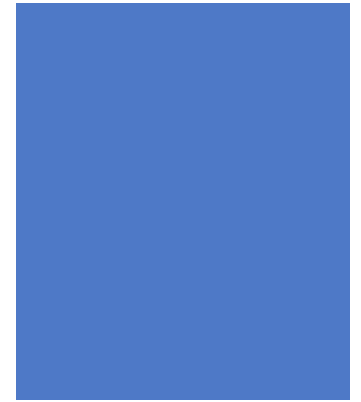


# Tema 7.

## Tipos estructurados (tuplas)



Curso 2020/2021  
Programación



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

# 1. Tuplas

- Una **tupla** es similar a una lista, pero inmutable: una vez creada no se pueden modificar sus elementos
- Una tupla se define mediante una lista de expresiones, separadas por comas y, opcionalmente, encerradas entre paréntesis.

Ejemplo:

```
a = (1, 7, 3)
```

O bien:

```
a = 1, 7, 3
```

- Podemos consultar tanto la longitud de una tupla como sus elementos:

```
len(a)
```

3

```
a[0]
```

1

# 1. Tuplas

- Recorrido de tuplas:

```
a = (1, 2, 3)
for i in a:
    print(i)
```

1  
2  
3

```
a = (1, 2, 3)
for i in range(len(a)):
    print(a[i])
```

1  
2  
3

- Pueden tener valores de diferentes tipos
- Python ofrece un potente mecanismo de asignación de tuplas: empaquetado y desempaquetado de tuplas

```
x = ('Luis', 20, 7.5)      # empaquetado
(nombre, edad, nota) = x   # desempaquetado
print(nombre, edad, nota)
```

Luis 20 7.5

# 1. Empaquetado/desempaquetado de tuplas

- Permite asignar una tupla de expresiones (a la derecha del =) a una tupla de variables (a la izquierda del =)
  - Se puede usar para asignar valores a varias variables simultáneamente:

```
x, y = 1, 2  
print(x, y)
```

1 2

- Por ejemplo, el siguiente código intercambia los valores de las variables x e y:

```
x, y = y, x  
print(x, y)
```

2 1

# 1. Empaquetado/desempaquetado de tuplas

- Puede usarse también en las funciones para devolver múltiples valores:

```
def minMax(a):  
    minimo = maximo = a[0]    # calcula el mínimo y el máximo de una lista  
    # inicializamos el mínimo y el máximo con el 1er elemento  
    for i in range(1, len(a)):  
        # recorremos el resto de elementos de la lista  
        if a[i] < minimo:  
            # si encontramos uno menor que el mínimo actual  
            minimo = a[i]    # actualizamos el mínimo  
        elif a[i] > maximo:  
            # y si encontramos uno mayor que el máximo actual  
            maximo = a[i]    # actualizamos el máximo  
    return (minimo, maximo)    # devolvemos una tupla con el mínimo y el máximo
```

```
lista = [5, 2, 3, 8, 7]    # definimos la lista  
(minimo, maximo) = minMax(lista)    # llamamos a la función y desempaquetamos el resultado  
print(minimo, maximo)    # imprimimos el mínimo y el máximo de la lista
```

# 1. Indexación de tuplas: también funciona el operador de corte

- Funciona igual que a Strings
- El operador de corte se denota con dos índices separados por dos puntos (:), dentro de los corchetes de indexación
- La expresión `a[i:j]` devuelve la subparte formada por los elementos desde `a[i]` hasta `a[j-1]`

```
>>> t = (0,1,2,3,4,5,6)
>>> t[2:5]
(2, 3, 4)
>>> t[0:4]
(0, 1, 2, 3)
>>> |
```

# 1. Indexación de tuplas

- También se pueden utilizar índices negativos: los valores negativos acceden al tupla de derecha a izquierda. El último elemento de la tupla tiene índice  $-1$ , el penúltimo,  $-2$ , y así sucesivamente

```
>>> t = ("hola", 5, [3,4], "Wow!")
>>> t[-1]
'Wow!'

>>> t[-3]
5

>>> t[-8]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
IndexError: tuple index out of range

>>> t[-2]
[3, 4]

>>>
```

# 1. Operaciones con tuplas

- Concatenación (+)

```
>>> (1,2,3,4) + (5,6,7)
(1, 2, 3, 4, 5, 6, 7)
```

- Repetición (\*)

```
>>> (1,2,3)*4
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> 2*(1,1)
(1, 1, 1, 1)
```

- Comparaciones (== , !=, <, <=, > y >=)

- Las tuplas se comparan posición por posición
- El primer elemento de la primera tupla se compara con el primer elemento de la segunda tupla:
  - si no son iguales (es decir, el primero es mayor o menor que el segundo), entonces ese es el resultado de la comparación,
  - de lo contrario, se considera el segundo elemento, luego el tercero y así sucesivamente.



# Comparaciones (== , !=, <, <=, > y >=) de tuplas

```
>>> (1,2,3) < (1,2)
False
>>> (1,2,3) < (1,2,4)
True
>>> (1,2,3) < (1,2,4)
True
>>> (1,2,3) < (1,5)
True
>>> (1,2,3) < (2,5)
True
>>> (3,2,3) < (2,5)
False
>>> (1,2,3) < (1,2)
False
```

# Tuplas son inmutables



La  
diferencia  
con listas

```
>>> ls = [1,2,3,4,5]
>>> tp = (1,2,3,4,5)
>>>
>>> ls[2] = 10
>>> ls
[1, 2, 10, 4, 5]
>>> tp[2] = 10
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> del ls[3]
>>> ls
[1, 2, 3, 5]
>>> del tp[3]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

## 2. Pasar argumentos a funciones

- una función con 2 **parámetros**:

- nombre
- msg

```
def saludar(nombre, msg):  
    """Este función imprime un saludo a la persona  
    con nombre | con el mensaje msg"""  
  
    print("Hola", nombre + ', ' + msg)
```

- lo llamamos con 2 **argumentos posicionales**:

- “Tanja”
- “Bon dia!”

```
>>> saludar("Tanja", "Bon dia!")  
Hola Tanja, Bon dia!
```

- estos valores se asignan a los paramteros según su posición
  - 1er parametro -> primer argumento
  - 2do parametro -> segundo argumento

- Si no lo llamamos con exactamente 2

```
>>> saludar ("Tanja")  
Traceback (most recent call last):  
  File "<pyshell>", line 1, in <module>  
TypeError: saludar() missing 1 required positional argument: 'msg'
```

# Valores de argumentos predeterminado

(default values)

- una función con 2 **parámetros**
  - nombre
  - msg (**predeterminado**)

```
def saludar(nombre, msg="Bon dia!"):
    """Este función imprime un saludo a la persona
    con nombre con el mensaje msg.

    Si el argumento msg no esta dado,
    sale el valor predeterminado: Bon dia! """

    print("Hola", nombre + ', ' + msg)
```

- lo podemos llamar con 1 o 2 **argumentos posicionales**:

```
>>> saludar("Tanja")
Hola Tanja, Bon dia!
```

```
>>> saludar("Tanja", "Goodmorning!")
Hola Tanja, Goodmorning!
```

# Cambiar el orden posicional de argumentos

(usando palabras clave - keywords)

- Python permite que se invoquen funciones utilizando los nombres de los parametros como palabras clave.

```
>>> saludar(nombre="Tanja", msg = "Goodnight!")
Hola Tanja, Goodnight!
>>> saludar(msg = "Goodnight!", nombre="Tanja")
Hola Tanja, Goodnight!
>>> saludar(nombre="Tanja")
Hola Tanja, Bon dia!
>>> saludar("Tanja", msg = "Goodnight!")
Hola Tanja, Goodnight!
>>> saludar(nombre="Tanja", "Goodnight!")
File "<pyshell>", line 1
SyntaxError: positional argument follows keyword argument
```

una vez usado argumentos con palabra clave, ya no podemos usar posicionales!

# Cantidad arbitraria de argumentos (varargs)

- A veces, no sabemos de antemano el número de argumentos que se pasarán a una función.
- En la definición de la función podemos usar un asterisco (\*) antes del nombre del parámetro
  - estos parametros no tienen palabras clave
  - y tienen que ir despues de la lista de los parametros posicionales
- Usando el \*, la variable que asociamos con el \* se convierte en un iterable, entonces podemos iterar sobre él
- Sintaxis:


```
def functionname ([formal_args,] *var_args_tuple):  
    """documentación"""  
  
    funcionalidad
```



# Ejemplo

```
def saludar(msg, *nombres):  
    """Este función imprime un saludo a todas las  
    personas en *nombres con el mensaje msg.  
  
    Si el argumento msg no esta dado,  
    sale el valor predeterminado: Bon dia! """  
  
    for nombre in nombres:  
        print("Hola", nombre + ', ' + msg)
```

iterar sobre  
nombres



```
>>> saludar("Bon dia!", "Tanja", "Alvaro", "Luis", "Daniel")  
Hola Tanja, Bon dia!  
Hola Alvaro, Bon dia!  
Hola Luis, Bon dia!  
Hola Daniel, Bon dia!
```

```
>>> saludar("Bon dia!")  
>>> |
```

La tupla nombres permanece vacía si no se especifican argumentos adicionales durante la llamada a la función.

# Cantidad arbitraria de argumentos CON keyword (kwargs)

- Con kwargs podemos pasar una cantidad arbitraria de argumentos con keywords (palabras clave).
- En la definición de la función podemos usar 2 asteriscos (\*\*) antes del nombre del parámetro
  - estos parametros tienen que ir despues de los varargs
- *kwargs* son como un diccionario que asigna cada keyword al valor que pasamos junto a ella.
- Sintaxis:

```
def functionname ([formal_args,] *var_args_tuple, **kwargs):  
    """documentación"""  
  
    funcionalidad
```



# Ejemplo

```
def saludar(**nombres_msgs):  
    """Este función imprime un saludo a todas las  
    personas en los keys de **nombres_msgs con el mensaje en  
    los valores.  
    """  
  
    for nm in nombres_msgs:  
        print("Hola", nm + ', ' + nombres_msgs[nm])
```

```
>>> saludar(Tanja = "Goedemorgen!", Alvaro = "¡Buenas días!", Luis = "Bon dia!", Daniel = "Goodmorning!")  
Hola Tanja, Goedemorgen!  
Hola Alvaro, ¡Buenas días!  
Hola Luis, Bon dia!  
Hola Daniel, Goodmorning!
```