ESCUELA POLITÉCNICA NACIONAL

cec·epn CENTRO DE EDUCACIÓN CONTINUA

## UNIDAD 4

# Funciones, tuplas, diccionarios y procesamiento de datos.

python™

Ing. Efraín Montenegro

# Listas.

Se pueden crear de dos formas, con los corchetes o usando el constructor de la clase, list(iterable). En este caso, el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto iterable puede ser o una secuencia, un contenedor que soporte la iteración o un objeto iterador.

El tipo str también es un tipo secuencia. Si pasamos un string al constructor list() creará una lista cuyos elementos son cada uno de los caracteres de la cadena:

```
>>> vocales = list('aeiou')
>>> vocales
['a', 'e', 'i', 'o', 'u']
```

```
>>> lista_1 = []  # Opción 1
>>> lista_2 = list()  # Opción 2
```

# Operaciones mutables en listas.

**append** Añade un único elemento al final de la lista.

```
x = [1, 2]
x.append('h')
```
Output: [1, 2, 'h']

**extend** Añade otra lista al final de una lista.

```
x = [1, 2]
x.extend([3, 4])
```
Output:[1, 2, 3, 4]

**insert** Inserta un nuevo elemento en una posición determinada de la lista.

```
x = [1, 2]
x.insert(0, 'y')
```
Output: ['y', 1, 2]

**del** Elimina el elemento ubicado en el indice descrito, a su vez tiene la capacidad de eliminar una sección de elementos de la lista.

```
x = [1, 2, 3]
del x[1]
```
Output:[1, 3]

```
y = [1, 2, 3, 4, 5]
del y[:2]
```
Output:[3, 4, 5]

**remove** Remueve la primer coincidencia del elemento especificado.

```
x = [1, 2, 'h', 3, 'h']
x.remove('h')
```
Output:[1, 2, 3, 'h']

**reverse** Invierte el orden de los elementos de la lista.

```
x = [1, 2, 'h', 3, 'h']
x.reverse()
```
Output:['h', 3, 'h', 2, 1]

**sort** Ordena los elementos de la lista de menor a mayor, este comportamiento puede modificarse mediante el parámetro reverse=True.

```
x = [3, 2, 1, 4]
x.sort()
```
Output:[1, 2, 3, 4]

```
y = ['R', 'C', 'Python', 'Java', 'R']
y.sort(reverse=True)
Output:
['R', 'R', 'Python', 'Java', 'C']
```

Se debe realizar sobre listas que contengan elementos del mismo tipo de dato

## Operaciones inmutables en listas.

Permiten trabajar con listas sin alterar o modificar su definición previa

**sorted** Permite ordenar los elementos de una lista de menor a mayor, no se encuentra limitado a las listas.

```
x = [5, 2, 9, 0]
print(sorted(x))
[0, 2, 5, 9]
```

**+** Permite concatenar o unir dos listas diferentes en nueva lista.

```
x = [1, 2, 3]
y = [4, 5, 6]
print(x + y)
```
Output:[1, 2, 3, 4, 5, 6]

* Permite replicar una lista hasta la cantidad de veces indicada.

```
x = [1, 2, 3]
print(x * 3)
Output:
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

**min** Retorna el elemento más pequeño dentro de una lista.

**max** Retorna el elemento más grande dentro de una lista.

```
x = [40, 100, 3, 9, 4]
print(min(x))  print(max(x))
Output:3       Output:100
```

**index** Retorna la posición en la lista del elemento especificado.

```
x = [10, 30, 20]
print(x.index(30))
Output:1
```

**count** Retorna la cantidad de veces que el elemento especificado se encuentra en la lista.

```
x = [10, 30, 20, 30, 30]
print(x.count(30))
Output:3
```

**sum** Realiza la suma de los elementos de la lista.

```
x = [2.5, 3, 3.5]
print(sum(x))  Output: 9.0
```

**in** Determina si un elemento especifico se encuentra en una lista

```
x = ['h', 2, 'a', 6, 9]
print('a' in x) Output: True
```

# Tuplas.

Son muy similares a las listas y comparten varias de sus funciones y métodos integrados, aunque su principal diferencia es que son **inmutables.** En lugar de inicializarse con corchetes se lo hace con ().

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

Para convertir a tipos tuplas debe usar la función *tuple()*, la cual está integrada en el interprete Python.

```
lista = [1, 2, 3]
tupla = tuple(lista)
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

También pueden declararse sin (), separando por , todos sus elementos.

```
tupla = 1, 2, 3
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

Las operaciones y métodos asociados son los mismos que en las listas, considerando su propiedad de inmutables.

# Diccionarios.

Son estructura de datos y un tipo de con características especiales que permite almacenar cualquier tipo de valor como enteros, cadenas, listas e incluso otras funciones. Identifican cada elemento por una clave (Key).

Para definirlo junto con los miembros que va a contener, se encierra el listado de valores entre llaves, las parejas de clave y valor se separan con comas, y la clave y el valor se separan con *:* .

```
punto = {'x': 2, 'y': 1, 'z': 4}
```

Es posible declararlo vacío y luego ingresar los valores, se lo declara como un par de llaves sin nada en medio, y luego se asignan valores directamente a los índices.

```
materias = {}
materias["lunes"] = [6103, 7540]
materias["martes"] = [6201]
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = [6201]
```

Para buscar en las listas, se utiliza un algoritmo de comparación que tarda cada vez más a medida que la lista se hace más larga. En cambio, para buscar en diccionarios se utiliza un algoritmo tipo hash, que se basa en realizar un cálculo numérico sobre la clave del elemento. Tiene una propiedad importante: sin importar cuántos elementos tenga el diccionario, el tiempo de búsqueda es aproximadamente igual.

Pueden tener diferentes tipos de valores, y el acceso a los mismos será mediante su key, ya que no están indexados:

```
diccionario = {'nombre' : 'Carlos', 'edad' : 22, 'cursos': ['Python','Django','JavaScript'] }
```

```
print diccionario['nombre'] #Carlos
print diccionario['edad']#22
print diccionario['cursos'] #['Python','Django','JavaScript']
```

Los arreglos internos de un diccionario si están indexados:

```
print diccionario['cursos'][0]#Python
print diccionario['cursos'][1]#Django
print diccionario['cursos'][2]#JavaScript
```

# Operaciones en Diccionarios.

**dict ()** Permite crear un diccionario, si es factible, devuelve un diccionario de datos estructurado.

```
dic =  dict(nombre='nestor', apellido='Plasencia', edad=22)

dic → {'nombre' : 'nestor', 'apellido' : 'Plasencia', 'edad' : 22}
```

**zip()** Usa dos elementos iterables, ya sea una cadena, una lista o una tupla. Se devolverá un diccionario relacionando el elemento i-esimo de cada uno de los iterables.

```
dic = dict(zip('abcd',[1,2,3,4]))
dic →   {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
```

**items()** Devuelve una lista de tuplas, cada tupla se compone de la clave y su valor.

```
dic =   {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
items = dic.items()
items → [('a',1),('b',2),('c',3),('d',4)]
```

**keys()** Retorna una lista de elementos, con las claves del diccionario.

```
dic =  {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
keys= dic.keys()
keys→ ['a','b','c','d']
```

**values()** Retorna una lista de elementos con los valores del diccionario.

```
dic =  {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
keys= dic.keys()
keys→ ['a','b','c','d']
```

**clear()** Elimina todos los ítems del diccionario.

```
In [98]: dic_1= {"a":1,"b":2,"c":3,"d":4}
In [101]: print(dic_1.clear())
None
```

**copy()** Retorna una copia del diccionario original.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
dic1 = dic.copy()
dic1 → {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
```

**get()** Devuelve el valor de la clave. Si no lo encuentra, devuelve un objeto none.

```
dic = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
valor = dic.get('b')
valor → 2
```

**update()** Usa dos diccionarios, si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
dic 1 = {'a' : 1, 'b' : 2, 'c' : 3 , 'd' : 4}
dic 2 = {'c' : 6, 'b' : 5, 'e' : 9 , 'f' : 10}
dic1.update(dic 2)
dic 1 → {'a' : 1, 'b' : 5, 'c' : 6 , 'd' : 4 , 'e' : 9 , 'f' : 10}
```

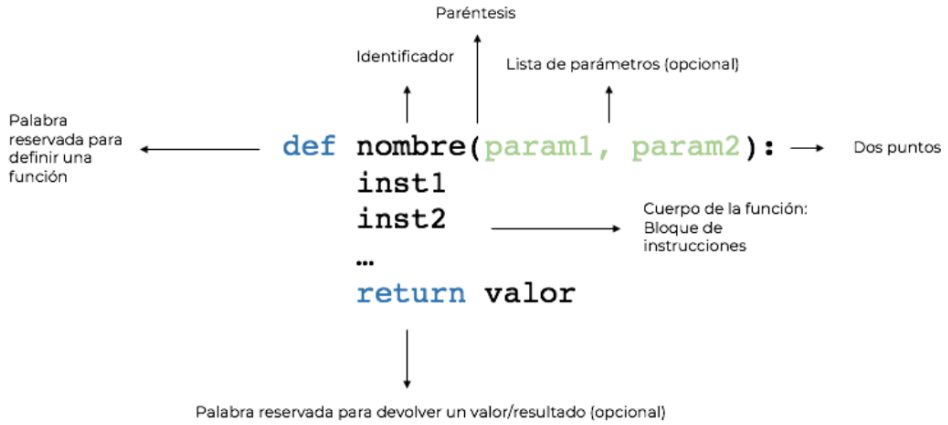
# Funciones.

Son bloques de código con un nombre asociado, que reciben cero o más argumentos como entrada, tienen dos objetivos:

- Dividir y organizar el código en partes más sencillas - modularización.
- Encapsular el código que se repite a lo largo de un programa para ser reutilizado - reutilización.

Se utiliza la palabra reservada *def*. A continuación viene el nombre de la función que es el que se utilizara también para llamarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. Por último, como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:).

Tras los dos puntos se incluye el cuerpo de la función que son el conjunto de instrucciones que se encapsulan en dicha función y le dan significado. En último lugar y de manera opcional, se añade la instrucción con la palabra reservada *return* para devolver un resultado.



Cuando la primera instrucción de una función es un string encerrado entre tres comillas simples ''' o dobles """, a dicha instrucción se le conoce como *docstring*. Es una cadena que se utiliza para documentar la función.

```
>>> def hola(arg):
...     """El docstring de la función"""
...     print "Hola", arg, "!"
...
>>> hola("Plone")
Hola Plone !
```

## return

La declaración *return* se usa para salir de la función de Python. Las dos formas más comunes en las que se usa esta declaración son:

1. Devolver un valor de una función después de que ha salido o se ha ejecutado. Es posible usar el valor más adelante en el programa.

```
def add(a, b):
    return a+b

value = add(1,2)
print(value)
3
```

2. Cuando queremos detener la ejecución de la función en un momento dado.

```
def add(a, b):
    if(a == 0):
        return
    elif(b == 0):
        return
    else:
        sum = a + b
        return sum
value = add(0,2)
print(value)
None
```

## Retorno implícito.

```
def solution():
    name = "john"

    if(name == "john"):
        print('My name ',name)
solution()
My name john
```

- No es obligatorio escribir la declaración return. Siempre que sale de cualquier función se llama a return, inclusive de forma implícita.
- Se usa el valor de None solo si no ha especificado la declaración return. El valor None significa que la función ha completado su ejecución y no devuelve ningún dato. A estas funciones se les conoce como ***void***.
- Si ha especificado la instrucción return sin ningún parámetro, también es lo mismo que return None.

## Retorno explícito.

Al agregar una declaración *return* dentro del código, se denomina tipo de retorno explícito. Gracias a esto es posible pasar un valor calculado por una función y almacenarlo dentro de una variable para su uso posterior o detener la ejecución de la función en puntos definidos o con base en condiciones.

## Argumentos y parámetros.

Al definir una función los valores los cuales se reciben se denominan parámetros, pero durante la llamada los valores que se envían se denominan argumentos.

```
>>> def f(x):
        return x**2 + 1

>>> f(4)
17
```

## Múltiples argumentos.

Cada función en Python recibe un determinado número de argumentos, se debe separar con comas dentro de la definición:

```
def myfunction(first, second, third):
    # Haz algo con las tres variables
```

```
def restar_bc(a,b,c):
    resultado = a - b * c
    print("Valor de a es", a)
    print("Valor de b es", b)
    print("Valor de c es", c)
    return(resultado)
```


ESCUELA
POLITÉCNICA
NACIONAL


cec·epn
CENTRO DE EDUCACIÓN CONTINUA


python™