Introducción a la programación con Python

Billy M. Vanegas B.

Ingeniero Informático

billy@billyclasstime.com

https://billyclasstime.com

Hola!
Presentación de los alumnos

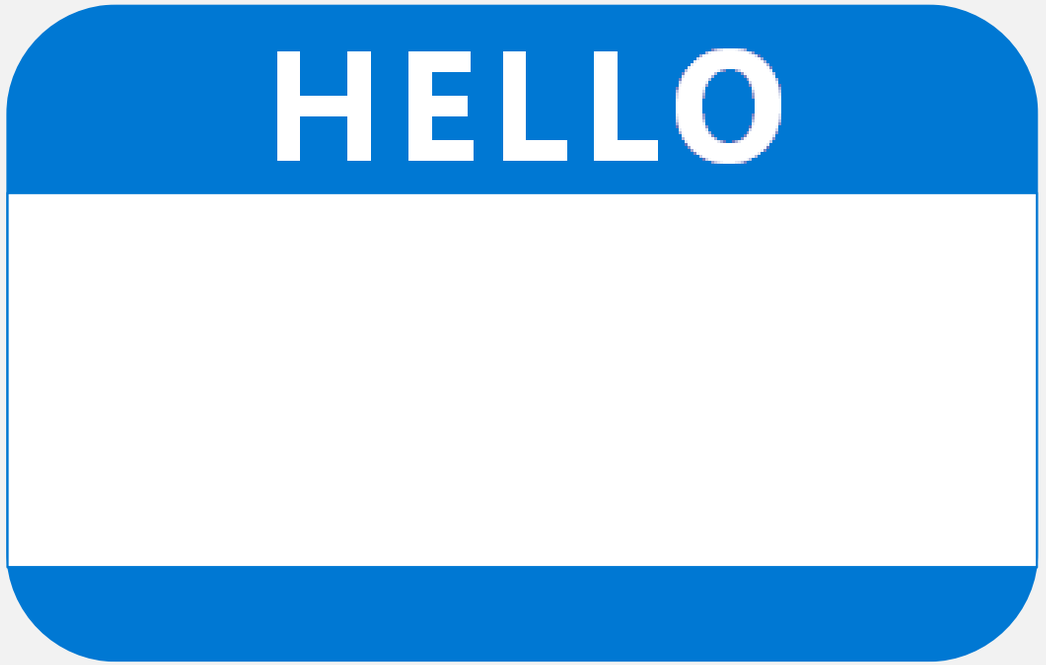
Vamos a familiaricémonos

Su nombre

Formación

Conocimientos sobre programación

Espectativas de este Modulo 2



Agenda

1 - Introducción

2– Entorno de trabajo

3 – Variables y sentencias de control

4 – Colecciones y JSON

5 – Funciones clases y módulos

6 – Trabajando con Ficheros

7 – Acceso a base de datos

8 – Consumiendo ApiRESTful

1 - Introducción

WIKIPEDIA
La enciclopedia libre

Artículo | Discusión | Leer | Editar | Ver historial

Python

Python es un lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código, se utiliza para desarrollar aplicaciones de todo tipo, ejemplos: Instagram, Netflix, Panda 3D, entre otros.[2] Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Administrado por la Python Software Foundation, posee una licencia de código abierto, denominada Python Software Foundation License.[3] Python se clasifica constantemente como uno de los lenguajes de programación más populares.

Python

Desarrollador(es)	Python Software Foundation *Sitio web oficial*
Información general	
Extensiones comunes	`.py` , `.pyc` , `.pyd` , `.pyo` , `.pyw` , `.pyz` , `.pyi`
Paradigma	Multiparadigma: orientado a objetos, imperativo, funcional, reflexivo
Apareció en	1991
Diseñado por	Guido van Rossum
Última versión estable	3.10.2[1] (14 de enero de 2022 (3 meses y 8 días))
Sistema de tipos	Fuertemente tipado, dinámico
Implementaciones	CPython, IronPython, Jython, Python for S60, PyPy, ActivePython, Unladen Swallow
Dialectos	Stackless Python, RPython
Influido por	ABC, ALGOL 68, C, Haskell, Icon, Lisp, Modula-3, Perl, Smalltalk, Java
Ha influido a	Boo, Cobra, D, Falcon, Genie, Groovy, Ruby, JavaScript, Cython, Go Latino
Sistema operativo	Multiplataforma
Licencia	Python Software Foundation License

[editar datos en Wikidata]

Filosofia de Python

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.

- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una —y preferiblemente solo una— manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que seas holandés
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

2 – Entorno de trabajo

Instalación de Python

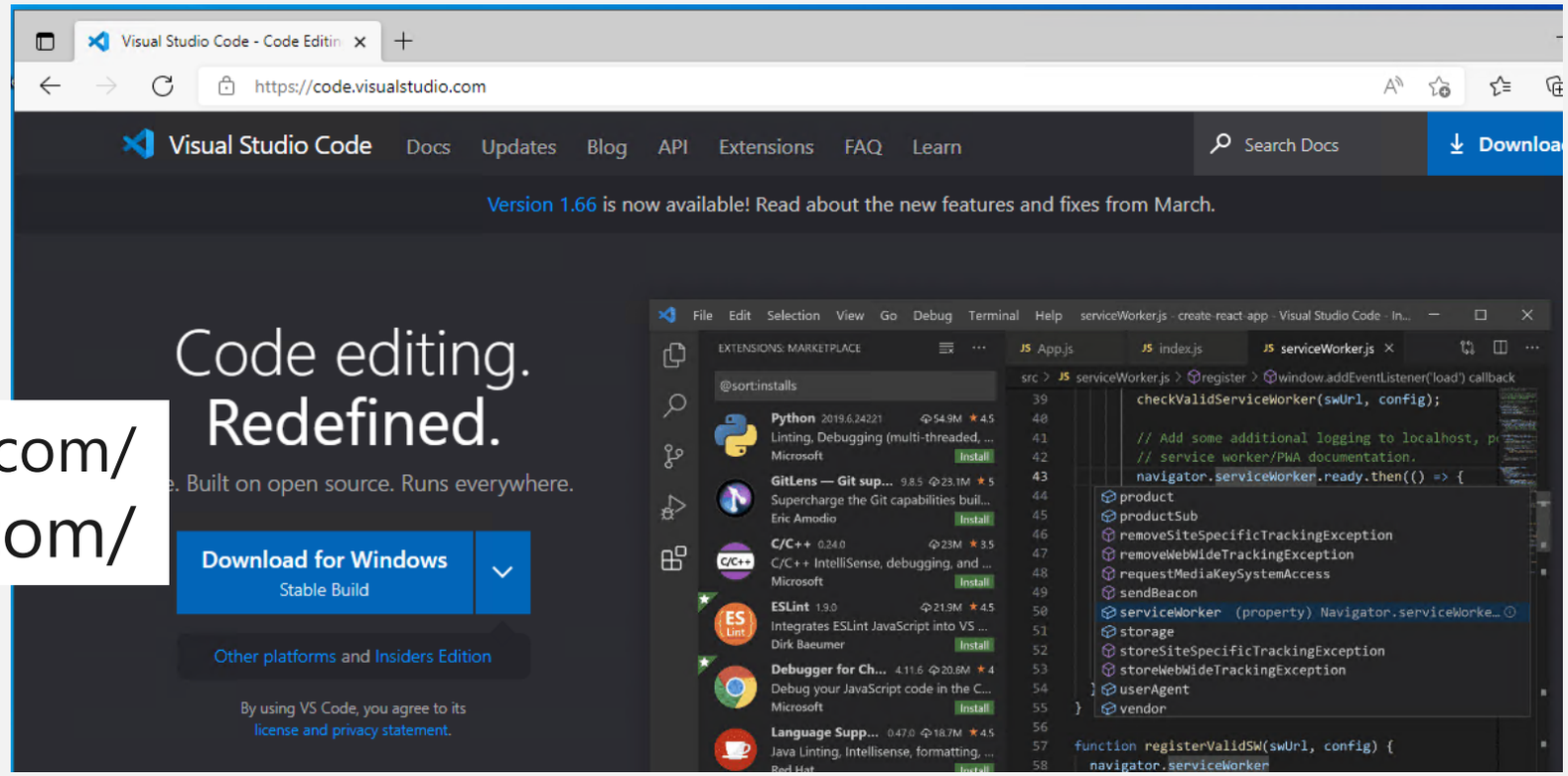
Tenemos dos opciones para descargar Python:

- https://www.python.org/downloads/
- https://www.anaconda.com/distribution/

Anaconda es una Suite de código abierto que abarca una serie de aplicaciones, librerías y conceptos diseñados para el desarrollo de la Data Science con Python.

Instalación de Visual Studio

Diferentes IDE para trabajar con Python, una buena elección puede ser Visual Studio o Visual Studio Code.



- https://visualstudio.microsoft.com/
- https://code.visualstudio.com/

Visual Studio Code
instalar **Python extension for Visual Studio Code** desarrollada por Microsoft.

Extensiones en Visual Studio Code

instalar **Python extension for Visual Studio Code** desarrollada por Microsoft.
Python - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=ms-python.python

instalar **Jupyter** desarrollada por Microsoft.
Jupyter - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter

Entorno de Trabajo

Un entorno de Python es un contexto en el que se ejecuta el código de Python.

Un entorno consta de un intérprete, una biblioteca (normalmente la biblioteca estándar de Python) y un conjunto de paquetes instalados

Entorno de trabajo

Entornos globales, cada instalación de Python (por ejemplo, Python 2.7, Python 3.6, Python 3.7, Python 3,9 Anaconda 4.4.0, etc., mantiene su propio *entorno global*.

Cada entorno está formado por el intérprete de Python correspondiente, su biblioteca estándar, un conjunto de paquetes preinstalados y cualquier paquete adicional que instale mientras ese entorno está activado.

Al instalar un paquete en un entorno global, este pasa a estar disponible para todos los proyectos que usan ese entorno.

3 – Variables y Sentencias de Control

Sintaxis y Estructuras de Control

COMENTARIOS

```
#Comentarios
text = """Escribamos valores alfanumericos
       en distintas lineas. """
```

- # para comentar llneas de código
- """ """ para bloque de valores alfanuméricos que puede tener varias líneas

Sintaxis y Estructuras de Control

DECLARACION DE VARIABLES

```
numero = 10
Numero = 20
saludo = "Hola mundo"
print (numero)
print (Numero)
print (Numero + numero)
print ("saludos" + saludo)
print (type(numero))
print (type(saludo))
```

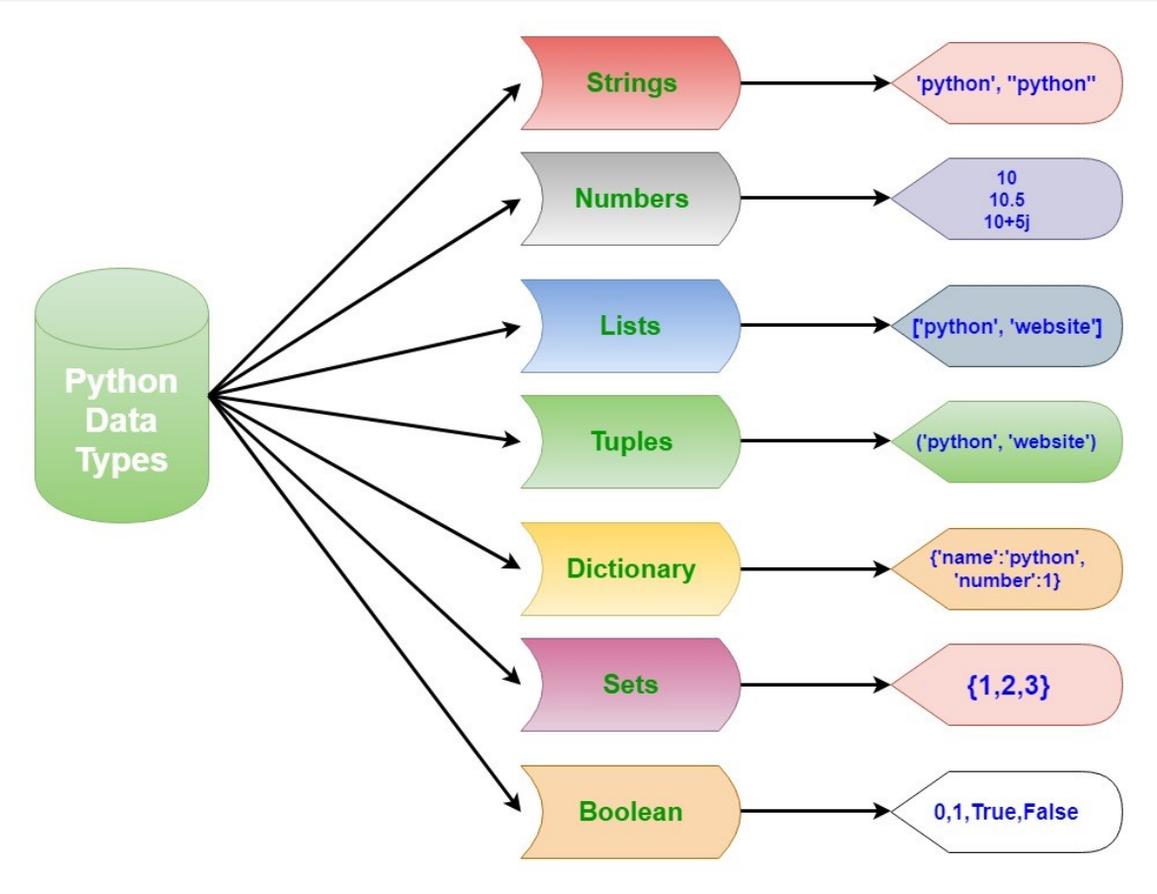
10
20
30
saludos:Hola mundo
<class 'int'>
<class 'str'>

- No especificamos tipo
- Un variable puede ser cualquier palabra a la que asignamos un valor
- El nombre de la variable siempre comienza por guion bajo o letra, nunca por número
- Diferencia mayüsculas de minüsculas
- **`print`** para mostrar el contenido de la variable en la pantalla
- **`type`** para mostrar el tipo de la variable
- **`del`** para eliminar una variable

Sintaxis y Estructuras de Control

Tipos de Datos

Algunos de los datos más communes en Python



Sintaxis y Estructuras de Control

Tipos de Datos (Cont.)

Resumen tipos de datos en Python

Tipo	Clase	Notas	Ejemplo
`str`	Cadena	Inmutable	`'Cadena'`
`unicode`	Cadena	Versión Unicode de `str`	`u'Cadena'`
`list`	Secuencia	Mutable, puede contener objetos de diversos tipos	`[4.0, 'Cadena', True]`
`tuple`	Secuencia	Inmutable, puede contener objetos de diversos tipos	`(4.0, 'Cadena', True)`
`set`	Conjunto	Mutable, sin orden, no contiene duplicados	`{4.0, 'Cadena', True}`
`frozenset`	Conjunto	Inmutable, sin orden, no contiene duplicados	`frozenset([4.0, 'Cadena', True])`
`dict`	Mapping	Grupo de pares clave:valor	`{'key1': 1.0, 'key2': False}`
`int`	Número entero	Precisión fija, convertido en *long* en caso de overflow.	`42`
`long`	Número entero	Precisión arbitraria	`42L` o `456966786151987643L`
`float`	Número decimal	Coma flotante de doble precisión	`3.1415927`
`complex`	Número complejo	Parte real y parte imaginaria *j*.	`(4.5 + 3j)`
`bool`	Booleano	Valor booleano verdadero o falso	`True` o `False`

- Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.
- Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Sintaxis y Estructuras de Control

Asignación simultanea

```
#Asignación simultanea
a=5
b=10
print("Paso1. Valores iniciales")
print(a)
print(b)
```

```
Paso1. Valores iniciales
5
10
```

Descubre la potencia de la asignación simultanea entre variables

Intercambia los valores de las variables a y b para que a contenga 10 y b contenga 5.

Sintaxis y Estructuras de Control

CONVERSIONES

```
a=5
b="25"
c="25.7"
print("Número:" + Str(a))
print(int(b))
print(float(c))
```

```
Número: 5
25

25.7
```

- `str(n)` convierte variables numéricas en texto
- `int(s)` convierte un texto en un valor numérico entero
- `float(s)` convierte un texto en un valor numérico con parte decimal

Sintaxis y Estructuras de Control

CADENA DE CARACTERES

```
cadena="Hola mundo"
print(cadena)
print(cadena[2])
print(cadena[2:])
print(cadena[:2])
print(cadena[2:6])
print(cadena[-2])
print(len(cadena))
```

```
Hola mundo
l
la mundo
Ho
la m
d
10
```

- Una secuencia inmutable de caracteres Unicode
- Es un tipo secuencial e inmutable, un objeto de este tipo no se puede modificar después de haber sido creado.
- Si podemos cambiar el valor de una variables de tipo *String*
- Si podemos leer el contenido total o parcial de una cadena de caracteres
- `len(s)` retorna el número de caracteres de una cadena de texto

Descubre las funciones de las cadenas pulsando punto después del nombre de la variable (Diapositiva No. 38)

Sintaxis y Estructuras de Control

Formateando Cadenas

```
mensaje = "Mundo"
print ("Hola "+mensaje + " !!!")
print ("Hola {}!!!".format(mensaje))
print ("Hola {s} !!!".format(s=mensaje))
print (f"Hola {mensaje}!!!")
numero = 10/3
print (numero)
print ("Hola{n:1.2f}!!!".format(n=numero))
```

```
Hola Mundo !!!
Hola Mundo !!!
Hola Mundo !!!
Hola Mundo !!!
3.333333333335
Hola 3.33 !!!
```

Utiliza `Input()` para preguntar el nombre usuario al cual debes saludar de forma personalizada utilizando `format`.

Sintaxis y Estructuras de Control

Fechas y Horas

```
from datetime import datetime
datenow1 = datetime.now().date()
print("Fecha:",datenow1)
datenow2 = datetime.now()
print("Fecha:",datenow2)
print("Año:",datenow2.year)
print("Mes:",datenow2.month)
print("Dia:",datenow2.day)
print(f"Hora:{datenow2.hour}:{datenow2.minute}")
```

```
Fecha: 2022-04-25
Fecha: 2022-04-25 16:26:03.567149
Año: 2022
Mes: 4
Dia: 25
Hora:16:26
```

Para trabajar con variables que almacenan fechas y horas tenemos que importar el módulo *datetime*.

`Now()` retorna la fecha actual incluyendo la hora.

`Now().date()` retorna la fecha actual.

Sintaxis y Estructuras de Control

Parse Fechas

```python
from datetime import datetime
fecha = "10-11-2022"
obj = datetime.strptime(fecha,'%m-%d-%Y').date()
print(obj)
print(f"{obj.day}-{obj.month}-{obj.year}")
```

```
2022-10-11
11-10-2022
```

Recibe el nombre de **`Parse`**, a la acción de analizar gramaticalmente un valor.

`strptime()` convierte los valores de texto en fecha.

Sintaxis y Estructuras de Control

Formato de Fechas

```
from datetime import datetime
fecha = datetime.now()
print(fecha.strftime("%A %d %b %Y"))
```

```
Monday 25 Apr 2022
```

El formato de fechas es ampliamente utilizado para datos de fechas en diferentes culturas y países.

`strftime()` dar formato a un valor de fechas

Sintaxis y Estructuras de Control

Comodines para formato y conversion de fechas

%a	Nombre corto para un día de la semana	vi.
%A	Nombre largo para un día de la semana	viernes
%w	Dia de la semana en número	5 es viernes
%d	Dia en el mes - desde el 1 hasta el 31 (Según el mes)	29
%b	Nombre del mes versión corta	abr.
%B	Nombre del mes versión larga	abril
%m	Numero del mes 01-12	4
%y	Año versión corta sin siglo	22
%Y	Año versión larga	2022
%H	Hora formato 00-23	10
%I	Hora formato 00-12	10
%p	AM/PM	AM
%M	Minutos 00-59	43
%S	Segundos 00-50	23
%f	Microsegundos	230203
%z	UTC offset	200
%Z	Zona horaria	UTC+02:00
%J	Numero del dia en el año	119
%U	Número de la semana del año (Iniciando semana en domingo	17
%W	Número de la semana del año (Iniciando semana en lunes)	17
%c	versión local de fecha y hora	29/04/2022 0:00:00
%x	versión local de la fecha	29/04/2022
%X	Local versión of time	0:00:00
%%	Un caracter tanto porciento	%

Los comodines se utilizan para transformar texto en fechas y formatear su representación.

Sintaxis y Estructuras de Control

Fechas en español

```
from datetime import datetime
import locale
locale.setlocale(locale.LC_TIME, 'es_ES.UTF-8')

dt =datetime.strptime('29/04/2022','%d/%m/%Y') #datetime.now()

print(dt)
print('\nDirectívas\n------------------------------')
print('Día de semana corto        ' + dt.strftime(': %a'))
print('Día de la semana largo     ' + dt.strftime(': %A'))
print('Número de dia de la semana' + dt.strftime(': %w'))
print('Día del mes                ' + dt.strftime(': %d'))
print('Nombre del mes corto       ' + dt.strftime(': %b'))
```

```
Directívas
------------------------------
Día de semana corto        : vi.
Día de la semana largo     : viernes
```

Para localizar (cambiar de cultura) una ejecución del interprete de Python importamos `locate`

Sintaxis y Estructuras de Control

Tiempo

```
import time
print("Time:",time.time())
print(time.localtime(time.time()))
print("Año:",time.localtime(time.time()).tm_year)
print("Minutos:",time.localtime(time.time()).tm_min)
print("Milisegundos:",int(time.time()*1000.0))
print(time.asctime(time.localtime(time.time())))
```

```
Time: 1650906051.0794837
time.struct_time(tm_year=2022, tm_mon=4, tm_mday=25,
tm_hour=19, tm_min=0, tm_sec=51, tm_wday=0,
tm_yday=115, tm_isdst=1)
Año: 2022
Minutos: 0
Milisegundos: 1650906051081
Mon Apr 25 19:00:51 2022
```

Para trabajar con variables que almacenan tiempo importamos el módulo *time*.

`time()` retorna el tiempo actual

`localtime()` retorna una representación local del tiempo

`asctime()` retorna una representación del tiempo alfanumérica, con fecha, hora y día de la semana.

Sintaxis y Estructuras de Control

Zonas horarias

```python
from datetime import datetime, timedelta
from pytz import timezone
import pytz

#print (pytz.all_timezones)
print(datetime.now(pytz.timezone('Asia/Tokyo')))
print(datetime.now(pytz.timezone('Europe/Madrid')))
```

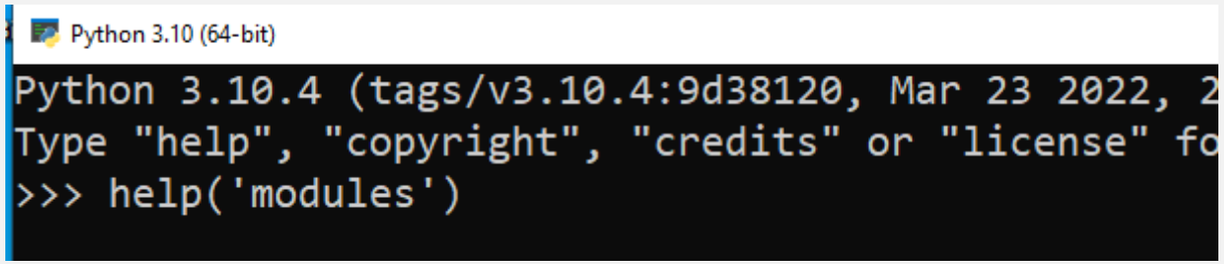
```
2022-04-26 02:16:38.391250+09:00
2022-04-25 19:16:38.394230+02:00
```

Para trabajar con diferentes zonas horarias hacemos uso del módulo *pytz*.

Este módulo debe ser instalado porque no viene por defecto.

`pip` utilidad para disponer y gestionar módulos externos de Python.

`pip install pytz` instala el módulo pytz



Instalar PIP

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
py get-pip.py
```

Variables del sistema

```
C:\Users\REPLAZAR POR VUESTRO USUARIO\AppData\Local\Programs\Python\Python310\Scripts
```

Actualizar pip

```
py -m pip install -U pip
```

Comprobar (Deberan existir los modulos datetime y locale, entre otros)

```
py
help('modules')
```

Instalar el módulo para las zonas horarias

```
pip install pytz
```

Verificar que ha quedado instalado en Python

```
py
help('modules')
```

Ver el fichero NotasPIP.md

Sintaxis y Estructuras de Control

Operadores

Los operadores son símbolos que le indican al intérprete que realice una operación específica, como aritmética, comparación, lógica, etc.

Literales booleanos	
Verdadero	True
Falso	False

Operadores lógicos	
and	Y Lógico
or	O Lógico
not	negación de verdad

Operadores a nivel de bits	
&	and
|	Or
~	not
^	xor
>>	desplazamiento a la derecha
<<	desplazamiento a la izquierda

Operadores numéricos	
+	Suma
-	Diferencia o negación
*	Producto
/	División con decimales
%	Resto
//	División entera
**	Potencia

Operadores de asignación	
=	Igual
+=	Incremento
-=	decremento

Operadores de comparación	
==	igual
!=	distinto
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
is	mismo objeto
is not	distinto objeto

Operadores de pertenencia	
in	pertenece a la colección
not in	no pertenece a la colección

Sintaxis y Estructuras de Control

Sentencia de decision If/Else

```
a = 10
b = 20
if (a>b):
    print (f"el número mayor es {a}")
else:
    if (b>a):
        print(f"El número mayor es {b}")
    else:
        print ("los dos números son iguales")
```

```
El número mayor es 20
```

Las sentencias de decisión determinan el flujo del programa tras evaluar una expresión de comparación

`if` determina la condición y el bloque de sentencias que cumple la condicion

`else` marca el bloque de sentencias que no cumple la condición

`elif` sinmplica un else if (SiNo entonces si)

`Python` no tiene la estructura **`según caso`**.

Sintaxis y Estructuras de Control

Sentencia de repetición FOR

```python
lenguajes = ['python','c','c++','java']
for lenguaje in lenguajes:
    print(lenguaje)
for numero in range(3):
    print(numero)
for numero in range(len(lenguajes)):
    print(f"P:{numero} - V:{lenguajes[numero]}")
```

```
python
c
c++
java
0
1
2
P:0 - V:python
P:1 - V:c
P:2 - V:c++
P:3 - V:java
```

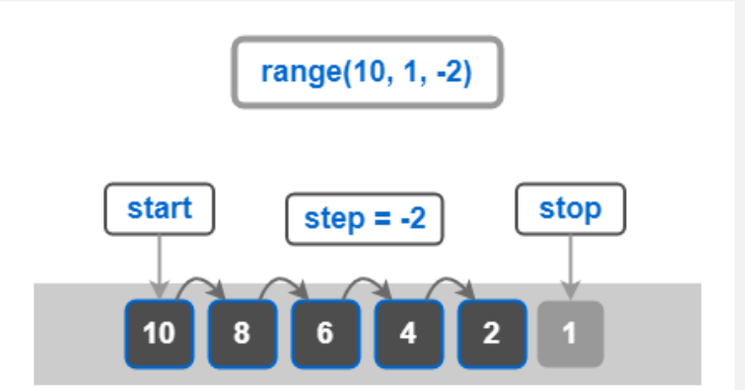
Las sentencia de repetición *for* ejecutan bloques de código de forma iterativa.

La sentencia de repetición for se usa para recorrer y trabajar con las colecciones.

`range(n),` permite definir el inicio, el final y el paso.

`continue` con la siguiente operación de incremento

`Break` finaliza `el` *for*



Sintaxis y Estructuras de Control

Sentencia de repetición While

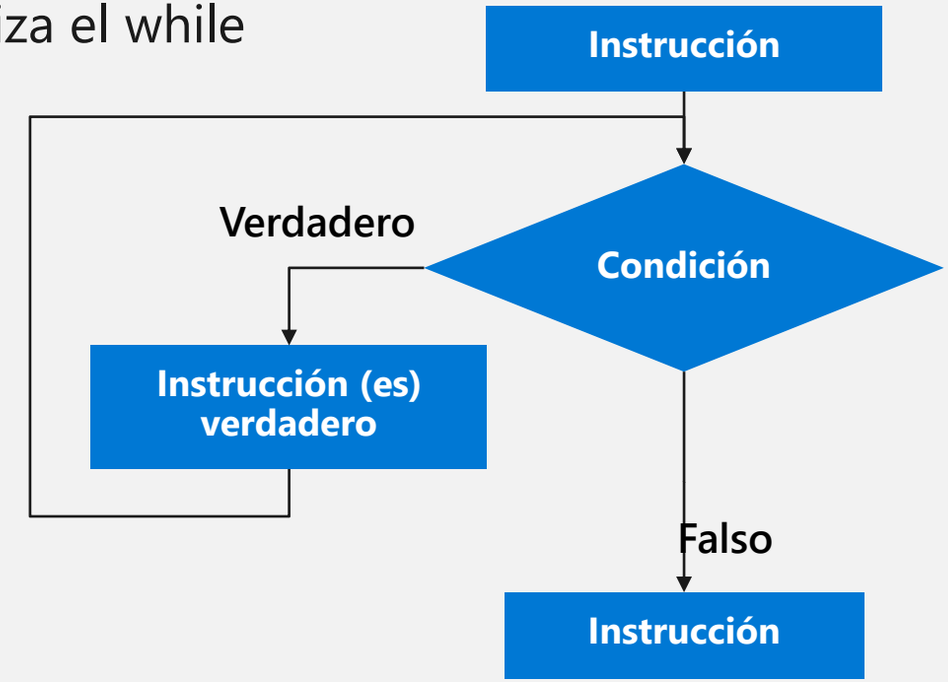
```
print("Instrucción antes del while")
valor = 0
while(valor < 5):
    valor += 1
    if (valor == 3):
        continue
    print(f"Valor actual:{valor}")
print("Instrucción después del while")
```

```
Instrucción antes del while
Valor actual:1
Valor actual:2
Valor actual:4
Valor actual:5
Instrucción después del while
```

La sentencia de repetición `while`, realiza iteraciones mientras se cumpla la una condición.

`continue`, continua con la siguiente iteración.

`Break` finaliza el while



`Python` no tiene `Repetir Hasta`

Sintaxis y Estructuras de Control

Control de excepciones

```
numero1 = 100
numero2 = 0

try:
    print(numero1/numero2)
except ZeroDivisionError:
    print("Error al dividir por cero.")
except:
    print("Error")
else:
    print("La división se calculo correctamente.")
finally:
    print("Fin del programa")
```

```
Error al dividir por cero.
Fin del programa
```

`try` permite controlar las excepciones producidas en un bloque de código.

`except` bloque de instrucciones que se ejecutan cuando se produce una excepción

`else` bloque de instrucciones que se ejecutan al finalizar el `try` si no se produce una excepción

`finally` bloque de instrucciones que se ejecutan siempre que finaliza el `try`, `except` o `else`

Ejercicios

Implementar el ejercicio de algoritmos de fundamentales en python

1. Hola Mundo
2. A partir de un número ingresado diga si es mayor o menor o igual a 9
3. A partir de un número ingreado diga si el mismo es par o impar
4. Ingresar dos números y devuelva el resultado de la suma entre ambos
5. Sumar todo los números pares entre 2 y 100
6. Ingresar un número y muestre todos los divisors del mismo
7. Determinar si un alumno aprueba o suspende un curso, sabiendo que aprobará si su promedio de tres calificaciones es mayor o igual a 4.0; suspende en caso contrario. Deberá permitir ingresar las tres calificaciones y luego calcular su promedio
8. Crear un algoritmo que permita ingresar un nombre y una cantidad numérica para escribir este nombre tantas veces como su cantidad ingresada.
9. Sumar todos los números naturales comprendidos entre 1 y 50.
10. Leer tres números; si el primero es negativo, debe imprimir la multiplicación de los tres y si no lo es, imprimirá la suma.
11. Si un número ingresado es primo o no. (Un número es primo si es divisible únicamente por 1 y por sí mismo).
12 . Sumar los dígitos de un número ingresado. Ejemplo: Si se ingresa 123, debería devolver 6.

4 – Colecciones y JSON

Colecciones y JSON

Colecciones, str

```
ubicacion = 'Madrid capital de España'
print(ubicacion.upper())
print(ubicacion.lower())
aes = 'aaaaaa'
nueva = aes.replace('a', 'b', 3)
print(nueva)
print(ubicacion.split(' '))
ubicacion = '\t\t Madrid capital de España'
print(ubicacion)
print(ubicacion.strip())
print(ubicacion.count('a'))
print(ubicacion.strip().endswith('paña'))
print(ubicacion.strip().upper().startswith('MADRID'))
datos="Pais:{}, capital:{}"
print(datos.format("España",'Madrid'))
```

`upper()` convierte una cadena en mayúsculas
`lower()` convierte una cadena en minúsculas
`find()` Busca en la cadena un valor específico y devuelve la posición donde se encontró
`replace()` Devuelve una cadena donde un valor especificado se reemplaza con un valor especificado
`split()` Divide la cadena en el separador especificado y devuelve una lista
`strip()` Devuelve una versión recortada de la cadena.
`count()` Devuelve el número de veces que aparece un valor especificado en una cadena
`endswith()` Devuelve verdadero si la cadena termina con el valor especificado
`startwith()` Devuelve verdadero si la cadena comienza con el valor especificado
`format()` Formatea valores especificados en una cadena

Colecciones y JSON

Colecciones, Listas

```
colores=["Red","Yellow","Green"]
colores.append("Black")
colores.insert(2,"Orange")
print("Lista de colores:",colores)
print("Posicion color Yellow:",colores.index("Yellow"))
colores.remove("Yellow")
print("Color en la posición 2:",colores[2])
print("Lista de colores:",colores)
print(colores*2)
```

```
Lista de colores: ['Red', 'Yellow', 'Orange',
'Green', 'Black']
Posicion color Yellow: 1
Color en la posición 2: Green
Lista de colores: ['Red', 'Orange', 'Green',
'Black']
['Red', 'Orange', 'Green', 'Black', 'Red',
'Orange', 'Green', 'Black']
```

- Una Lista es una colección de elementos ordenados con un índice de base 0
- En una Lista se pueden añadir, eliminar y modificar elementos

`extend` une dos listas en una

`sort` ordena los elementos con base al valor

`reverse` invierte el orden de los elementos en base a su índice

`pop(i)` elimina el elemento de la posición i.

`remove(value)` elimina el primer elemento coincidente con el valor

Se puede duplicar la lista tantos elementos multiplicándola por un número

Colecciones y JSON

Colecciones, Tuplas

```
numeros = (17,89,21,988,42,429,32,834)
print(numeros[4])
print(list(enumerate(numeros)))
print(max(numeros))
print(min(numeros))
print(sum(numeros))
print(numeros*2)
```

```
42
[(0, 17), (1, 89), (2, 21), (3, 988),
(4, 42), (5, 429), (6, 32), (7, 834)]
988
17
2452
(17, 89, 21, 988, 42, 429, 32, 834, 17,
89, 21, 988, 42, 429, 32, 834)
```

- Una tupla es una colección de elementos ordenados con un índice 0.

- En una tupla no se pueden añadir, eliminar y modificar elementos.

Se puede duplicar la tubla tantos veces multiplicándola por un número

Colecciones y JSON

Colecciones, conjuntos

```
ciudades = {"madrid","albacete","sevilla"}
ciudades.add("valencia")
print("Conjunto de ciudades:",ciudades)
ciudades.discard("albacete")
print("Conjunto de ciuades:",ciudades)
for ciudad in ciudades:
    print(ciudad)
print(len(ciudades))
```

```
Conjunto de ciudades: {'valencia',
'sevilla', 'albacete', 'madrid'}
Conjunto de ciuades: {'valencia',
'sevilla', 'madrid'}
valencia
sevilla
madrid
3
```

- Un conjunto es una colección de elementos sin índice, se dice que esta desordenado

- En un conjunto se pueden añadir y eliminar elementos. (**`add`** y **`discard`**)

- Para acceder a los valores tenemos que recorrer la colección mediante un **`for`**.

No se pueden duplicar los conjuntos al ser únicos en el conjunto. (No se admite el operador *)

Colecciones y JSON

List Comprehension

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print('Nueva lista a partir de recorrer la primera:',newlist)

newlist = [x for x in fruits if "a" in x]
print('Nueva lista con LC',newlist)
```

- La comprensión de listas o **List Comprehension,** ofrece una sintaxis más corta cuando desea crear una nueva lista basada en los valores de una lista existente.

- Sintaxis:
newlist = [expresión `for` el elemento en iterable `if` *condición* == `True`]

- La *condición* `Condición` es como un filtro que solo acepta los elementos que se valoran como Verdadero (`True`).

```
Nueva lista a partir de recorrer la primera: ['apple', 'banana', 'mango']
Nueva lista con LC ['apple', 'banana', 'mango']
```

Colecciones y JSON

Colecciones, diccionarios

```
dicc= {"red":"rojo", "blue":"blue", "green":"verde"}
dicc["black"]="negro"
print(dicc)
dicc.pop("blue")
print(dicc)
print(dicc["red"])
for key in dicc:
    print(key,'->',dicc[key])
```

```
{'red': 'rojo', 'blue': 'blue', 'green': 'verde',
'black': 'negro'}
{'red': 'rojo', 'green': 'verde', 'black': 'negro'}
rojo
red -> rojo
green -> verde
black -> negro
```

- Un diccionario es un conjunto de elementos indexados por su clave

- Cada elemento del *Diccionario* se compone de clave y valor.

- Cuando recorremos un Diccionario mediante `for` los valores que obtenemos son las claves.

- `get(k, “ ”)` muestra un valor de una clave o uno alternativo si la clave no existe.

No se puede duplicar un diccionario con *

Colecciones y JSON

JavaScript Object Notation - JSON

```
import json
citricos = ["limon","naranja","pomelo","lima"]
citricosJSON=json.dumps(citricos)
print("Json de citricos:",citricosJSON)

listacitricos = json.loads(citricosJSON)
print(listacitricos[2])
```

```
Json de citricos: ["limon", "naranja", "pomelo", "lima"]
pomelo
```

- JSON es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo.

- Está basado en un subconjunto del Lenguaje de Programación JavaScript, JSON es un formato de texto que es completamente independiente del lenguaje desde 2019.

- **`dumps`** serializamos un objeto a JSON
- **`loads`** desearilizamos un JSON a objeto

5 – Funciones, clases y módulos

Funciones, Clases y Objetos

Funciones

```
def saludar(nombre):
    print(f'hola {nombre}!!!')
def sumar(a,b):
    return (a+b)
def addColores(colores,color):
    colores.append(color)
    return True

saludar('Billy')
print(sumar(4,3))
colores=["azul","rojo","verde"]
addColores(colores,'negro')
print(colores)
```

```
hola Billy!!!
7
['azul', 'rojo', 'verde', 'negro']
```

- Una Función es un bloque de código que se ejecuta cuando es llamado. Es la implementación de los subalgoritmos vistos en fundamentos

- `def` es la palabra reservada de Python para crear una funciones

- `return` es la palabra reservada de Python para retornar un resultado desde una Función

- Argumento es el valor enviado a la función

- Parámetro es el valor entre paréntesis de la función

- Las colecciones y otros objetos son parámetros por referencia

Funciones, Clases y Objetos

Funciones

```
def ProcesoInicial():
    print('Paso 1 desde aqui')

def SaludarATodos(*nombres):
    for i in nombres:
        print(f'Hola:{i}')

def SaludarATodosDict(**nombres):
    for i in nombres:
        print(f'Saludar a todos Dict:{i},{nombres[i]}')

ProcesoInicial()

SaludarATodos('Franco','Mario')
SaludarATodos('Mariano')
SaludarATodos('Paris','Dakota','Helena','Rocio')
SaludarATodosDict(nombre="Billy",apellidos="Vanegas")
```

- Puede establecer *param para definir N parámetros en la función o **param para tratarlos como diccionarios.

- Puede definir valores por defecto en los parámetros param=<valor por defecto>

```
def ciudades(ciudad='Oslo'):
    print(ciudad)

ciudades('Bogotá')
ciudades()
```

```
Bogotá
Oslo
```

Funciones, clases y objetos

Funciones Lambda

```
def saludar(nombre):
    print(f'hola {nombre} !!!')

saludarLambda = lambda nombre:print(f'hola {nombre} !!!')

saludar('Billy')
saludarLambda('Carola')
```

```
hola Billy !!!
hola Carola !!!
```

- Una Función Lambda es una función pequeña y anónima

- `lambda` es la palabra reservada de Python para crear una funciones lambda

Funciones, Clases y Objetos

Funciones Asíncronas

```python
import asyncio

async def saludo():
    print('Hola  ... ')
    await asyncio.sleep(4)
    print(' ... world')

#python 3.7+
asyncio.run(saludo())
print('siguiente instrucción del programa')
```

```
Hola ...
...world
siguiente instrucción del programa
```

- **`asyncio`** es una biblioteca para escribir código utilizando la sintaxis async / await.
- Se utiliza como base para múltiples marcos asincrónicos de Python que proporcionan redes y servidores web de alto rendimiento, bibliotecas de conexión de bases de datos, colas de tareas distribuidas, etc.

Funciones, Clases y Objets

Modulos

```python
#Modulo modulo.py
def saludar(nombre):
    print(f'Hola {nombre}!!')

#Programa principal App.py
import modulo
modulo.saludar("Billy")

#Programa principal App.py
from modulo import saludar
saludar('Billy -01')

#Programa Principal App.py
from modulo import saludar as saludo
saludo('Billy -02')
```

```
Hola Billy!!
Hola Billy -01!!
Hola Billy -02!!
```

- Un módulo es un conjunto de funciones y clases definidas en un fichero que puedes reutilizar en diferentes aplicaciones de Python

- **`import`** es la palabra reservada de Python para importar y reutilizar las definiciones de otros ficheros (módulos)

- **`from/import`** para importar una definición especifica de un módulo

- **`from/import/as`** para importar una definición especifica de un módulo y crearla un alias

Funciones, Clases y Objetos

Clases

```python
class Persona:
    Nombre = ""
    Apellidos = ""

    def __init__(self, nombre, apellidos):
        self.Nombre = nombre
        self.Apellidos = apellidos

    def Saludar(self):
        print(f'Hola {self.Nombre} !!!')

profesor = Persona("Billy","Vanegas")
print(profesor.Nombre)
profesor.Saludar()
```

```
Billy
Hola Billy !!!
```

- Una clase es un constructor de objetos
- `class` es la palabra reservada de Python para crear una clase
- Las clases puede contener variables, funciones y constructores
- Las funciones y los constructores pueden estar sobrecargados
- `__init__` es el nombre especial para la función constructor
- `self` es un parámetro especial que permite acceder al objeto mismo

Funciones, Clases y Objetos

Atributos de Instancia y Atributos de Clase

```python
class student:
    school = "San Patric"       #Atributos de clase
    totalStudents = 0           #Atributos de clase
    def __init__(self, name, std, roll_no):
        self.nm = name          #Atributos de instancia
        self.std = std          #Atributos de instancia
        student.totalStudents +=1
    def getData(self):            #metodo getter
        print("Student name: ", self.nm)
        print("Standard: ", self.std)
    def setData(self, name, std):#metodo setter
        self.nm = name
        self.std = std
print("The school name is:", student.school)
stud_1 = student("Om", "4th", 9)
stud_1.getData()
stud_2 = student("Hari", "5th", 14)
stud_2.getData()
print("The totals students are:", student.totalStudents)
```

- Python a diferencia de otros lenguajes utiliza el concepto de Atributo de clase y de instancia para diferenciar el uso de variables estáticas.

- Atributos de clase son variables dentro de la clase que comparten valores entre instancias (variables estáticas)

- Atributos de instancia son valores únicos para cada objeto creado a partir de la clase

- `getters` en Python son métodos utilizados para asignar valores a los atributos de instancia

- `setters` en Python son métodos utilizados para recuperar valores de los atributos de instancia

Funciones, Clases y Objetos

Convención para métodos privados

```
class Vehiculo():
    #Clase Vehiculo
    def __privado(self):
       #Metodo supuestamente privado"""
        print("Soy privado")
    def un_metodo(self):
        #Metodo supuestamente publico desde el que se
        # accede a __privado
        self.__privado()
g1 = Vehiculo()
#g1.__privado()  Una excepción no se encuentra metodo
g1._Vehiculo__privado()
g1.un_metodo()
```

- Para cumplir con la encapsulación Python utiliza como convención el uso de las barras bajas en el nombramiento de las funciones de las clases.

- Convencionalmente un método (función) que sea nombrada con dos barras bajas serán marcadas como privadas.

- Esta misma convención se utiliza para los atributos de instancia y marcarlos como privados

Funciones, Clases y Objetos

Funcion property() para atributos de instancia

```
class partner:
    def __init__(self):
        self._age = 0
    def get_age(self):    # get value of _age
        print("getter method called")
        return self._age
    def set_age(self, a): # set value of _age
        print("setter method called")
        if (a<18):
            raise ValueError("age not valid")
        self._age = a
    def del_age(self):    # delete _age attribute
        del self._age
    age = property(get_age, set_age, del_age)
mark = partner()
mark.age = 18
mark._age=15
print(mark.age)
```

- En Python, **`property()`** es una función integrada que crea y devuelve un objeto de propiedad.

- Un objeto de propiedad tiene tres métodos, **`getter(), setter() y delete()`**.

- La sintaxis del uso de **`property()`** es
  property(fget,fset,fdel,doc) donde:
  fget recupera el valor
  gset establece el valor
  fdel elimina el valor
  doc es opcional y se usa para documentar

- Por convención, para establecer el enlace entre el atributo de instancia y los métodos de propiedad, este se nombra con _ , ejemplo **`_age`**

Funciones, Clases y Objetos

Herencia de Clases

```python
class Persona:
    Nombre = None
    Apellidos = None
    def __init__(self, nombre, apellidos):
        self.Nombre = nombre
        self.Apellidos = apellidos

class Alumno(Persona):
    Curso = None
    def __init__(self, nombre, apellidos, curso):
        self.Curso = curso
        super().__init__(nombre,apellidos)

est = Alumno('Yolanda', 'Merizalde','Cloud computing')
print(f'{est.Nombre} {est.Apellidos} - {est.Curso}')
```

```
Yolanda Merizalde - Cloud computing
```

- La herencia nos permite definir una clase que hereda todos los métodos y propiedades de otra clase.

- La clase principal es la clase de la que se hereda, también llamada **`clase base`**.

- La clase hija es la clase que hereda de otra clase, también llamada **`clase derivada`**.

- La función **`super`**() nos permite acceder desde la clase derivada a funciones y propiedades de la clase base.

Funciones clases y Objetos

Polimorfismos con Classes

```python
class Figura:
    def dibujar(self,nombre):
        print(f'Dibujando un:{nombre}')
class Triangulo(Figura):
    def __init__(self,nombre):
        self.nombre = nombre
class Rectangulo(Figura):
    def __init__(self,nombre):
        self.nombre = nombre
class Cuadrado(Figura):
    def __init__(self,nombre):
        self.nombre = nombre
def DibujarFigura(figura):
    figura.dibujar(figura.nombre)
triangulo = Triangulo('Triangulo verde')
cuadrado = Cuadrado('Cuadrado rojo')
rectangulo = Rectangulo('Rectangulo amarillo')
DibujarFigura(triangulo)
DibujarFigura(cuadrado)
DibujarFigura(rectangulo)
```

- Capacidad de un objeto para adoptar múltiples formas.

- Se utiliza como funcionalidad compartida de objetos de una misma clase

En el ejemplo se ha utilizado la función `DibujarFigura` con un parámetro que tiene el método común a todos.

6 – Trabajando con ficheros

7 – Acceso a base de datos

8 – Consumiendo Api RESTful