UD02 – Lenguaje Python

1. Python

Python es un lenguaje de propósito general que en los últimos años se ha ido haciendo cada vez más popular en áreas como data sciences o la inteligencia artificial.

2. Bondades de Python

Python tiene varias propiedades que lo han convertido en un lenguaje muy potente y fácil de aprender. Estas propiedades son las siguientes:

- **Tipado dinámico**: Python no necesita que definamos el tipo de las variables cuando las inicializamos como pasa, por ejemplo, en Java o C. Cuando inicializamos una variable Python le asigna el tipo del valor que le estamos asignando. Incluso, durante la ejecución, una misma variable podría contener valores con distintos tipos. Esta propiedad hace que sea más sencillo aprender a usarlo, aunque hace que sea más difícil detectar errores asociados con los tipos de datos.
- **Lenguaje multiparadigma**: Python permite aplicar diferentes paradigmas de programación como son la programación orientada a objetos, como Java o C++, programación imperativa, como C, o programación funcional, como Haskell.

2. Bondades de Python

- **Interpretado/scripts**: otra ventaja es que podemos ejecutar Python de forma interpretada o usando scripts. Es decir, puedo abrir una consola de Python y escribir y ejecutar las instrucciones una a una o, por otro lado, puedo crear un fichero que almacene todo el programa.
- **Extensible**: por último, Python cuenta con una gran cantidad de módulos y librerías que podemos instalar para incluir nuevas funcionalidades. Sin embargo, como Python esta implementado usando C++, podemos crear nuevos módulos en C++ e incluirlo en Python haciendo que el lenguaje sea extensible a nuevos módulos.

Identificadores

Los identificadores son nombres que asignaremos a elementos, como funciones, variables, etc., para hacerles referencia más adelante en el código. Estos identificadores tienen que seguir las siguientes reglas:

- Pueden ser una combinación de números (0-9), letras mayúsculas (A-Z), letras minúsculas (a-z) y el símbolo de guion bajo (_).
- No pueden comenzar por un dígito.
- No pueden utilizarse símbolos especiales: @, !, #, etc.
- Pueden tener cualquier longitud.
- Python distingue las mayúsculas de las minúsculas.

```
# Identificadores correctos.
variable = 1
otra_variable = 'Otra'
_mas_variables = [1, 3, 4]
Variable_4 = True


# Identificadores incorrectos.
@variable = 3
2_variable ='Está mal.'
```

Palabras reservadas

Python, como todos los lenguajes de programación, tiene un conjunto de palabras reservadas que no se pueden utilizar como identificadores:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield.

Comentarios

Los comentarios son partes del código que el intérprete de Python no ejecuta. Estos comentarios nos permiten escribir aclaraciones en el código.En Python hay dos sintaxis diferentes:

- **Comentarios de línea**: para aquellos comentarios que solo ocupen una línea se utiliza el símbolo # al principio de la misma.
- **Comentarios de bloque**: para los comentarios que ocupen más de una línea se utilizan triples comillas (simples o dobles) al principio y al final del comentario.

```
# Ejemplo de comentario de línea.

"""
Ejemplo de comentario de bloque.
Todas las líneas pertenecen al comentario.
"""

'''
Otro ejemplo de
comentario de bloque.
'''
```

Sangría

Una de las diferencias que tiene Python con respecto a otros lenguajes,, es que no utiliza llaves ({,}) para diferenciar bloques de código. Se utiliza la sangría de bloques, es decir, hacer una separación hacia la derecha del código que pertenece a un bloque. Esta sangría está determinada por 4 espacios según la guía de estilos PEP 8.

Al ser obligatorio, puede llevar a errores de ejecución si no se ha hecho la sangría correctamente.

```
# Ejemplo de una sangría:
variable = 1

if (variable == 1):
    print("Aquí pongo el código a ejecutar.")
    print("Tiene una sangría de 4 espacios.")
```

4. Almacenar valores

En este apartado explicaremos cómo se deben declarar las variables y las constantes en Python.

Variables

Una variable es un identificador al que le asignaremos un valor y, más adelante, llamando a ese identificador podremos consultar el valor.

En Python la creación de una variable se hace a través de una asignación. Haremos una asignación escribiendo el nombre de un identificador, seguido del símbolo = y el valor que deseamos almacenar en la variable:

```
identificador = [valor]
```

4. Almacenar valores

A diferencia de otros lenguajes, no es necesario definir el tipo de dato que almacenará la variable. Python infiere este tipo en el momento de ejecutar la asignación y le asignará el tipo de dato que mejor se adapte al valor que hayamos asignado. Además, una misma variable puede almacenar diferentes tipos de datos durante la ejecución.

- Para preguntar qué tipo de dato ha almacenado una variable, usaremos la función **type**.

```
# Ejemplo de variable:
n = 23
saludo = 'Hola! Qué tal?'
type(n) # Devuelve int
type(saludo) # Devuelve str
```

4. Almacenar valores

Constantes

Una constante es un tipo de variable cuyo valor no puede ser modificado durante toda la ejecución del programa.

En Python no existe ninguna palabra reservada que nos permita definir una constante. Sin embargo, existen dos posibles soluciones para declarar constantes. Una de ellas es utilizar una variable, escribiendo el identificador en mayúsculas para que sepamos que esa variable no puede ser modificada. Esta solución se refiere a los estilos y tendremos que estar pendientes de que esa constante nunca cambia de valor.

```
# Ejemplo de constantes:
IVA = 0.21

precio = 25
precio_final = precio + (precio * IVA)
```

4. Almacenar valores

Otra solución muy utilizada es crear un script de Python, accesible desde nuestro proyecto, donde se almacenen todas las constantes. A continuación, importamos ese script con la instrucción import para poder llamar a esas constantes.

```
# Importamos el fichero de constantes.
import Constantes

precio = 25
precio_final = precio + (precio * Constantes.IVA)
```

5. Tipos de datos

Es necesario conocer los tipos de datos que podemos encontrarnos en Python para saber qué operadores podemos aplicar. A continuación, describiremos los tipos de datos básicos y los operadores que podemos aplicar a cada uno de estos tipos.

Datos numéricos

- **Enteros** (`int`): `26`, `0b1101` (base binaria), `0x3f4a` (base hexadecimal).
- **Flotante** (`float`): `3.14`, `5.`, `-67.763`
- **Complejos** (`complex`): `0.117j`

Usando estos tipos de datos, podemos aplicar los siguientes tipos de operadores: operadores aritméticos, operadores de asignación y operadores de bits.

5. Tipos de datos

Operadores aritméticos:

- **Suma** (+): devuelve como resultado la suma de dos números.

```
3.13 + 6 # Devuelve 9.13
```

- **Resta** (-): devuelve como resultado la resta de dos números.

```
3.13 - 6 # Devuelve -2.87
```

- **Multiplicación** (*): devuelve como resultado la multiplicación de dos números.

```
3.13 * 10 # Devuelve 31.3
```

- **División** (/): devuelve como resultado la división de dos números.

```
3.13 / 10 # Devuelve 0.313
```

- **División entera** (//): devuelve como resultado la división entera de dos números.

  Es decir, el resultado será la únicamente la parte entera de la división.

```
3 // 10 # Devuelve 0
```

- **Módulo** (%): devuelve como resultado el valor del resto obtenido de la división entera entre dos números.

```
3 % 10 # Devuelve 3
```

- **Exponente** (**): devuelve como resultado el valor exponencial de una base con respecto al exponente:

```
3 ** 2 # Devuelve 9
```

Operadores de asignación:

- **Asignación simple** (=): asigna a la variable del lado izquierdo el valor definido en la parte derecha.

```
resultado = 10 # resultado vale 10
```

- **Suma y asignación** (+=): el operador suma, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado += 10 # resultado vale 20
```

- **Resta y asignación** (-=): el operador resta, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado -= 10 # resultado vale 0
```

- **Multiplicación y asignación** (*=): el operador multiplica, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado *= 10 # resultado vale 100
```

- **División y asignación** (/=): el operador divide, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado /= 10 # resultado vale 1
```

- **División entera y asignación** (//=): el operador realiza la división entera al valor de la variable con respecto al valor definido en el lado derecho:

```
resultado = 14 # resultado vale 14
resultado //= 10 # resultado vale 1
```

- **Módulo y asignación** (%=): el operador asigna a la variable el resto de la división entera entre el valor de la variable y el valor definido en el lado derecho de la operación:

```
resultado = 14 # resultado vale 14
resultado %= 10 # resultado vale 4
```

- **Exponente y asignación** (**=): el operador asigna a la variable el resultado del exponente entre el valor de la variable y el valor de la derecha de la operación:

```
resultado = 3 # El resultado vale 3
resultado **= 2 # El resultado vale 9
```

Operaciones de bits:

- **AND** (&): operador lógico *and* a nivel de bits.

```
4 & 5 # El resultado será 4
```

- **OR** (|): operador lógico *or* a nivel de bits.

```
4 | 5 # El resultado será 5
```

- **XOR** (^): operador lógico *xor* a nivel de bits.

```
4 ^ 5 # El resultado será 1
```

- **Mover bits a la izquierda** (<<): operador que mueve todos los bits a la izquierda tantas posiciones como se indique en el lado derecho del operador.

```
4 << 1 # El resultado será 8
```

- **Mover bits a la derecha** (>>): operador que mueve todos los bits a la derecha tantas posiciones como se indique en el lado derecho del operador.

```
4 >> 1 # El resultado será 2
```

Datos booleanos

El tipo de datos booleano es un tipo binario cuyos valores solo pueden ser True o False. Se les puede aplicar los siguientes operadores lógicos, de comparación y de identidad:

- **AND** (and): operador lógico *and*.

```
True and False # Devolverá False
```

- **OR** (or): operador lógico *or*.

```
True or False # Devolverá True
```

- **NOT** (not): operador de negación lógica.

```
not True # Devolverá False
```

- **Menor** (<): operador que devuelve `True` si el valor de la izquierda es menor que el valor de la derecha. En caso contrario devolverá `False`.

```
5 < 7 # Devolverá True
```

- **Menor o igual** (<=): operador que devuelve `True` si el valor de la izquierda es menor o igual que el valor de la derecha. En caso contrario devolverá `False`.

```
7 <= 7 # Devolverá True
```

- **Mayor** (>): operador que devuelve `True` si el valor de la izquierda es mayor que el valor de la derecha. En caso contrario devolverá `False`.

```
5 > 7 # Devolverá False
```

- **Mayor o igual** (>=): operador que devuelve `True` si el valor de la izquierda es mayor o igual que el valor de la derecha. En caso contrario devolverá `False`.

```
7 >= 7 # Devolverá True
```

- **Igual** (==): operador que devuelve `True` si el valor de la izquierda es igual que el valor de la derecha. En caso contrario devolverá `False`.

```
5 == 7 # Devolverá False
```

- **Distinto** (!=): operador que devuelve `True` si el valor de la izquierda es distinto que el valor de la derecha. En caso contrario devolverá `False`.

```
7 != 7 # Devolverá False
```

Cadenas de caracteres

Las cadenas de texto son secuencias de caracteres encapsuladas con comillas simples ('') o comillas dobles (""). Podemos aplicarle diferentes operadores.

Para este apartado crearemos una variable llamada mensaje e iremos viendo las diferentes operaciones que podemos aplicar:

mensaje = "Esto es un mensaje de prueba para el curso de Python.“

Las cadenas de texto funcionan como una lista de caracteres. Por lo tanto, podemos obtener el carácter que existen en una posición concreta de la cadena.

```
mensaje[34] # Devuelve 'e'
```

Este operador no solo nos permite acceder a una única posición, sino que también podemos introducir un rango de caracteres.

```
CADENA[POSICION_INICIAL:POSICION_FINAL]
```

```
mensaje[4:15] # Devuelve ' es un mens'
```

Si dejamos la POSICION_INICIAL vacía, Python nos devolverá los caracteres desde el comienzo de la cadena hasta la POSICION_FINAL, y viceversa.

```
mensaje[:20] # Nos devolverá 'Esto es un mensaje d'
mensaje[40:] # Nos devolverá 'so de Python.'
```

En ambas posiciones se pueden incluir números negativos. En estos casos, Python empezará a contar las posiciones desde el lado derecho.

```
mensaje[-10:] # Desde la posición -10 (es decir, 10 desde la derecha)
hasta el final. Nos devolverá 'de Python.'
mensaje[:-10] # Desde la posición inicial, hasta la posición -10 (es
decir, 10 desde la derecha). Nos devolverá 'Esto es un mensaje de prueba
para el curso '
```

Operadores de cadenas de texto

- **Concatenar** (+): este operador devuelve una cadena de caracteres uniendo los caracteres de dos cadenas.

```
"Hola, " + "estoy bien" # Devolverá 'Hola, estoy bien'
```

- **Multiplicar** (*): este operador nos permite repetir una cadena de caracteres tantas veces como indiquemos en la parte derecha del operador.

```
"Hoy hace sol! " * 3 # Devolverá 'Hoy hace sol! Hoy hace sol! Hoy hace sol! '
```

5. Tipos de datos

- **len()**: esta función nos permite obtener la longitud de la cadena de caracteres, es decir, nos devuelve cuantos caracteres están contenidos en la cadena.

```
len(mensaje) # Devuelve 53
```

- **find()**: permite obtener la primera posición donde se encuentra la subcadena que pasamos por parámetro dentro de la cadena de texto original. En caso de que la subcadena no exista dentro de la cadena original, nos devolverá -1.

```
mensaje.find('Python') # Devolverá 46
mensaje.find('Java') # Devolverá -1
```

- **upper()**: convierte todos los caracteres de la cadena de texto en mayúsculas.

```
mensaje.upper() # Devolverá 'ESTO ES UN MENSAJE DE PRUEBA PARA EL CURSO DE PYTHON.'
```

- **lower()**: este operador convierte todos los caracteres de la cadena de texto en minúsculas.

```
mensaje.lower() # Devolverá 'esto es un mensaje de prueba para el curso de python.'
```

- **replace()**: nos permite modificar el contenido de la cadena de caracteres. Para ello, introducimos dos parámetros. El primero de ellos contiene la subcadena que queremos sustituir. El segundo contiene la cadena de texto que sustituirá a la primera subcadena. Si la primera subcadena no existe en la cadena de texto original, no habrá ningún cambio.

```
mensaje.replace("curso", "seminario") # Devolverá 'Esto es un mensaje de prueba para el seminario de Python.'
mensaje.replace("Java", "C++") # Devolverá 'Esto es un mensaje de prueba para el curso de Python.'
```

Tipo None

El tipo None se utiliza para definir que el valor de una variable es nada o ninguna cosa. Hay que tener cuidado con este tipo de datos, ya que es común utilizarlo cuando queremos declarar una variable, pero no le queremos asignar ningún valor.

```
variable = None
type(variable) # Devolverá que es de tipo NoneType
```

Es importante saber que None es un tipo de dato propio con su significado y que lo debemos diferenciar de valores por defecto de otros tipos como el booleano (False) o numérico (0).

```
None == False # Devolverá False
```

6. Transformación de tipos de datos

Vamos a hacer un repaso de las transformaciones de tipos de datos permitidas en Python.

- **Convertir a cadena de caracteres** (`str`): convierte un objeto que se pasa por parámetro a cadena de caracteres.

```
valor = 10
str(valor) # Devolverá ‘10’
```

- **Convertir a valor entero** (`int`): convierte un objeto que se pasa por parámetro al tipo entero. En caso de ser un número decimal, nos devolverá únicamente la parte entera. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = ‘Hola’
int(valor1) # Devolverá 10
int(valor2) # Devolverá un error de tipo ValueError
```

6. Transformación de tipos de datos

- **Convertir a valor flotante** (`float`): convierte un objeto que se pasa por parámetro al tipo flotante. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 30
valor2 = 'Hola'
float(valor1) # Devolverá 30.0
float(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor complejo** (`complex`): convierte un objeto que se pasa por parámetro al tipo complejo. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
complex(valor1) # Devolverá 10.78 + 0j
complex(valor2) # Devolverá un error de tipo ValueError
```

6. Transformación de tipos de datos

- **Convertir a valor complejo** (`complex`): convierte un objeto que se pasa por parámetro al tipo complejo. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
complex(valor1) # Devolverá 10.78 + 0j
complex(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor booleano** (`bool`): convierte un objeto que se pasa por parámetro al tipo booleano. Su funcionamiento es el siguiente:
  - Si no se pasa ningún parámetro, devolverá `False`.
  - En el resto de los casos devolverá `True` excepto si: el valor del parámetro es `0`, si es una secuencia vacía de alguna estructura de datos, si se pasa el tipo `None`, si se pasa el valor `False`.

```
valor1 = 10.78
valor2 = False
valor3 = 0
bool(valor1) # Devolverá True
bool(valor2) # Devolverá False
bool(valor3) # Devolverá False
```

7. Listas, tuplas, diccionarios y conjuntos

Veremos las estructuras de datos que nos proporciona Python para almacenar valores más complejos.

Listas

Una lista es una estructura que nos permite tener un conjunto de objetos separados por comas. Esta estructura de datos es **mutable**, es decir, podemos cambiar el valor de una lista que hemos creado como, por ejemplo, cambiar el orden de los elementos, eliminar elementos, etc.

En una lista **pueden existir elementos de diferentes tipos o estructuras de datos**. Para declarar una lista, escribimos **entre corchetes ([])** un conjunto de elementos separados por comas.

```
lista = [3, 'Hola', True]

lista_vacia = []
```

Acceso a los datos de la lista según índice:

```
lista = [3, 'Hola', True]
lista[2] # Devolverá el valor True

lista[:2] # Devolverá [3, 'Hola']
lista[1:] # Devolverá ['Hola', True]

lista[-1] # Devolverá True
lista[-2:] # Devolverá ['Hola', True]
```

Funciones aplicables a listas:

- **len**: devuelve la longitud de una lista, es decir, el número de elementos incluidos en una lista.

```
lista = [3, 'Hola', True]
len(lista) # Devolverá 3
```

- **index**: devuelve la posición que ocupa un elemento dentro de una lista.

```
lista = [3, 'Hola', True]
lista.index('Hola') # Devolverá 1
```

- **insert**: inserta un elemento dentro de una lista en la posición que le indicamos.

```
lista = [3, 'Hola', True]
lista.insert(1, 'Adiós')
lista # Nos mostrará [3, 'Adiós', 'Hola', True]
```

- **append**: inserta un elemento al final de la lista. En el caso de que pongamos una lista de elementos, la función lo insertará como un elemento único.

```
lista = [3, 'Hola', True]
lista.append([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, [3, 4]]
```

- **extend**: permite agregar un conjunto de elementos en una lista. A diferencia del método anterior, si incluimos una lista de elementos, se agregarán cada uno de los elementos a la lista.

```
lista = [3, 'Hola', True]
lista.extend([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, 3, 4]
```

- **remove**: elimina el elemento que pasamos por parámetro de la lista. En caso de que este elemento estuviese repetido, solo se eliminará la primera copia.

```
lista = [3, 'Hola', True, 'Hola']
lista.remove('Hola')
lista # Nos mostrará [3, True, 'Hola']
```

- **count**: devuelve el número de veces que se encuentra un elemento en una lista.

```
lista = [3, 'Hola', True, 'Hola']
lista.count('Hola')  # Nos devolverá 2
```

- **reverse**: este método nos permite invertir la posición de todos los elementos de la lista.

```
lista = [3, 'Hola', True]
lista.reverse()
lista # Nos mostrará [True, 'Hola', 3]
```

- **sort**: ordena los elementos de una lista. Por defecto, este método los ordena en orden creciente. Para ordenarlo de forma decreciente, hay que incluir el parámetro (`reverse=True`). ¡Ojo! La lista debe contener elementos del mismo tipo.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.sort()
lista # Nos mostrará [0, 1, 4, 5, 6, 7, 9]
lista.sort(reverse=True)
lista # Nos mostrará [9, 7, 6, 5, 4, 1, 0]
```

- **pop**: elimina y devuelve el elemento que se encuentra en la posición que se pasa por parámetro. En caso de que no se pase ningún valor por parámetro, eliminará y devolverá el último elemento de la lista.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.pop(1) # Devolverá 4
lista # Nos mostrará [6, 1, 9, 7, 0, 5]
```

Tuplas

Al igual que las listas, las tuplas son conjuntos de elementos separados por comas. Sin embargo, a diferencia de las listas, las tuplas son **inmutables**, es decir, no se pueden modificar una vez creadas.

```
tupla = 'Hola', 3.4, True, 'Hola'
```

Desempaquetado de tuplas:

```
w, x, y, z = tupla # w = 'Hola', x = 3.4, y = True, z = 'Hola'
```

Se accede a los elementos de la tupla de la misma manera que se hacía con las listas.

```
tupla[1] # Nos mostrará 3.4
tupla[1:] # Nos mostrará (3.4, True, 'Hola')
```

Métodos más utilizados en tuplas:

- **len**: método que devuelve la longitud de la tupla.

```
tupla = 'Hola', 3.4, True, 'Hola'
len(tupla) # Devolverá 4
```

- **count**: número de veces que se encuentra un elemento en una tupla.

```
tupla.count('Hola') # Devolverá 2
```

- **index**: devuelve la posición que ocupa un elemento dentro de una tupla. En caso de que el elemento este repetido, devolverá la primera posición donde aparece el objeto.

```
tupla.index('Hola') # Devolverá 0
```

Diccionarios

Los diccionarios conforman una estructura que enlaza los elementos almacenados con **claves** (keys) en lugar de índices. Es decir, para acceder a un objeto es necesario hacerlo a través de su clave.

Se trata de un conjunto de **pares (clave, valor), donde las claves son únicas**. Para crear un diccionario definiremos un conjunto de elementos clave valor delimitados por llaves (**{}**):

```
diccionario = {
      'clave1': 'Mi primer valor',
      'clave3': 'Y, como no, mi tercer valor',
      'clave2': 'Este es mi segundo valor'
}
```

Accedemos a los valores a través de la clave:

```
diccionario['clave1'] # Devolverá 'Mi primer valor'
```

Para crear nuevos elementos en los diccionarios usamos la misma forma de acceso a un elemento, pero asignando un nuevo valor:

```
diccionario['clave_nueva'] = ‘nuevo valor’
```

Podemos crear diccionarios vacíos usando las llaves, pero sin insertar ningún elemento, o usando la función dict():

```
diccionario = dict()
```

Podemos eliminar un elemento del diccionario usando la instrucción del e indicando qué elemento del diccionario queremos eliminar.

```
del diccionario['clave1'] # Eliminará el elemento con clave 'clave1'
```

Funciones más utilizadas en diccionarios:

- **list**: devuelve una lista de todas las claves incluidas en un diccionario. Si queremos la lista ordenada, usaremos la función **sorted** en lugar de **list**.

```
list(diccionario) # Devolverá ['clave1', 'clave3', 'clave2']
sorted(diccionario) # Devolverá ['clave1', 'clave2', 'clave3']
```

- **in**: comprueba si una clave se encuentra en el diccionario.

```
'clave3' in diccionario # Devolverá True
```

Conjuntos

Los conjuntos son **colecciones no ordenadas** de elementos. Además, los conjuntos no contienen repetición de elementos, es decir, **se eliminan todos los elementos duplicados**. Para crear un conjunto podemos indicar un conjunto de objetos **separados por comas y delimitados por llaves {}.**

```
conjunto = {'audi', 'mercedes', 'seat', 'ferrari', 'ferrari', 'renault'}
```

Para crear un conjunto vacío podemos crearlo usando las llaves sin insertar ningún valor o la función set().

```
conjunto = set()
conjunto # Nos mostrará {}
```

Al no ser una lista ordenada, no podemos acceder a sus elementos a través de su índice; esto nos devolverá un error de tipo.

Algunas funciones para conjuntos:

- **union**: operación matemática para obtener la unión de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.union(conjunto2) # Devolverá {1, 2, 3, 4, 5}
```

- **intersection**: operación matemática para obtener la intersección de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.intersection(conjunto2) # Devolverá {3}
```

- **difference**: operación matemática para obtener la diferencia del conjunto original con respecto al conjunto que se pasa por parámetro, es decir, los elementos del primer conjunto que no están en el segundo.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.difference(conjunto2) # Devolverá {1, 2}
```

- **in**: comprueba si un elemento se encuentra dentro de un conjunto.

```
1 in conjunto1 # Devolverá True
```

- **len**: devuelve la longitud del conjunto.

```
len(conjunto1) # Devolverá 3
```

8. Condicionales e iterativas

```
if (EXPRESION_LOGICA):
    sentencia_1
    ...
elif (EXPRESION_LOGICA_2):
    sentencia_1
    ...
elif (EXPRESION_LOGICA_3):
    sentencia_1
    ...
...
else:
    sentencia_1
    sentencia_2
    ...
```

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
elif (number == 5):
    print("Soy el número 5")
else:
    print("Soy menor o igual que 5")
```

8. Condicionales e iterativas

```
while (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
else:
    sentencia_1
    sentencia_2
    ...
```

```
numero = 5
fin = 0

while(numero > fin):
    numero -= 1
    print(numero)
else:
    print("Ya he acabado!")
```

8. Condicionales e iterativas

```
for VARIABLE in OBJETO:
    sentencia_1
    sentencia_2
    ...

else:
    sentencia_1
    sentencia_2
    ...
```

```
numeros = [5, 4, 3, 2, 1, 0]

for numero in numeros:
    print(numero)
else:
    print("Ya he acabado!")
```

```
texto = 'Hola mundo'


for caracter in texto:

    print(caracter)
```

```
lista = ['texto', 5, (23, 56)]


for elemento in lista:

    print(elemento)
```

8. Condicionales e iterativas

En las sentencias `for`, una forma de recorrer un objeto, como una lista, es a través de sus índices.

- Para poder hacer esto, en Python se utiliza la instrucción **range**.
- Esta instrucción nos devuelve un rango de números desde un número inicial hasta uno final, y con la separación entre números que hayamos seleccionado

```
# Secuencia de números de 0 a NUMERO_FINAL con paso 1
range(NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso 1
range(NUMERO_INICIAL, NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso PASO
range(NUMERO_INICIAL, NUMERO_FINAL, PASO)
```

- El siguiente ejemplo mostrará los caracteres que ocupan una posición par en la cadena:

```
cadena = 'Hola mundo'

# Comenzamos en 0, hasta la longitud de la cadena y con paso = 2
for i in range(0, len(cadena), 2):
    print(cadena[i])
```

8. Condicionales e iterativas

Break/Continue

- **break**: esta sentencia rompe la ejecución del bucle en el momento en que se ejecute.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Rompemos el bucle!')
        break

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Rompemos el bucle!
```

- **Continue**: esta instrucción permite saltarnos una iteración del bucle sin que se rompa la ejecución final.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Me salto una vuelta')
        continue

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Me salto una vuelta,
6, 7, 8, 9
```

8. Condicionales e iterativas

Iteradores

Otra forma de recorrer los elementos de un objeto en Python es utilizando iteradores. Un iterador es un objeto que, al aplicarlo sobre otro objeto iterable, como son las cadenas de texto o los conjuntos, nos permite obtener el siguiente elemento por visitar.

Para crear un iterador sobre un objeto usamos la instrucción iter(OBJETO) y se lo asignamos a una variable. Una vez creado, podemos visitar los elementos del OBJETO con la función next() y pasando por parámetro el identificador del iterador.

```
cadena = 'Hola'
iterador = iter(cadena)

next(iterador) # Devolverá 'H'
next(iterador) # Devolverá 'o'
next(iterador) # Devovlerá 'l'
next(iterador) # Devolverá 'a'
```

Las funciones nos permiten encapsular un bloque de instrucciones para que podamos utilizarlo varias veces dentro de nuestros programas.

Definición

```
def NOMBRE_FUNCION(ARG1, ARG2,...):
    sentencia_1
    sentencia_2
    ...
    return OBJETO_DEVOLVER
```

En el caso de que nuestra función no necesitase parámetros, dejaremos los paréntesis vacíos.

```
def area_triangulo(base, altura):
    area = (base * altura) / 2
    return area
```

```
area_triangulo(6, 5) # Devolverá 15.0
```

A la hora de llamar a una función existen dos formas para introducir los argumentos de la función:

- **Argumentos por posición:** los argumentos se envían en el mismo orden en el que se han definido los parámetros. Esta es la forma que hemos escogido para llamar a la función `area_triangulo` en el ejemplo anterior.
- **Argumentos por nombre:** los argumentos se envían utilizando los nombres de los parámetros que se han asignado en la función. Para ello, usaremos el nombre del parámetro, seguido del símbolo igual (=) y del argumento. En el ejemplo anterior sería de la siguiente forma:

```
area_triangulo(base=10, altura=4) # Devolverá 20.0
```

Cuando una función tiene definido unos parámetros, es **obligatorio** que, a la hora de llamarlo, **la función reciba el mismo número de argumentos**. En caso de que no recibiese alguno de esos argumentos, Python devolvería un error de tipo.

Sin embargo, en el momento de declarar una función, **podemos asignar un valor por defecto** a cada parámetro. Este valor por defecto se usará solo si no se ha indicado un argumento en el parámetro correspondiente.

```
def area_triangulo(base=10, altura=10):
    area = (base * altura) / 2
    return area
```

```
area_triangulo() # Devolverá 50.0
```

En algunas ocasiones, puede que necesitamos definir una función que necesite un número variable de argumentos. Para estos casos Python nos permite usar los **parámetros indeterminados** en las funciones. Existen dos formas de asignar los argumentos de este tipo:

- **Argumentos por posición:** se deben definir los parámetros como una lista dinámica. Para ello, a la hora de definir el parámetro, se incluirá un asterisco (*) antes del nombre del parámetro. Los parámetros indeterminados se recibirán por posición. A estos parámetros se les puede pasar cualquier tipo de dato en cada función. Un ejemplo de su uso sería el siguiente:

```
def imprime_numeros(*args):
    for numero in args:
        print(numero)

imprime_numeros(1, 7, 89, 46, 9394)
```

- **Argumentos por nombre:** para recibir varios argumentos por nombre, sin saber la cantidad, es necesario definir los parámetros como un diccionario dinámico. Para ello se usa dos asteriscos (**) antes del nombre del parámetro. Un ejemplo de su uso sería el siguiente:

```
def imprime_valores(**args):
    for argumento in args:
        print(argumento, '=>', args[argumento])

imprime_valores(arg1='Hola', arg2=[2,3,4], arg3=876.98)
```

Cuando devolvemos más de un valor en una función, lo que obtenemos es una tupla con todos los valores. Por este motivo, si queremos que cada uno de los resultados esté en una variable distinta, es necesario hacer un desempaquetado de la tupla.

```
def ejemplo():
    return "Hola", 3546, [3,90]
```

```
var1, var2, var3 = ejemplo() # Nos dará var1 = “Hola”, var2 = 3546, var3 =
[3, 90]
```

Debemos **documentar bien las funciones** implementadas, para recordar su función o mejorar la legibilidad de cara a otros programadores.

Para poder documentar las funciones se utilizan los **docstring**. En Python todos los objetos cuentan con una variable por defecto llamada **doc**, y nos permite acceder a la documentación de dicho objeto. Para documentar una función usando los docstring, únicamente tenemos que incluir un comentario inmediatamente después de la cabecera de la función.

- Podemos usar la sentencia **help()** con el nombre de la función para saber su documentación.

```
def potencia(base, exponente):
    """
    Función que calcula la potencia de dos números.

    Argumentos:
    base -- base de la operación.
    exponente -- exponente de la operación.
    """
    return base ** exponente
```

```
help(potencia)
```

```
Help on function potencia in module __main__:

potencia(base, exponente)
    Función que calcula la potencia de dos números.

    Argumentos:
    base -- base de la operación.
    exponente -- exponente de la operación.
```

Funciones incluidas en Python:

https://docs.python.org/es/3/library/functions.html

https://docs.python.org/es/3/library/index.html

Existen muchas librerías en Python, con funciones para múltiples usos.

Ejemplos de importación de módulos y librerías externas:

- Descargadas de la red (PySide6) o creadas por nosotros mismos (para agrupar funciones o constantes).

```
#Importar un módulo completo
import numpy
from numpy import *

#Importar submódulos de un módulo
from numpy import square, subtract
```

```
#Asignar pseudónimo a un módulo
from tensorflow import keras as k
import tensorflow as tf
```

```
#Importar un módulo de un paquete (PySide6)
from PySide6.QtWidgets import *

#Importar solo ciertos submódulos o nombres del módulo
from PySide6.QtWidgets import QApplication, QWidget, QLabel
```

Podemos crear un módulo y utilizar sus funciones importándolo desde nuestro código.

- Por ejemplo, podemos tener un módulo llamado circunferencia.py que tenga definidas las funciones perímetro y área:

```python
# Módulo circunferencia.py
import math


def perimetro(radio):
    return 2 * math.pi * radio


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

- En nuestro código lo importamos y utilizamos sus funciones:

```python
import circunferencia

circunferencia.perimetro(5) # Devolverá 31.41592653589793
```

Una carpeta con un conjunto de módulos, con un fichero __init__.py, será un paquete:

```
figuras/
├── __init__.py
├── circunferencia.py
└── poligono.py
```

```
import figuras


figuras.circunferencia.area(5)
figuras.poligono.area(10, 5)
```

```
from paquete import modulo1, modulo2, …, moduloN
from paquete import *
```

```
from figuras import *


circunferencia.area(5)
poligono.area(10, 5)
```

10. Objetos y clases

Una **clase** es, digamos, una estructura que tienen que seguir los objetos de dicha clase. En una clase se les define qué operaciones pueden hacer y qué características tienen. Para crear una clase en Python utilizaremos la palabra reservada **class**, seguida del nombre de la clase y de dos puntos (:).

Una clase puede tener definidos unos atributos y unos métodos:

```
class Libro:
    titulo = 'Don Quijote de la Mancha'
    autor = 'Miguel de Cervantes'
    isbn = '0987-7489'
    editorial = 'Mi Editorial'
    paginas = 934
    edicion = 34

    def imprime(self):
        print(self.titulo + " - " + self.autor)
```

10. Objetos y clases

Al utilizar el parámetro **self**, no necesitamos pasar ningún atributo a la hora de ejecutar un método, ya que hace referencia a la propia instancia del objeto desde el que se ejecuta.

```
mi_libro.imprime() # Devolverá Don Quijote de la Mancha - Miguel de Cervantes
```

Un método especial en las clases es el método **__init__().** Este método se ejecuta en el momento en que creamos un nuevo objeto. El comportamiento de esta función es muy similar al de los **constructores.**

Los parámetros que se incluyan en este método deberán hacerlo como argumentos a la hora de crear una instancia de dicha clase.

```
class Libro:

    def __init__(self, titulo, autor, isbn, editorial, paginas, edicion):
        self.titulo = titulo
        self.autor = autor
        self.isbn = isbn
        self.editorial = editorial
        self.paginas = paginas
        self.edicion = edicion

    def imprime(self):
        print(self.titulo + " - " + self.autor):
```

```
mi_libro = Libro('Don Quijote de la Mancha', 'Miguel de Cervantes', '0987-7489', 'Mi Editorial', 934, 34)
```

```
lista = [2, 3, 4]

try:
    print(lista[5])
except:
    print("No existe esa posición")
```

```
try:
    raise TypeError("Este es un ejemplo de error personalizado")
except:
    print("He detectado un error")
```

```
lista = [2, 3, 4]

try:
    print(lista[2])
except:
    print("No existe esa posición")
else:
    print("Está perfecto")
finally:
    print("Hemos terminado el bloque try-except")

print("Sigo con la ejecución")
```

El resultado de ejecutar este ejemplo será el siguiente:

```
4
Está perfecto
Hemos terminado el bloque try-except
Sigo con la ejecución
```

12. __name__ // __main__

Cuando un intérprete de Python lee un archivo de Python, **primero establece algunas variables especiales**. Luego ejecuta el código desde el archivo.

Una de esas variables se llama **__name__** .

Partiendo de que los archivos de Python se llaman módulos y se identifican mediante la extensión de archivo .py, cuando el intérprete ejecuta un módulo, establece el valor de la variable **__name__** cómo **__main__ .**

```
# Python file one module

print("File one __name__ is set to: {}" .format(__name__))
```

```
File one __name__ is set to: __main_
```

12. __name__ // __main__

Si importamos otro módulo y lo llamamos en el módulo principal que estamos ejecutando, este segundo módulo tendrá **__name__ = nombre del módulo.**

```
# Python module to import

print("File two __name__ is set to: {}" .format(__name__))
```

Creo e importo

Ejecuto file_one

```
# Python module to execute
import file_two

print("File one __name__ is set to: {}" .format(__name__))
```

```
File two __name__ is set to: file_two
File one __name__ is set to: __main__
```

12. __name__ // __main__

Se suele utilizar esta variable así:

```
if __name__ == "__main__":
    Do something here
```

```
# Python module to execute
import file_two

print("File one __name__ is set to: {}" .format(__name__))

if __name__ == "__main__":
    print("File one executed when ran directly")
else:
    print("File one executed when imported")
```

Podemos usar un bloque if __name__ == "__name__" para permitir o evitar que se ejecuten partes del código cuando sean importados los módulos.