PROGRAMACIÓN III

Clase 3

Clase 2 → Clase 3 → Clase 4

Controladores de flujo

- Estructuras control.
- Condicionales: sentencia if.
- Iterativas: sentencia while y for.
- Operadores lógicos y relacionales.

Cadenas y listas

- Cadenas de caracteres.
- Métodos de listas.
- f-strings
- Índices y slicing (rebanadas).
- Tipo de datos compuestos.
- Listas. Métodos.
- Tipos de datos mutables e inmutables.
- Tuplas, diccionarios, conjuntos

Funciones

- Funciones. Concepto.
- Llamada a función.
- Retorno y envío de valores.
- Parámetros, argumentos, valor y referencia.
- Parámetros mutables e inmutables.
- Parámetros por defecto
- Docstring.
- Funciones Lambda/Anónima. Cade

CADENA Y LISTAS

Cadenas de caracteres

Python, al igual que la mayoría de los lenguajes de programación actuales, provee un tipo de datos específico para tratar las cadenas de caracteres (strings).
Se trata de un tipo de dato con longitud variable, ya que deben adecuarse a la cantidad de caracteres que albergue la cadena.
Este tipo de datos posee una buena cantidad de métodos y propiedades que facilita su uso.

Cadenas de caracteres

Una cadena de caracteres está compuesta por cero o más caracteres. Las cadenas pueden delimitarse con comillas simples o dobles.

```
# Definición de cadenas usando comillas dobles
dia1 = "Lunes"
x = ""          # x es un string de longitud cero

# Definición de cadenas usando comillas simples
dia2 = 'Martes'
z = "121"       # z contiene dígitos, pero es un string
```

Cadenas de caracteres

Una ventaja del hecho de poder delimitar cadenas con comillas simples o dobles es que si usamos comillas de una clase, las de otra clase puede utilizarse como parte de la cadena:

Uso de comillas simples y dobles

```
print("Mi perro 'Toby'") # Mi perro 'Toby'
print('Mi perro "Toby"') # Mi perro "Toby"
```

Una cadena puede replicarse con el operador *:

Replicación

```
risa = 'ja'
carcajada = risa*5  # jajajajaja
asteriscos = "*"*10 # **********
```

Cadenas de caracteres

También pueden usarse triples comillas simples o dobles, que proveen un método sencillo para crear cadenas usando más de una línea de código:

Cadenas delimitadas por comillas dobles y triples

```
# Definición de cadenas usando comillas dobles triples:
cadena1 = """En Python es posible definir
cadenas de caracteres utilizando más de una
línea de código"""

# Definición de cadenas usando comillas simples triples:
cadena2 = '''Por supuesto, se puede hacer
lo mismo utilizando comillas simples'''
```

Cadenas de caracteres | Concatenación

Para concatenar dos o más cadenas se utiliza el operador + (más):

Concatenación de cadenas

```
nombre= input("Ingrese su nombre: ")
saludo= "Hola "+ nombre
print(saludo)
```

Terminal

```
Ingrese su nombre: Pedro
Hola Pedro
```

El mismo operador se usa para sumar números o concatenar cadenas. Pero no podemos utilizarlo con datos mixtos, porque se obtiene un error. Este error se puede evitar mediante las funciones de conversión

Concatenación de cadenas

```
var1 = 3 + 5          # 8 (entero)
var2 = "3" + "5"      # 35 (cadena)
var3 = 3 + "5"        # TypeError
var4 = str(3) + "5"   # 35 (cadena)
var5 = 3 + int("5")   # 8 (entero)
```

Cadenas de caracteres | Comparación

Dos cadenas se pueden comparar mediante los operadores relacionales.

Comparación de cadenas

```
cadena1 = "Hola"
cadena2 = "Programacion 3"
print(cadena1 < cadena2)
print(cadena1 == cadena2)
print(cadena1 > cadena2)
```

Terminal

```
True
False
False
```

La comparación es case sensitive, es decir, se distingue entre mayúsculas y minúsculas.

Concatenación de cadenas

```
cadena1 = "Hola"
cadena2 = "hola"
print(cadena1 == cadena2)
```

Terminal

```
False
```

Cadenas de caracteres

En Python disponemos de la función **len()**, que retorna la cantidad de caracteres que contiene un string:

Función len()

```
nombre = "Programacion 3"
print(len(nombre))    # Se imprime 14
```

Se accede a los elementos de la cadena utilizando subíndices:

Uso de subíndices

```
cadena = "Hola Programacion 3"
print(cadena[0])
print(cadena[5])
print(cadena[-1])
print(cadena[2])
```

Terminal

```
H
P
3
l
```

El primer carácter tiene subíndice cero. Si usamos subíndices negativos, se cuentan desde el final de la cadena.

Cadenas de caracteres | Métodos

Las cadenas tienen una serie de **métodos** (funciones) que simplifican el desarrollo de código. Tres de estos métodos son:

- ✓ **.upper()**: devuelve la cadena con todos sus caracteres en mayúsculas.
- ✓ **.lower()**: devuelve la cadena con todos sus caracteres en minúsculas.
- ✓ **.capitalize()**: devuelve la cadena con su primer carácter en mayúscula y todos los demás en minúsculas.

Métodos propios de las cadenas

```
cadena = "programacion 3"
print(cadena.upper())              #PROGRAMACION 3
print(cadena.lower())              #programacion 3
print(cadena.capitalize())         #Programacion 3
```

Existen los métodos **.isupper()** e **.islower()** que devuelven **True** si todos los caracteres alfabéticos de una cadena están en mayúsculas o minúsculas respectivamente.

Cadenas de caracteres | Métodos

Una **"rebanada"** es un subconjunto de una cadena que se obtiene mediante la definición de un índice **inicio** y/o **fin**. El subconjunto devuelto incluye el valor del índice de inicio, pero no el valor final. Un tercer valor permite determinar un **paso**, que incluso puede ser negativo:

Rebanadas

```
cadena = "¡Hola mundo!"
print(cadena[6:11])   # mundo
print(cadena[2:12:2]) # oamno
print(cadena[6:])     # mundo!
print(cadena[:5])     # ¡Hola
print(cadena[:])      # ¡Hola mundo!
print(cadena[::2])    # ¡oamno
print(cadena[::-1])   # !odnum aloH¡
```

Cadenas de caracteres | in / not in

Los operadores de pertenencia se utilizan para comprobar si un caracter o

in	Devuelve True si el valor se encuentra en una secuencia; False en caso contrario.
not in	Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario.

Ejemplos

```
cadena = "Hola mundo"
print("H" in cadena)        #True
print("c" in cadena)        #False
print("Hola" in cadena)     #True
print("A" not in cadena)    #True
print("u" not in cadena)    #False
```

Cadenas de caracteres | for , min() y max()

Un **bucle for** puede iterar sobre una cadena, y la variable recibe en cada iteración uno de los caracteres de la misma:

For con cadenas

```
cadena = "Python"
for letra in cadena:
   print(letra)
```

Terminal

```
P
y
t
h
o
n
```

min() y max() devuelven el elemento con el código ASCII más pequeño o más grande respectivamente:

Mix() y Max

```
cadena = "Programador"
print(max(cadena))
print(min(cadena))
```

Terminal

```
r
P
```

Cadenas de caracteres | .join(), .split() y .replace()

cadena.join(separador) devuelve una cadena con el separador entre cada carácter.
cadena.split(separador) convierte una cadena en una lista, y **cadena.replace**(viejo,nuevo,max) reemplaza una cadena por otra hasta un máximo. Si se omite max reemplaza todas las apariciones.

.join()

```
cadena = "12345"
cadena = '-'.join(cadena)
print(cadena)
#1-2-3-4-5
```

.split()

```
cadena = "hola mundo"
lista = cadena.split(" ")
print(lista)  #['hola', 'mundo']
```

.replace()

```
cadena = "hola mundo"
cadena = cadena.replace('mundo', 'clase')
print(cadena) #hola clase
```

Cadenas de caracteres | Detección de tipos

cadena.isalpha() devuelve **True** si todos los caracteres de una cadena son alfabéticos.
cadena.isdigit() devuelve **True** si todos los caracteres de una cadena son dígitos
cadena.isalnum() devuelve **True** si todos los caracteres de una cadena son alfabéticos o numéricos.

.isalpha()

```
cad1 = "Python"
cad2 = "Python3"
print(cad1.isalpha())
# True
print(cad2.isalpha())
# False
```

. isdigit()

```
cad1 = "1234"
cad2 = "1234a"
print(cad1.isdigit())
# True
print(cad2.isdigit())
# False
```

. isalnum()

```
cad1 = "12Ab"
cad2 = "12Ab%"
print(cad1.isalnum())
# True
print(cad2.isalnum())
# False
```

Cadenas de caracteres | title() y center()

cadena.title() devuelve una cadena en minúsculas con el primer carácter de cada palabra en mayúsculas. **cadena.center(ancho, relleno)** devuelve una cadena en el ancho especificado. El resto de la cadena se rellena con espacios o con el carácter opcional relleno.

.title()

```
cad1 = "aprendiendo programación"
print(cad1.title())
# Aprendiendo Programación
cad1 = "Este es un TEXTO"
print(cad1.title())
# Este Es Un Texto
```

. center()

```
cad1 = "Hola"
cad2 = cad1.center(10,"*")
print(cad2)
# ***Hola***
```

Cadenas de caracteres | Alineación

cadena.ljust(ancho, [relleno]) y **cadena.rjust(ancho, [relleno])** devuelven cadenas alineadas a izquierda o derecha, respectivamente. El resto de la cadena se rellena con espacios o con el carácter opcional relleno.
cadena.zfill(ancho) devuelve una cadena alineada a la derecha en el ancho especificado. El comienzo de la cadena se rellena con ceros.

.ljust()

```
cad1 = "Python"
cad1 = cad1.ljust(10, '-')
print(cad1)
# Python----
```

.rjust()

```
cad1 = "Python"
cad1 = cad1.rjust(10, '-')
print(cad1)
# ----Python
```

.zfill()

```
cad1 = "120"
cad1 = cad1.zfill(8)
print(cad1)
# 00000120
```

Cadenas de caracteres | Recorte

cadena.lstrip(str) y **cadena.rstrip(str)** devuelven cadenas a las que se les han quitado los caracteres indicados por **str** a la izquierda o a la derecha, respectivamente. **cadena.strip(str)** devuelve una cadena sin los caracteres indicados en **str** al inicio y al final de la cadena.

.lstrip()

```
cad1 = "---Hola-Mundo----"
cad1 = cad1.lstrip('-')
print(cad1)
# Hola-Mundo----
```

.rstrip()

```
cad1 = "---Hola-Mundo----"
cad1 = cad1.rstrip('-')
print(cad1)
# ---Hola-Mundo
```

. strip()

```
cad1 = "---Hola-Mundo----"
cad1 = cad1.strip('-')
print(cad1)
# Hola-Mundo
```

Cadenas de caracteres | Búsquedas

cadena.find(<str>,[[inicio],[fin]]) devuelve la posición donde encuentra str en la cadena. Si no lo encuentra devuelve -1. Se puede indicar los subíndices desde (inicio) y hasta (fin) donde buscar.

<cadena>.rfind(<str>,[[inicio],[fin]]) es similar a find, pero busca la última aparición.

.find()

```
cadena = "hola mundo"
lista = cadena.split(" ")
print(lista)  #['hola', 'mundo']
```

.rfind()

```
cadena = "hola mundo hola"
pos = cadena.rfind("hola")
print(pos) #11
```

Cadenas de caracteres | f-Strings

f-Strings tiene una sintaxis simple y fluida que simplifica la tarea de dar formato a cadenas de texto. Para mostrar variables se coloca el nombre de las variables entre llaves {}, en una cadena que antepone f a su contenido. Al ejecutar el código, todos los nombres de las variables se reemplazan por sus respectivos valores:

f-Strings

```
legajo = 12212
nombre = "María"
nota = 10
print(f"Legajo: {legajo} Nombre: {nombre} Nota: {nota}")
# Legajo: 12212 Nombre: María Nota: 10
```

Cadenas de caracteres |metodo str.format()

Las llaves y caracteres dentro de las mismas (llamados campos de formato) son reemplazadas con los objetos pasados en el método

Uso de str.format()

```
print("Esta es la comision del turno {} de la materia {} ".format('mañana', 'programacion III'))
# Esta es la comisión del turno mañana de la materia programacion III
print("Esta es la comision del turno {0} de la materia {1} ".format('mañana', 'programacion III'))
# Esta es la comisión del turno mañana de la materia programacion III
print("El resultado de la suma de {} y {} es igual a {}".format(3, 10, 3+10))
```

Listas

Una lista es una secuencia ordenada de elementos. Pueden tener elementos del mismo tipo o combinar distintos tipos de datos, aunque esto último es poco frecuente.

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, con la particularidad de ser mutables. Esto último quiere decir que su contenido se puede modificar después de haber sido creadas.

Listas

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:

f-Strings

```
numeros = [1,2,3,4,5] #Lista de números
dias = ["Lunes", "Martes", "Miércoles"] #Lista de strings
elementos = [] #Lista vacía
sublistas = [ [1,2,3], [4,5,6] ] # lista de listas
```

Las listas se suelen nombrar en plural. Para incluir una lista como parte de otra, basta con incluirla separada por comas de los otros elementos.

Listas | Acceso por subíndice

Las listas se pueden imprimir directamente, y el acceso a sus elementos se hace mediante 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**

Acceso por subíndices

```
dias = ["Lunes", "Martes", "Miércoles"]
print(dias)
print(dias[0])
print(dias[1])
print(dias[-1])
print(dias[3])
```

Terminal

```
['Lunes', 'Martes', 'Miércoles']
Lunes
Martes
Miércoles
IndexError: list index out of range
```

Listas | Recorrer listas con for y While

Es posible recorrer una lista utilizando **for** y **range** o **while** para generar la secuencia de índices:

For con listas

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

For con WHILE

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

len(lista) retorna la cantidad de elementos que posee una lista y resulta muy útil, entre otras cosas, para determinar la cantidad de ciclos de un bucle.

Listas | Recorrer listas con for.. in

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:

For..ir

```
vocales = ['a','e','i','o','u']
# El bucle recorre la lista
for i in vocales:
   print(i)
```

Terminal

```
a
e
i
o
u
```

Listas | Desempaquetado y concatenado

El proceso de desempaquetado consiste en asignar cada elemento de una lista a una variable. Además, las listas pueden concatenarse con el operador suma:

Desempaquetado

```
dias = ["Lunes", "Martes", "Miércoles"]
d1, d2, d3 = dias
print(d1)
print(d2)
print(d3)
```

Terminal

```
Lunes
Martes
Miércoles
```

Concatenado

```
lista1 = [1,2,3]
lista2 = [4,5,6]
lista3 = lista1 + lista2
print(lista3)
```

Terminal

```
[1, 2, 3, 4, 5, 6]
```

Listas | max(), min() y sum()

La función **max()** devuelve el mayor elemento de una lista.
La función **min()** devuelve el menor elemento de una lista.
La función **sum()** devuelve la suma de los elementos de una lista:

max(), min() y sum()

```
lista = [3,4,5,6]
print(max(lista))
print(min(lista))
print(sum(lista))
```

Terminal

```
6
3
18
```

Listas | in / not in y list()

Los operadores de pertenencia **in / not** in permiten determinar si un elemento está o no en una lista. La función **list()** convierte cualquier secuencia a una lista. Se puede utilizar con rangos, cadenas y otros.

in / not in

```
lista = list(range(6))
print(lista)
cadena = "Hola"
print(list(cadena))

lista2 = [3,4,5,6]
print(4 in lista2)
print(8 in lista2)
print("A" not in lista2)
```

Terminal

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

['H', 'o', 'l', 'a']


True
False
True
```

Listas | .append() e .insert()

El método **append()** agrega un elemento al final de la lista.

. append()

```
lista = [3,4,5]
lista.append(6)
print(lista)
```

Terminal

```
[3, 4, 5, 6]
```

insert(<pos>, <elemento>) inserta un elemento en una posición determinada:

. insert()

```
lista=[3,4,5]
lista.insert(0,2)
print(lista)
lista.insert(3,25)
print(lista)
```

Terminal

```
[2, 3, 4, 5]
[2, 3, 4, 25, 5]
```

Listas | .pop() y .remove()

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.
remove(<valor>) elimina un elemento en la lista, identificado por su valor.

. pop()

```
lista = [6,9,8]
lista.pop()
#Resultado: [6,9]
```

. pop(posicion)

```
lista = [3,4,5]
lista.pop(1)
#Resultado: [3,5]
```

.remove(valor)

```
lista = [3,4,5]
lista.remove(3)
#Resultado: [4,5]
```

Listas | .index(), count() y reverse()

index(<valor>) busca un valor y devuelve su posición. Admite como argumento adicional un índice inicial a partir de donde comenzar la búsqueda. **count()** devuelve la cantidad de repeticiones de un elemento, cero si no lo encuentra. Y **reverse()** Invierte el orden de los elementos de una lista.

. index(valor)

```
lista = [3,4,5]
print(lista.index(5))
#Resultado: 2
```

. count()

```
lista = [3,4,5,3,5,8,5]
print(lista.count(5))
# Resultado: 3
print(lista.count(2))
# Resultado: 0
```

.reverse()

```
lista = [3,4,5]
lista.reverse()
print(lista)
# Resulado: [5,4,3]
```

Listas | .sort() y clear()

sort() ordena los elementos de la lista, de menor a mayor**. sort(reverse=True)** ordena la lista de mayor a menor. En ambos casos, el orden depende del tipo de dato contenido en la lista. Y **clear()** elimina todos los elementos de la lista.

. sort()

```
lista = [5, 1, 7, 2]
lista.sort()
print(lista)
#Resultado: [1,2,5,7]
```

sort(reverse=True)

```
lista = [5, 1, 7, 2]
lista.sort(reverse=True)
print(lista)
#Resultado: [7,5,2,1]
```

.clear()

```
lista = [3,4,5]
lista.clear()
print(lista)
#Resultado: []
```

Tuplas

Las tuplas en Python son conjunto de elementos separados por comas y encerrados entre paréntesis. Los paréntesis no son obligatorios. Las tuplas son inmutables y en general contienen una secuencia heterogénea de elementos. Podemos pensar en una tupla como si fuese una lista, pero recordando que su inmutabilidad hace que muchos de sus métodos y funciones no puedan utilizarse.

Tuplas

Las tuplas se pueden crear por extensión, o mediante un **empaquetado (pack)**:

Creación de una tupla

```
# Creación: Por extensión
tupla1 = ('uno', 'dos', 'tres')
# Creación: Mediante empaquetado
tupla2 = 'Palotes, Juan de', (1930, 11, 13), 3000936
```

Y se pueden “desempaquetar” (**unzip**), volcando su contenido a variables. Se requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la tupla.

Desempaquetado de una tuplas:

```
nombre,  nacimiento, dni  =( 'Palotes, Juan de', (1930, 11, 13), 3000936)
```

Tuplas | Acceso

Ejemplo de tupla

```
()                                                  # tupla vacía
'un valor',                                         # tupla con un valor
('uno', 'dos', 'tres')                              # cadenas
('Palotes, Juan de', (1930, 11, 13), 3000936)       # datos de persona
```

Se accede a los elementos desempaquetando, mediante un índice o usando rebanadas (**slices**):

Acceso a elementos de una tuplas

```
tupla = ('Palotes, Juan de', (1930, 11, 13), 3000936)
nombre, fecha, dni = tupla # Desempaque
# Acceso por índices:
print('Nombre: ', tupla[0], '. Fecha nac.: ', tupla[1], '. DNI: ', tupla[2])
print(tupla[::]) # Acceso mediante rebanadas
```

Conjuntos

Los conjuntos en Python son una metáfora de los conjuntos del área de la matemática. Al igual que aquellos, en Python son conjuntos de elementos únicos, no ordenados y sus elementos no son mutables.
Se definen sus elementos encerrándolos entre llaves y separados por comas.
No puede accederse a los elementos de un conjunto a través de un subíndice pues sus elementos no están ordenados.

Conjuntos | Conjuntos

Ejemplos de conjuntos:

```
set()                                                  # conjunto vacío
{'un valor'}                                           # conjunto con un valor
{'uno', 'dos', 'tres'}                                 # conjunto de cadenas
{'Palotes, Juan de', (1930, 11, 13), 3000936}          # datos de persona
```

Se accede a los elementos iterando

Acceso a elementos de un diccionario:

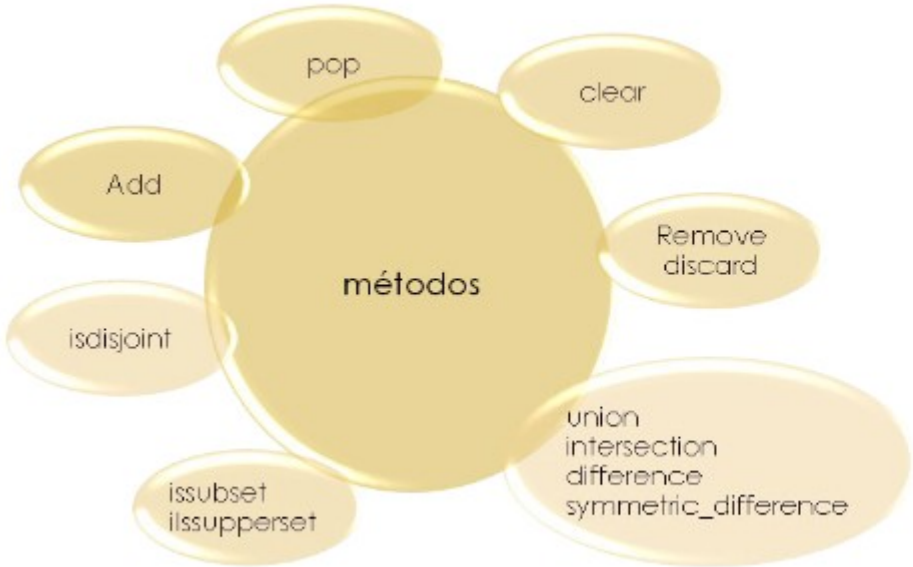
```
conjunto = {'Empleado', (1930, 11, 13), 36}
a = set(conjunto)
print(a)
[print (elem) for elem in a]
print(a)
```

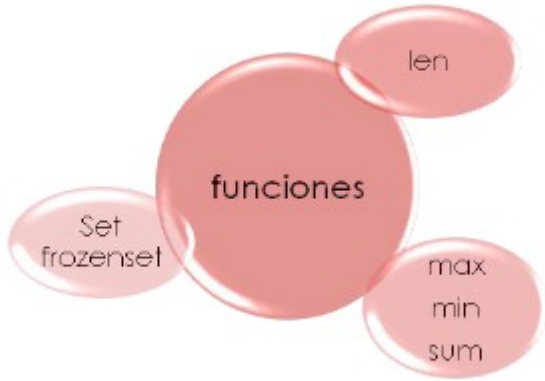
Terminal

```
{'Empleado', 36, (1930, 11, 13)}
Empleado
36
(1930, 11, 13)
{'Empleado', 36, (1930, 11, 13)}
```

Conjuntos | Métodos, funciones y operadores

Los siguientes diagramas resumen los métodos, funciones y operadores disponibles para los conjuntos:





Diccionarios

Los diccionarios en Python son un conjunto no ordenado de pares clave: valor. Estas claves son únicas, y si se intenta guardar un valor a una clave ya existente se pierde dicho valor.
Las claves pueden ser cualquier elemento de tipo inmutable (cadenas, números o tuplas (si estás sólo contienen cadenas, números o tuplas).
Se representan como una lista de pares (las claves y valores) separados por comas y encerrados entre llaves.

Diccionarios

Los diccionarios se pueden crear mediante la simple enumeración de los elementos, o por comprensión:

Creación de un diccionario

```
# Creación: Por extensión
diccionario  = {'Juan': 56, 'Ana': 15}
# Creación: Por compresión
diccionario  = {x: x ** 2 for x in (2, 4, 6)}
```

Para acceder a las claves Python utiliza un método de hash. +info

Ejemplos de diccionarios:

```
{}                          # diccionario vacío
{'Juan': 56}                # diccionario de un elemento
{'Juan': 56, 'Ana': 15}     # diccionario de dos elementos
```

Diccionarios | Acceso

Puede accederse a un diccionario de diferentes maneras:

- ✓ A las claves, utilizando método **keys()**
- ✓ A los valores, utilizando la clave como índice
- ✓ A la clave-valor, utilizando método **ítems()**
- ✓ No es posible obtener porciones de un diccionario usando [:]

Ejemplos de diccionarios:

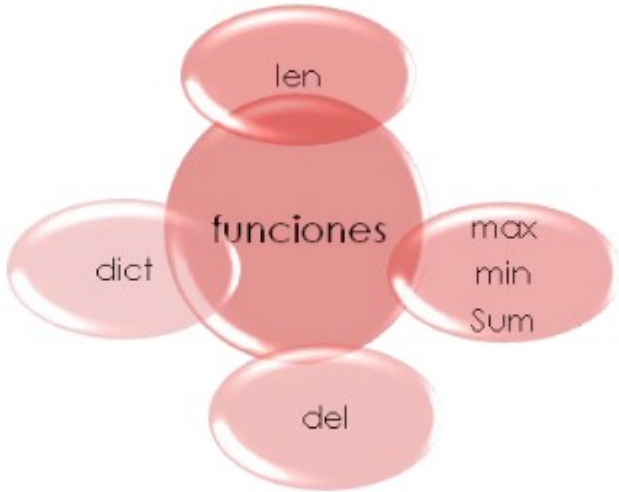
```
diccionario = {1: 'uno', 2:'dos', 3:'tres'}
print(diccionario.keys())
for i in diccionario.keys():
    print(diccionario[i])
for clave, valor in diccionario.items():
    print(clave, ':', valor, end= '; ')
```

Terminal

```
dict_keys([1, 2, 3])
uno
dos
tres
1 : uno; 2 : dos; 3 : tres;
```

Diccionarios | Métodos, funciones y operadores

Los siguientes diagramas resumen los métodos, funciones y operadores disponibles para los diccionarios:



MATERIAL EXTRA

Artículos de interés

Material de extra:

- ✓ Listas
- ✓ Diccionarios
- ✓ Tuplas
- ✓ Conjuntos

Videos:

- ✓ Listas, tuplas y diccionarios, en Píldoras Informáticas
- ✓ Conjuntos (primera parte) y conjuntos (segunda parte) en Programación ATS