones en cadenas de texto.

Estas son solo algunas de las bibliotecas de módulos estándar más utilizadas en Python. Sin embargo, Python ofrece muchas más bibliotecas útiles para diferentes propósitos, como manejo de archivos, networking, procesamiento de datos, interfaces gráficas, entre otros.

5. DIR

La función dir() es una función integrada de Python que se utiliza para obtener una lista de nombres definidos actualmente en un módulo, un objeto o en el ámbito actual. Proporciona una forma conveniente de explorar y obtener información sobre los atributos y métodos disponibles en un objeto o módulo.

La sintaxis básica de la función dir() es la siguiente:

```
dir([objeto])
```

Donde [objeto] es un argumento opcional que representa el objeto del cual deseas obtener la lista de nombres. Si no se proporciona ningún objeto, dir() se aplicará al ámbito actual.

A continuación, se muestran algunos ejemplos de cómo usar la función dir():

Usar dir() en el ámbito global para obtener una lista de nombres definidos actualmente:

```
print(dir())
```

```
SALIDA :    ['__builtins__',    '__doc__',    '__loader__',
'__name__', '__package__', '__spec__', 'x', 'y', 'z']
```

Usar dir() en un módulo para obtener una lista de nombres definidos en ese módulo:

```
import math

print(dir(math))
```

```
SALIDA: ['__doc__', '__loader__', '__name__', '__package__',
'__spec__',  'acos',  'acosh',  'asin',  'asinh',  'atan',
'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh',
'degrees',  'dist',  'e',  'erf',  'erfc',  'exp',  'expm1',
'fabs',  'factorial',  'floor',  'fmod',  'frexp',  'fsum',
'gamma',  'gcd',  'hypot',  'inf',  'isclose',  'isfinite',
'isinf',  'isnan',  'ldexp',  'lgamma',  'log',  'log10',
'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
```

```
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'tau', 'trunc']
```

Usar dir() en un objeto para obtener una lista de atributos y métodos disponibles para ese objeto:

```
lista = [1, 2, 3]

print(dir(lista))
```

```
SALIDA:     ['__add__',     '__class__',     '__contains__',
'__delattr__',    '__delitem__',    '__dir__',    '__doc__',
'__eq__',   '__format__',   '__ge__',   '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__',   '__init_subclass__',   '__iter__',   '__le__',
'__len__',   '__lt__',   '__mul__',   '__ne__',   '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__',   '__setattr__',   '__setitem__',   '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy',
'count',  'extend',  'index',  'insert',  'pop',  'remove',
'reverse', 'sort']
```

Al ejecutar estos ejemplos, obtendrás una lista de nombres disponibles en el ámbito actual, en el módulo math y en el objeto lista, respectivamente. La lista resultante incluirá nombres de variables, funciones, clases y otros atributos definidos.

Es importante tener en cuenta que la función dir() solo muestra los nombres definidos en un objeto, pero no proporciona detalles sobre su funcionalidad. Para obtener información adicional sobre un nombre en particular, puedes consultar la documentación o utilizar otras funciones de exploración, como help().

6. PAQUETES

En Python, un paquete es una forma de organizar y estructurar módulos relacionados. Un paquete es simplemente una carpeta (directorio) que contiene uno o más archivos de módulo de Python, junto con un archivo especial llamado __init__.py. El archivo __init__.py puede estar vacío o puede contener código que se ejecuta cuando el paquete se importa.

Los paquetes en Python brindan una manera conveniente de agrupar módulos relacionados, lo que facilita su organización y reutilización en diferentes proyectos. Algunas ventajas de usar paquetes son:

- Organización lógica: Los paquetes permiten agrupar módulos relacionados en una estructura jerárquica, lo que facilita la navegación y comprensión de un conjunto de módulos.
- Evitar conflictos de nombres: Al agrupar los módulos en paquetes, se pueden evitar colisiones de nombres, ya que cada módulo se importa utilizando su nombre completo que incluye el nombre del paquete.
- Facilidad de reutilización: Los paquetes proporcionan una forma de empaquetar y distribuir código para su reutilización en diferentes proyectos. Los paquetes pueden instalarse y utilizarse en múltiples proyectos sin tener que copiar y pegar el código.

Para crear un paquete en Python, debes seguir estos pasos:

Crear una carpeta (directorio) para el paquete. El nombre de la carpeta será el nombre del paquete.

Agregar un archivo __init__.py dentro de la carpeta. Este archivo puede estar vacío o contener código de inicialización del paquete.

Agregar los módulos relacionados al paquete. Cada módulo es un archivo .py dentro de la carpeta.

```
mi_paquete/
    __init__.py
    modulo1.py
    modulo2.py
```

Una vez que hayas creado el paquete, puedes utilizarlo en tus programas importando los módulos que necesitas utilizando la sintaxis import:

```
import mi_paquete.modulo1
from mi_paquete import modulo2

mi_paquete.modulo1.funcion1()
modulo2.funcion2()
```

Ejemplo:

Supongamos que tienes un proyecto relacionado con geometría que requiere cálculos de áreas y perímetros de diferentes figuras. Puedes crear un paquete llamado geometría que contenga módulos para diferentes tipos de figuras geométricas. Veamos cómo se puede estructurar:

```
geometria/
    __init__.py
    formas/
        __init__.py
        cuadrado.py
        circulo.py
        triangulo.py
    calculadora.py
```

En este ejemplo, el paquete geometría contiene una carpeta llamada formas, que a su vez contiene los módulos cuadrado.py, circulo.py y triangulo.py. Además, tenemos el módulo calculadora.py que se encuentra en el nivel superior del paquete.

El archivo __init__.py en la carpeta geometría y en la subcarpeta formas indica que es un paquete y puede estar vacío o contener código de inicialización.

Vamos a ver cómo se podrían implementar algunos de los módulos:

cuadrado.py

```
def calcular_area(lado):
    return lado ** 2


def calcular_perimetro(lado):
    return 4 * lado
```

circulo.py

```
import math
```

```
def calcular_area(radio):
    return math.pi * radio ** 2


def calcular_perimetro(radio):
    return 2 * math.pi * radio
```

calculadora.py

```
from geometria.formas.cuadrado import calcular_area as area_cuadrado
from geometria.formas.cuadrado import calcular_perimetro as perimetro_cuadrado
from geometria.formas.circulo import calcular_area as area_circulo
from geometria.formas.circulo import calcular_perimetro as perimetro_circulo


lado = 4
radio = 2


area = area_cuadrado(lado)
perimetro = perimetro_cuadrado(lado)
print(f"Cuadrado - Área: {area}, Perímetro: {perimetro}")


area = area_circulo(radio)
perimetro = perimetro_circulo(radio)
print(f"Círculo - Área: {area}, Perímetro: {perimetro}")
```

En el archivo calculadora.py, importamos las funciones calcular_area y calcular_perimetro de los módulos cuadrado y circulo utilizando la sintaxis from módulo import funcion. Luego, realizamos cálculos de área y perímetro para un cuadrado y un círculo.

Al ejecutar el archivo calculadora.py, obtendrás la salida:

```
Cuadrado - Área: 16, Perímetro: 16
```

Círculo - Área: 12.566370614359172, Perímetro: 12.566370614359172