Data Analytics

-

Python

gamma
tech school

Python - Marco teórico

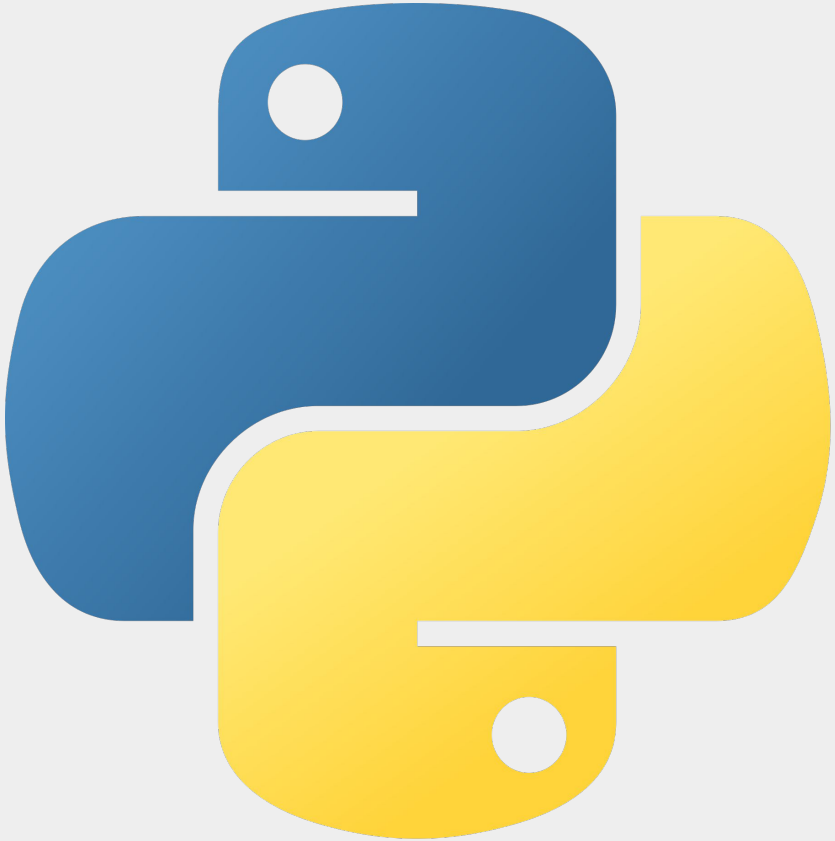
Índice

1. Introducción a Python

Python es uno de los lenguajes de programación dinámicos más populares que existen. Su autor principal, **Guido Van Rossum**, empezó a desarrollarlo a finales de la década de los 80.

Hizo la primera publicación de Python en alt.sources el **20 de febrero de 1991**.

Como curiosidad, su nombre es debido a que van Rossum era aficionado de la banda *Monty Python*.



1. Introducción a Python

Características

- **Lenguaje interpretado**
  Se ejecuta línea a línea. Si hay un error detiene la ejecución en esa línea, así se encuentran errores con rapidez.
- **Fácil de utilizar**
  Utiliza palabras, similares al inglés. No utiliza llaves, utiliza sangría.
- **Tipeado dinámicamente**
  No hay que anunciar el tipo de variable, Python lo determina en el tiempo de ejecución.
- **De alto nivel**
  Es más cercano a los idiomas humanos que otros lenguajes de programación.
- **Orientado a objetos**
  Lo considera todo como un objeto aunque también admite la programación estructurada y funcional.

```
setOfNumbers = []

print("How many random numbers do you want to generate?")
max = int(input())

for i in range (max):
    setOfNumbers.append(random.randrange(1,101,1))
setOfNumbers.sort()
print(setOfNumbers)

print("Which number do you want to find in the set of random numbers")
searchNumber = int(input())

firstPos = 0
lastPos = max-1
found = False

while (not found and firstPos <= lastPos):
    midPos = int((firstPos + lastPos)/2)

    if (searchNumber == setOfNumbers[midPos]) :
        found = True
    else :
        if (searchNumber < setOfNumbers[midPos]):
            lastPos = midPos - 1
        else :
            firstPos = midPos + 1


if (found) :
    print("Your item is in the list")
else :
    print("Your item is not in the list")
```

1. Introducción a Python

Beneficios

- **Fácil comprensión y lectura**, su sintaxis es parecida a la del inglés
- **Mayor productividad**, necesita menos líneas de código
- **Gran biblioteca**, no es necesario escribir código desde cero
- **Se utiliza fácilmente con otros lenguajes de programación**
- **Millones de desarrolladores**, hay una gran comunidad y es fácil obtener ayuda
- **Millones de recursos**, es muy fácil encontrar todo tipo de tutoriales
- **Compatible con muchos sistemas operativos**, como Windows, MacOS, Linux, Unix…

1. Introducción a Python

Principales librerías

- **Numpy**: facilita operaciones matemáticas
- **Pandas**: especializada en manipulación y análisis de datos
- **Matplotlib**: para la generación de gráficos
- **Seaborn**: también para gráficos, basada en matplotlib
- **Requests**: permite hacer peticiones HTTP
- **BeatifulSoup**: facilita la extracción de información HTML
- **Selenium**: webdriver para automatizar pruebas
- **Scikit-learn:** aprendizaje automático

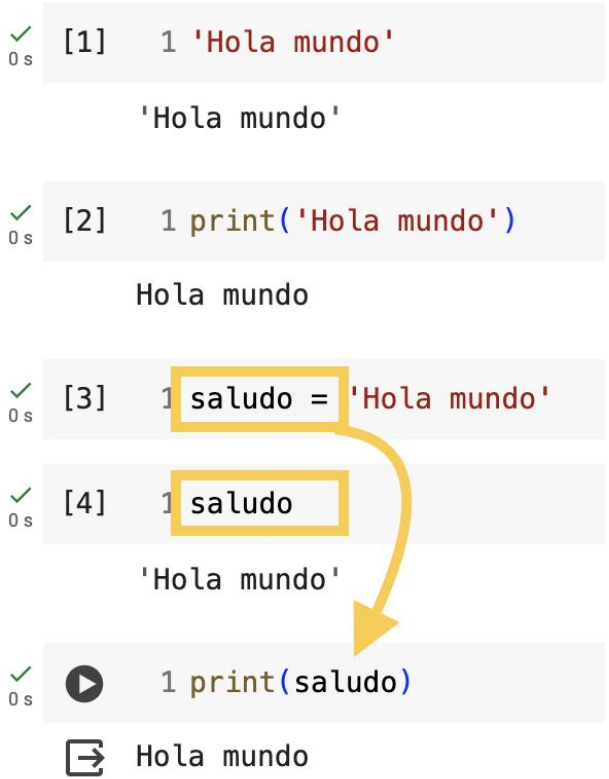
2. Fundamentos: variables y tipos de datos

Variable

Una variable es el nombre que se le da a algo. Es una manera de almacenar algo que vamos a usar recurrentemente para facilitar la lectura y el uso de nuestro código.

En una variable podemos almacenar cadenas de texto, números, listas, sets, tuplas, diccionarios, DataFrames, gráficos…

Facilita la interacción con aquello que hayamos almacenado en la variable y te permitirá ser más eficiente trabajando.



2. Fundamentos: variables y tipos de datos

Datos numéricos

- **INT**: son números enteros. Se pueden escribir como son, en formato binario, octal o hexadecimal.
- **FLOAT**: *más o menos*, representan el conjunto de números reales. Es más complejo de explicar pero es el tipo numérico que permite números decimales.
- **COMPLEX**: los números complejos tienen una parte real y otra imaginaria y, ambas, se representan con un FLOAT.

➔ Se puede hacer operaciones con todos los números indistintamente del tipo que sean.

➔ El tipo más pequeño se convertirá automáticamente al tipo superior.

int → float → complex

➔ Si haces una operación, como por ejemplo, 1+1.5 se va a traducir a 1.0+1.5

```
int = 1
```

```
float = 3.5
```

```
complex = 0j
```

```
str = 'Esto es Python'
```

```
list = [1, 3.5, 'Hola', {'Python':'código'}]
```

```
tuple = (1, 3.5, 'Hola')
```

```
set = {1, 2, 3, 4, 5}
```

```
dict = {'Academia':'Gamma', 'ciudad':'Madrid'}
```

2. Fundamentos: variables y tipos de datos

Otros tipos básicos

- **BOOL**: es la clase que representa los valores *booleanos*. Devuelve true o false.
- **STR**: es una cadena de caracteres conocida como *string*. Se escribe entre comillas simples ‘’ o dobles “”

Otros tipos complejos

- **LIST**: es una secuencia mutable de valores.
- **TUPLE**: es una secuencia inmutable de valores.
- **SET**: conjunto de elementos únicos y ordenados, en un conjunto no se puede repetir un mismo elemento.
- **DICT**: es un contenedor que se compone por parejas de **clave:valor**

```
int = 1
```

```
float = 3.5
```

```
complex = 0j
```

```
str = 'Esto es Python'
```

```
list = [1, 3.5, 'Hola', {'Python':'código'}]
```

```
tuple = (1, 3.5, 'Hola')
```

```
set = {1, 2, 3, 4, 5}
```

```
dict = {'Academia':'Gamma', 'ciudad':'Madrid'}
```

2. Fundamentos: variables y tipos de datos

Estructura de datos

- **Datos básicos**: numéricos, booleanos, strings (cadenas de texto).
- **Datos complejos**: secuencias (list, tuple, range), mapas (dict), conjunto (set), iteradores, clases, instancias, excepciones.

```
int = 1
```

```
float = 3.5
```

```
complex = 0j
```

```
str = 'Esto es Python'
```

```
list = [1, 3.5, 'Hola', {'Python':'código'}]
```

```
tuple = (1, 3.5, 'Hola')
```

```
set = {1, 2, 3, 4, 5}
```

```
dict = {'Academia':'Gamma', 'ciudad':'Madrid'}
```

3. Operadores

Hay diferentes tipos de operadores:

- **Operadores lógicos o booleanos**
- **Operadores de comparación**
- **Operadores aritméticos**
- **Operadores de asignación**
- **Operadores de pertenencia**
- **Operadores de identidad**

Operadores lógicos o booleanos

`and` --> Si el primer operando es verdadero evalúa el segundo

`or` --> Si el primer operando es falso evalúa el segundo

`not` --> Tiene menos prioridad. Si es falso, va a devolver TRUE

```
[1]  1 # Ejemplo: 1 = True, 0 = False
     2 1 and 0
```

```
0
```

```
[2]  1 1 or 0
```

```
1
```

```
[3]  1 not 0
```

```
True
```

3. Operadores de comparación

Operadores de comparación : van a hacer la comparación y devolverán `True` o `False` en función si se cumple o no la condición.

`<` --> menor que

`>` --> mayor que

`<=` --> menor o igual que

`>=` --> mayor o igual que

`==` --> igual que

`!=` --> distinto a

```
1 1 < 2
```

```
True
```

```
1 1 > 2
```

```
False
```

```
1 len('Hola') <= 4
```

```
True
```

```
1 3 >= 5
```

```
False
```

```
1 'Hola' == 'Pepe'
```

```
False
```

```
1 'Hola' != 'Pepe'
```

```
True
```

3. Operadores aritméticos

Operadores aritméticos: muy utilizados. Para hacer operaciones.

`+` --> suma

`-` --> resta

`*` --> multiplicación

`/` --> división. Devuelve un `float`

`%` --> módulo. Devuelve el resto de la división

`//` --> devuelve el cociente entero de la división

`**` --> potencia

```
1 3 + 2 # suma
```

```
5
```

```
1 3 - 2 # resta
```

```
1
```

```
1 3 * 2 # multiplicación
```

```
6
```

```
1 3 / 1 # división, devuelve float
```

```
3.0
```

```
1 6 % 2 # devuelve resto
```

```
0
```

```
1 6 // 2 # devuelve cociente entero
```

```
3
```

```
1 3 ** 2 # potencia
```

```
9
```

3. Operadores de asignación

Operadores de asignación: el operador de asignación es `=`, que se usa para asignar valor a una variable.

Existen los operadores de asignación compuestos, que realizan una operación sobre la variable que se asignan.

Realizan sobre la variable y el valor que haya a la derecha del `=` la operación que se indique delante del igual.

```
x += 2  # equivalente a x + 2
x -= 2  # equivalente a x - 2
x *= 2  # equivalente a x * 2
x /= 2  # equivalente a x / 2
x %= 2  # equivalente a x % 2
x //= 2 # equivalente a x // 2
x **= 2 # equivalente a x ** 2
x &= 2  # equivalente a x &= 2
x |= 2  # equivalente a x | 2
x ^= 2  # equivalente a x ^ 2
x >>= 2 # equivalente a x >> 2
x <<= 2 # equivalente a x << 2
```

3. Operadores de pertenencia/identidad

Pertenencia

- Utilizados para comprobar si un valor, o una variable, se encuentran en una secuencia: list, tuple, dict, set o str.
- Devuelven True o False, en función de si se confirma o no la pertenencia de ese valor dentro de esa secuencia.

Identidad

- Sirven para comprobar que dos variables sean, o no, un mismo elemento.

Operador	Descripción
in	Si el valor se encuentra en la secuencia devuelve True.
not in	Si el valor **no** se encuentra en la secuencia devuelve False.

Operador	Descripción
is	Devuelve True si ambos operandos hacen referencia al mismo objeto.
is not	Devuelve False si los operandos hacen referencia a objetos distintos.

3. Operadores jerarquía

Se debe tener en cuenta la jerarquía de las operaciones.

1. Paréntesis ()
2. Exponenciación **
3. Signos (positivo o negativo) +num -num
4. Multiplicación * - División / - División entera // - Módulo %
5. Suma `+` - Resta `-`

Además, hay jerarquía en las operaciones de comparación, asignación, identidad, pertenencia y lógicas.

Después de las operaciones aritméticas se realizan las de comparación.

Luego las booleanas (not, and, or)

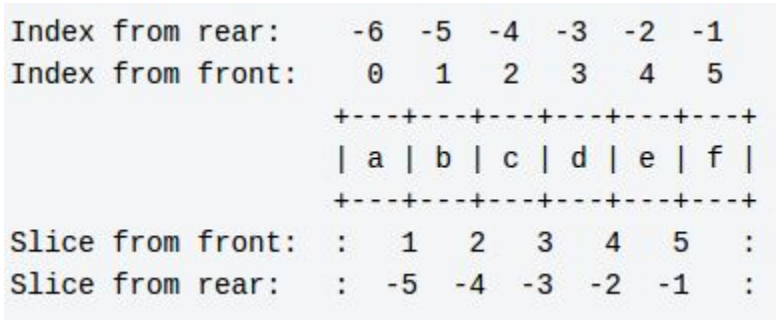
4. ¿Cómo funciona: índice?

El índice

- Permite acceder, modificar o consultar elementos dentro de una secuencia.
- Siempre empieza a contar desde el 0.
  - Primer elemento: `[0]`
  - Segundo elemento: `[1]`
- Los índices negativos indican las últimas posiciones.
  - Última `[-1]`
  - Penúltima `[-2]`

El slicing

- `('start':'stop':'step')`
  - **start**: en qué posición empieza, incluyendo el elemento.
  - **stop**: en qué posición acaba, excluyendo el elemento.
  - **step**: determina el incremento entre índices en una subsecuencia.
- Indicar un slicing vacío implica coger todos los elementos, de principio a fin.



4. ¿Cómo funciona: string?

El tipo *str*

- Representa texto.
- Se conoce como **string** o cadena de texto.
- Es una secuencia **inmutable** de caracteres.
- No puede modificarse una vez creada.

Cómo declarar una *string*

- Siempre se declara entre comillas simples o dobles(`''` ó `""`)
- Puedes hacer una **string multilínea** con tres comillas, simples o dobles, al principio y al final (`'''...'''` ó `"""..."""`)
- Puedes sumar cadenas, y se concatenarán.
- Puedes multiplicarlas, y se repetirán.

4. ¿Cómo funciona: string?

Formato de una string

Hay varias maneras de dar formato a una cadena. Desde Python 3.6 en adelante, la más eficiente es la **f-string**

- **Operador `%`:** se puede utilizar el operador % para indicar el lugar de las diferentes variables.
- **Método .format()**: añade al hueco que se deja entre claves `{ }` el valor pertinente en el orden que se indique.
- **f-string**: permite incluir entre las claves `{ }` el valor que se indique, sí, **dentro** de la propia string.

4. ¿Cómo funciona: string?

Métodos de cadenas

- **.capitalize():** convierte la primera letra en mayúscula.
- **.count():** cuenta las veces que aparece una subcadena.
- **.format():** formatea la cadena de texto.
- **.isalnum()/.isalpha()/.islower()/.isupper():** para hacer comprobaciones de si es alfanumérica, alfabética, está en minúsculas, en mayúsculas… (hay muchos más).
- **.join():** para hacer uniones.
- **.lower()/.upper():** convertir toda la cadena en minúsculas/mayúsculas.
- **.replace(**‘quiero reemplazar esto’**,** ‘por esto otro’**):** para hacer reemplazos en una cadena.
- **.split():** trocea la cadena en elementos de una lista, por el lugar que se le indique.
- **.strip():** elimina caracteres especificados a inicio y fin de la cadena.

A parte de estos métodos, hay muchos más. Recuerda visitar el manual con frecuencia para tener siempre a mano todos los métodos.

https://docs.python.org/es/3/library/stdtypes.html#string-methods

4. ¿Cómo funciona: list?

El tipo *list*

- Versátiles y fundamentales para la programación en este lenguaje.
- Agrupa valores, pudiendo ser de diferentes tipos.
- Es una secuencia **ordenada**.
    - Visita la documentación

Declarar una lista

- Puedes usar corchetes vacíos `[ ]`
- Corchetes, separando valores con comas: `['a'], ['a', 'b', 'c']`
- Lista comprimida: `[x for x in iterable]`
- Con el constructor `list()` o `list(iterable)`.

4. ¿Cómo funciona: list?

Características

- **Mutabilidad**: son mutables, puedes agregar, modificar o eliminar elementos de una lista.
- **Ordenadas**: los elementos aparecerán en la lista en el mismo orden que hayan sido añadidos.
- **Indexación y slicing**: puedo acceder a elementos individuales o a subsecuencias.
- **Heterogénea**: una lista puede contener elementos de tipos diferentes, incluidos complejos como otras listas, diccionarios, o cualquier objeto definido por el usuario.
- **Dinámicas**: las listas pueden crecer o encogerse durante el proceso de ejecución.
- **Anidación**: pueden contener otras listas como elementos, permitiendo crear estructuras de datos matriciales más complejas, como las listas de listas.
- **Iterables**: se puede recorrer sus elementos utilizando un bucle for, u otros elementos de indexación.
- **Flexibilidad para su inclusión en estructuras compuestas**: Pueden ser usadas como valores en diccionarios, elementos de sets (aunque las listas mismas no pueden ser incluidas en sets directamente debido a su mutabilidad, pero pueden ser convertidas a tuplas), y más.

4. ¿Cómo funciona: list?

Otros métodos útiles para listas

- **.split()**: divide una string en elementos de una lista.
- **‘’.join()**: une los elementos de una lista en una cadena.

Su uso conjunto es muy útil para el procesamiento de texto, permitiéndote realizar tareas como tokenización, limpieza de datos, reestructuración de texto…

Sorted()

La función **sorted(***iterable, * , key=None, reverse=True***)** sirve para ordenar los elementos de un iterable (lista, tupla, diccionario) en un orden específico.

Métodos de listas

- **.append(elemento)**: Agrega un elemento al final de la lista.
- **.extend(iterable)**: Extiende la lista agregando todos los elementos del iterable.
- **.insert(i, elemento)**: Inserta un elemento en la posición especificada.
- **.remove(elemento)**: Elimina el primer elemento cuyo valor es igual al especificado.
- **.pop([i])**: Elimina el elemento en la posición dada y lo devuelve. Si no se especifica índice, .pop() elimina y devuelve el último elemento de la lista.
- **.clear()**: Elimina todos los elementos de la lista.
- **.index(elemento[, start[, end]])**: Devuelve el índice del primer elemento con el valor especificado.
- **.count(elemento)**: Cuenta el número de veces que aparece el elemento en la lista.
- **.sort([key=función, reverse=False])**: Ordena los elementos de la lista in situ CUIDADO **devuelve un None**.

4. ¿Cómo funciona: tuple?

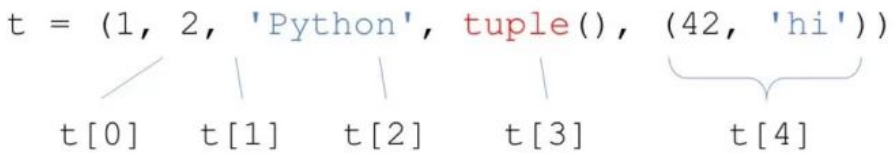
El tipo *tuple*

- Colección ordenada e **inmutable** de elementos.
- Pueden entenderse como una versión inmutable de las listas.
- Pueden contener elementos compuestos y objetos, como listas, otras tuplas…
    - Visita la documentación

Tuples in Python

Declarar una tupla

- Se pueden crear utilizando paréntesis **( )** y separando los elementos por comas.
- Con el constructor **tuple()**.
- Si es de un único elemento elem, o (elem,)
- Si es de varios elementos a, b, c, o (a, b, c)



4. ¿Cómo funciona: tuple?

Características de una tupla

- **Inmutables**: no se pueden modificar después de su creación.
- **Ordenadas**: los elementos tienen un orden definido, el cual no cambiará.
- **Permiten duplicados**: pueden contener elementos duplicados.
- **Indexación**: se pueden acceder a sus elementos por su índice.

4. ¿Cómo funciona: tuple?

Operaciones con tuplas

- **Indexación y slicing**: se puede acceder a los elementos de las tuplas por sus índices y obtener sub-tuplas.
- **Concatenación y repetición**: aunque las tuplas son inmutables, se pueden combinar y repetir para conseguir nuevas tuplas.

Inmutabilidad y uso práctico

- **Por qué son inmutables**: discute la importancia de la inmutabilidad en contextos donde se requiere que los datos no cambien, como claves de diccionarios.
- **Desempaquetado de tuplas:** muestra cómo asignar los elementos de una tupla a variables.
- **Tuplas como retorno de funciones**: las funciones pueden devolver múltiples valores para permitir múltiples valores de retorno.

Métodos y funciones

- **Funciones incorporadas**: pueden usarse con las tuplas.
  - **len()**
  - **max()**
  - **min()**
  - **sum()**
- **Métodos de tuplas**: las tuplas tienen menos métodos disponibles que las listas, debido a su inmutabilidad.
  - **.count()**
  - **.index()**

4. ¿Cómo funciona: set?

El tipo *set*

- Un set es una colección desordenada, mutable e indexada de elementos únicos. Almacenan múltiples variables en un único elemento.
    - Visita la documentación

Creación de sets

- Se pueden crear **sets** utilizando llaves { } o la función **set()**
    - **ATENCIÓN:** tenéis que crearlo con **set()**. Si utilizas { } crearás un diccionario vacío.

```
#### Creating Sets ####

# Defining sets
set1={2,2,3,1,1,4,4}
set2={5,5,3,4,5,5}
set3={'a','c','c','b','b'}

print(type(set1))

# printing the sets
print(set1)
print(set2)
print(set3)
```

```
<class 'set'>
{1, 2, 3, 4}
{3, 4, 5}
{'b', 'a', 'c'}
```

4. ¿Cómo funciona: set?

Características de un set

- **Desordenados**: Los elementos no mantienen el orden en el que son declarados.
- **Únicos**: No permiten elementos duplicados, automáticamente filtran duplicados.
- **Mutables**: Puedes añadir o eliminar elementos de un set.
- **No indexados**: No se puede acceder a los elementos de un set por un índice o clave.

4. ¿Cómo funciona: set?

Operaciones básicas con sets

- **Agregar elementos:**
  - **.add()**: para añadir elementos
  - **.update()**: para añadir múltiples elementos
- **Eliminar elementos:**
  - **.remove(*element*):** elimina el elemento, si no lo encuentra lanza un error.
  - **.discard(*element*):** elimina el elemento, si no lo encuentra no hace nada.
  - **.pop():** elimina un elemento al azar.
  - **.clear():** elimina todos los elementos.
- **Longitud del set:**
  - **len()**

Operaciones de conjuntos:

- **Unión:** para combinar sets sin duplicados.
  - | o **.union()**
- **Intersección:** para obtener elementos presentes en ambos sets.
  - **&** o **.intersection()**
- **Diferencia:** para obtener elementos únicos en diferentes sets.
  - **-** o **.difference()**
- **Diferencia simétrica:** para elementos que están en uno de los sets pero no en ambos.
  - ^ o **.symmetric_difference()**

4. ¿Cómo funciona: set?

Iteraciones básicas con sets

- **Bucle for:** para iterar sobre los elementos del set.
- Comprobar si un elemento está presente con **in**.

Métodos útiles con sets

- **.clear():** vacía el set.
- **.copy()**: devuelve una copia del set.
- **.issubset()/.issuperset()/.isdisjoint():** para hacer comparaciones entre sets.

Uso práctico del set

- **Eliminación de Duplicados**: Cómo utilizar sets para filtrar duplicados de una lista.
- **Operaciones de Conjunto en Análisis de Datos**: Ejemplos de cómo los sets pueden ser útiles para análisis rápidos y filtrado de datos únicos.
- **Relaciones entre Conjuntos**: Uso de sets para modelar y analizar relaciones matemáticas o lógicas.

4. ¿Cómo funciona:dict?

El tipo *dict*

- Es una estructura muy utilizada en Python y también muy versátil.
- Permite almacenar pares `clave:valor`
- Son indexados. Sirven para almacenar datos que están conectados de alguna manera.
- La clave es inmutable.

Crear un diccionario

- Puedes usar { } o **dict()**
- Dentro van valores clave:valor separados por comas

```
1  # Create a dictionary
2
3  my_dict = {'Alex': 5,
4             'Ben' : 10,
5             'Carly': 12,
6             'Danielle': 7,
7             'Evan' : 6}
8  my_dict
```

```
{'Alex': 5, 'Ben': 10, 'Carly': 12, 'Danielle': 7, 'Evan': 6}
```

Documentación diccionario

4. ¿Cómo funciona:dict?

Características de un diccionario

- **Ordenados**: Los elementos no se almacenan el orden de inserción.
- **Mutables**: Puedes cambiar, agregar o eliminar pares `clave:valor` después de que el diccionario ha sido creado.
- **Indexados por claves**: Utilizas claves únicas para acceder a los valores, no índices.
- **Claves únicas**: Cada clave debe ser única en el diccionario

4. ¿Cómo funciona:dict?

Operaciones con los diccionarios

- **Acceder a un valor**
  - **diccionario[‘clave’]**
  - si la clave no existe en el diccionario la va a añadir.
  - si la clave ya existe se modificará su valor.
- **Métodos para ver las *key:value***
  - **.keys():** devuelve una lista de las claves.
  - **.values()**: devuelve una lista de los valores.
  - **.items()**: devuelve una lista de tuplas con los pares `clave:valor`.

Métodos de un diccionario

- **Otros métodos:**
  - **.clear()**: elimina todos los elementos del diccionario.
  - **.copy()**: hace una copia superficial del diccionario.
  - **.pop(*key*)**: si *key* está en el diccionario lo elimina y nos devuelve su valor.
  - **.popitem()**: elimina el último par key:value y devuelve su valor. En versiones anteriores a Python 3.7 eliminará un par key:value al azar.
- **Otras funciones:**
  - **list(*dict*)**: devuelve una lista de todas las claves del diccionario.
  - **len(*dict*)**: devuelve el número de elementos almacenados en un diccionario.

5. Estructura de control: condicionales

- **Toma de decisiones**: evalúan una condición, si se cumple se realiza la instrucción.
- **Control de flujo**: en función de las condiciones se ejecutan unas partes u otras del código.
- **Filtran datos**: selecciona o descarta datos basándose en si cumplen o no ciertos criterios.
- **Validan** los datos introducidos externamente.

```
if condicion1:
    # Bloque de código que se ejecuta si condicion1 es verdadera
elif condicion2:
    # Bloque de código que se ejecuta si condicion1 es falsa y condicion2 es verdadera
else:
    # Bloque de código que se ejecuta si condicion1 y condicion2 son falsas
```

Aplicaciones comunes en análisis de datos

1. **Filtrar conjuntos de datos**: Por ejemplo, seleccionar solo las filas de un DataFrame que cumplan ciertas condiciones.
2. **Clasificación de datos**: Categorizar o etiquetar datos basados en rangos o condiciones específicas.
3. **Limpieza de datos**: Identificar y manejar valores nulos, atípicos o errores en los datos.
4. **Transformaciones condicionales**: Aplicar diferentes transformaciones o cálculos a un conjunto de datos basándose en condiciones específicas.

5. Estructura de control: condicionales

```
a=7

if a==0:
    print('🤪')

elif (a>5 and not a%2) or a == 7:
    print('Aprobado')

else:
    print('Cate')
    if a == -8:
        print(f'has sacado un {a}')
```

Sentecia if…… :

- hace una evaluación de la condición.
- si es True va a hacer esta instrucción.

elif……:

- abreviatura de *else if*.
- puede haber cero o más bloques *elif*.

else:

- es opcional
- si todas las condiciones previas son falsas
- qué hacer con el resto de casos.

5. Estructura de control: condicionales

Ejecución:

- las condiciones se ejecutan **de arriba a abajo**.
- cuando se cumple una de las condiciones se ejecuta la instrucción asociada.
- cuando una condición es `True` el resto de condiciones no se van a evaluar.

Indentación:

- el código en Python usa la indentación para definir los bloques de código.
- son espacios al inicio de una línea de código.
- se utilizan niveles de indentación, que por norma general son **4 espacios**.
- debe tener consistencia, no puede haber diferentes espacios para un mismo nivel.
- otros lenguajes de programación utilizan claves `{}`
- se utiliza con los bloques de código asociados, como:
  - condiciones `if`...`elif`...`else`
  - bucles `for` y `while`
  - funciones `def`
  - clases `class`

5. Estructura de control: bucle *for*

¿Qué es un bucle *for*?

- se utiliza para recorrer objetos **iterables** elemento a elemento.
- en cada ciclo del bucle hará las instrucciones u operaciones sobre el elemento de ese ciclo.
- lo hará por cada elemento que tenga el iterable que va a recorrer.
- cuando se ejecute sobre el último elemento finalizará.

```
n=5
for i in range(1,n*2):
    if i<=n:
        for j in range(i,0, -1):
            print('*', end='')
    else:
         for j in range(1,n*2+1-i):
            print('*', end='')

    print()
```

¿Qué es un objeto *iterable*?

- objeto que se puede recorrer elemento a elemento.
    - string
    - lista
    - tupla
    - set
    - diccionario (de una manera un poco particular)

5. Estructura de control: bucle *for*

Bucle y la clase *range*

- **range()** permite simular el bucle basado en una secuencia numérica.
- se le pueden pasar hasta **tres argumentos** ¿recuerdas el slicing `[start:stop:step]`? Pues es muy parecido.
    - **range(max)**: empieza en 0 y acaba en max-1.
    - **range(min, max)**: empieza en min y acaba en max-1.
    - **range(min, max, step)**: empieza en min, acaba en max-1 y los valores se incrementan de step en step.

5. Estructura de control: bucle *for*

Recorrer los elementos de un iterable

- la sintaxis es la siguiente:
  - ```
    for element in iterable:
        #código de lo que hago
    ```
- el iterable será el objeto que quiero recorrer.
- 'element' es el elemento dentro del objeto.
  - puedes darle el nombre que quieras.

Break y continue

- **break**: podemos finalizar el bucle y salir de él.
  - útil cuando hay condiciones y si se cumple una no quieres que continúe.
- **continue**: salta al siguiente paso de la iteración, la siguiente repetición, ignorando todas las sentencias que le siguen.

Bucle for para *diccionarios*

- los diccionarios tienen tres métodos que hacen listas.
  - **dict.keys()**: devuelve una lista de sus **key**
  - **dict.values()**: devuelve una lista de sus **value**
  - **dict.items()**: devuelve una lista de tuples con los pares **key:value**
- Al tener todas las key, value, o ambas, en una lista ya podemos recorrer el diccionario completo y hacer las operaciones que sean oportunas.
    ```

5. Estructura de control: bucle *while*

¿Qué es un bucle *while*?

- se ejecuta mientras se cumpla una **condición**.
- se utiliza cuando no se conoce cuántas iteraciones tiene que hacer.
- puede haber varios tipos de condición:
  - literales
  - valor de una variable
  - resultado de una expresión
  - valor devuelto por una función

Peligro de bucle infinito

- es esencial mantener el control de la condición.
- si siempre se da que la condición es **True** va a estar iterando infinitamente.
- *break* y *continue* en el uso de las condiciones, en caso de tenerlas dentro del bucle, pueden ayudarnos a tener mayor control.

```
a=0


while a<25:
    a+=5   # a = a + 5
    print(f'El valor de a es {a}')

print('\nhasta aquí mi bucle')
```

5. Estructura de control: bucles

TIPS para un buen bucle

- utiliza nombres para los elementos lo más descriptivos posible, así tu código será mucho más legible.
- utiliza **enumerate(*iterable*)** para recorrer el iterable recorriendo el índice y el valor del elemento.
- las **listas comprimidas** son más concisas.
- utiliza **zip(*iterable1*, *iterable2*)** para operar sobre múltiples secuencias simultáneamente.

5. Estructura de control: bucles

TIPS para un buen bucle

- utiliza nombres para los elementos lo más descriptivos posible, así tu código será mucho más legible.
- utiliza **enumerate(*iterable*)** para recorrer el iterable recorriendo el índice y el valor del elemento.
- las **listas comprimidas** son más concisas.
- utiliza **zip(*iterable1*, *iterable2*)** para operar sobre múltiples secuencias simultáneamente.

CUIDADO CON EL POP

```
#en una lista.pop() elimina el último

#en un set.pop() elimina el primero

#en un diccionario.pop() le tengo que decir qué key
debe eliminar
```

6. Funciones

Método vs función

Método

- similar a una función, pero asociado a un objeto o clase.
- se invocan sobre un objeto.
- pueden acceder a la información de ese objeto.

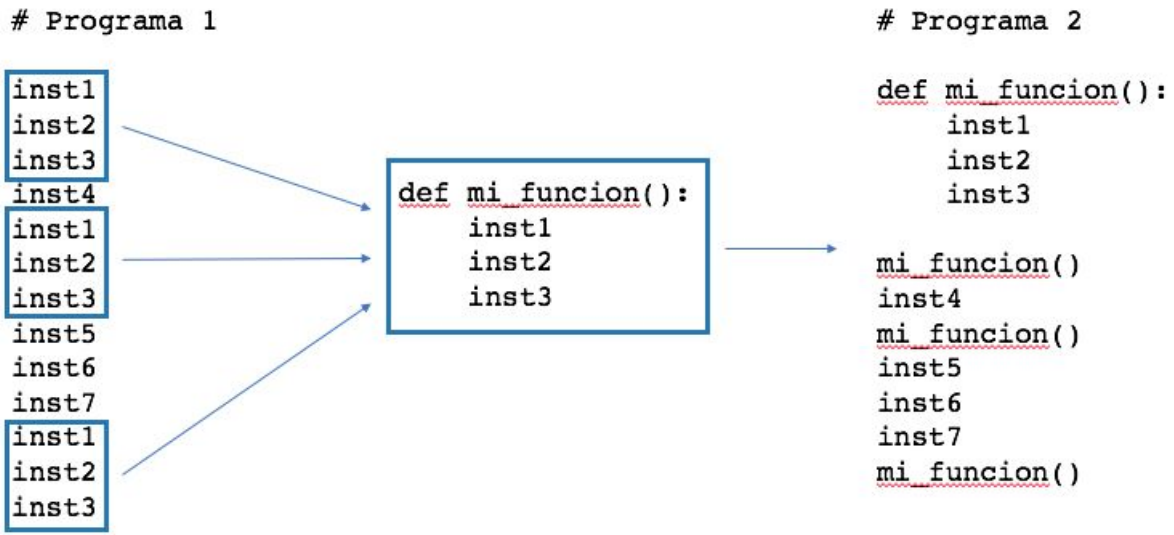
Función

- bloque de código reutilizable.
- diseñada para una tarea específica.
- se definen utilizando la palabra **def**
- se pueden llamar en cualquier parte del código una vez ya está definida.

6. Funciones

Objetivos

- dividir el código en partes más sencillas.
- encapsular el código que se repite.



6. Funciones

Partes de una función

- Al definirla: **def** nombre_función(parámetros)**:**
- **nombre**: al llamarla usaremos su nombre.
- **parámetros**: variables que va a necesitar.
- **docstring**: texto multilínea que se incluye para poder tener una explicación sobre cómo funciona.
- **código**: contiene las instrucciones.
- **return:** retorno de la función. Su resultado.

6. Funciones

Para definir una función:

```
def nombre_funcion(argumento/s de entrada):
    # los dos puntos y la indentacion implican estar en la funcion
    realiza un accion
    return()   # devuelven un/os valor/es
```

Ejecución de la función:

```
nombre_función(argumento1, argumento2, ...)
```

o, si quieres guardar el resultado en una variable: 💾

```
resultado = nombre_función(argumento1, argumento2, ...)
```

6. Funciones

Llamada a una función

- se usa el nombre de la función y entre paréntesis todos aquellos **argumentos** necesarios, separados por comas.
- los argumentos son aquellos datos que le das a la función para que los utilice.
    - pueden ser obligatorios u opcionales.
- la función puede devolverte un valor. Para capturarlo almacénalo en una variable.