OFICIOS DIGITALES

PROGRAMACIÓN PYTHON

Sintaxis Básica

LIC. GUILLERMO ALFARO

IPFL
Instituto Provincial de Formación Laboral

GOBIERNO DE LA PROVINCIA DE BUENOS AIRES

Unidad 2
Fundamentos de programación en Python

Introducción

En este capítulo veremos los fundamentos básicos del lenguaje de programación Python, haciendo énfasis en la sintaxis, los tipos de datos, las estructuras de control de flujo y todos los elementos básicos necesarios para la creación de código.

Python es un lenguaje de programación de alto nivel y propósito general, creado en 1991.

Actualmente es uno de los lenguajes más utilizados principalmente por su simpleza y su versatilidad, lo que hace de este, uno de los lenguajes más indicados para iniciarse en el mundo de la programación.

Los programas en este lenguaje suelen ser bastante compactos, por lo general suelen ser más cortos que el mismo programa escrito en otros lenguajes, como por ejemplo C.

Es un lenguaje interpretado, multiplataforma y orientado a objetos.

Python nos permite la codificación orientada a objetos, en forma imperativa y en menor medida, la programación funcional.

Otro detalle no menor es la gratuidad de su intérprete que tiene versiones para casi cualquier plataforma.

Características

En lo que su versatilidad se refiere, Python nos permite crear, desde aplicaciones de uso local o web hasta entrenar algoritmos de Machine Learning y hoy en día es el lenguaje

más utilizado en todo lo que se refiere a inteligencia artificial, redes neuronales y ciencia de datos.

Sus principales características son:

- Sintaxis legible.
- Versatilidad.
- Es interpretado.
- Posee múltiples implementaciones.
- Multiparadigma.
- Posee tipado dinámico.

Que sea de sintaxis legible, significa que podemos leer e interpretar con relativa facilidad el código escrito por otros programadores, captando rápidamente cual es la función del mismo y sus principales elementos.

Cuando hablamos de versatilidad, nos referimos a que podemos desarrollar aplicaciones, ya sea para equipos móviles, computadoras hogareñas, podemos hacer SCRIPTING, levantar servidores y páginas web y aplicar toda su potencia en el análisis de datos o sencillez para representar, mediante código, ideas complejas de manera muy sencilla.

Estas aplicaciones del lenguaje, a su vez, están potenciadas por una enorme comunidad de programadores e investigadores que hay por detrás y que posibilita que tengamos a nuestro alcance un sinfín de documentos, ejemplos e información de todo tipo.

Para quienes no recuerden la diferencia entre lenguaje compilado e interpretado, al decir que Python es interpretado, decimos que necesita de un intérprete para traducirle al dispositivo el código que genera el programador.

Esto nos permite entre otras cosas, poder ejecutar el código en bloques sin necesidad de hacer una ejecución completa, permitiendo el avance en nuestro desarrollo de una manera mucho más ágil detectando y corrigiendo errores y verificando si el resultado de la ejecución de dicho bloque es la esperada o no.

Cabe aclarar que, la ejecución de un programa compilado es más rápida que la de un programa interpretado.

Otro de sus atributos es que posee múltiples implementaciones en otros lenguajes, como por ejemplo Java, permitiendo escribir código para ser utilizado en dicho lenguaje o por el contrario, hacer uso de las diferentes librerías de Java.

Un paradigma de programación es un estilo, una forma de escribir código de manera sistemática obedeciendo determinadas reglas o imposiciones de dicho paradigma, lo que nos aporta el beneficio de generar código de manera más ordenada, lo que redunda en las ventajas a la hora de mantener o actualizar nuestros desarrollos.

Python admite trabajar con varios paradigmas, como por ejemplo la programación orientada a objetos, la programación estructurada o la programación funcional.

En este curso en particular, nos vamos a enfocar en lo que es la programación orientada a objetos para poder sacar un mejor provecho de toda la potencia de este lenguaje.

Finalmente, cuando decimos que Python tiene un tipado dinámico, significa que, cuando definimos una variable u otro almacén temporal de datos, no debemos especificar el tipo de datos que va a contener, pudiendo en una misma variable cargar un dato numérico, una cadena de texto o cualquiera de los tipos que maneja el lenguaje.

Resumiendo lo visto hasta, podemos armar un cuadro resumen de las ventajas y desventajas de Python, atributos a tener en cuenta al momento de decidir que herramientas vamos a utilizar para llevar a buen término nuestro desarrollo.

Ventajas	Desventajas
Facilidad de uso	Ejecución lenta
Popularidad	No se tiene control de la gestión de memoria
Open Source	Requiere testing en tiempo de ejecución
Fácil integración con sistemas empresariales	
Desarrollo asincrónico	

Versiones de Python.

Python posee dos versiones diferentes del intérprete actualmente en uso

- → Python 2
- → Python 3

Si bien la versión 2 fue deprecada, es decir, ya no tiene más actualizaciones y su uso está totalmente desalentado, se continúa usando ya que hay muchos sistemas armados en base a esta versión aún funcionando y su desarrollo sobra la nueva implica en muchos casos, un desarrollo desde cero.

Elementos y conceptos básicos de programación.

En este punto vamos a hacer una rápida referencia sobre los principales elementos que utilizamos para escribir código independientemente del lenguaje antes de verlos directamente con Python.

El más básico con el que trabajaremos es la variable.

Una variable es una referencia a un espacio de memoria dónde se puede almacenar algún tipo de información, básicamente nos permiten etiquetar dicha información para hacer más legible nuestro código y para poder acceder a esta información de una manera sencilla y sin errores.

Veamos algunos ejemplos haciendo foco en definir correctamente cada elemento al que hagamos referencia, ya que en programación es sumamente importante que no existan dudas respecto a que elemento o parte del mismo estamos haciendo referencia, ya que no puede existir, en programación, ningún tipo de dualidad en lo que a definición se refiere.

```
# Ejemplo de variables.

numero = 15

texto = "Curso de Python"

lista = [1,2,'a']

nulo = None
```

A simple vista vemos que todas las variables, independientemente del tipo de que se trate, se definen utilizando tres elementos, a la izquierda el identificador o nombre que le demos, a continuación, el operador de asignación, que es el símbolo = y a la derecha del mismo, el valor que queremos asignarle a la variable o, para ser más precisos, el valor que queremos almacenar en una determinada posición de memoria a la cual hará referencia la variable que estamos definiendo.

Tipos de datos.

En Python tenemos a nuestra disposición un interesante conjunto de tipos de datos, que podemos catalogar y definir por grupos en base a nuestra necesidad.

Por ejemplo, veamos a continuación, la forma más sencilla de definir los grupos de tipos de datos con algunos tipos de ejemplo.

- Numéricos:
  - Enteros, coma flotante, complejos.
- Secuenciales:
  - Listas, Tuplas.
- Mapeo:
  - Diccionarios.
- Colecciones:
  - Conjuntos
- Texto:
  - Caracteres, cadenas de texto.
- Booleano:
  - Verdadero, falso.

Entre las características del lenguaje habíamos mencionado el tipado dinámico, esto significa que no es necesario indicar el tipo de dato al definir la variable, el intérprete lo deduce en base al dato que guardemos en dicha variable y esto, además, posibilita que una misma variable, pueda ser utilizada para diferentes tipos de datos.

```
#  Tipado dinámico.

variable_1 = 15

variable_1 = "Hola mundo"

variable_1 = [4,5,'abc']
```

Como podemos apreciar, a la variable que llamamos “variable_1”, comenzamos asignándole un valor numérico, luego de lo cual, le asignamos una cadena de texto y finalmente una lista, desentendiéndonos por completo de la definición del tipo de dato, ya que de eso se encarga el intérprete.

Sintaxis de Python.

Para los ejemplos del siguiente tema trabajaremos con Jupyter, pero el código es válido para ser ejecutado en cualquier intérprete.

→ IMPORTANTE: Python es un lenguaje CASE SENSITIVE

1 - Tabulaciones.

A diferencia de otros lenguajes de programación, en los que se utilizan símbolos especiales para delimitar estructuras de control de flujo o funciones, en Python usamos la tabulación para delimitar y estructurar los bloques de código.

```
paises = ['Argentina','Brasil','Uruguay']

for i in paises:
    print(i)
```

```
Argentina
Brasil
Uruguay
```

Si detenernos en el análisis del código, cosa que haremos más adelante, el ejemplo nos muestra una lista de elementos y a continuación una estructura que se encarga de recorrerla y mostrar por pantalla cada uno de sus elementos.

Cuando creamos este tipo de estructuras o cualquier otra en base a tabulaciones, lo hacemos utilizando los dos puntos, para que el intérprete sepa que debe agregar una tabulación mientras estamos escribiendo y a su vez, cuando detecta una tabulación, sabe que en ese lugar comienza una estructura o función.

Se puede dar el caso que necesitemos contener una estructura de código dentro de otra, por ejemplo, la siguiente estructura de repetición, contiene dentro de sí, una estructura de decisión, y por cada estructura que vayamos agregando, también deberemos agregar nuevas tabulaciones.

```
numeros = [1,2,3,4]

for x in numeros:
    if x%2 == 0:
        print("{} es un número par".format(x))
    else:
        print("{} es un número impar".format(x))
```

```
1 es un número impar
2 es un número par
3 es un número impar
4 es un número par
```

Podemos apreciar como la estructura IF comienza luego de la tabulación perteneciente al FOR y dentro del mencionado IF, la sentencia PRINT comienza luego de las dos tabulaciones, la del FOR y la del IF.

→ Probar agregando un nuevo PRINT con una y con dos tabulaciones y ver qué sucede en cada ejecución.

2 – Comentarios

Para los programadores, los comentarios suelen ser el tipo de documentación más sencilla de mantener y más útil a la hora de mantener o reformar un desarrollo de software.

En Python, cualquier texto precedido por la marca HASH (símbolo numeral) es ignorado por el intérprete, por lo que se usa para incluir en nuestro código los mencionados comentarios sin perjuicio de la ejecución o eficiencia del mismo.

```
# Creamos una lista de números
numeros = [1,2,3,4]

# Iteramos la lista
for x in numeros:
    if x%2 == 0: # Verificamos si el número es par.
        print("{} es un número par".format(x))
    else:
        print("{} es un número impar".format(x))
```

Otro uso muy común de los comentarios es el de excluir ciertos bloques de código para que el intérprete los ignore, pero sin borrarlos, práctica generalmente usada para la detección de errores.

```
paises = ['Argentina','Brasil','Uruguay']

#for i in paises:
#    print(i)
```

3 – Función HELP

Esta función se utiliza para mostrar una breve descripción del uso de una función específica. Si utilizamos la función HELP sin argumentos, se nos abre una consola de ayuda.

```
def par():
    '''
    La función PAR sirve para determinar si un número es PAR o IMPAR
    '''
    if x%2 == 0:
        print("{} es un número par".format(x))
    else:
        print("{} es un número impar".format(x))
```

```
help(par)
```

```
Help on function par in module __main__:

par()
    La función PAR sirve para determinar si un número es PAR o IMPAR
```

4 – Tipado dinámico

Ya explicamos a que nos referimos cuando hablamos de tipado dinámico, veamos ahora un par de ejemplos en los que vamos a reutilizar una misma variable, para usarla como referencia a diferentes tipos de datos.

```
variable = 1
print(type(variable))
```

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

```
variable = "Hola mundo"
print(type(variable))
```

```
<class 'str'>
```

```
variable = [1,2,3]
print(type(variable))
```

```
<class 'list'>
```

Como podemos apreciar, una misma variable hace referencia a distintos tipos de datos, como, por ejemplo, un valor numérico entero, una cadena de texto y una lista.

→ **La función TYPE nos informa el tipo de dato de un objeto dado.**

Las variables son nombres de objetos dentro de un determinado espacio de nombres, la información del tipo de datos se almacena dentro del propio objeto.

Lo que no podemos hacer es utilizar cualquier operación con cualquier tipo de datos, por ejemplo:

```
print(5+'5')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_63044/3285321692.py in <module>
----> 1 print(5+'5')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Por esto se considera a Python como un lenguaje fuertemente tipificado, esto quiere decir que cada objeto tiene un tipo (o clase) específico y las conversiones implícitas de tipo se van a realizar únicamente cuando no haya posibilidad alguna de ambigüedad.

Particularmente en nuestro ejemplo, el intérprete no tiene forma de saber si lo que queremos obtener es el STRING 55 o el entero 10, y ante esa ambigüedad, Python nos da error.

Podemos, en caso de ser necesario trabajar con excepciones.

Si bien lo veremos más adelante y lo utilizaremos bastante, nos adelantamos un poco y definimos a una excepción como un evento que se produce ante una situación que el intérprete no puede resolver y que, mediante código, podemos anticipar y resolverla nosotros mismos a conveniencia nuestra.

```
try:
    print(5+'5')
except Exception as e:
    print('No se puede realizar la operación solicitada.')
```

```
No se puede realizar la operación solicitada.
```

Existen casos, sin embargo, en los que no hay ambigüedad alguna, por ejemplo, si intentamos realizar una operación entre un numero entero y otro de coma flotante, Python no ve problema alguno para aplicar el tipo de conversión requerido, convirtiendo el entero en coma flotante.

```
a = 30
b = 4.6
print(a/b)
```

```
6.521739130434783
```

5 – Variables

Vamos a ver ahora las variables y lo vamos a hacer directamente sobre código, para explicar bien en detalle cómo es su funcionamiento en Python y que es lo que realmente sucede cuando hacemos determinadas operaciones.

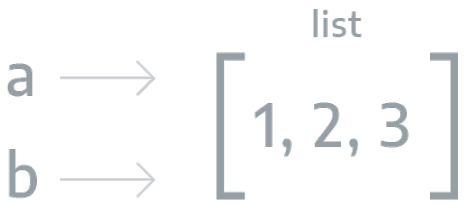
Por ejemplo:

```
a = [1,2,3]
b = a
```

En algunos lenguajes, el código anterior provocaría que los datos cargados de la lista [1,2,3] originalmente asignados a la variable a, se copien para la variable b.

En Python eso no sucede, ya que a y b se refieren al mismo objeto y hacen referencia al mismo espacio de memoria.

Los datos no están duplicados, existe un solo juego de ellos, y lo que tenemos es más de una referencia apuntando hacia ellos.



Podemos corroborar de manera muy fácil este postulado, agregando un elemento a nuestra lista original, la lista a y ver si ese nuevo elemento aparece en la lista b.

Para ello vamos a adelantarnos nuevamente con un tema que profundizaremos más adelante, y vamos a hacer uso del método APPEND que lo que hace es agregar un elemento al final de una lista.

```
a = [1,2,3]
b = a

a.append('x')
print(b)
```

```
[1, 2, 3, 'x']
```

Como podemos ver, a pesar de haber realizado la inserción de un nuevo elemento en la lista a, también lo vemos en la lista b, lo que verifica el postulado de que son solo referencias a un mismo espacio de memoria.

5.a – COPY

Puede darse el caso en que sea necesario tener una copia de los datos, y no solo varias referencias del mimo.

Para que una variable haga referencia a los datos y no a la memoria, tenemos que hacerlo de manera explícita al momento de crear la variable y para ello tenemos el método COPY, que deberemos importar.

```
from copy import copy
a = [1,2,3]
c = copy(a)
a.append('x')
print(c)
```

```
[1, 2, 3]
```

Ahora, cuando creamos la variable c, lo hacemos mediante la función COPY, por lo que no va a ser creada como referencia sino como una copia de la variable a con sus propios datos en una posición de memoria distinta.

Si hacemos la misma verificación que en el punto anterior, vemos ahora que al modificar el contenido de la variable a, no se ve afectado el contenido de la variable c.

5.b – DEEPCOPY

Existen ocasiones en las que las variables almacenan estructuras complejas.

En el ejemplo anterior, cuando vimos COPY, trabajamos con una lista que tenía solamente elementos simples.

Modifiquemos el código para ver qué sucedería si la lista, además de contener datos numéricos enteros o caracteres, tuviera una lista entre sus elementos, y probemos de hacer dos copias, una utilizando COPY y otra utilizando DEEPCOPY.

```
from copy import copy
from copy import deepcopy
lista1 = [1,2,3,['a','b']]

lista2 = copy(lista1)
lista3 = deepcopy(lista1)

print("Lista original : {}".format(lista1))
print("Lista copy: {}".format(lista2))
print("Lista deepcopy: {}".format(lista3))
```

```
Lista original : [1, 2, 3, ['a', 'b']]
Lista copy: [1, 2, 3, ['a', 'b']]
Lista deepcopy: [1, 2, 3, ['a', 'b']]
```

En ambas copias, al menos en apariencia, vemos que obtuvimos el mismo resultado, pero vamos a comprobar, de manera sencilla, que, en realidad, las copias que hicimos tienen una diferencia importante.

Para el caso de la copia realizada con COPY, al momento de copiar el elemento de la lista original que era una lista, lo que hizo fue copiar la referencia de dicha lista a una posición de memoria y no la lista en sí.

```
from copy import copy
from copy import deepcopy
lista1 = [1,2,3,['a','b']]
```

Y lo vamos a comprobar de manera muy sencilla.

Vamos a modificar esta pequeña lista, en la lista original, y luego verificaremos si en las copias, dicha modificación se ve reflejada.

```
from copy import copy
from copy import deepcopy
lista1 = [1,2,3,['a','b']]

lista2 = copy(lista1)
lista3 = deepcopy(lista1)

lista1[3][0] = "Hola"
lista1[3][1] = " mundo"

print("Lista original : {}".format(lista1))
print("Lista copy: {}".format(lista2))
print("Lista deepcopy: {}".format(lista3))
```

```
Lista original : [1, 2, 3, ['Hola', ' mundo']]
Lista copy: [1, 2, 3, ['Hola', ' mundo']]
Lista deepcopy: [1, 2, 3, ['a', 'b']]
```

Como podemos observar, en la copia que realizamos con COPY, esta pequeña lista dentro de la lista original, no se copió como dato, sino como referencia a una posición de memoria, mientras que en la copia realizada con DEEPCOPY, si se copió como dato puro, por lo tanto, cualquier modificación que realicemos en la lista original, la lista 1, va a tener efecto sobre la lista 2 pero no sobre la lista 3.

Siempre refiriéndonos a los elementos complejos, si nuestra lista original está compuesta de elementos simples, bastará realizar la copia con un simple COPY, pero si sabemos que en su interior contiene elementos complejos o podría llegar a contenerlos, lo recomendable es unas DEEPCOPY.

Tipos de datos.

Antes de comenzar con los tipos de datos de Python, tomémonos unos minutos para entender el método FORMAT de la clase STRING, que estuvimos usando en algunos ejemplos pero que no lo definimos formalmente.

En Python casi todo es un objeto, incluso las cadenas de texto, y como tal, tienen sus propios métodos.

El método FORMAT, lo que nos permite es reemplazar en una determinada cadena de texto, los símbolos { } por una variable, quedando de esta forma, nuestra cadena de texto con cierta flexibilidad ya que el texto no será estático.

```
nombre = "Guillermo"

print("Bienvenido {} a la clase de Python".format(nombre))
```

```
Bienvenido Guillermo a la clase de Python
```

En el ejemplo anterior, vemos como le pasamos la variable nombre como argumento al método FORMAT,

1 – Numéricos

Dentro de los tipos de datos numéricos, los más básicos y habituales con los que vamos a trabajar, son los números enteros, los números reales o de coma flotante y en menor medida, los números complejos.

Python también nos permite hacer conversiones de tipo, que veremos más adelante y trabajar con otras bases, y utilizar, por ejemplo, numeración octal o hexadecimal.

Veamos un ejemplo de los tres primeros tipos:

```
# Entero - Decimal - Complejo
n1 = 2
n2 = 2.4
n3 = 4 + 5j

print("n1 es del tipo: {}".format(type(n1)))
print("n2 es del tipo: {}".format(type(n2)))
print("n3 es del tipo: {}".format(type(n3)))
```

```
n1 es del tipo: <class 'int'>
n2 es del tipo: <class 'float'>
n3 es del tipo: <class 'complex'>
```

En el caso particular de los complejos, podemos obtener por separado la parte real y la parte imaginaria de la siguiente manera:

```
n3 = 4 + 5j

print("La parte real de n3 es: {}".format(n3.real))
print("La parte imaginaria de n3 es: {}".format(n3.imag))
```

```
La parte real de n3 es: 4.0
La parte imaginaria de n3 es: 5.0
```

1.a – Operadores aritméticos.

Si hablamos de valores numéricos, ya sea en Python o en cualquier otro lenguaje, tenemos que conocer y saber cómo trabajan los operadores aritméticos.

Dichos operadores nos permiten realizar diferentes operaciones entre los distintos tipos de datos numéricos.

Operador	Nombre	Ejemplo	Resultado
+	Suma	15 + 20	35
-	Resta	15 - 7	8
-	Negativo	-5	-5
*	Multiplicación	8 * 4	32
**	Exponente	3 ** 2	9
/	División	24 / 3	8
//	División entera	17 // 2	7
%	Resto	29 % 3	2

Su uso es muy intuitivo y no difiere mucho de un lenguaje a otro, el siguiente cuadro los resume y agrega un ejemplo de uso de cada uno con el resultado que deberíamos obtener en cada caso:

2 – STRING o cadenas de texto

Básicamente tenemos tres formas de declarar un STRING en Python, teniendo en cuenta que siempre, este tipo de datos, va encerrado entre comillas:

1. Utilizando triple comillas dobles para cadenas de varias líneas.
2. Utilizando comillas simples.
3. Utilizando comillas dobles.

```
a= """
    Esto me permite generar una cadena de texto de varias líneas.
        Incluso reconoce la tabulación.
    """
print(a)
```

```
    Esto me permite generar una cadena de texto de varias líneas.
        Incluso reconoce la tabulación.
```

Veamos un ejemplo de cada uno:

```
b = "Hola mundo"
c = 'Chau mundo'
print(b)
print(c)
```

```
Hola mundo
Chau mundo
```

Como vemos, en el último ejemplo, para las cadenas simples, es indistinto utilizar comillas simples o dobles.

Si quisiéramos que, en nuestro texto, se vean comillas por pantalla, lo que debemos hacer es encerrar todo el texto en el tipo de comillas que no necesitamos mostrar, por ejemplo, si queremos mostrar texto con comillas simples, deberemos encerrar toda nuestra cadena entre comillas dobles.

```
b = "Hola 'mundo'"
c = 'Chau "mundo"'
print(b)
print(c)
```

```
Hola 'mundo'
Chau "mundo"
```

2.a – Operaciones básicas con string.

Las cadenas de texto se almacenan como listas, esto nos permite poder realizar una serie de operaciones sobre las mismas, como si de lista se tratase además de un detalle muy importante, al igual que las listas, contienen índices.

En el siguiente ejemplo, podemos ver cómo obtener un elemento de la cadena accediendo al mismo mediante su índice:

```
texto = 'Curso de Python'
print(texto[0])
```

```
C
```

Vemos que el elemento cuyo índice es el cero, es la letra C.

→ Probar recorrer una cadena de texto como si fuese una lista.

Veamos ahora ejemplos de las operaciones más comunes que podemos realizar con un par de cadenas de texto.

2.a.1 – Suma

```
# Suma o concatenación
a = "Curso"
b = "Python"

print("La suma entre {} y {} es: {}".format(a,b,a+b))
```

```
La suma entre Curso y Python es: CursoPython
```

Vemos que el resultado de sumar o concatenar ambas variables de texto, nos da como resultado ambos textos unidos sin un espacio entre ellos.

2.a.2 – Join

La función JOIN nos va a permitir concatenar ambas variables de texto, pero nos da la posibilidad de intercalar lo que necesitemos entre ambas, por ejemplo, un espacio:

```
# Función JOIN
a = "Curso"
b = "Python"

' '.join([a,b])
```

```
'Curso Python'
```

Podemos usarla con variables como en este ejemplo o directamente con el texto a unir.

→ Probar JOIN usando otro carácter que no sea un espacio.

2.a.3 – Len

La función LEN nos permite obtener el largo de la cadena de caracteres.

El espacio es un carácter, por lo tanto, también será tenido en cuenta a la hora de contar los caracteres de una cadena de texto.

```
#Función LEN
a = "Curso de Python"
print("El largo del texto {} es: {}".format(a,len(a)))
```

```
El largo del texto Curso de Python es: 15
```

2.a.4 – Strip

Este método nos permite eliminar espacios al inicio y al final de la cadena de texto, y los caracteres especiales.

```
#Método STRIP
a = "   Curso de Python % # $"
print(a)
print(a.strip())
print(a.strip('% # $'))
```

```
   Curso de Python % # $
Curso de Python % # $
Curso de Python
```

2.a.5 – Split.

El método SPLIT nos permite dividir una cadena de texto, por ejemplo, un registro de un archivo de datos en formato CSV, donde cada elemento viene separado por una coma, y hacer de cada campo del registro, un elemento de una lista y a partir de ahí, hacer todas las operaciones que vimos hasta ahora y las que veremos cuando toquemos el tema de las listas.

```
#Método SPLIT
a = "Guillermo,Alfaro,Programador"
print(a.split(','))
```

```
['Guillermo', 'Alfaro', 'Programador']
```

2.a.6 – Upper y lower.

Estos métodos nos permiten convertir una cadena enteramente en mayúsculas o minúsculas.

```
a = "Hola"
b = "Mundo"

print("Todo a mayúsculas: {}".format(a.upper()))
print("Todo a minúsculas: {}".format(b.lower()))
```

```
Todo a mayúsculas: HOLA
Todo a minúsculas: mundo
```

Manejo básico de fechas.

Para el manejo básico de fechas a alto nivel, de manera muy sencilla, podemos utilizar el módulo DATETIME.

Veamos algunos ejemplos sencillos de uso:

```
19:11:29
```

→ El método STRFTIME nos permite darle formato a la salida de la fecha de acuerdo a nuestra conveniencia.

Podemos armar el formato de salida como más nos convenga, veamos un ejemplo sencillo para ver todos los datos que obtuvimos, en una sola salida:

```
# Fecha completa
fecha_completa = ahora.strftime("%d/%m/%Y, %H:%M:%S")
print("Fecha y hora: {}".format(fecha_completa))
```

```
Fecha y hora: 12/02/2022, 19:11:29
```