Tema 10:

Ficheros, programación estructurada y modular (8h)

10.1. Lectura y escritura de ficheros de texto

10.2. Gestión de ficheros CSV

10.3. Gestión de ficheros JSON

10.4. Programación estructurada y modular. Introducción a las funciones

10.5. Conceptos avanzados sobre funciones

10.6. Tipos especiales de funciones

10.7. Ejemplo de programación estructurada y modular

Profesor: Salvador Martínez Bolinches

Centro: IES Font de Sant Lluís

Año: 2025

10.1. Lectura y escritura de ficheros de texto

Función open()

La función open() se utiliza normalmente con dos parámetros (fichero con el que vamos a trabajar y modo de acceso) y nos devuelve un objeto de tipo fichero.

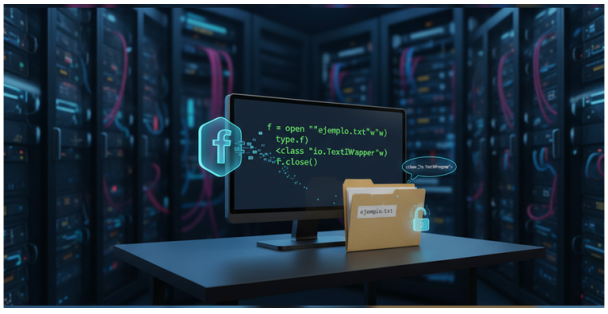
```
>>> f = open("ejemplo.txt","w")
>>> type(f)
<class '_io.TextIOWrapper'>
>>> f.close()
```

Modos de acceso

Los modos que podemos indicar son los siguientes:

Modo	Comportamiento	Puntero
r	Solo lectura	Al inicio del archivo
rb	Solo lectura en modo binario	
r+	Lectura y escritura	Al inicio del archivo
rb+	Lectura y escritura binario	Al inicio del archivo
w	Solo escritura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb	Solo escritura en modo binario. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
w+	Escritura y lectura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb+	Escritura y lectura binaria. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
a	Añadido (agregar contenido). Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
ab	Añadido en modo binario. Crea si éste no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
a+	Añadido y lectura. Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
ab+	Añadido y lectura en binario. Crea el archivo si no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.

Como podemos comprobar, podemos trabajar con ficheros binarios y con ficheros de texto.



Codificación de caracteres

Si trabajamos con fichero de texto, podemos indicar también el parámetro `encoding` que será la codificación de caracteres utilizadas al trabajar con el fichero, por defecto se usa la indicada en el sistema:

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

Y por último, también podemos indicar el parámetro `errors` que controla el comportamiento cuando se encuentra con algún error al codificar o decodificar caracteres.

```
# Abrir un archivo de texto en modo lectura con codificación UTF-8
# y manejo de errores que ignora los caracteres no decodificables.

fichero = open("datos.txt", mode="r", encoding="utf-8", errors="ignore")

# Leer el contenido del fichero
contenido = fichero.read()

# Mostrar el contenido
print(contenido)

# Cerrar el fichero al terminar
fichero.close()
```

Explicación de los parámetros:

- `"datos.txt"` → nombre del fichero a abrir.
- `mode="r"` → modo de acceso:
  - `"r"`: lectura
  - `"w"`: escritura (sobrescribe)
  - `"a"`: añadir al final
  - `"r+"`: lectura y escritura
- `encoding="utf-8"` → define la codificación del texto (UTF-8 es estándar y recomendable).
- `errors="ignore"` → ignora los errores de decodificación (otras opciones son `"strict"`, `"replace"`, `"backslashreplace"`, etc.).

Objeto fichero

Al abrir un fichero con un determinado modo de acceso con la función `open()` se nos devuelve un objeto fichero. El fichero abierto siempre hay que cerrarlo con el método `close()`:

```
>>> f = open("ejemplo.txt","w")
>>> type(f)
<class '_io.TextIOWrapper'>
>>> f.close()
```

Se pueden acceder a las siguientes propiedades del objeto file:

- `closed`: retorna `True` si el archivo se ha cerrado. De lo contrario, `False`.
- `mode`: retorna el modo de apertura.
- `name`: retorna el nombre del archivo
- `encoding`: retorna la codificación de caracteres de un archivo de texto

Podemos abrirlo y cerrarlo en la misma instrucción con la siguiente estructura:

```
>>> with open("ejemplo.txt", "r") as archivo:
...     contenido = archivo.read()
>>> archivo.closed
True
```

Métodos principales

Métodos de lectura

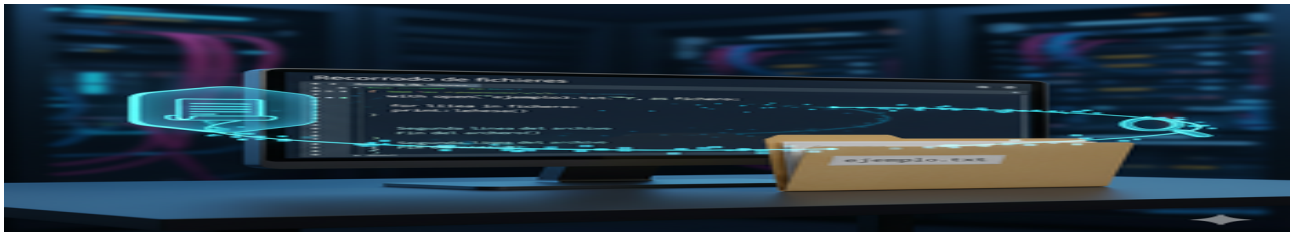
```
>>> f = open("ejemplo.txt","r")
>>> f.read()            # Devuelve una cadena con el contenido del fichero
'Hola que tal\n'

>>> f = open("ejemplo.txt","r")
>>> f.read(4)           # Devuelve una cadena con x (4) caracteres del fichero
'Hola'
>>> f.read(4)
' que'
>>> f.tell()            # Devuelve la posición del puntero del fichero
8
>>> f.seek(0)
>>> f.read()
'Hola que tal\n'

>>> f = open("ejemplo2.txt","r")
>>> f.readline()
'Línea 1\n'
>>> f.readline()        # Devuelve una línea del fichero y apunta a la siguiente
'Línea 2\n'
>>> f.seek(0)           # Posiciona el puntero del fichero (en 0)
0
>>> f.readlines()       # Devuelve una lista con las líneas del fichero
['Línea 1\n', 'Línea 2\n']
```

Métodos de escritura

```
>>> f = open("ejemplo3.txt","w")
>>> f.write("Prueba 1\n")     # Crea el archivo, lo escribe y da la pos del puntero
9
>>> print("Prueba 2\n",file=f)
>>> f.writelines(["Prueba 3","Prueba 4"])      # Escribe las líneas de la lista
>>> f.close()
>>> f = open("ejemplo3.txt","r")
>>> f.read()
'Prueba 1\nPrueba 2\n\nPrueba 3Prueba 4'
```

Recorrido de ficheros

```
>>> with open("ejemplo3.txt","r") as fichero:
...     for linea in fichero:
...         print(linea)
```



10.2. Gestión de ficheros CSV

Módulo CSV

El módulo csv nos permite trabajar con ficheros CSV.

Un fichero CSV (comma-separated values) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o por otro carácter).

```
1 4/5/2015 13:34,Apples,73
2 4/5/2015 3:41,Cherries,85
3 4/6/2015 12:46,Pears,14
4 4/8/2015 8:59,Oranges,52
5 4/10/2015 2:07,Apples,152
6 4/10/2015 18:10,Bananas,23
7 4/10/2015 2:40,Strawberries,98
```

Leer ficheros CSV

Para leer un fichero CSV utilizamos la función `reader()`:

```
>>> import csv
>>> fichero = open("ejemplo1.csv")
>>> contenido = csv.reader(fichero)
>>> list(contenido)
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
['4/10/2015 2:40', 'Strawberries', '98']]
>>> list(contenido)          # el puntero está situado al final del fichero
[]                           # con fichero.seek(0) moveríamos el puntero
>>> fichero.close()
```

Podemos guardar la lista obtenida en una variable y acceder a ella indicando fila y columna.

```
...
>>> datos = list(contenido)
>>> datos[0][0]
'4/5/2015 13:34'
>>> datos[1][1]
'Cherries'
>>> datos[2][2]
'14'
```

Por supuesto podemos recorrer el resultado:

```
...
>>> for row in contenido:
    print("Fila "+str(contenido.line_num)+" "+str(row))

Fila 1 ['4/5/2015 13:34', 'Apples', '73']
Fila 2 ['4/5/2015 3:41', 'Cherries', '85']
Fila 3 ['4/6/2015 12:46', 'Pears', '14']
Fila 4 ['4/8/2015 8:59', 'Oranges', '52']
Fila 5 ['4/10/2015 2:07', 'Apples', '152']
Fila 6 ['4/10/2015 18:10', 'Bananas', '23']
Fila 7 ['4/10/2015 2:40', 'Strawberries', '98']

# Si se lee varias veces, line_num se va acumulando, no se resetea
```

Veamos otro ejemplo un poco más complejo:

```
1 Año,Marca,Modelo,Descripción,Precio
2 1997,Ford,E350,"ac, abs, moon",3000.00
3 1999,Chevy,"Venture ""Extended Edition""","",4900.00
4 1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
5 1996,Jeep,Grand Cherokee,"MUST SELL!
6 air, moon roof, loaded",4799.00
```

```
>>> import csv
>>> fichero = open("ejemplo2.csv")
>>> contenido = csv.reader(fichero,quotechar='"')    # “ separa los datos
>>> for row in contenido:
...   print(row)
...
['Año', 'Marca', 'Modelo', 'Descripción', 'Precio']
['1997', 'Ford', 'E350', 'ac, abs, moon', '3000.00']
['1999', 'Chevy', 'Venture "Extended Edition"', '', '4900.00']
['1999', 'Chevy', 'Venture "Extended Edition, Very Large"', '', '5000.00']
['1996', 'Jeep', 'Grand Cherokee', 'MUST SELL!\nair, moon roof, loaded',
'4799.00']
```

Escribir ficheros CSV

Para escribir un fichero CSV utilizamos la función `writer()` y los métodos `writerow` o `writerows`:

```
>>> import csv
>>> fichero = open("ejemplo3.csv","w")
>>> contenido = csv.writer(fichero)
>>> contenido.writerow(['4/5/2015 13:34', 'Apples', '73'])
>>> contenido.writerows(['4/5/2015 3:41', 'Cherries', '85'],['4/6/2015 12:46',
'Pears', '14'])
>>> fichero.close()

$ cat ejemplo3.csv
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
```

```
1 4/5/2015 13:34,Apples,73
2 4/5/2015 3:41,Cherries,85
3
```

10.3. Gestión de ficheros JSON

El módulo json nos permite gestionar ficheros con formato JSON (JavaScript Object Notation).

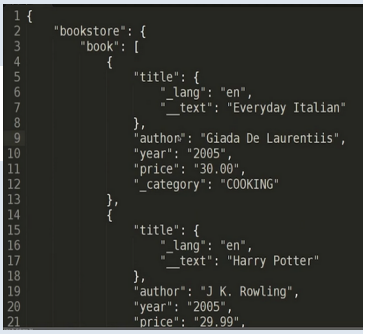
JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Leer ficheros json

Desde una cadena de caracteres, con el método `loads`:

```
>>> import json
>>> datos_json='{"nombre":"carlos","edad":23}'
>>> datos = json.loads(datos_json)
>>> type(datos)
<class 'dict'>
>>> print(datos)
{'nombre': 'carlos', 'edad': 23}
```

Desde un fichero, con el método `load`:




```
>>> with open("ejemplo1.json") as fichero:
...   datos=json.load(fichero)
>>> type(datos)
<class 'dict'>
>>> datos
{'bookstore': {'book': [{'_category': 'COOKING', 'price':
'30.00', 'author': 'Giada De Laurentiis', 'title': {'__text': 'Everyday
Italian', '_lang': 'en'}, 'year': '2005'}, {'_category': 'CHILDREN', 'price':
'29.99', 'author': 'J K. Rowling', 'title': {'__text': 'Harry Potter', '_lang':
'en'}, 'year': '2005'}, {'_category': 'WEB', 'price': '49.99', 'author': ['James
McGovern', 'Per Bothner', 'Kurt Cagle', 'James Linn', 'Vaidyanathan Nagarajan'],
'title': {'__text': 'XQuery Kick Start', '_lang': 'en'}, 'year': '2003'},
{'_category': 'WEB', 'price': '39.95', 'author': 'Erik T. Ray', 'title':
{'__text': 'Learning XML', '_lang': 'en'}, 'year': '2003'}]}}
```

```
datos["bookstore"]["book"][0]["price"]
'30.00'
```

Escribir ficheros json

```
>>> datos = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie','felineIQ': None}
>>> fichero = open("ejemplo2.json","w")
>>> json.dump(datos,fichero)
>>> fichero.close()

cat ejemplo2.json
{"miceCaught": 0, "name": "Zophie", "felineIQ": null, "isCat": true}
```

10.4. Programación estructurada y modular. Introducción a las funciones

Introducción a la programación estructurada y modular

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

La programación estructural y modular se lleva a cabo en python3 con la definición de funciones.

```python
#!/usr/bin/env python
def factorial(n):
    """Calcula el factorial de un número"""
    resultado = 1
    for i in range(1,n+1):
        resultado*=i
    return resultado

if __name__ == '__main__':
    print(factorial(6))
```

Definición de funciones

Veamos un ejemplo de definición de función:

```
>>> def factorial(n):
...     """Calcula el factorial de un número"""   # Documentación de la función
...     resultado = 1                              # Mostrada al ejecutar help(factorial)
...     for i in range(1,n+1):
...       resultado*=i
...     return resultado
```

Podemos obtener información de la función:

```
>>> help(factorial)
Help on function factorial in module __main__:
factorial(n)
    Calcula el factorial de un número
```

Y para utilizar la función:

```
>>> factorial(5)
120
```

`factorial` es un objeto de la clase `function`.

Ámbito de variables. Sentencia global

Una variable **local** se declara en su ámbito de uso (en el programa principal y dentro de una función) y una **global** fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
>>> def operar(a,b):
...     global suma
...     suma = a + b
...     resta = a - b
...     print(suma,resta)
...
>>> operar(4,5)
9 -1
>>> resta
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resta' is not defined
>>> suma
9
```

```
1 #!/usr/bin/env python
2 def operar(a,b):
3     global suma
4     suma = a + b
5     resta = a - b
6     print(suma,resta)
7
8 if __name__ == '__main__':
9     operar(4,5)
10    print(suma)
11    print(resta)
12
```

Podemos definir variables globales, que serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas:

```
>>> PI = 3.1415
>>> def area(radio):
...     return PI*radio**2
...
>>> area(2)
12.566
```

Parámetros formales y reales

- **Parámetros formales**: Son las variables que recibe la función, se crean al definir la función. Su contenido lo recibe al realizar la llamada a la función de los parámetro reales. Los parámetros formales son variables locales dentro de la función.
- **Parámetros reales**: Son la expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los parámetros formales.

Datos inmutables	Datos mutables

`def f(a):`
`  a=5`
 	
`a=1`
 	
`f(a)`
 	
`a`
`1`
	`def f(lista):`
	`  lista.append(5)`
	
	`lista=[1,2,3]`
	
	`f(lista)`
	
	`lista`
	`[1, 2, 3, 5]`

Paso de parámetro por valor o por referencia

En Python el paso de parámetros es siempre por referencia. El lenguaje no trabaja con el concepto de variables sino objetos y referencias. Al realizar la asignación a = 1 no se dice que “a contiene el valor 1” sino que “a referencia a 1”. Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

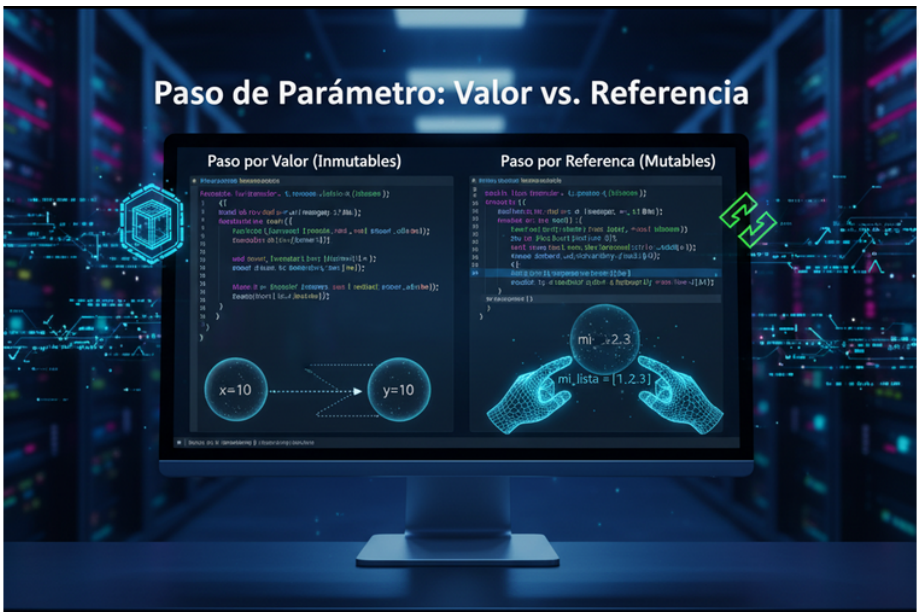
Evidentemente si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

Sin embargo si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, no es recomendable hacerlo en Python. En otros lenguajes es necesario porque no tenemos opción de devolver múltiples valores, pero como veremos en Python podemos devolver tuplas o lista con la instrucción `return`.



Llamadas a una función

Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar. La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función. Si la función no tiene una instrucción `return` el tipo de la llamada sera `None`.

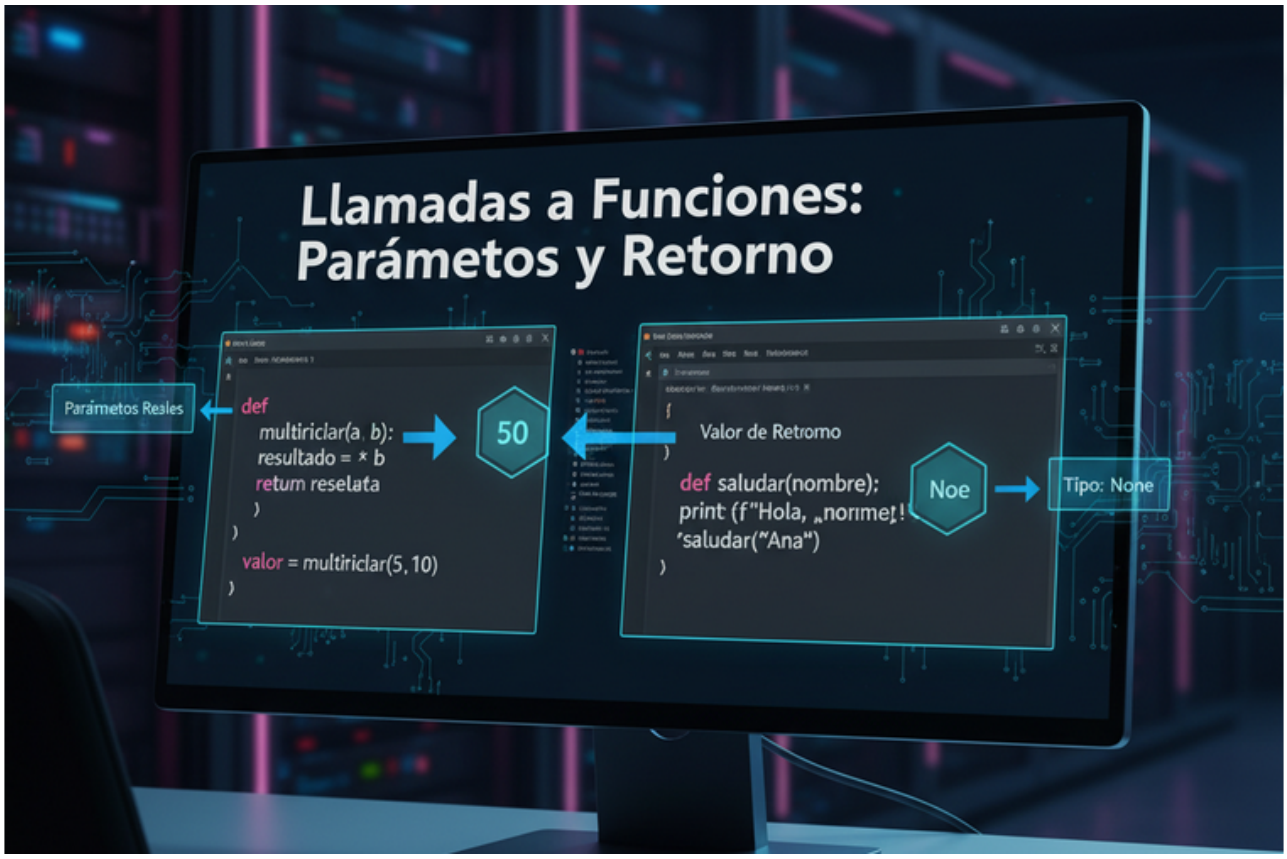
```
>>> def cuadrado(n):
...   return n*n

>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Cuando estamos definiendo una función estamos creando un objeto de tipo `function`.

```
>>> type(cuadrado)
<class 'function'>
```

Y por lo tanto puedo guardar el objeto función en otra variable:

```
>>> c=cuadrado
>>> c(4)
16
```



10.5. Conceptos avanzados sobre funciones

Tipos de argumentos: posicionales o keyword

Tenemos dos tipos de parámetros: los **posicionales** donde el parámetro real debe coincidir en posición con el parámetro formal:

```
>>> def sumar(n1,n2):
...   return n1+n2
...
>>> sumar(5,7)
12
>>> sumar(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() missing 1 required positional argument: 'n2'
```

Además podemos tener parámetros con valores por defecto:

```
>>> def operar(n1,n2,operador='+',respuesta='El resultado es '):
...   if operador=="+":
...     return respuesta+str(n1+n2)
...   elif operador=="-":
...     return respuesta+str(n1-n2)
...   else:
...     return "Error"
...
>>> operar(5,7)
'El resultado es 12'
>>> operar(5,7,"-")
'El resultado es -2'
>>> operar(5,7,"-","La resta es ")
'La resta es -2'
```

Los parámetros **keyword** son aquellos donde se indican el nombre del parámetro formal y su valor, por lo tanto no es necesario que tengan la misma posición. Al definir una función o al llamarla, hay que indicar primero los argumentos posicionales y a continuación los argumentos con valor por defecto (keyword).

```
>>> operar(5,7)    # dos parámetros posicionales
>>> operar(n1=4,n2=6)    # dos parámetros keyword
>>> operar(4,6,respuesta="La suma es")     # dos parámetros posicionales y uno keyword
>>>  operar(4,6,respuesta="La  resta  es",operador="-")       #  dos  parámetros posicionales y dos keyword
```

Parámetro *

Un parámetro * entre los parámetros formales de una función, nos obliga a indicar los parámetros reales posteriores como keyword:

```
>>> def sumar(n1,n2,*,op="+"):
...   if op=="+":
...     return n1+n2
...   elif op=="-":
...     return n1-n2
...   else:
...     return "error"
...
>>> sumar(2,3)
5
>>> sumar(2,3,"-")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() takes 2 positional arguments but 3 were given
>>> sumar(2,3,op="-")
-1
```

Argumentos arbitrarios (*args y **kwargs)

Para indicar un número indefinido de argumentos posicionales al definir una función, utilizamos el símbolo *:

```
>>> def sumar(n,*args):
...   resultado=n
...   for i in args:
...     resultado+=i
...   return resultado
...
>>> sumar(2)
2
>>> sumar(2,3,4)
9
```

Para indicar un número indefinido de argumentos keyword al definir una función, utilizamos el símbolo **:

```
>>> def saludar(nombre="pepe",**kwargs):
...   cadena=nombre
...   for valor in kwargs.values():
...    cadena=cadena+" "+valor
...   return "Hola "+cadena
...
>>> saludar()
'Hola pepe'
>>> saludar("juan")
'Hola juan'
>>> saludar(nombre="juan",nombre2="pepe")
'Hola juan pepe'
>>> saludar(nombre="juan",nombre2="pepe",nombre3="maria")
'Hola juan maria pepe'
```

Por lo tanto podríamos tener definiciones de funciones del tipo:

```
>>> def f()
>>> def f(a,b=1)
>>> def f(a,*args,b=1)
>>> def f(*args,b=1)
>>> def f(*args,b=1,*kwargs)
>>> def f(*args,*kwargs)
>>> def f(*args)
>>> def f(*kwargs)
```

Desempaquetar argumentos: pasar listas y diccionarios

El caso contrario es cuando tenemos que pasar parámetros que los tenemos guardados en una lista o en un diccionario.

Para pasar listas utilizamos el símbolo `*`:

```
>>> lista=[1,2,3]
>>> sumar(*lista)
6
>>> sumar(2,*lista)
8
>>> sumar(2,3,*lista)
11
```

Podemos tener parámetros keyword guardados en un diccionario, para enviar un diccionario utilizamos el símbolo `**`:

```
>>> datos={"nombre":"jose","nombre2":"pepe","nombre3":"maria"}
>>> saludar(**datos)
'Hola jose maria pepe'
```

Devolver múltiples resultados

La instrucción `return` puede devolver cualquier tipo de resultados, por lo tanto es fácil devolver múltiples datos guardados en una lista o en un diccionario. Veamos un ejemplo en que devolvemos los datos en una tupla:

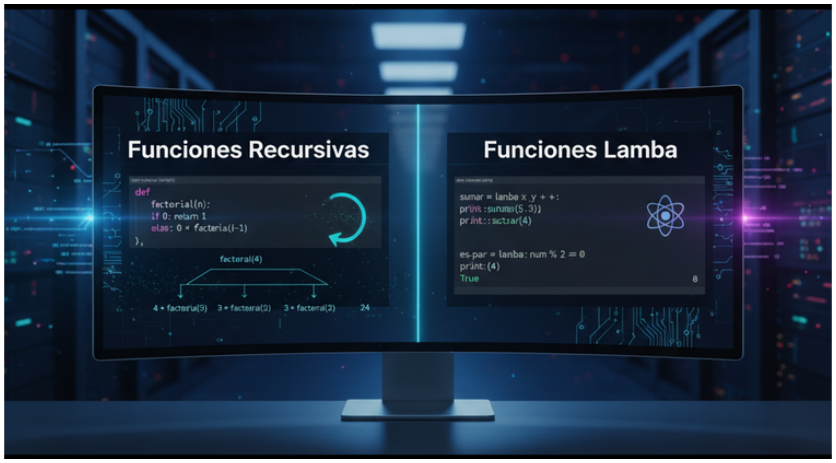
```
>>> def operar(n1,n2):
...   return (n1+n2,n1-n2,n1*n2)

>>> suma,resta,producto = operar(5,2)
>>> suma
7
>>> resta
3
>>> producto
10
```

10.6. Tipos especiales de funciones

Funciones recursivas

Una **función recursiva** es aquella que al ejecutarse hace llamadas a ella misma. Por lo tanto, debemos tener "un caso base" que hace terminar el bucle de llamadas. Veamos un ejemplo:

```
>>> def factorial(numero):
...     if(numero == 0 or numero == 1):
...         return 1
...     else:
...         return numero * factorial(numero-1)
...
>>> factorial(5)
120
```



Funciones lambda

Las **funciones lambda** nos sirven para crear pequeñas funciones anónimas, de una sola línea sobre la marcha.

```
>>> cuadrado = lambda x: x**2
>>> cuadrado(2)
```

Como podemos notar las funciones lambda no tienen nombre. Pero gracias a que lambda crea una referencia a un objeto función, la podemos llamar.

```
>>> lambda x: x**2
<function <lambda> at 0xb74469cc>
>>>
>>> (lambda x: x**2)(3)
9
```

Otro ejemplo:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Decoradores

Los decoradores son funciones que reciben como parámetros otras funciones y retornan como resultado otras funciones con el objetivo de alterar el funcionamiento original de la función que se pasa como parámetro. Hay funciones que tienen en común muchas funcionalidades, por ejemplo las de manejo de errores de conexión de recursos I/O (que se deben programar siempre que usemos estos recursos) o las de validación de permisos en las respuestas de peticiones de servidores, en vez de repetir el código de rutinas podemos abstraer, bien sea el manejo de error o la respuesta de peticiones, en una función decorador.

```
>>> def tablas(funcion):
...     def envoltura(tabla=1):
...         print('Tabla del %i:' %tabla)
...         print('-' * 15)
...         for numero in range(0, 11):
...             funcion(numero, tabla)
...         print('-' * 15)
...     return envoltura
...
>>> @tablas
... def suma(numero, tabla=1):
...     print('%2i + %2i = %3i' %(tabla, numero, tabla+numero))
...
>>> @tablas
... def multiplicar(numero, tabla=1):
...     print('%2i X %2i = %3i' %(tabla, numero, tabla*numero))
# Muestra la tabla de sumar del 1
suma()
# Muestra la tabla de sumar del 4
suma(4)
# Muestra la tabla de multiplicar del 1
multiplicar()
# Muestra la tabla de multiplicar del 10
multiplicar(10)
```

Funciones generadoras

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso.

```
>>> def par(inicio,fin):
...   for i in range(inicio,fin):
...     if i % 2==0:
...       yield i

>>> datos = par(1,5)
>>> next(datos)
2
>>> next(datos)
4

>>> for i in par(20,30):
...   print(i,end=" ")
20 22 24 26 28

>>> lista_pares = list(par(1,10))
>>> lista_pares
[2, 4, 6, 8]
```

10.7. Ejemplo de programación estructurada y modular

Partiendo del fichero csv `liga.csv` con los resultados de las jornadas de liga 2015-2016, realizar un programa que muestre la tabla de clasificación al final de la liga, en el que debe aparece el orden que ha quedado cada equipo, los partidos ganados, los empatados y perdidos, y por último los puntos conseguidos.

Para realizar este programa vamos a construir varias funciones:

- `LeerPartidos()`: Función que lee el fichero CSV y devuelve los datos del mismo en una lista de diccionarios.
- `impClasificacion(liga)`:Recibe la lista de diccionarios generado a parir de la función anterior, genera los datos de la clasificación y los imprime por pantalla.

Esta función utiliza interna las siguinetes funciones:

- `Equipos(datosliga)`: Función que recibe la lista de diccionarios con los datos de la liga y devuelve un conjunto con los equipos de la liga.
- `InfoEquipos(datosliga,equipos)`: Función que recibe la lista de diccionarios con los datos de la liga y el conjunto de equipos y devuelve una lista de tuplas, en cada tupla se guarda un equipo con los partidos ganados, empatados y perdidos y los puntos obtenidos.

  Esta función utiliza internamente:

  - `QuienGana(resultado)`: Función que recibe un resultado y devuelve un 0 si es un empate, un 1 si gana el equipo de casa y -1 si gana el equipo visitante.
  - `Puntos(info)`: Función que recibe una lista con los partidos ganados, empatados y perdidos y devuelve los puntos obtenidos.
- `Clasificacion(datos)`: Recibe la lista generada con la función anterior y la ordena según el número de puntos.

Fichero con la solución: clasificacion.py