Programación de Bases de Datos

Práctica 2A - Python

Introducción al lenguaje Python

Índice

1. Introducción

En esta sesión se hace una pequeña introducción al lenguaje Python para conocer los aspectos principales, teniendo en cuenta que ya se conocen los conceptos fundamentales de programación en otros lenguajes, tales como C/C++ o Java.

Se presentará el entorno de desarrollo propuesto, así como las principales cuestiones relacionadas con el desarrollo, como la inclusión de comentarios, especificación de variables, entrada/salida, la gestión de los tipos de datos típicos en Python como pueden ser las listas, diccionarios y conjuntos, las sentencias de bifurcación y las sentencias de bucles, finalizando con la implementación de funciones en Python.

El software requerido en esta sesión se resume en:

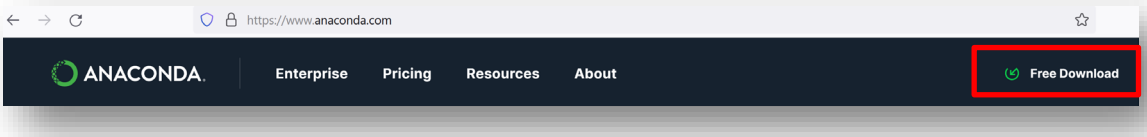
- Anaconda, que integra Jupyter Notebook
- Python

Como entregable (no de esta sesión, sino de sesiones posteriores) debe generarse una documentación donde se ilustre la instalación de todo el software, por lo que es recomendable ir haciendo capturas de pantalla y recopilando todas las imágenes y figuras mientras se realizan las instalaciones.

2. Preparación del entorno

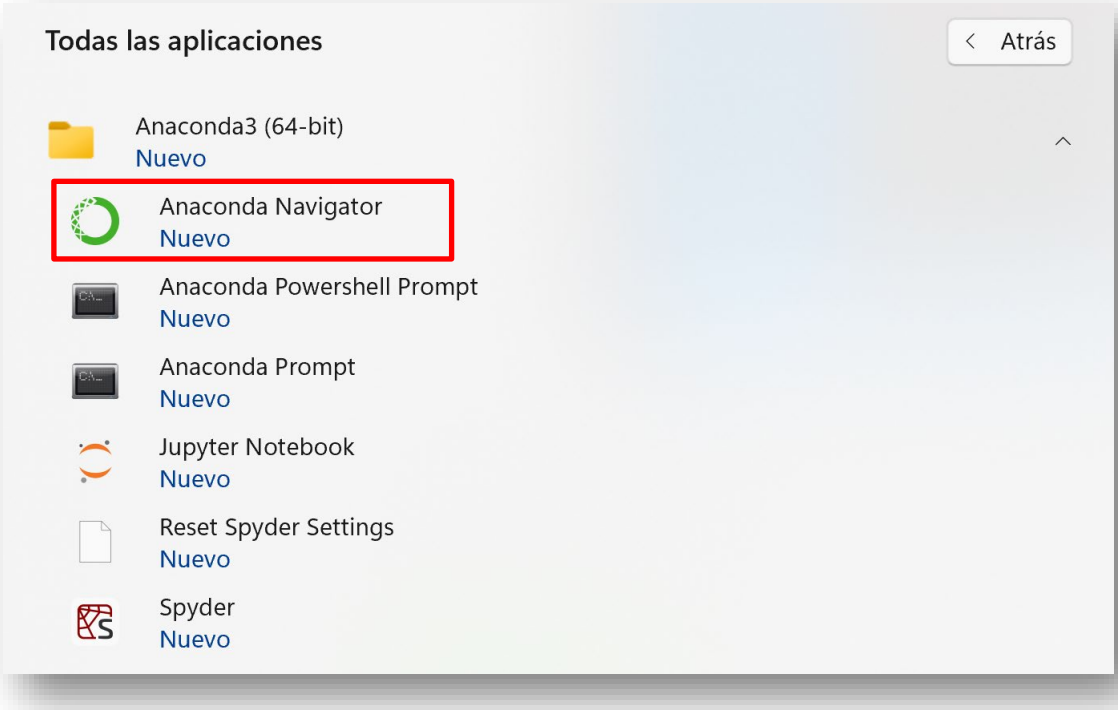
La suite Anaconda facilita la instalación y gestión de Jupyter Notebook, el entorno de programación que se usará para desarrollar los ejercicios en lenguaje Python. Esta suite es multiplataforma, y se puede instalar para Windows, Linux y Mac.

Para proceder a la instalación de Anaconda, es preciso ir a su web:
https://www.anaconda.com

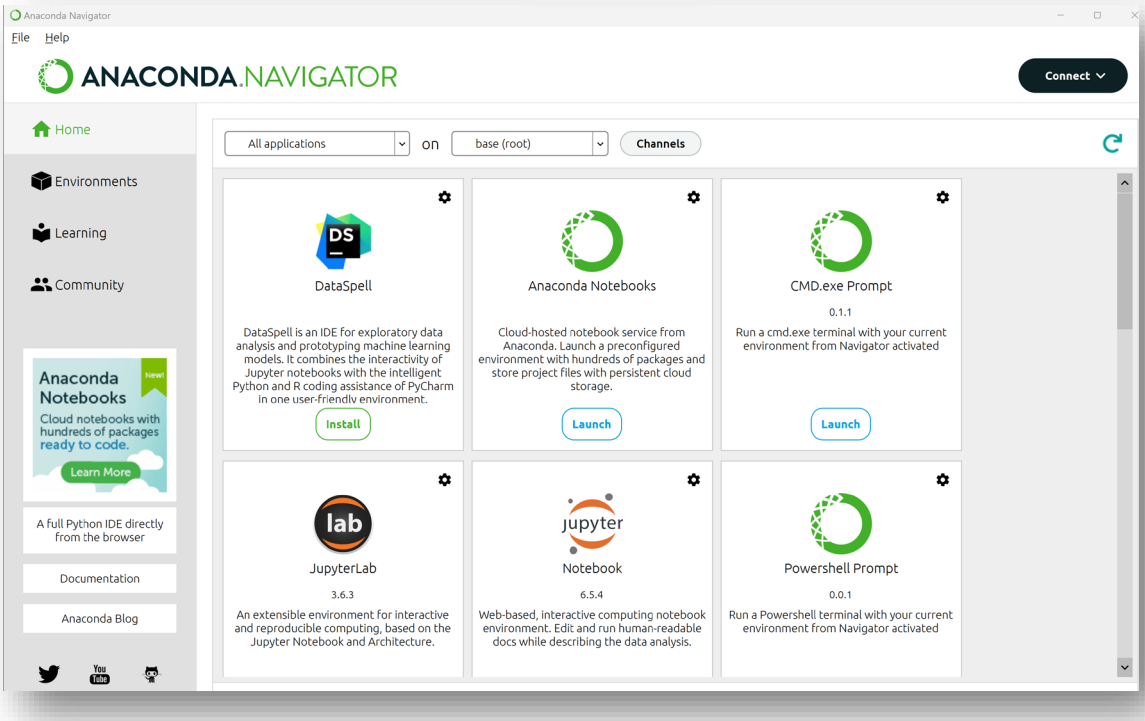


Y seleccionar "Free Download" y la versión adecuada:

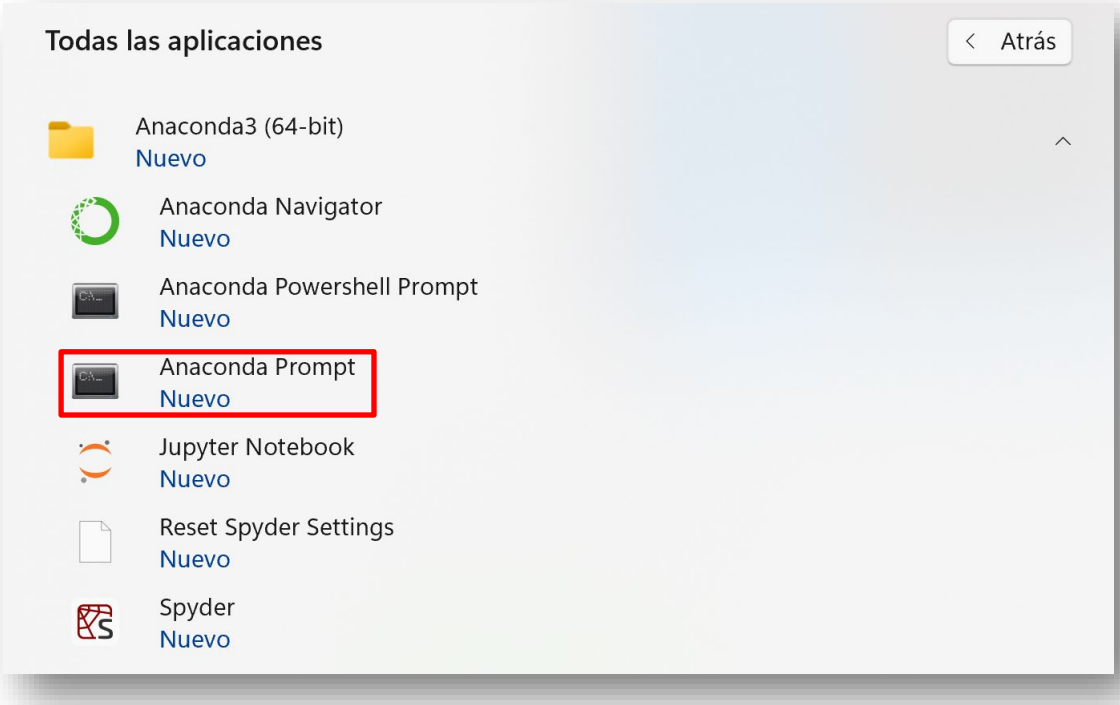
Siguiendo el asistente típico de instalación, se finalizará todo el proceso. Para iniciar Anaconda, bastará con seleccionar:



Así se accede a la Suite Anaconda:



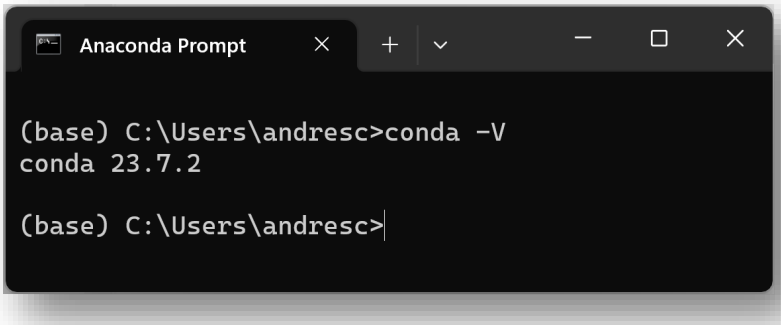
Para comprobar la versión instalada de Anaconda y de Python, se debe abrir la consola de Anaconda:



Tras abrir la consola de Anaconda, debe teclearse:

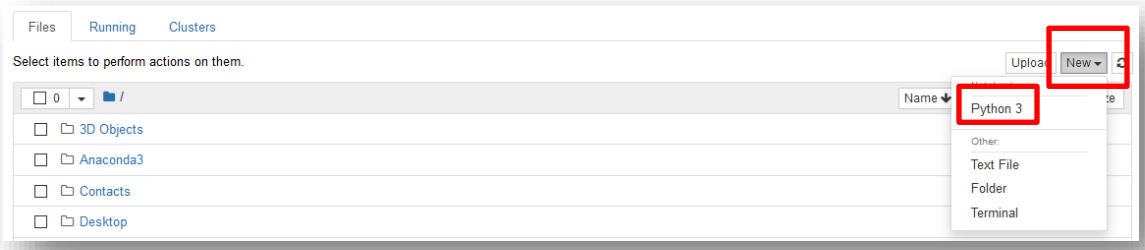
```
conda -V
python -V
```

Se obtiene la versión instalada de Anaconda y de Python:



Finalmente, para lanzar la ejecución del entorno de programación en Python, dentro de la Suite de Anaconda, debe seleccionarse “Jupyter Notebook”, bien en ANACONDA.NAVIGATOR (iniciando Anaconda) o bien iniciando Jupyter Notebook en el menú de Windows:

Y, a continuación, seleccionar el botón “New” y “Python 3”:



3. Introducción al lenguaje Python

Python es un lenguaje interpretado, multiparadigma y multiplataforma. En Python, la indentación (sangrado, en castellano) tiene una gran importancia, pues delimita los bloques de código. De esta manera, no es necesario introducir los clásicos `begin`/`end`, `{`/`}` e identificadores de bloques similares. Para ello, es obligatorio que todas las sentencias que conforme un mismo bloque se encuentren con el mismo sangrado, ya que, de otro modo, se producirá un error.

En los sucesivos apartados se presentan los principales conceptos de programación, teniendo en cuenta que se presupone que, para entender y asimilar estos conceptos, se conocen otros lenguajes de programación.

A continuación, se presentan los ejemplos que se ha visto en las clases de teoría.

3.1. Comentarios, variables y entrada/salida

Los **comentarios** se pueden especificar de dos formas:

- Anteponiendo el carácter `#` al comentario
- Escribiendo tres comillas dobles (`"""`) antes y después del comentario (que puede ocupar más de una línea).

```
# comentario
""" comentario en
varias líneas """
```

Para las **variables** en Python no es necesario especificar su tipo. La simple asignación de un valor condicionará el tipo de la variable, de modo que una misma variable puede reutilizarse para varias finalidades y con varios tipos diferentes, como puede verse en el siguiente fragmento de código, donde la variable `i` asume un valor numérico entero, una cadena, un numérico real y, finalmente, un valor booelano.

La **salida** en pantalla se realiza con `print()`, como también puede apreciarse en este fragmento de código:

```
# -------------------------
# variables, entrada/salida
# -------------------------

# variables no tipadas
i = 24
print(i)
i = "Hola"
print(i)
i = 3.14
print(i)
i = False
print(i)

print ("Cadenas")
print (28) #números
```

Al ejecutar, se generará la siguiente salida:

```
24
Hola
3.14
False
Cadenas
28
```

Aunque se ha comentado que las variables en Python no necesitan tipo en su declaración, si el programador necesita declarar una variable e inicializarla con un valor concreto puede hacerlo como se muestra a continuación, donde la variable `num1` se declara de tipo entero y se inicializa a 12, y la variable `num2` se declara de tipo real y se inicializa a 2.5.

El fragmento también incluye la declaración de cadenas, que pueden especificarse indistintamente entre comillas dobles (`"`) o entre comillas simples (`'`). La concatenación de cadenas puede realizarse con +, aunque es obligatorio que lo que se concatene con el operador "+" sean de tipo cadena (por eso se usa la función `str()` para convertir el entero `num1` a cadena).

```
# declaración con tipo
num1=int(12)
num2=float(2.5)

# cadenas
txt1="Hola" #comillas dobles
txt2='Buenos días' #comillas simples
txt3=str(8)

print(num1)
print(num2)
print(txt1)
print(txt2)
print(txt3)

print(txt1+", "+txt2)
print(txt1+", "+txt2+txt3)

print(txt1+", "+txt2+str(num1)) #ojo no funciona sin str
#print(txt1+", "+txt2+num1) #ojo no funciona sin str
```

El fragmento de código anterior generará la siguiente salida:

```
12
2.5
Hola
Buenos días
8
Hola, Buenos días
Hola, Buenos días8
Hola, Buenos días12
```

Para complementar la **entrada/salida**, en Python además de **`print()`** se utiliza `input()`. En el siguiente ejemplo se ilustra también el funcionamiento de algunas funciones típicas de cadenas, como es la longitud mediante la función `len()`, o el hecho de que las cadenas comienzan en la posición 0 del vector, al igual que en lenguajes como C/C++.

Además, se ilustra que la función **`print`** permite separar argumentos mediante una "`,`", en lugar de tener que concatenar cadenas con el operador "+" como se mostró anteriormente.

```
nombre = input("Escribe tu nombre: ")
print(nombre)
print("Tu nombre tiene",len(nombre),"letras")
print("Tu nombre empieza por",nombre[0],"y termina por", nombre[len(nombre) -1])
```

El código anterior generará la siguiente salida:

```
Escribe tu nombre: Andrés
Andrés
Tu nombre tiene 6 letras
Tu nombre empieza por A y termina por s
```

El código completo para estos ejemplos se incluye en el archivo **python_01_variables_es.py**:

```
# comentario
""" comentario en
varias líneas """

# --------------------------
# variables, entrada/salida
# --------------------------

# variables no tipadas
i = 24
print(i)
i = "Hola"
print(i)
i = 3.14
print(i)
i = False
print(i)

print ("Cadenas")
print (28) #números

# declaración con tipo
num1=int(12)
num2=float(2.5)
```

```
# cadenas
txt1="Hola" #comillas dobles
txt2='Buenos días' #comillas simples
txt3=str(8)

print(num1)
print(num2)
print(txt1)
print(txt2)
print(txt3)

print(txt1+", "+txt2)
print(txt1+", "+txt2+txt3)

print(txt1+", "+txt2+str(num1)) #ojo no funciona sin str
#print(txt1+", "+txt2+num1) #ojo no funciona sin str

nombre = input("Escribe tu nombre: ")
print(nombre)
print("Tu nombre tiene",len(nombre),"letras")
print("Tu nombre empieza por",nombre[0],"y termina por",nombre[len(nombre)-1])
```

Y la salida de este script se muestra a continuación:

```
24
Hola
3.14
False
Cadenas
28
12
2.5
Hola
Buenos días
8
Hola, Buenos días
Hola, Buenos días8
Hola, Buenos días12
Escribe tu nombre: Andrés
Andrés
Tu nombre tiene 6 letras
Tu nombre empieza por A y termina por s
```

3.2. Tipos de datos: listas, diccionarios y conjuntos

Hay varios tipos de datos en Python, aunque se presentan tan solo tres: listas, diccionarios y conjuntos.

3.2.1. Listas

Las **listas** son estructuras de datos en las que el orden de los elementos determina su acceso. Se puede acceder a estos elementos mediante un índice que determina su posición, especificando esta posición entre corchetes [ ].

El siguiente fragmento de código genera dos listas de clientes, correspondientes a 2019 y 2020:

```
# -------------------------
# listas
# -------------------------

Clientes2020=["Juan","Luis","María","Carmen"]
Clientes2019=["Pedro","Adela"]
print(Clientes2020)
print(Clientes2019)
```

La impresión de estas dos listas generará la siguiente salida:

```
['Juan', 'Luis', 'María', 'Carmen']
['Pedro', 'Adela']
```

Las operaciones típicas con listas en Python son las siguientes:

- `lista[i]`: Devuelve el elemento que está en la posición i de la lista
- `lista.pop(i)`: Devuelve el elemento en la posición i de una lista y luego lo borra
- `lista.append(elemento)`: Añade elemento al final de la lista
- `lista.insert(i, elemento)`: Inserta elemento en la posición i
- `lista.extend(lista2)`: Fusiona lista con lista2
- `lista.remove(elemento)`: Elimina la primera vez que aparece el elemento

El siguiente fragmento imprime el segundo elemento de la lista (el especificado por el índice 1), y luego vuelve a imprimir este segundo elemento de la lista y, a la vez, lo saca de ella, mediante la opción `pop()`:

```
print(Clientes2020[1]) #El primer elemento es el 0
print(Clientes2020.pop(1)) # sacar al elemento 1
```

Generará la siguiente salida:

```
Luis
Luis
```

El siguiente fragmento de código añade un nuevo elemento a la lista, que se incluirá al final de la misma:

```
Clientes2020.append("Pablo") # añadir al final
print(Clientes2020)
```

Obteniéndose la siguiente salida:

```
['Juan', 'María', 'Carmen', 'Pablo']
```

De igual modo, es posible insertar un elemento en cualquier posición de la lista, por ejemplo en la posición especificada por el índice 1:

```
Clientes2020.insert(1,"Laura") # insertar en la posición 1
print(Clientes2020)
```

Apreciándose la siguiente salida:

```
['Juan', 'Laura', 'María', 'Carmen', 'Pablo']
```

También es posible aunar dos listas en una sola, mediante `extend()`:

```
Clientes2020.extend(Clientes2019) # juntar los clientes de 2019 y 2020
print(Clientes2020)
```

De modo que, ahora, la lista `Clientes2020` quedará como se muestra:

```
['Juan', 'Laura', 'María', 'Carmen', 'Pablo', 'Pedro', 'Adela']
```

Para eliminar un elemento de la lista (que efectivamente exista) se usa `remove()`:

```
Clientes2020.remove("Pablo") # eliminar elemento
print(Clientes2020)
```

De modo que Pablo se habrá eliminado de la lista de `Clientes2020`:

```
['Juan', 'Laura', 'María', 'Carmen', 'Pedro', 'Adela']
```

Como último ejemplo con listas, mostrar también que la función `len()` en este caso devuelve el número de elementos de una lista:

```
# Número de elementos
print("El número de clientes es",len(Clientes2020))
```

Imprimiendo el número de clientes especificado según la longitud de la lista:

```
El número de clientes es 6
```

El script completo se incluye en el archivo **python_02_listas.py**:

```
# --------------------------
# listas
# --------------------------

Clientes2020=["Juan","Luis","María","Carmen"]
Clientes2019=["Pedro","Adela"]
print(Clientes2020)
print(Clientes2019)

""" Operaciones con listas
    lista[i]: Devuelve el elemento que está en la posición i de la lista
    lista.pop(i): Devuelve el elemento en la posición i de una lista y luego lo borra
    lista.append(elemento): Añade elemento al final de la lista
    lista.insert(i, elemento): Inserta elemento en la posición i
    lista.extend(lista2): Fusiona lista con lista2
    lista.remove(elemento): Elimina la primera vez que aparece elemento
"""

print(Clientes2020[1]) #El primer elemento es el 0
print(Clientes2020.pop(1)) # sacar al elemento 1
Clientes2020.append("Pablo") # añadir al final
print(Clientes2020)
Clientes2020.insert(1,"Laura") # insertar en la posición 1
print(Clientes2020)
```

```
Clientes2020.extend(Clientes2019) # juntar los clientes de 2019 y 2020
print(Clientes2020)

Clientes2020.remove("Pablo") # eliminar elemento
print(Clientes2020)

# Número de elementos
print("El número de clientes es",len(Clientes2020))
```

La salida por consola se muestra a continuación:

```
['Juan', 'Luis', 'María', 'Carmen']
['Pedro', 'Adela']
Luis
Luis
['Juan', 'María', 'Carmen', 'Pablo']
['Juan', 'Laura', 'María', 'Carmen', 'Pablo']
['Juan', 'Laura', 'María', 'Carmen', 'Pablo', 'Pedro', 'Adela']
['Juan', 'Laura', 'María', 'Carmen', 'Pedro', 'Adela']
El número de clientes es 6
```

3.2.2. Diccionarios

En cuanto a estructuras de datos de tipo **diccionario**, recordar que son estructuras que no siguen ningún orden, sino que vienen asociados con parejas `clave:valor`.

El siguiente ejemplo ilustra un diccionario de personas, que se identifican por una clave y, como valor, el nombre de la persona.

```
# --------------------------
# diccionarios
# --------------------------

dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}
print(dicc)
```

Al imprimir el diccionario `dicc` se obtienen todas las parejas `clave:valor`:

```
{'abc': 'Antonio', 'qwe': 'Javier', 'asd': 'Alba', 'zxc': 'Esther'}
```

Como sucedía con las listas, las operaciones típicas con diccionarios en Python son las siguientes:

- `diccionario.get('clave')`: Devuelve el valor que corresponde con la clave
- `diccionario.pop('valor')`: Devuelve el valor que corresponde con la clave, y luego borra la clave/valor
- `diccionario.update({'clave':'valor'})`: Inserta una clave o actualiza su valor si ya existiera
- `'clave' in diccionario`: Devuelve True/False si la clave existe en el diccionario.
- `'valor' in diccionario.values()`: Devuelve True/False si valor existe en el diccionario

El siguiente fragmento de código obtiene el elemento (función `get()`) cuya clave es `asd` y lo imprime. A continuación, obtiene el elemento cuya clave es `asd` y lo elimina del diccionario (función `pop()`).

```
print(dicc.get("asd"))
print(dicc.pop("asd")) # eliminar elemento
print(dicc)
```

Estas acciones se aprecian al mostrar el diccionario, ya que Alba no aparece en el diccionario:

```
Alba
Alba
{'abc': 'Antonio', 'qwe': 'Javier', 'zxc': 'Esther'}
```

Como ejemplo de inserción mediante `update()`, se presenta el siguiente fragmento de código, donde la primera sentencia `update()` inserta al elemento Mario, al no existir previamente el elemento con clave `mmm` en el diccionario, mientras que el segundo `update()` sí que modifica, al existir ya el elemento con clave `qwe` en el diccionario:

```
dicc.update({"mmm":"Mario"}) #inserta, al no existir
dicc.update({"qwe":"Javi"}) #modifica, al existir ya
print(dicc)
```

Obteniéndose:

```
{'abc': 'Antonio', 'qwe': 'Javi', 'zxc': 'Esther', 'mmm': 'Mario'}
```

Por último, es posible comprobar si una clave existe en el diccionario, mediante **`in`**:

```
print("mmm" in dicc) # True
print("xxx" in dicc) # False
```

Mostrando la siguiente salida:

```
True
False
```

Para comprobar si un valor existe en el diccionario, además de **`in`** es preciso especificar que se trata de buscar en los valores del diccionario, mediante `values()`

```
print("Javi" in dicc.values()) # True
print("Fran" in dicc.values()) # False
```

Mostrando la siguiente salida:

```
True
False
```

El script completo se incluye en el archivo **python_03_diccionarios.py**:

```
# --------------------------
# diccionarios
# --------------------------

dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}
```

```
print(dicc)

""" Operaciones con diccionarios
    diccionario.get('clave'): Devuelve el valor que corresponde con la clave
    diccionario.pop('valor'): Devuelve el valor que corresponde con la clave, y luego borra
clave/valor
    diccionario.update({'clave':'valor'}): Inserta una clave o actualiza su valor si ya
existiera
    'clave' in diccionario: Devuelve True/False) si la clave existe en el diccionario.
    'valor' in diccionario.values(): Devuelve True/False si valor existe en el diccionario
"""


print(dicc.get("asd"))
print(dicc.pop("asd")) # eliminar elemento
print(dicc)

dicc.update({"mmm":"Mario"}) #inserta, al no existir
dicc.update({"qwe":"Javi"}) #modifica, al existir ya
print(dicc)

print("mmm" in dicc) # True
print("xxx" in dicc) # False

print("Javi" in dicc.values()) # True
print("Fran" in dicc.values()) # False
```

La salida por consola se muestra a continuación:

```
{'abc': 'Antonio', 'qwe': 'Javier', 'asd': 'Alba', 'zxc': 'Esther'}
Alba
Alba
{'abc': 'Antonio', 'qwe': 'Javier', 'zxc': 'Esther'}
{'abc': 'Antonio', 'qwe': 'Javi', 'zxc': 'Esther', 'mmm': 'Mario'}
True
False
True
False
```

3.2.3. Conjuntos

La última estructura de datos de Python que se presenta es el tipo **Conjunto**, que se define encerrando los elementos entre llaves `{ }`.

El siguiente fragmento de código define tres conjuntos diferentes, imprimiéndolos en pantalla.

```
# -------------------------
# conjuntos
# -------------------------

pares={2,4,6,8,10}
impares={1,3,5,7,9}
primos={1,2,3,5,7}

print(pares)
print(impares)
print(primos)
```

La salida en pantalla será la siguiente:

```
{2, 4, 6, 8, 10}
{1, 3, 5, 7, 9}
{1, 2, 3, 5, 7}
```

Como otros tipos de datos, los conjuntos definen una serie de operaciones básicas, como son las siguientes:

- A | B: Unión entre conjuntos
- A & B: Intersección entre conjuntos
- A – B: Diferencia entre conjuntos (los elementos que están en A pero no están en B)
- A ^ B: Diferencia simétrica entre el conjunto A y B (los elementos que están en A o en B pero no en los dos)

Estas operaciones se ilustran en el siguiente código:

```
print("Pares e impares",pares|impares)
print("Impares primos",impares&primos)
print("Impares no primos",impares-primos)
print("Diferencia simétrica",impares^primos)
```

Que provoca la siguiente salida:

```
Pares e impares {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Impares primos {1, 3, 5, 7}
Impares no primos {9}
Diferencia simétrica {2, 9}
```

El script completo se incluye en el archivo **python_04_conjuntos.py**:

```
# -------------------------
# conjuntos
# -------------------------

pares={2,4,6,8,10}
impares={1,3,5,7,9}
primos={1,2,3,5,7}

""" Operaciones con conjuntos
    A | B: Unión entre conjuntos
    A & B: Intersección entre conjuntos
    A – B: Diferencia entre conjuntos (los elementos que están en A pero no están en
B)
    A ^ B: Diferencia simétrica entre el conjunto A y B (los elementos que están en A
o en B pero no en los dos)
"""

print(pares)
print(impares)
print(primos)

print("Pares e impares",pares|impares)
print("Impares primos",impares&primos)
print("Impares no primos",impares-primos)
print("Diferencia simétrica",impares^primos)
```

La salida por consola se muestra a continuación:

```
{2, 4, 6, 8, 10}
{1, 3, 5, 7, 9}
{1, 2, 3, 5, 7}
Pares e impares {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Impares primos {1, 3, 5, 7}
Impares no primos {9}
Diferencia simétrica {2, 9}
```

3.3. Sentencias de bifurcación

En Python se dispone de la sentencia de bifurcación **`if`**, que puede ir con su correspondiente **`else`**, o con varias condiciones anidadas mediante **`elif`**, como se ilustra a continuación. Indicar que no existe sentencia de bifurcación de tipo **`switch`**/**`case`** como en otros lenguajes de programación, y, en caso de necesitarse, debe implementarse mediante sucesivos **`if`**.

Importante destacar los dos puntos **`:`** al final de cada condición.

```
# -------------------------
# bifurcación: if
# -------------------------

# Importancia de indentación para definir bloques (no hay {})
# No hay sentencia de bifurcación switch/case

pos=int(input("¿En qué posición terminaste la carrera? "))

if pos == 1:
    print("¡Enhorabuena!")
    print("¡Ganaste!")
elif pos == 2:
    print("¡Muy bien!")
    print("¡Quedar segundo es un buen puesto!")
else:
    print("Lo importante es participar")
    print("Sigue entrenando")
```

La salida sería la que se muestra a continuación. Suponiendo que se haya contestado con un 1 a la pregunta que se formula, mostraría el texto relacionado con esa condición en la sentencia **`if`**/**`elif`**/**`else`**.

```
¿En qué posición terminaste la carrera? 1
¡Enhorabuena!
¡Ganaste!
```

3.4. Sentencias de bucles

En Python se implementan dos tipos de bucles: **`while`** y **`for`**. El bucle **`repeat`** queda fuera, ya que se puede reproducir fácilmente mediante un **`while`**. Importante destacar los dos puntos (**`:`**) tras la condición que gobierna el bucle.

3.4.1. Bucle while

El uso del bucle **while** es muy sencillo, ya que únicamente se precisa la inicialización de la variable que gobierna el bucle antes de su comprobación, y su modificación en el cuerpo del bucle. El incremento de una variable puede hacerse de la forma `i=i+1`, y también de la forma `i+=1`, como se observa a continuación:

```
# -------------------------
# bucle: while
# -------------------------

i=1
while(i<11):
    print("8 X",i,"=",8*i)
    i=i+1 # también es posible i+=1
```

La salida sería la siguiente:

```
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
8 X 10 = 80
```

3.4.2. Bucle for

Para el bucle **for** hay que especificar el rango en el que varía la variable que lo gobierna, pudiendo ser un rango creciente o decreciente (en cuyo caso, hay que especificar que el incremento de la variable será de `-1`):

```
# -------------------------
# bucle: for
# -------------------------

for i in range(1,11):
    print("5 X",i,"=",5*i)

for i in range(10,0,-1):
    print("3 X",i,"=",3*i)
```

Los bucles anteriores muestran las tablas de multiplicar del 5 (en orden creciente) y del 3, en orden decreciente:

```
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
```

```
5 X 10 = 50
3 X 10 = 30
3 X 9 = 27
3 X 8 = 24
3 X 7 = 21
3 X 6 = 18
3 X 5 = 15
3 X 4 = 12
3 X 3 = 9
3 X 2 = 6
3 X 1 = 3
```

3.4.3. Recorrido de listas y diccionarios

El recorrido de **listas** y **diccionarios** se realiza mediante bucles. En el caso de las **listas**, la variable que gobierna el bucle y el uso de `in` permite acceder a cada elemento de la lista, como se aprecia en el primer ejemplo del siguiente fragmento de código. También es posible recorrer una lista siguiendo el modelo habitual de otros lenguajes de programación, estableciendo el rango entre la posición inicial (`0`) y la última de la lista (la determinada por la longitud de la lista, `len(Clientes2020)`), accediendo a la posición de la lista según el índice `i` encerrado entre corchetes `[]` (`Clientes2020[i]`), como ilustra el segundo bucle del ejemplo:

```
# Recorrer listas
Clientes2020=["Juan","Luis","María","Carmen"]

for i in Clientes2020:
    print(i)

for i in range(0,len(Clientes2020)):
    print("Cliente",i,Clientes2020[i])
```

El primer bucle `for` recorre la lista mostrando simplemente los elementos que forman parte de la lista, mientras que el segundo bucle imprime la cadena `"Cliente"` seguido del entero que gobierna el bucle y del elemento i-ésimo de la lista. Lo importante es comprobar que se accede al elemento i-ésimo de dos formas distintas:

```
Juan
Luis
María
Carmen
Cliente 0 Juan
Cliente 1 Luis
Cliente 2 María
Cliente 3 Carmen
```

En el caso del recorrido de **diccionarios**, es posible utilizar la clave para gestionar los recorridos de esta estructura de datos. Así, en el primer bucle del siguiente fragmento de código se aprecia que es posible recorrer el diccionario haciendo uso simplemente de un `in`, mientras que en el segundo, explícitamente se especifica que el recorrido se basa en centrar las búsquedas indiscutiblemente sobre las *claves*, por si hubiese alguna duda (`dicc.keys()`). El resultado será el mismo en ambos casos, lo que se desea destacar es que, si no se especifica nada, se entiende que las búsquedas en diccionarios se centran en las *claves*, y no en los *valores*, y que se obtienen las *claves* del diccionario.

```
#Recorrer diccionarios (claves)
```

```
dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}

for clave in dicc:
    print("Clave:",clave)

for clave in dicc.keys():
    print("Clave (con keys):",clave)
```

La salida generada será la siguiente:

```
Clave: abc
Clave: qwe
Clave: asd
Clave: zxc
Clave (con keys): abc
Clave (con keys): qwe
Clave (con keys): asd
Clave (con keys): zxc
```

Si se desean recuperar los *valores* del diccionario, se tienen dos posibilidades: bien se obtienen las *claves* y se recuperan los *valores* asociados a esas claves (`dicc[clave]`) como ilustra el primer bucle `for` del siguiente código, o bien se obtienen los valores directamente del diccionario, especificando ahora que se realicen las búsquedas sobre los valores (`dicc.values()`), como muestra el segundo bucle `for`.

```
#Recorrer diccionarios (valores)
for clave in dicc:
    print("Valor:",dicc[clave])

for valor in dicc.values():
    print("Valor (con values):",valor)
```

El resultado recorre el diccionario, imprimiendo los valores basándose en la clave (primer bucle) o en el propio valor (segundo bucle):

```
Valor: Antonio
Valor: Javier
Valor: Alba
Valor: Esther
Valor (con values): Antonio
Valor (con values): Javier
Valor (con values): Alba
Valor (con values): Esther
```

Como último ejemplo, se propone obtener la *clave* y el *valor* de un diccionario, bien centrándose en obtener las claves del diccionario y, con ellas, los valores (primer bucle), o bien obteniendo directamente claves y valores mediante `dicc.items()`:

```
#Recorrer diccionarios (claves y valores)
for clave in dicc:
    print("Clave:",clave,"- Valor:",dicc[clave])

for clave,valor in dicc.items():
    print("Clave (con items):",clave,"- Valor (con items):",valor)
```

Como se aprecia en la salida, de ambas formas es posible acceder a las *claves* y *valores* del diccionario.

```
Clave: abc - Valor: Antonio
Clave: qwe - Valor: Javier
Clave: asd - Valor: Alba
Clave: zxc - Valor: Esther
Clave (con items): abc - Valor (con items): Antonio
Clave (con items): qwe - Valor (con items): Javier
Clave (con items): asd - Valor (con items): Alba
Clave (con items): zxc - Valor (con items): Esther
```

El script completo se incluye en el archivo **python_05_if_bucles.py**:

```
# --------------------------
# bifurcación: if
# --------------------------

# Importancia de indentación para definir bloques (no hay {})
# No hay sentencia de bifurcación switch/case

pos=int(input("¿En qué posición terminaste la carrera? "))

if pos == 1:
    print("¡Enhorabuena!")
    print("¡Ganaste!")
elif pos == 2:
    print("¡Muy bien!")
    print("¡Quedar segundo es un buen puesto!")
else:
    print("Lo importante es participar")
    print("Sigue entrenando")

# --------------------------
# bucle: while
# --------------------------

i=1
while(i<11):
    print("8 X",i,"=",8*i)
    i=i+1 # también es posible i+=1

# no hay bucle "repite"

# --------------------------
# bucle: for
# --------------------------

for i in range(1,11):
    print("5 X",i,"=",5*i)

for i in range(10,0,-1):
    print("3 X",i,"=",3*i)

# Recorrer listas
Clientes2020=["Juan","Luis","María","Carmen"]

for i in Clientes2020:
    print(i)

for i in range(0,len(Clientes2020)):
```

```
    print("Cliente",i,Clientes2020[i])

#Recorrer diccionarios (claves)
dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}

for clave in dicc:
    print("Clave:",clave)

for clave in dicc.keys():
    print("Clave (con keys):",clave)

#Recorrer diccionarios (valores)
for clave in dicc:
    print("Valor:",dicc[clave])

for valor in dicc.values():
    print("Valor (con values):",valor)

#Recorrer diccionarios (claves y valores)
for clave in dicc:
    print("Clave:",clave,"- Valor:",dicc[clave])

for clave,valor in dicc.items():
    print("Clave (con items):",clave,"- Valor (con items):",valor)
```

La salida por consola se muestra a continuación:

```
¿En qué posición terminaste la carrera? 1
¡Enhorabuena!
¡Ganaste!
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
8 X 10 = 80
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
3 X 10 = 30
3 X 9 = 27
3 X 8 = 24
3 X 7 = 21
3 X 6 = 18
3 X 5 = 15
3 X 4 = 12
3 X 3 = 9
3 X 2 = 6
3 X 1 = 3
Juan
```

```
Luis
María
Carmen
Cliente 0 Juan
Cliente 1 Luis
Cliente 2 María
Cliente 3 Carmen
Clave: abc
Clave: qwe
Clave: asd
Clave: zxc
Clave (con keys): abc
Clave (con keys): qwe
Clave (con keys): asd
Clave (con keys): zxc
Valor: Antonio
Valor: Javier
Valor: Alba
Valor: Esther
Valor (con values): Antonio
Valor (con values): Javier
Valor (con values): Alba
Valor (con values): Esther
Clave: abc - Valor: Antonio
Clave: qwe - Valor: Javier
Clave: asd - Valor: Alba
Clave: zxc - Valor: Esther
Clave (con items): abc - Valor (con items): Antonio
Clave (con items): qwe - Valor (con items): Javier
Clave (con items): asd - Valor (con items): Alba
Clave (con items): zxc - Valor (con items): Esther
```

3.5. Funciones

Las funciones se especifican con **`def`**, continuando con el nombre de la función y, entre paréntesis, los argumentos. Importante también destacar los dos puntos (`:`) al finalizar la declaración de la cabecera de función. El cuerpo de la función lo formarán todo el bloque de sentencias especificado con el mismo sangrado.

La llamada a una función puede hacerse invocando a la función y asignando el valor de retorno a una función. En el siguiente fragmento de código se ilustra la definición de una función sin argumentos (`Saludo`), que no retorna nada, así como la forma en que se invoca, asignado a la variable `llamada` el valor de retorno (imprimirá `None`).

```
# -------------------------
# funciones
# -------------------------

# Funciones sin argumentos
def Saludo():
    print("Hola Mundo!")

llamada=Saludo()
print (llamada)
```

Como se ha comentado, al invocar a la función se imprime la cadena `"Hola Mundo!"`, y si se imprime la variable que ha provocado la llamada a esta función, como no retorna ningún valor, se imprime `None`.

```
Hola Mundo!
None
```

Cuando la función recibe argumentos, éstos se especifican entre paréntesis en la declaración de la función. Las dos funciones del siguiente fragmento de código reciben dos números y realizan la suma de ellos: la primera función imprime directamente la suma mediante la propia función, no retornando ningún valor (al imprimir la variable `resul` que invoca a la función, imprimirá de nuevo `None`). Sin embargo, la segunda función suma los dos valores y retorna la suma, por lo que la llamada a esta segunda función puede imprimirse directamente, ya que ahora sí recibe un valor concreto.

```
# Funciones con argumentos
def Suma1(x,y):
    z=x+y
    print(z)

resul=Suma1(5,3)
print(resul)

# Funciones que retornan un valor
def Suma2(x,y):
    z=x+y
    return(z)

print(Suma2(20,4))
```

Como se ha comentado, la primera función imprime directamente la suma de los dos valores que recibe, no retornando ningún valor, de modo que cuando se imprime la variable que invoca la llamada a la función se imprime `None`. La segunda función sí retorna la suma, por lo que se puede imprimir directamente la llamada a la función.

```
8
None
24
```

Como último ejemplo de funciones, se propone el paso de parámetros de tipo lista y diccionarios a las funciones. Siguiendo la filosofía de Python, no puede ser más sencillo: simplemente se especifica como parámetro la lista o el diccionario cuando se invoca la llamada. En la definición de la función, se gestionará adecuadamente qué hacer con la lista o el diccionario (en este caso, simplemente se ha optado por recorrer la lista o el diccionario, como se ha comentado con anterioridad).

```
# Enviando listas como parámetros
def Listar_Clientes(lista):
    for x in lista:
        print(x)
Clientes2020=["Juan","Luis","María","Carmen"]
Listar_Clientes(Clientes2020)

# Enviando diccionarios como parámetros
def Listar_dicc(dic):
    for clave in dic:
        print(clave,":",dic[clave])
dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}
Listar_dicc(dicc)
```

La ejecución de este código genera la siguiente salida, según la lista y diccionario generados:

```
Juan
Luis
María
Carmen
abc : Antonio
qwe : Javier
asd : Alba
zxc : Esther
```

El script completo se incluye en el archivo **python_06_funciones.py**:

```
# -------------------------
# funciones
# -------------------------

# Funciones sin argumentos
def Saludo():
    print("Hola Mundo!")

llamada=Saludo()
print (llamada)

# Funciones con argumentos
def Suma1(x,y):
    z=x+y
    print(z)

resul=Suma1(5,3)
print(resul)

# Funciones que retornan un valor
def Suma2(x,y):
    z=x+y
    return(z)

print(Suma2(20,4))

# Enviando listas como parámetros
def Listar_Clientes(lista):
    for x in lista:
        print(x)
Clientes2020=["Juan","Luis","María","Carmen"]
Listar_Clientes(Clientes2020)

# Enviando diccionarios como parámetros
def Listar_dicc(dic):
    for clave in dic:
        print(clave,":",dic[clave])
dicc = {"abc":"Antonio", "qwe":"Javier", "asd":"Alba", "zxc":"Esther"}
Listar_dicc(dicc)
```

La salida por consola se muestra a continuación:

```
Hola Mundo!
None
8
None
```

```
24
Juan
Luis
María
Carmen
abc : Antonio
qwe : Javier
asd : Alba
zxc : Esther
```

3.6. Excepciones

Unas últimas pinceladas sobre el uso de excepciones en Python, que siguen la misma filosofía de otros lenguajes de programación. Básicamente, se definen los bloques clásicos (**try**/**except**), lanzándose excepciones definidas por el sistema o definidas por el propio usuario. El archivo **python_07_excepciones.py** contiene el siguiente código:

```
# -------------------------
# excepciones
# -------------------------

def dividir(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        return resultado
    except ZeroDivisionError:
        raise ZeroDivisionError("El divisor no puede ser cero")
print(dividir(128,2))
print(dividir(128,4))
```

En este caso se invoca a la función con valores que no interrumpan la correcta ejecución del programa, generándose la salida:

```
64.0
32.0
```

4. Estudio a realizar

En esta sesión práctica únicamente se propone probar el código fuente y modificarlo para aprender y practicar con el lenguaje.

En la siguiente sesión será preciso documentar el proceso de instalación realizado en la sesión actual, por lo que se recomienda realizar las capturas necesarias y documentar brevemente el proceso de instalación, para tenerlo ya de cara a la siguiente sesión.