

```
1
2 Programming 'Language' = {
3
4     Introducción = [Python,
5                     Micropython]
6
7
8
9
10
11 }
12
13
14
```



```
1
2
3 Clase 03 = {
4
5     Presentación = [Les
6                     damos la bienvenida al
7                     curso]
8
9
10
11
12 }
13
14
```



# Clases = {

## Clase 01

Breve historia de Python y su Filosofía. Principios de diseño de Python (PEP 20). Instalación y Configuración de Python y entornos de desarrollo (IDE).

## Clase 02

Sintaxis Básica y Estructuras de Control. Variables, tipos de datos y operadores. Estructuras de control (if, for, while).



## Clase 03

**Estructuras de Datos. Listas, tuplas, diccionarios y conjuntos. Manipulación y métodos asociados.**

## Clase 04

Funciones y Módulos. Definición y uso de funciones. Importación y creación de módulos.



# Listas {

```
# Una lista es una secuencia ordenada de elementos. Cada elemento es un
# dato. Pueden ser del mismo o distinto tipo, aunque esto último es poco
# frecuente. Tienen la particularidad de ser mutables, contrario a las
# tuplas. Esto último quiere decir que su contenido se puede modificar
# después de haber sido creadas.

# Las listas en Python son un tipo contenedor compuesto, y se usan para
# almacenar conjuntos de elementos relacionados.

# Junto a las tuplas, diccionarios y conjuntos, constituyen uno de los
# tipos de datos más versátiles del lenguaje.

# Las listas se crean asignando a una variable una secuencia de elementos
# encerrados entre corchetes [ ] y separados por comas. Se puede crear una
# lista vacía, y las listas pueden ser elementos de otras listas.
```

}



# Listas {

```
# Como parte de las recomendaciones de estilo en Python, se sugiere que  
# las listas usen nombres en plural.
```

```
numeros = [1,2,3,4,5] # Lista de números
```

```
dias = ["Lunes", "Martes", "Miércoles"] # Lista de strings
```

```
elementos = [] # Lista vacía
```

```
matriz_listas_2 = [ [1,2,3] , [4,5,6] ] # Matriz de 2 dimensiones
```

```
matriz_listas_3 = [ [[1,2,3],[4,5,6]] , [[7,8,9],[1,2,3]] ]  
# Matriz de 3 dimensiones
```

```
}
```



# Listas – Acceso por subíndices {

# Para acceder a los datos de la lista lo hacemos mediante los subíndices. El primer elemento tiene subíndice cero. Un subíndice negativo hace que la cuenta comience desde atrás. Un subíndice fuera de rango genera un error: out of range. Por ejemplo, armemos una lista con 5 caracteres del tipo string.

```
letras_1 = [ a , b , c , d , e ]
```



```
letras_2 = [ "a" , "b" , "c" , "d" , "e" ]
```



}



# Listas – Acceso por subíndices {

	len(lista) = 5				
lista	a	b	c	d	e
Subíndice (index)	0	1	2	3	4
Subíndice negativo (negative index)	-5	-4	-3	-2	-1

```
lista = [ "a", "b", "c", "d", "e" ]
```

```
print(lista[1])      # b
```

```
print(lista[3])      # d
```

```
print(lista[-1])     # e
```

```
print(lista[-3])     # c
```

```
print(lista[8])      # IndexError: list index out of range
```



# Listas – Acceso por slices {

# Un slice o "rebanada" es una técnica utilizada para extraer una subsección de una secuencia, como una lista, una cadena o una tupla. La sintaxis básica para realizar un slicing es

# **secuencia** [ **inicio** : **fin** : **paso** ]

# **inicio** es el índice donde comienza la rebanada (incluido).

# **fin** es el índice donde termina la rebanada (excluido).

# **paso** es el intervalo entre los elementos seleccionados.

Si se omite inicio, se asume que es el principio de la secuencia. Si se omite fin, se asume que es el final de la secuencia. Si se omite paso, se asume que es 1.

}





# Listas – Acceso por slices {

```
lista = [ "a", "b", "c", "d", "e" ]

print(lista[1:3])    # ['b', 'c']
print(lista[0:3])    # ['a', 'b', 'c']
print(lista[1:-1])   # ['b', 'c', 'd']
print(lista[1:])      # ['b', 'c', 'd', 'e']
print(lista[:3])      # ['a', 'b', 'c']
print(lista[:])       # ['a', 'b', 'c', 'd', 'e']
print(lista[::2])     # ['a', 'c', 'e']
print(lista[1:3:1])   # ['b', 'c']
print(lista[1:3:2])   # ['b']
print(lista[1:4:2])   # ['b', 'd']
```

}



# Listas – función len() {

```
# La función len() se utiliza para obtener la longitud o el número de
# elementos de una lista. Retorna un valor entero que representa la cantidad
# de elementos.
```

```
lista = [ "a", "b", "c", "d", "e" ]
```

```
print(len(lista))    # 5
```

```
# Podríamos usar len() para acceder al último elemento de la lista, aunque
# debemos considerar que el largo de la lista es 4, pero la última posición
# es 3:.
```

```
print(lista[len(lista)]) # IndexError: list index out of range
```

```
print(lista[len(lista)-1]) # e
```

```
}
```



# Listas – Recorrer {

# Es posible recorrer una lista utilizando for (con un range) o while para generar la secuencia de índices. Con la secuencia de índices generada, vamos a recorrer elemento por elemento para poder utilizarlo.

```
lista = [2,3,4,5,6]
suma = 0
for i in range(len(lista)):
    suma = suma + lista[i]
print(suma) #20
```

```
lista = [2,3,4,5,6]
suma = 0
i = 0
while i < len(lista):
    suma = suma + lista[i]
    i = i + 1
print(suma) #20
```

}



# Listas – Recorrer {

# También se puede iterar en forma directa los elementos de la lista, sin necesidad de generar la secuencia de subíndices. En este caso la variable `i` toma el elemento de la lista.

```
lista = [2,3,4,5,6]
suma = 0
for e in lista:
    suma = suma + e
print(suma) #20
```

```
vocales = ['a','e','i','o','u']

for e in vocales:
    print(e, end=" ") # a e i o u
```

}



# Listas – Desempaquetado {

```
# El proceso de desempaquetado de una lista en Python, también conocido  
# como "unpacking", es una técnica que permite asignar los elementos de una  
# lista (o cualquier otra secuencia) a variables individuales de manera  
# directa y concisa. Esto se logra mediante la asignación múltiple, donde se  
# especifican varias variables en el lado izquierdo de la asignación y una  
# lista o tupla en el lado derecho.
```

```
dias = ["Lunes", "Martes", "Miércoles"]
```

```
d1, d2, d3 = dias
```

```
print(d1)
```

```
print(d2)
```

```
print(d3)
```

```
}
```



# Listas – Concatenado {

# El concatenado de listas en Python es el proceso de unir dos o más listas para formar una sola lista. Esto se puede lograr utilizando el operador `+` o el método `extend()`.

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
```

```
lista_concatenada = lista1 + lista2
print(lista_concatenada)
# [1, 2, 3, 4, 5, 6]
```

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
```

```
lista1.extend(lista2)
print(lista1)
# [1, 2, 3, 4, 5, 6]
```

}



# Listas – max(), min() y sum() {

# Las funciones **max()**, **min()** y **sum()** en Python son funciones integradas que se utilizan para realizar operaciones comunes en secuencias como listas, tuplas, y otros iterables.

La función **max()** devuelve el elemento más grande de una secuencia o el más grande de varios argumentos.

La función **min()** devuelve el elemento más pequeño de una secuencia o el más pequeño de varios argumentos.

La función **sum()** devuelve la suma de todos los elementos en una secuencia. Puede tomar un segundo argumento opcional que se añade al resultado final.

```
numeros = [1, 2, 3, 4, 5]
print(max(numeros)) # 5
print(min(numeros)) # 1
print(sum(numeros)) # 15
```

}



# Listas – list {

# La función **list()** en Python se utiliza para crear una lista a partir de un iterable, como una cadena, tupla, conjunto, o cualquier otro objeto que se pueda iterar. También se puede usar para copiar una lista existente.

```
cadena = "abcde"
lista_desde_cadena = list(cadena)
print(lista_desde_cadena)
# ['a', 'b', 'c', 'd', 'e']
```

```
numeros = [1, 2, 3, 4, 5]
copia_numeros = list(numeros)
print(copia_numeros)
# [1, 2, 3, 4, 5]
```

```
rango = range(2, 6)
lista_desde_rango = list(rango)
print(lista_desde_rango)
# [2, 3, 4, 5]
```

}





# Listas – in / not in {

# Los operadores **in** y **not in** en Python se utilizan para verificar la presencia de un elemento en una secuencia, como una lista, tupla, cadena, conjunto, o diccionario.

# En una lista

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

print(3 in numeros) # True

print(6 in numeros) # False

# En una cadena

cadena = "hola mundo"

print("hola" in cadena) # True

print("adios" in cadena) # False

# En una lista

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

print(3 not in numeros) # False

print(6 not in numeros) # True

# En una cadena

cadena = "hola mundo"

print("hola" not in cadena) # False

print("adios" not in cadena) # True

}



# Listas – `.append()` e `.insert()` }

# El método `.append()` añade un elemento al final de la lista.

```
numeros = [1, 2, 3]
numeros.append(8)
print(numeros) # [1, 2, 3, 8]
```

# El método `.insert()` añade un elemento en una posición específica de la lista. Toma dos argumentos: el índice en el que se debe insertar el elemento y el elemento en sí. `insert(<posicion>, <elemento>)`

```
numeros = [1, 2, 3]
numeros.insert(1, 1.5) # Inserta 1.5 en el índice 1
print(numeros) # [1, 1.5, 2, 3]
```

}



# Listas – .pop() {

**# pop(<posición>)** Elimina un elemento en una posición determinada de la lista. Si no se pasa un argumento, pop() elimina el último elemento de la lista.

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

**# Eliminar y devolver el último elemento**

```
ultimo = numeros.pop()
```

```
print(ultimo) # 5
```

```
print(numeros) # [1, 2, 3, 4]
```

**# Eliminar y devolver el elemento en el índice 1**

```
segundo = numeros.pop(1)
```

```
print(segundo) # 2
```

```
print(numeros) # [1, 3, 4]
```

```
}
```



# Listas – .remove() {

# El método **.remove()** elimina la primera aparición de un valor específico en la lista. Si el valor no se encuentra en la lista, se lanza un **ValueError**.

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

# Eliminar la primera aparición del valor 3

```
numeros.remove(3)
```

```
print(numeros)      # [1, 2, 4, 5]
```

# Eliminar la primera aparición del valor 6

```
numeros.remove(6)  # ValueError: list.remove(x): x not in list
```

```
}
```



# Listas – .index() {

```
# El método .index() devuelve el índice de la primera aparición de un
valor especificado en la lista. Si el valor no se encuentra en la lista,
se lanza un ValueError.
```

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

```
# Obtener el índice de la primera aparición del valor 3
```

```
indice = numeros.index(3)
```

```
print(indice) # 2
```

```
# Intentar obtener el índice de un valor que no está en la lista
lanza un ValueError
```

```
indice = numeros.index(6)
```

```
print(indice) # ValueError: 6 is not in list
```

```
}
```



# Listas – .count() {

# El método **.count()** devuelve el número de veces que un valor especificado aparece en la lista.

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

# Contar el número de veces que el valor 3 aparece en la lista

```
conteo = numeros.count(3)
```

```
print(conteo) # 3
```

# Contar el número de veces que el valor 6 aparece en la lista

```
conteo = numeros.count(6)
```

```
print(conteo) # 0
```

```
}
```



# Listas – `.reverse()` y `.clear()` }

```
# El método .reverse() invierte el orden de los elementos en la lista.  
Este método modifica la lista original y no devuelve ningún valor.  
  
numeros = [1, 2, 3, 4, 5]  
  
numeros.reverse()  
print(numeros)  # [5, 4, 3, 2, 1]  
  
# El método .clear() elimina todos los elementos de una lista, dejándola  
vacía. Este método modifica la lista original y no devuelve ningún valor.  
  
numeros = [1, 2, 3, 4, 5]  
  
numeros.clear()  
print(numeros)  # []
```

}



# Listas – .sort() {

# El método **.sort()** ordena los elementos de una lista en su lugar (modifica la lista original). Por defecto, ordena los elementos en orden ascendente, pero se puede especificar un orden diferente utilizando el parámetro **reverse**.

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

# Ordenar la lista en orden ascendente

```
numeros.sort()
```

```
print(numeros) # [1, 2, 3, 4, 5]
```

# Ordenar la lista en orden descendente

```
numeros.sort(reverse=True)
```

```
print(numeros) # [5, 4, 3, 2, 1]
```

```
}
```





# Tuplas {

# Al igual que las lista, las **tuplas** en Python son un tipo de estructura de datos que permite almacenar una colección ordenada de elementos. A diferencia de las listas, las tuplas son **inmutables**, lo que significa que una vez creadas, sus elementos no pueden ser modificados, añadidos o eliminados. Las tuplas se utilizan cuando se necesita una colección de elementos que no debe cambiar a lo largo del tiempo.

## # Características de las tuplas

# **Inmutables:** No se pueden modificar después de su creación.

# **Ordenadas:** Mantienen el orden de los elementos.

# **Permiten duplicados:** Pueden contener elementos duplicados.

# **Heterogéneas:** Pueden contener elementos de diferentes tipos de datos.

# Las tuplas se pueden crear utilizando paréntesis **()** o la función **tuple()**.

}



# Tuplas {

```
1      # Creación de una tupla vacía
2      tupla_vacia = ()
3
4      # Creación de una tupla con elementos
5      tupla = (1, 2, 3, "a", "b", "c")
6
7      # Creación de una tupla sin paréntesis (packing o empaquetado)
8      tupla_sin_parentesis = 1, 2, 3, "a", "b", "c"
9
10     # Creación de una tupla a partir de un iterable
11     tupla_desde_lista = tuple([1, 2, 3, "a", "b", "c"])
12
13     print(tupla)                # (1, 2, 3, 'a', 'b', 'c')
14     print(tupla_sin_parentesis) # (1, 2, 3, 'a', 'b', 'c')
15     print(tupla_desde_lista)    # (1, 2, 3, 'a', 'b', 'c')
```



# Tuplas – Acceso y desempaquetado {

```
# Al trabajar con tuplas, tambien vamos a poder acceder a sus elementos  
individuales con los subindices o desempaquetar sus elementos como vimos  
anteriormente.
```

```
tupla = (1, 2, 3, "a", "b", "c")
```

```
print(tupla[0]) # Acceso al primer elemento
```

```
print(tupla[-1]) # Acceso al último elemento
```

```
print(tupla[1:4]) # Acceso a un rango de elementos (slicing)
```

```
# Desempaquetado
```

```
a, b, c, d, e, f = tupla
```

```
print(a) # 1
```

```
print(e) # b
```

```
}
```



# Tuplas - Métodos {

# Las tuplas en Python tienen un conjunto limitado de métodos debido a su inmutabilidad.

# **count()**: Devuelve el número de veces que un valor especificado aparece en la tupla.

# **index()**: Devuelve el índice de la primera aparición de un valor especificado en la tupla. Si el valor no se encuentra, lanza un ValueError.

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

```
conteo = tupla.count(2)
```

```
print(conteo) # 3
```

```
indice = tupla.index(3)
```

```
print(indice) # 2
```

```
}
```



# Tuplas - Operaciones {

**# Concatenación:** Se pueden concatenar dos o más tuplas utilizando el operador +.

```
tupla1 = (1, 2, 3)
```

```
tupla2 = (4, 5, 6)
```

```
tupla_concatenada = tupla1 + tupla2
```

```
print(tupla_concatenada) # (1, 2, 3, 4, 5, 6)
```

**# Repetición:** Se puede repetir una tupla un número específico de veces utilizando el operador \*.

```
tupla = (1, 2, 3)
```

```
tupla_repetida = tupla * 3
```

```
print(tupla_repetida) # (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
}
```



# Diccionarios {

# En Python, un diccionario es una estructura de datos que almacena pares de clave-valor. Los diccionarios son mutables, lo que significa que se pueden modificar después de su creación. Las claves en un diccionario deben ser únicas e inmutables (por ejemplo, cadenas, números o tuplas), mientras que los valores pueden ser de cualquier tipo.

## # Características de los diccionarios

# **Mutable:** Se pueden modificar después de su creación.

# **Desordenadas:** No mantienen un orden específico de los elementos.

# **Claves únicas:** Cada clave en un diccionario debe ser única.

# **Acceso rápido:** Permiten un acceso rápido a los valores mediante las claves.

Los diccionarios se pueden crear utilizando llaves `{}` o la función `dict()`.

}



# Diccionarios {

```
# Creación de un diccionario con elementos
```

```
diccionario = {  
    "nombre": "Juan",  
    "edad": 30,  
    "ciudad": "Madrid"  
}
```

```
# Creación de un diccionario utilizando la función dict()
```

```
diccionario_funcion = dict(nombre="Juan", edad=30, ciudad="Madrid")
```

```
print(diccionario)
```

```
# {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Madrid'}
```

```
print(diccionario_funcion)
```

```
# {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Madrid'}
```



# Diccionarios - Acceso {

```
1
2
3     diccionario = {
4         "nombre": "Juan",
5         "edad": 30,
6         "ciudad": "Madrid"
7     }
8
9     # Acceso a un valor mediante su clave
10    print(diccionario["nombre"]) # Juan
11    print(diccionario["edad"])   # 30
12    print(diccionario["pais"])   # Error
13
14    # Acceso a un valor utilizando el método get()
15    print(diccionario.get("ciudad")) # Madrid
16    print(diccionario.get("pais", "No disponible"))
17    # No disponible (valor por defecto)
```





# Diccionarios - Modificación {

```
1  diccionario = {
2      "nombre": "Juan",
3      "edad": 30,
4      "ciudad": "Madrid"
5  }
6
7  # Añadir un nuevo par clave-valor
8  diccionario["pais"] = "España"
9
10 # Modificar un valor existente
11 diccionario["edad"] = 31
12
13 # Eliminar un par clave-valor
14 del diccionario["ciudad"]
15 print(diccionario) # {'nombre': 'Juan', 'edad': 31, 'pais': 'España'}
```



# Diccionarios – Métodos {

```
# Existen varios metodos para trabajar con diccionarios.  
  
# keys(): Devuelve una vista de las claves del diccionario.  
  
# values(): Devuelve una vista de los valores del diccionario.  
  
# items(): Devuelve una vista de los pares clave-valor del diccionario.  
  
# update(): Actualiza el diccionario con pares clave-valor de otro  
diccionario o iterable.  
  
# pop(): Elimina y devuelve el valor asociado a una clave especificada.  
  
# clear(): Elimina todos los elementos del diccionario.
```

```
}
```



# Diccionarios - Métodos {

```
1     diccionario = {
2         "nombre": "Juan",
3         "edad": 30,
4         "ciudad": "Madrid"
5     }
6
7     # Obtener todas las claves
8     print(diccionario.keys()) # dict_keys(['nombre', 'edad', 'ciudad'])
9
10    # Obtener todos los valores
11    print(diccionario.values()) # dict_values(['Juan', 30, 'Madrid'])
12
13    # Obtener todos los pares clave-valor
14    print(diccionario.items()) # dict_items([('nombre', 'Juan'), ('edad', 30), ('ciudad', 'Madrid')])
```



# Diccionarios - Métodos {

```
1  diccionario = {
2      "nombre": "Juan",
3      "edad": 30,
4      "ciudad": "Madrid"
5  }
6  # Actualizar el diccionario con otro diccionario
7  diccionario.update({"pais": "España", "edad": 31})
8  print(diccionario)
9  # {'nombre': 'Juan', 'edad': 31, 'ciudad': 'Madrid', 'pais': 'España'}
10
11 # Eliminar y devolver el valor asociado a una clave
12 edad = diccionario.pop("edad")
13 print(edad) # 31
14 print(diccionario)
15 # {'nombre': 'Juan', 'ciudad': 'Madrid', 'pais': 'España'}
```



# Diccionarios – in {

```
diccionario = {  
    "nombre": "Juan",  
    "edad": 30,  
    "ciudad": "Madrid"  
}  
  
print("nombre" in diccionario) # True  
print("pais" in diccionario)   # False  
  
print("Juan" in diccionario.values()) # True  
print("Barcelona" in diccionario.values()) # False
```

```
}
```



# Diccionarios – not in {

```
1
2
3     diccionario = {
4         "nombre": "Juan",
5         "edad": 30,
6         "ciudad": "Madrid"
7     }
8
9     print("pais" not in diccionario) # True
10    print("edad" not in diccionario) # False
11
12    print("Barcelona" not in diccionario.values()) # True
13    print("Juan" not in diccionario.values()) # False
14 }
```



# Conjuntos {

```
# Un conjunto (set) es una colección desordenada de elementos únicos. Los
conjuntos se utilizan para almacenar múltiples elementos en una sola
variable, pero a diferencia de las listas o tuplas, no permiten elementos
duplicados. Los conjuntos son mutables, lo que significa que se pueden
modificar después de su creación, aunque los elementos del conjunto deben
ser inmutables (por ejemplo, números, cadenas, tuplas). No puede accederse
a los elementos de un diccionario a través de un subíndice pues sus
elementos no están ordenados.
```

## # Características de los conjuntos

```
# Desordenados: No mantienen un orden específico de los elementos.
```

```
# Elementos únicos: No permiten elementos duplicados.
```

```
# Mutables: Se pueden modificar después de su creación.
```

```
# Elementos inmutables: Los elementos del conjunto deben ser inmutables.
```

```
} Los conjuntos se pueden crear utilizando llaves {} o la función set().
```



# Conjuntos {

```
# Creación de un conjunto vacío
conjunto_vacio = set()

# Creación de un conjunto con elementos
conjunto = {1, 2, 3, 4, 5}

# Creación de un conjunto a partir de una lista
conjunto_desde_lista = set([1, 2, 3, 4, 5, 5, 5])
print(conjunto_desde_lista)
# {1, 2, 3, 4, 5} (los duplicados se eliminan)

# Creación de un conjunto a partir de una cadena
conjunto_desde_cadena = set("hola")
print(conjunto_desde_cadena)
# {'h', 'o', 'l', 'a'} (el orden puede variar)
```





# Conjuntos - | y & {

```
# Los conjuntos soportan varias operaciones matemáticas como unión,  
intersección, diferencia y diferencia simétrica.
```

```
conjunto1 = {1, 2, 3, 4, 5}
```

```
conjunto2 = {4, 5, 6, 7, 8}
```

```
# Unión (elementos presentes en cualquiera de los conjuntos)
```

```
union = conjunto1 | conjunto2
```

```
print(union) # {1, 2, 3, 4, 5, 6, 7, 8}
```

```
# Intersección (elementos presentes en ambos conjuntos)
```

```
interseccion = conjunto1 & conjunto2
```

```
print(interseccion) # {4, 5}
```

```
}
```



# Conjuntos - - y ^ {

```
conjunto1 = {1, 2, 3, 4, 5}
```

```
conjunto2 = {4, 5, 6, 7, 8}
```

```
# Diferencia (elementos presentes en el primer conjunto pero no en el segundo)
```

```
diferencia = conjunto1 - conjunto2
```

```
print(diferencia) # {1, 2, 3}
```

```
# Diferencia simétrica (elementos presentes en cualquiera de los conjuntos, pero no en ambos)
```

```
diferencia_simetrica = conjunto1 ^ conjunto2
```

```
print(diferencia_simetrica) # {1, 2, 3, 6, 7, 8}
```

```
}
```



# Conjuntos - Métodos {

```
# Existen varios metodos para trabajar con conjuntos.  
  
# add(): Añade un elemento al conjunto.  
  
# remove(): Elimina un elemento del conjunto. Lanza un KeyError si el  
# elemento no está presente.  
  
# discard(): Elimina un elemento del conjunto si está presente. No lanza un  
# error si el elemento no está presente.  
  
# pop(): Elimina y devuelve un elemento aleatorio del conjunto.  
  
# clear(): Elimina todos los elementos del conjunto.  
  
# update(): Añade múltiples elementos al conjunto.
```

```
}
```



# Conjuntos - Métodos {

```
conjunto = {1, 2, 3}

# Añadir un elemento al conjunto
conjunto.add(4)
print(conjunto)  # {1, 2, 3, 4}

# Eliminar un elemento del conjunto
conjunto.remove(2)
print(conjunto)  # {1, 3, 4}

# Eliminar un elemento del conjunto si está presente
conjunto.discard(3)
print(conjunto)  # {1, 4}
```

}



# Conjuntos - Métodos {

```
conjunto = {1, 2, 3}

# Eliminar y devolver un elemento aleatorio del conjunto
elemento = conjunto.pop()
print(elemento) # 1 (puede variar)
print(conjunto) # {2, 4}

# Eliminar todos los elementos del conjunto
conjunto.clear()
print(conjunto) # set()

# Añadir múltiples elementos al conjunto
conjunto.update([5, 6, 7])
print(conjunto) # {5, 6, 7}
```

}



```
1
2
3 Aprender a programar
4
5 es aprender a pensar.
6
7
8
9
10
```

```
11 { Steve Jobs; }
12
13
14
```



```
1
2
3
4
5 { Nos vemos en la
6 proxima clase }
7
8
9
10
11
12
13
14
```

