



## Unidad 1: Introducción a Estructuras de Datos

### Estructuras de Datos

#### FACULTAD DE CIENCIAS DE LA ADMINISTRACIÓN - UNER

In [1]:

```
from datetime import datetime
print(f"{'PRIMER' if datetime.now().month < 7 else 'SEGUNDO'} CUATRIMESTRE DE {datetime.now().y
```

PRIMER CUATRIMESTRE DE 2024 🤖

# Funciones y módulos

## Objetivos

- Entender cómo definir funciones.
- Comprender las distintas técnicas para separar las responsabilidades de una aplicación.
- Conocer la importancia de la reutilización de código.
- Definir funciones recursivas.

## Temas a desarrollar:

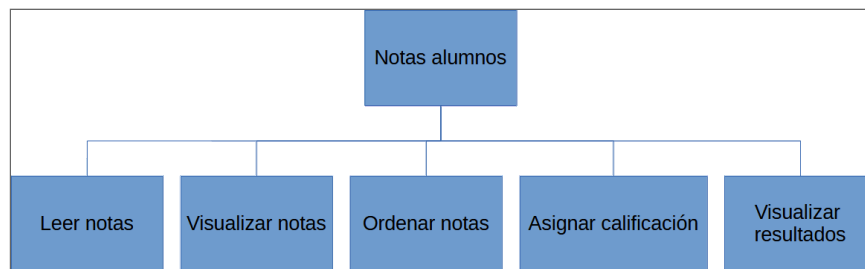
- Modularización. Definición. Funciones. Definición.
- Parámetros y argumentos. Técnicas de diseño top-down y bottom-up.
- Recursividad. Definición.

## Modularización y diseño descendente

- Uno de los métodos fundamentales para resolver un problema es dividirlo en problemas más pequeños llamados **subproblemas**.
- Estos problemas pueden ser divididos repetidamente en problemas más pequeños hasta que sean solucionados.
- La técnica de dividir el problema principal en subproblemas se denomina frecuentemente, **divide y vencerás**.
- El método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se denomina **diseño descendente (top-down design)** debido a que se comienza en la parte superior con un problema general y se diseñan soluciones específicas a sus subproblemas.
  - Cada subproblema es deseable que sea **independiente** de los restantes y se denomina **módulo**.
  - El problema principal se resuelve con el programa principal (también llamado conductor del programa) y los subproblemas (**módulos**) mediante subprogramas.

## Modularización y diseño descendente (2)

- Un **subprograma** realiza una tarea concreta que se describe con una serie de instrucciones.
- La resolución de un problema comienza con una **descomposición modular** y luego nuevas descomposiciones de cada módulo en un proceso denominado **refinamiento sucesivo** (stepwise).
- **Ejemplo:**
  - Dadas las puntuaciones de una clase de informática, ordenar las puntuaciones (notas) en orden descendente; a continuación visualizar la calificación alcanzada basada en la puntuación. Realizar el análisis y el algoritmo.



# Funciones

- En el contexto de programación, una **función** es un **nombre** que se le asigna a una **secuencia de sentencias** que llevan a cabo un cómputo.
- Una **función** nos permite definir un bloque de código **reutilizable** que se puede ejecutar muchas veces dentro de nuestro programa.
- El resultado es un código simple, elegante y más legible, lo que facilita la depuración y limita los errores de escritura.

## Llamadas a funciones

- Ya previamente vimos invocaciones a funciones cuando usamos la función `type(42)`.

In [2]:

```
print(type(42))
```

```
<class 'int'>
```

- El nombre de la función es `type`. La expresión entre paréntesis es llamada el **argumento** de la función. El **resultado**, para esta función es el tipo del argumento.
- Es común decir que la función **"toma"** un **argumento** y **"retorna"** un **resultado**.

## Funciones matemáticas

- El **módulo math** nos provee las funciones matemáticas más comunes.
- Un **módulo** es un archivo que contiene una colección de funciones relacionadas. Antes de que podamos usar las funciones de un módulo, tenemos que **importarlo** con la sentencia `import`.

In [3]:

```
import math
print(math)
```

<module 'math' (built-in)>

- `import math` crea un objeto de **tipo módulo** que se llama `math`.
- Para acceder a las funciones del módulo, se tiene que especificar el nombre del módulo y el nombre de la función, separados por el símbolo punto `.`.

In [4]:

```
signal_power = int(input("Ingrese un valor para señal: "))
noise_power = int(input("Ingrese un valor para ruido: "))

ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
radians = 0.7
height = math.sin(radians)

print(f"ratio: {ratio}, decibels: {decibels}, radians: {radians}, height: {height}")
```

ratio: 0.11764705882352941, decibels: -9.294189257142927, radians: 0.7, height: 0.644217687237691

- En el ejemplo usamos `math.log10` para computar la relación señal/ruido en decibeles. El módulo `math` también provee de la función `log`, que computa logaritmos en base `e`.



## Creación de funciones

- **Python** nos permite crear nuestras propias funciones. Una **definición de función** especifica el **nombre** para la nueva función y una secuencia de sentencias que se van a ejecutar cuando la función sea invocada.

Ejemplo:

In [5]:

```
def imprimir_letra():  
    print("Hey Jude, don't make it bad.")  
    print("Take a sad song and make it better.")
```

- La definición de las funciones se ejecuta tal cual el resto de las sentencias, no generan ninguna salida y el efecto que tienen es crear un **objeto función**.
- La palabra reservada que indica que comienza la definición de una función es `def`. El nombre de la función es `imprimir_letra`.
- Las reglas para los nombres de función son los mismos para los nombres de variables.
- Los paréntesis vacíos luego del nombre de la función indican que no recibe ningún argumento.
- La primer línea de la definición de la función llama **cabecera**; el resto **cuerpo**. La **cabecera** tiene que finalizar con el carácter `:` (dos puntos) y el cuerpo de la función tiene que estar indentado. El cuerpo puede contener cualquier número de sentencias.
- La sintaxis para invocar funciones creadas por nosotros es la misma que para las funciones integradas. `imprimir_letra()`

In [6]:

```
imprimir_letra()
```

```
Hey Jude, don't make it bad.  
Take a sad song and make it better.
```

## Creación de funciones (2)

- Una vez que hemos definido una función podemos usarla dentro de otras funciones.
- Por ejemplo, para repetir la letra varias veces podemos escribir una función llamada `repetir_letra`:

In [7]:

```
def repetir_letra():  
    imprimir_letra()  
    imprimir_letra()  
  
# Y Luego Llamar a la función repetir_Letra:  
repetir_letra()
```

```
Hey Jude, don't make it bad.  
Take a sad song and make it better.  
Hey Jude, don't make it bad.  
Take a sad song and make it better.
```

## Parámetros y argumentos

- Existen funciones que requieren un argumento. Por ejemplo, cuando llamamos a la función `math.sin` pasamos un número como argumento.
- Otras funciones requieren más de un argumento. Por ejemplo, `math.pow`, necesita base y exponente.
- Dentro de la función, los argumentos son asignados a variables que se llaman **parámetros**.
- Los **argumentos y parámetros** se los llaman **parámetros actuales y formales** respectivamente.
- Aquí la definición de una función que acepta argumentos:

In [8]:

```
def imprimir_dos_veces(nombre):  
    """  
        La función asigna el argumento a un parámetro que se llama nombre.  
        Cuando la función es invocada, imprime el valor del parámetro dos veces.  
        Funciona con cualquier valor que pueda imprimirse por pantalla.  
    """  
    print(nombre)  
    print(nombre)
```

## Alcance de variables y parámetros

- Cuando creamos una variable dentro de una función su alcance es local, esto quiere decir que solo existe dentro de la función. Por ejemplo:

In [9]:

```
def concat_imprimir(part1, part2):  
    """  
    Esta función toma dos argumentos, los concatena, e imprime los resultados dos veces  
    """  
    cat = part1 + part2  
    imprimir_dos_veces(cat)  
  
linea1 = input("Escriba un texto: ")  
linea2 = input("Escriba otro texto: ")  
concat_imprimir(linea1, linea2)
```

```
texto1texto2  
texto1texto2
```

- Los parámetros son locales a la función. Cuando `concat_imprimir` termina, la variable `cat` es destruida. Si intentamos imprimir su valor obtendremos una excepción:

In [10]:

```
print(cat)
```

```
-----  
NameError                                Traceback (most recent call last)  
Cell In[10], line 1  
----> 1 print(cat)  
  
NameError: name 'cat' is not defined
```

## Funciones que no retornan valores

- Algunas funciones que hemos usado, tales como las funciones matemáticas, **retornan valores** (en algunos contextos se llaman **funciones que retornan valores** o **fructíferas**).
- En funciones, como `imprimir_dos_veces`, se lleva a cabo una acción pero **NO se retorna un valor**.
- Estas funciones son llamadas **funciones que no retornan valores**.
- Cuando llamamos a una función que retorna valores, casi siempre tenemos que hacer algo con el resultado. Por ejemplo, puede ser que queramos asignarlo a una variable o utilizarlo como parte de una expresión.

In [6]:

```
import math
calculo = (math.sqrt(5) + 1) / 2
print(calculo)
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 calculo = (math.sqrt(5) + 1) / 2
      2 print(calculo)

NameError: name 'math' is not defined
```

- Cuando llamamos a cualquier función en modo interactivo, **Python** muestra un resultado.
- Pero en un script si **llamamos a una función** que **devuelve resultados** el **valor de retorno se pierde**.
- Las funciones que retornan vacío puede ser que muestren algo en la pantalla o tengan algún efecto pero no tienen valor de retorno. Si asignamos el resultado a una variable, obtendremos un valor especial que es `None`.
- `None` no es lo mismo que `"None"`. `None` es un valor que tiene su tipo especial

In [12]:

```
print(type(None))
```

<class 'NoneType'>

## ¿Por qué funciones?

- Hay varias razones por las cuales vale la pena dividir un programa en funciones:
  - Crear una nueva función nos da la oportunidad de nombrar un grupo de sentencias lo que hace **más fácil de leer y depurar nuestros programas**.
  - Una función hace el programa más pequeño eliminando código repetitivo. Más tarde si tenemos que hacer un cambio **solo tenemos que hacerlo en un único lugar**.
  - Dividir un programa largo en funciones nos permite depurar las partes que lo componen una a la vez y luego ensamblarlo en un todo que funcione.
  - Las funciones bien diseñadas, generalmente son utilizadas por muchos programas. Una vez que escribimos y depuramos podemos reutilizarlas.

# Recursividad

- La **recursividad** o **recursión** nos permite resolver problemas o tareas donde las mismas pueden ser divididas en subtareas cuya funcionalidad es la misma. Dado que los subproblemas a resolver son de la misma naturaleza, **se puede usar la misma función para resolverlos**.
- Una **función recursiva** es aquella que se llama repetidamente a sí misma hasta llegar a un punto de salida.
- La **recursión** puede ser utilizada como una **alternativa a la repetición o estructura repetitiva**.
- El uso de la **recursión** es particularmente idóneo para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos (sobre todo de cálculo.).
- Una **función recursiva** es similar a una tradicional solo que tiene dos secciones de código claramente divididas:
  - La sección en la que la **función se llama a sí misma**.
  - Por otro lado, tiene que existir siempre una **condición de terminación** en la que la función **retorna** sin volver a llamarse. Es muy importante porque **de lo contrario**, la función **se llamaría de manera indefinida**.



## Cuenta regresiva

- Reveamos la solución iterativa de la cuenta regresiva para luego realizar su implementación recursiva.

### Implementación usando while

In [16]:

```
def cuenta_descendente(n):  
    while n > 0:  
        print (n)  
        n = n - 1  
    print ("It's the final countdown!!! Tarata taataa!!!")  
  
cuenta_descendente(5)
```

```
5  
4  
3  
2  
1  
It's the final countdown!!! Tarata taataa!!!
```

### Implementación recursiva

In [15]:

```
def cuenta_descendente_rec(n):  
    if n == 0:  
        print("It's the final countdown! Tarata taataa!!!")  
    else:  
        print(n)  
        cuenta_descendente_rec(n - 1)  
  
cuenta_descendente_rec(5)
```

```
5  
4  
3  
2  
1  
It's the final countdown! Tarata taataa!!!
```

# Factorial

- Muchas funciones matemáticas se definen **recursivamente**. Un ejemplo de ello es el factorial de un número entero  $n$ .

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1 & \text{si } n > 0 \end{cases} \quad 0! = 1$$

- Si se observa la fórmula anterior cuando  $n > 0$ , es fácil definir  $n!$  en función de  $(n-1)!$  Por ejemplo, 5:
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
  - $4! = 4 \times 3 \times 2 \times 1 = 24$
  - $3! = 3 \times 2 \times 1 = 6$
  - $2! = 2 \times 1 = 2$
  - $1! = 1 \times 1 = 1$
  - $0! = 1 = 1$
- En términos generales sería:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)! & \text{si } n > 0 \end{cases}$$

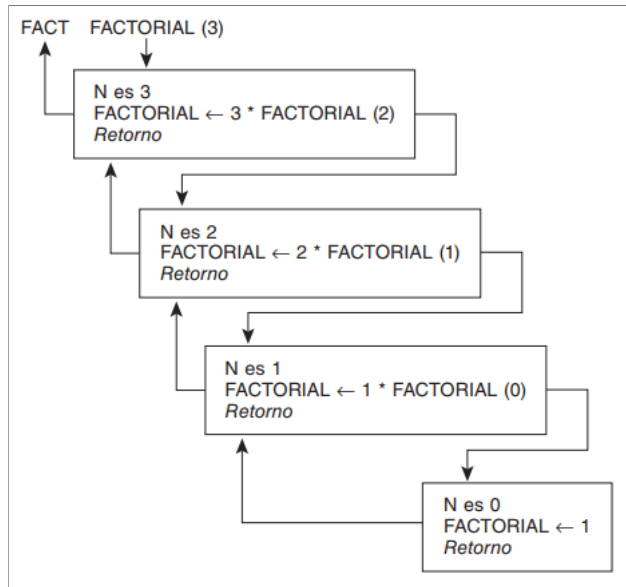
## Factorial (2)

In [18]:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
  
print(factorial(3))
```

6

## Factorial (3)



## Comprobación de tipos

- ¿Qué sucede si llamamos a factorial con 1.5 como argumento?

In [ ]:

```
factorial (1.5)
```

- Tiene todo el aspecto de una recursión infinita Pero, ¿Cómo ha podido ocurrir?
- Hay una condición de salida o caso base: cuando `n == 0`. Pero el valor de n nunca coincide con el caso base.
- Para solucionar esto podemos usar la función `isinstance()` para determinar si el tipo del parámetro es entero y también controlar que el parámetro sea positivo:

In [17]:

```
def factorial_control(n):  
    if not isinstance(n, int):  
        print('La función factorial solo se aplica a enteros.')  
        return None  
    elif n < 0:  
        print('La función factorial solo se aplica a enteros positivos.')  
        return None  
    elif n == 0:  
        return 1  
    else:  
        return n * factorial_control(n-1)  
  
print(factorial_control(5))
```

120

## Espacios de nombres y alcance de variables

- Un nombre de variable puede referirse a diferentes cosas, dependiendo de dónde se use.
- Los programas de **Python** tienen varios **espacios de nombres**. Un **espacio de nombres** es una sección dentro de la cual un nombre en particular es único y no está relacionado con el mismo nombre en otros espacios de nombres.
- Cada **función** define su propio **espacio de nombres**. Si define una variable con nombre `x` en un programa principal y otra variable llamada `x` en una función, se refieren a cosas diferentes.
  - Sin embargo, en caso de ser necesario, esto puede superarse. Se puede acceder a variables en otros espacios de nombres de varias maneras.
- La parte principal de un programa define el **espacio de nombres global**; por lo tanto, las variables en ese espacio de nombres son **variables globales**.
- Puede obtenerse el valor de una variable global desde dentro de una función:

In [ ]:

```
animal = 'carpincho'
def print_en_funcion():
    print('En función:', animal)

print('En el nivel superior:', animal)
print_en_funcion()
```

## Espacios de nombres y alcance de variables (2)

- Cambiamos un poco el ejemplo:

In [ ]:

```
animal = 'carpincho'
def cambiar_local():
    animal = 'benteveo' # variable local a función, oculta la global
    print('En función cambiar_local():', animal, id(animal))

print('En el nivel superior:', animal, id(animal))
cambiar_local()
```

- ¿Que pasó aquí? La primera línea asignó la cadena 'carpincho' a una variable **global** llamada `animal`.
- La función `cambiar_local()` también tiene una variable llamada `animal`, pero está en su espacio de **nombres local**.
- Usamos la función `id()` para imprimir el valor único de cada objeto y probar que la variable `animal` dentro de `cambiar_local()` no es lo mismo que `animal` en el nivel principal del programa.

## Espacios de nombres y alcance de variables (3)

- Para acceder a la variable **global** en lugar de la local dentro de una función, debe ser explícito y usar la palabra clave **global**:

In [ ]:

```
animal = 'carpincho'
def cambiar_local():
    global animal
    animal = 'benteveo' # acceso a variable global
    print('En función cambiar_local():', animal, id(animal))

print('En el nivel superior:', animal, id(animal))
cambiar_local()
print('En el nivel superior:', animal, id(animal))
```

- Si no anteponemos `global` al nombre de la variable dentro de una función, **Python** usa el espacio de nombres local y la variable es local, desapareciendo después de completada la función.



## Orden y posición de parámetros

- Cuando definimos una función, en su forma más básica, indicamos los nombres de los parámetros formales.
- También se define el orden en que deben colocarse los argumentos (parámetros actuales) cuando se invoca.
- Debemos respetar este orden, si no queremos tener resultados no deseados.

In [19]:

```
def saludar(saludo, nombre):  
    print("Digo", saludo, "a", nombre)  
  
saludar("Hola", "Juan")  
saludar("Juan", "Hola")
```

```
Digo Hola a Juan  
Digo Juan a Hola
```

## Orden y posición de parámetros (2)

- Podemos alterar el orden de pasaje de parámetros, siempre y cuando digamos explícitamente el nombre del parámetro formal que recibe el valor pasado.

In [ ]:

```
# Definición saludar con 2 parámetros formales: saludo y nombre
def saludar(saludo, nombre):
    print("Digo", saludo, "a", nombre)
saludar(nombre="Juan", saludo="Hola")
```

- La ventaja es que podemos alterar el orden de argumentos.
- La desventaja es que, sí o sí, debemos conocer el nombre de los parámetros para invocar la función de esta manera.
- Se pueden pasar argumentos con nombres explícitos y sin nombres. En el caso de no tener nombre, se los ubica por posición.

In [ ]:

```
# Ejemplo de error ya que asignamos dos veces un valor al parámetro saludo (por posición y por
saludar("Juan", saludo="Hola")
```

## Parámetros opcionales y valores por defecto

- Podemos omitir el pasaje de algún argumento a una función. Como si el parámetro fuese "opcional"; sin embargo, en **Python** esto no es así.
- Se puede omitir el pasaje de un argumento, siempre y cuando, en la definición de la función, el parámetro correspondiente tenga un valor por defecto o default.

In [ ]:

```
# Def. saludar2 con 2 parámetros formales: nombre y saludo con default
def saludar2(nombre, saludo="Ciao"):
    print("Digo", saludo, "a", nombre)

saludar2("Giovanni")
saludar2("Juan")

saludar2("Juan", "Hola")

saludar2() # Error. Los parámetros sin defaults, son requeridos.
```

## Parámetros opcionales y valores por defecto

- Por correctitud y legibilidad, los parámetros con defaults, van al final.
- **Python** arroja error si esto no se respeta.

In [21]:

```
# Def. saludar2 con 2 parámetros formales: nombre y saludo con valor defecto
def ecurecta(a=1, c=0, x): # Error. Los p.requeridos después de defaults.
    print("A:",a,"C:",c,"X:",x)
    return a*x + c
print(ecurecta(3)) # SyntaxError: non-default argument follows default argument
```

Cell In[21], line 2

```
def ecurecta(a=1, c=0, x): # Error. Los p.requeridos después de defaults.
```

**SyntaxError:** non-default argument follows default argument

- Siempre los parámetros con defaults al final de la secuencia de parámetros.

In [20]:

```
def ecurecta(x, a=1, c=0): # ok.
    print("A:",a,"C:",c,"X:",x)
    return a*x + c
print(ecurecta(3)) # A: 1 C: 0 X: 3
```

```
A: 1 C: 0 X: 3
3
```

## Cantidad arbitraria de argumentos

- Hay situaciones en donde necesitamos definir funciones que reciban una cantidad arbitraria de argumentos.
- En **Python** tenemos algunas funciones built-in que permiten esto. Por ejemplo `print(...)`.

In [ ]:

```
print("Uno")
print("Uno", "Dos")
print("Uno", "Dos", "Tres")
print("Uno", "Dos", "Tres", "Cuatro", "Cinco")
print(1, .2e2, "Tres", ["Cua", "tro"], 5+5j)
```

## Cantidad arbitraria de argumentos (2)

- Para que una función pueda recibir un número arbitrario de argumentos, debemos definir parámetro con un operador `*` de unpacking.
- En **Python** cuando marquemos un parámetro con `*` significa que todos los argumentos que no podemos asociar con ningún parámetro, serán agregados como una **tupla** al parámetro unpacking.

In [30]:

```
def producto(a,b,*args): # obligatorios a y b; y un p.unpacking args
    res = a * b
    print(a, "*", b, "=", res)
    for val in args:
        print(res, "*", val, "=", res*val)
        res *= val
    return res

print("producto(2,3):")
print(producto(2,3))
print("producto(2,3,4,5):")
print(producto(2,3,4,5))
print("producto(2,.3e2,4.5,5.5):")
print(producto(2,.3e2,4.5,5.5))
```

```
producto(2,3):
2 * 3 = 6
6
producto(2,3,4,5):
2 * 3 = 6
6 * 4 = 24
24 * 5 = 120
120
producto(2,.3e2,4.5,5.5):
2 * 30.0 = 60.0
60.0 * 4.5 = 270.0
270.0 * 5.5 = 1485.0
1485.0
```

## Cantidad arbitrarios de argumentos por nombre

- Cuando usamos `*args` en la función se trata el parámetro formal como una **tupla**. No hay forma de identificar a cada elemento de esa **tupla**, salvo sea por la posición. Por ejemplo: `args[1]`, `args[5]`.
- Hay ocasiones que necesitamos pasar un numero arbitrario de argumentos que se puedan identificar con un nombre o clave.
- En estos casos, **Python** permite que pasemos a tipo especial de parámetro, al igual que la tupla `*args`, pero en este caso será una secuencia de clave -> valor, o **diccionario**.
- Usaremos el parámetro `**kwargs`. Pero para que se trate como un diccionario utilizaremos el prefijo `**` que es el operador de unpacking para diccionarios.
- Por ejemplo, definamos una función para imprimir un informe de notas y el promedio. Pasemos un nombre, un curso y distintas notas (asignatura→calificación).

In [31]:

```
def infoasig(nombre, curso, **kwargs):
    print("Informe de", nombre, "Curso:", curso)

    cant, prom = 0, 0
    for key in kwargs: # en diccionarios se itera por clave
        print("Materia",key,"Nota:",kwargs[key])
        prom += kwargs[key]
        cant += 1

    print("Promedio:",prom/cant)
infoasig("Giovanni", "1ro A",matemática=10,lengua=6,ingles=10,física=7,química=8)
```

```
Informe de Giovanni Curso: 1ro A
Materia matemática Nota: 10
Materia lengua Nota: 6
Materia ingles Nota: 10
Materia física Nota: 7
Materia química Nota: 8
Promedio: 8.2
```

## Cantidad arbitrarios de argumentos por nombre (2)

- El nombre `kwargs` es por convención (al igual que `args`). Pero para que se trate como un diccionario utilizaremos el prefijo `**` que es el operador de unpacking para diccionarios.
- En **Python**, los diccionarios se utilizan para almacenar valores de datos en pares **clave:valor**.
- Un diccionario es una **colección ordenada** (desde Python 3.7), **mutable** (o modificable) y que **no admite duplicados**.
- Si iteramos un diccionario con `for`, el elemento iterado es la **clave**.

In [ ]:

```
def funcdumb(**kwargs):  
    print(kwargs)  
    print(type(kwargs))  
    print(dir(kwargs))  
    for key in kwargs:  
        print("key:",key,"val:",kwargs[key],"tipo:",type(kwargs[key]))  
  
funcdumb(nombre="Tomas",edad=33,pref=('Azul','Ford','SUV'),presup=12000)
```



## Cantidad arbitrarios de argumentos por nombre (3)

- Se utiliza `*args` para pasar un número arbitrario de argumentos posicionales.
- Se utiliza `**kwargs` para pasar un número arbitrario de argumentos por nombre.
- Si usamos `*args` o `**kwargs` **NO** significa que vamos a recibir una tupla o diccionarios (u similar) como argumentos. Sino que hay varios argumentos que se capturan por estos contenedores.
- Por ejemplo, si lo tratamos con:

In [ ]:

```
def infoasig(nombre, curso, **kwargs): # Contenedor de params por nombre
    pass

def infoasig2(nombre, curso, kwargs): # Tratado como param posicional
    pass
```

## Secuencia y orden de parámetros

- **Python** permite trabajar con parámetros posicionales, parámetros posicionales no definidos, parámetros con valor por defecto (o por nombre) y parámetros con valor por defecto no definidos.
- Es decir que la signatura de una función en **Python** puede esquematizarse así:

In [ ]:

```
def nombre_funcion(parapos1,  
                  parapos2,  
                  *args,  
                  paranom1="val1",  
                  paranom2="val2",  
                  **kwargs):  
  
    pass
```

## Bibliografía

- Óscar Ramírez Jiménez: *"Python a fondo"* 1era Edición. Editorial Marcombo S.L.. 2021.
- Allen Downey. *"Think Python"*. 2da Edición. Editorial Green Tea Press. 2015.
- Bill Lubanovic. *"Introducing Python"*. 2da Edición. O' Reilly. 2020.
- Eirc Matthes: *"Python Crash Course"*. 1era Edición. Editorial No Starch Press. 2016.
- Zed A. Shaw: *"Learn Python 3 the Hard Way"*. 1era Edición. Editorial Addison-Wesley. 2017.
- Web - John Sturtz: *Python "for" Loops (Definite Iteration)*. **Enlace**.