MANUAL TATI

INTRODUCCIÓN: Python

¿Qué es Python?

- Solución que deriva de C# y Java
- Desarrollar lenguaje más amigable, siendo de alto nivel
- Solución para poco txt, misma funcionalidad

Instalación

- Paquetería web y complementos
  - https://www.python.org/downloads/
- Archivos nativos: extensión .py

INTRODUCCIÓN: IPython

¿Qué es IPython?

- Motor para correr Jupyter y archivos markdown
- Crear archivos con extensión .ipynb

Jupyter Notebook

Configuración en VSCode

- Instalar extensiones: Python, Jupyter, Jupyter Notebook Renderers, Pylance (todo dentro de un mismo bundle en la extensión Python)

---

1. MARKDOWN

Encabezados

- Usa símbolo # para indicar un título o subtítulo. Entre más signos # tenga el txt, menor será su tamaño.
- También podemos usar etiquetas HTML

Bloques de Citas

- Use el símbolo > para realizar Bloques de Citas de txt
- EJ: > Este es un
 bloque de citas
- También podemos usar la etiqueta <blockquote> de HTML

Sección de Código

- Use el símbolo ``` al inicio y al final del bloque de código. Ponga el nombre del lenguaje en el que se va a codificar después de los 3 primeros símbolos.
  - EJ: ```Python

str = "Este es un bloque de código"

print(str)

```

- También podemos usar la etiqueta <code> de HTML, sin embargo, no tendrá la paleta de colores que usa el lenguaje.

**Cursiva**

- Para cambiar el texto a cursiva, usar el símbolo * antes y después del texto
  - EJ: *Cursiva*

**Negritas**

- Para cambiar el texto a negritas, usar los símbolos ** antes y después del texto
  - EJ: **Negritas**

**Línea Horizontal**

- Para dibujar una línea horizontal, usar los símbolos --- (3 guiones)
- Alternativamente podemos usar la etiqueta <hr> de HTML.

**Listas Ordenadas**

- Para hacer listas ordenadas debemos insertar el num 1 junto con un . por cada inciso. Para darle hijos a algún inciso, tabular la línea hijo (la que se encuentra bajo al padre).
```

1. Padre 1

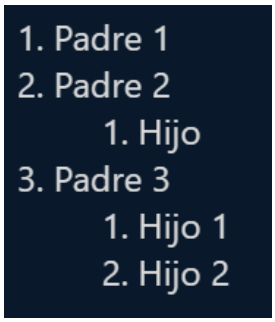
1. Padre 2

   1. Hijo

1. Padre 3

   1. Hijo 1

   1. Hijo 2



- También se pueden usar las etiquetas <ol> y <li> de HTML

Listas No Ordenadas

- Para hacer listas NO ordenadas debemos insertar el símbolo - o + o * en cada inciso. Para darle hijos a algún inciso, tabular la línea hijo (la que se encuentra bajo al padre)

* Item con *

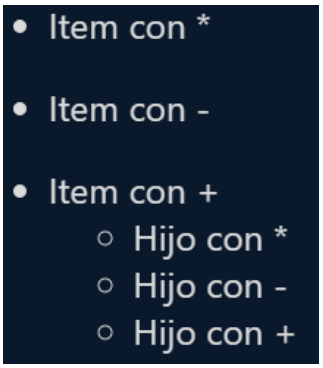
\- Item con \-

\+ Item con \+

   * Hijo con *

   \- Hijo con \-

   \+ Hijo con \+



- También se pueden usar las etiquetas <ul> y <li> de HTML

Links o Vínculos

- Para insertar vínculos se debe respetar la siguiente sintaxis: [txtDelLink](idDeLaSección_o_URL)
  - EJ: [Ir a Jupyter](https://jupyter.org/)
- También se puede usar la etiqueta <a> de HTML
  - EJ: <a href="https://www.google.com">Navegar a Google<a>

Tablas

- Para insertar tablas es posible usar el símbolo | como separador de colunas, y - para separar la primera fila (header) del resto de datos.

|Raza|Nombre|Edad|

|------|----------|-------|

|Bulldog|Nina|4|

|French|Copito|5|

Raza	Nombre	Edad
Bulldog	Nina	4
French	Copito	5

- También se pueden usar las etiquetas <table>, <tr>, <th>, y <td> de HTML

Imágenes y Videos

- Para insertar contenido multimedia se debe respetar la siguiente sintaxis: ![tituloDeLaImg](directorio/img.extension) **O** ![tituloDeLaImg](url)
  - EJ: ![Jupyter](https://cf.appdrag.com/dashboard-openvm-clo-b2d42c/uploads/Jupyter-Notebook-EF5w-udy4.png)
  - EJ2: ![GIF](https://i0.wp.com/www.printmag.com/wp-content/uploads/2021/02/4cbe8d_f1ed2800a49649848102c68fc5a66e53mv2.gif?fit=476%2C280&ssl=1)
  - EJ3: [![Video](http://img.youtube.com/vi/_sOKkON_UnQ/0.jpg)](http://www.youtube.com/watch?v=_sOKkON_UnQ)
- También se puede usar la etiqueta <img> de HTML
- OJO: De manera nativa, Markdown no cuenta con redimensionado de imgs

---

2. DATOS Y OPERADORES

2.1 VARIABLES

Python es un lenguaje de tipado dinámico. NO se necesitan declarar tipos de variables. Los nombres de las variables deben cumplir con esto:

- Sólo incluye caracteres alfanuméricos y guiones bajos (_)
- NO contener espacios en blanco
- NO comenzar con nums

El símbolo = se usa para asignar valores a variables

a) OJO: En Python, el nombre de las variables es sensible a mayúsculas y minúsculas. También existen palabras reservadas, las cuales NO podemos usar para nombrar variables.
b) El conjunto de identificador, símbolo de igualdad, y valor se le conoce como "expresión".

En Python, los tipos primitivos son nums, strings (cadenas de caracteres), booleanos (lógicos), tuplas, listas, y diccionarios.

Métodos

- Son funciones de clase (cada tipo de var incluye funciones asociadas a ellas por default)
- Se pueden checar los métodos disponibles de una var usando la función dir(nombreDeVar)
- Se puede mostrar la info completa de una var usando la función help(nombreDeVar)

2.2 NÚMEROS

Podemos asignar valores numéricos a variables. Existen 4 tipos de variables para almacenarlos dentro de Python:

a) Int (para nums enteros)
b) Float (para nums flotantes o aquellos que tienen decimales)
c) Complex (para nums complejos en formato “A + Bj” donde la unidad imaginaria se representa por "j")
d) ???

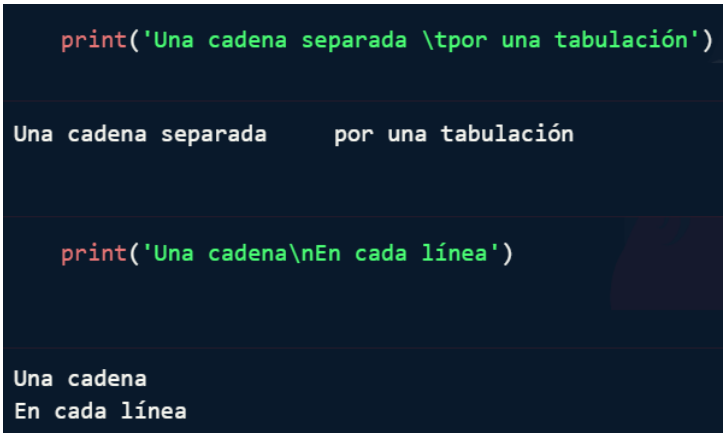
2.3 STRINGS

Son secuencias de caracteres dentro de comillas. No importa si usamos comillas simples (') o dobles (").

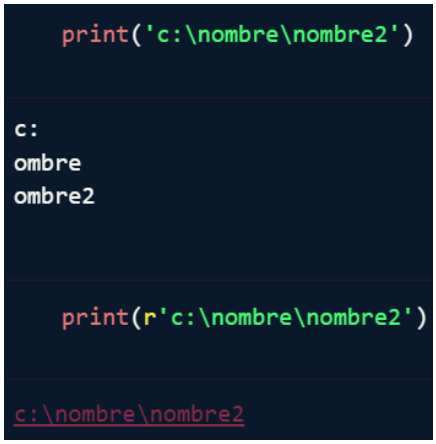
- OJO: No se puede combinar comillas
  - EJ: “Hola Mundo”

Print()

- Permite mostrar en pantalla un txt y editar su presentación con tabulaciones (\t) y saltos de línea (\n)
  - EJ:



- Para que una función print() nos muestre una cadena sin procesar, debemos indicar que la cadena es de tipo crudo (agregar "r" antes de las comillas)
    - EJ:



- También podemos indicar que queremos mostrar el txt en varias líneas usando 3 comillas dobles (""")

2.4 BOOLEANOS

Los datos booleanos son de tipo lógico; sólo tienen los valores de Verdadero y Falso. Las palabras reservadas para ellos son “True” y “False” (con mayúscula la primera letra).

2.5 COMPROBACIÓN DE TIPOS Y CONVERSIONES

Para poder comprobar el tipo de las variables y los valores usamos la función type(). Para convertir el tipo de un valor a otro tipo, se utiliza el nombre del tipo al que se quiere convertir, seguido del valor o la variable a convertir dentro de paréntesis.

- EJ: int(“5”) → Sale 5 como num entero
- EJ2: int(True) → Sale 1
- EJ3: str(2) → Sale 2 como string

OJO: En algunos casos no puede convertir de algún tipo a otro

2.6 OPERADORES

Los operadores aritméticos sirven para datos numéricos, así como otros tipos de datos como strings o listas.

a) Adición
    - Para strings, tuplas y listas se seguirá usando el formato (a + b).

b) Sustracción o Resta
c) Multiplicación
   - Para strings, tuplas y listas se seguirá usando el formato (a * b).
d) División
   - Se usará el formato (a / b).
e) División Entera Truncada
   - Nos da como resultado un valor int. Redondea hacia abajo los valores decimales. Se usa el formato (a // b).
f) Módulo o Residuo de la División
   - Sigue el formato (a % b).
g) Exponenciación
   - Se usará el formato (a ** b).
h) Operadores de Asignación (Acumuladores)
   - Nos facilita escribir código. Tenemos los siguientes:
     i. =
     ii. -=
     iii. +=
     iv. /=
     v. *=
     vi. %=
     vii. //=
     viii. **=
i) Operadores de Asignación (Contadores)
   - En Python no existen los operadores ++ o -- usados usualmente como contadores (en caso de desear incrementar o decrementar el valor de alguna variable en 1).

2.7 OTROS MÉTODOS

a) Lectura de Teclado

Se usa la función *input()* para poder asignar valores string a una variable usando el teclado. Se puede convertir el tipo de ese valor después.

b) Formateo

Existen 2 maneras para reemplazar "espacios" dentro de un sring, convirtiéndolo en una cadena dinámica.

1. Colocar **{ }** en los espacios donde se vaya a reemplazar por valores. Estos últimos se definen en el método .format(valor1, valor2, ..., valorN).

   a. También se pueden usar variables en este método.
   b. Se puede cambiar el orden de aparición de valores usando números dentro de las **{ }**, comenzando por el 0.
2. Colocar llaves dentro de un string y la "f" delante de este. Luego, colocar ***{variable}*** en los espacios correspondientes.

```
print('{} es el primer valor y {} es el segundo' .format("Hola", "Mundo"))
print('Se cambia el orden y {1} aparece primero, luego {0}' .format("Hola", "Mundo"))

nombre = 'Abdo'
x = True

print(f'El valor de {nombre} es {x}')
```

```
Hola es el primer valor y Mundo es el segundo
Se cambia el orden y Mundo aparece primero, luego Hola
El valor de Abdo es True
```

---

3. ESTRUCTURA DE DATOS

3.1 ÍNDICES Y SLICING

Índices

- Son números ordenados asociados a variables, con los cuales podemos identificar elementos basados en su posición.
  - Siempre comienza el conteo a partir del 0 (asociado al primer elemento)
  - También es posible usar números negativos, con los cuales es posible ir en reversa. EJ: El "-1" es el último elemento, el "-2" es el penúltimo elemento, etc.

Slicing

- Gracias a los índices podemos seleccionar secuencias de elementos definidos por la sintaxis [*indice1 : indice2 : paso(opcional)*].
  - En otras palabras, "corta" el valor de una variable, comenzando a partir del índice1, y terminando un valor antes del *indice2*.
  - El paso hace referencia a la cantidad de índices que se estará recorriendo cada vez que vaya al siguiente índice.
    - Por default es 1 (ir de uno en uno, recorriendo hacia la derecha)
- Cuando un rango de valores excluye alguno de sus límites, se dice que el rango es abierto
  - Abierto por la izq (se excluye *indice1* del rango).

- Cuando esto pasa, se hace el corte desde el inicio de la secuencia de elementos hasta el *indice2*.

- Abierto por la derecha (se excluye *indice2* del rango).
  - Cuando esto pasa, se hace el corte desde el indice1 hasta el final de la secuencia de elementos.

```
txt = 'Abdo Sabag'
```

```
print(txt[2:7])
print(txt[:7])
print(txt[2:])
print(txt[::2])
```

✓ 0.0s

```
do Sa
Abdo Sa
do Sabag
Ad aa
```

3.2 LISTAS

Son un tipo de dato compuesto de uno o más elementos.

- Siempre comienza el conteo a partir del 0 (asociado al primer elemento).
- Los elementos dentro de una lista pueden ser de todo tipo de dato.
- Se puede acceder a cada elemento de la lista usando su índice correspondiente.

```
strList = ["1", "2", "3", "4", "5"]
numList = [1, 2, 3, 4, 5]
print(type(strList))
print(numList[2])
```

```
<class 'list'>
3
```

Mutabilidad

- Las listas tienen la característica de poder modificar el valor de alguno de sus elementos (esto es conocido como mutabilidad).

```
numList[2] = 7
numList
```

```
[1, 2, 7, 4, 5]
```

Métodos

1. El método range() nos permite inicializar una lista de números dentro de cierto rango.
   a. Sintaxis: range(a, b, in)
      i. a = límite inferior de la secuencia de nums

ii. b = límite superior de la secuencia de nums
iii. in = unidad de incremento (opcional; cuánto se agregará de un elemento a otro; por default es 1)

2. El método append() nos permite añadir elementos a una lista.
   a. Sintaxis: append(***var_o_valor)***
   b. También podemos usar un + para agregar valores a una lista o juntar listas
3. El método len() nos permite conocer el num de elementos dentro de una lista.
   a. Sintaxis: len(***lista***)

```
rango = range(0, 20, 2)
listaRango = list(rango)
print(listaRango)
```

```
range(0, 20, 2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
listaRango.append("Hola Mundo")
print(listaRango)
print(listaRango + [3.14, 21, "Abdo", True])
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 'Hola Mundo']
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 'Hola Mundo', 3.14, 21, 'Abdo', True]
```

```
print(listaRango)
print(len(listaRango))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 'Hola Mundo']
11
```

4. También podemos aplicar ***Slicing*** dentro de las listas

```
listaX = ['Lista A', 34, 'Lista B', 56, 'Lista C', 78]
listaX[2:4] = ['Verde', 12, 'Azul', 23, 'Morado', 6]
listaX
```

```
['Lista A', 34, 'Verde', 12, 'Azul', 23, 'Morado', 6, 'Lista C', 78]
```

5. Se pueden crear listas de múltiples dimensiones

```
list1 = [0, 'a', False]
list2 = ['Hola', 12.12]
list3 = [['Abdo', 'Sabag', 22], ['Verde', 'Azul', 'Morado']]
multipleDim = [list1, list2, list3]
multipleDim
```

```
[[0, 'a', False],
 ['Hola', 12.12],
 [['Abdo', 'Sabag', 22], ['Verde', 'Azul', 'Morado']]]
```

3.3 TUPLAS

Son secuencias de valores. Los valores almacenados en ellas pueden ser de cualquier tipo y también están indexados.

Se diferencían con listas debido a:

- Después de su creación, sus valores no son modificables (es inmutable)
- Definimos sus valores entre ( ) en lugar de [ ]

```
tupla = ('Ene', 'Feb', 'Mar')
print(tupla)
tuplaMulti = ([1, 2, 3], 'a', 3.14, tupla)
print(tuplaMulti)
print(tuplaMulti[-2])
print(tuplaMulti[-1][1])
```

```
('Ene', 'Feb', 'Mar')
([1, 2, 3], 'a', 3.14, ('Ene', 'Feb', 'Mar'))
3.14
Feb
```

```
# Podemos convertir tuplas a listas para editar sus valores
nuevaLista = list(tupla)
nuevaLista[2] = 'Abr'
print(nuevaLista)
```

```
['Ene', 'Feb', 'Abr']
```

Métodos

1. El método index() devuelve la posición en la que se encuentra el valor especificado en la tupla. En caso de no encontrarlo, manda error.
2. El método count() devuelve la cantidad de veces que se encuentra el valor especificado en la tupla. En caso de no encontrarlo, manda 0.

3.4 CONJUNTOS

Son colecciones de datos que facilitan algunas operaciones, ya que sólo contienen datos únicos.

Se diferencían de listas y tuplas debido a:

- Cada elemento es único
- Los elementos no están ordenados
- Los elementos deben estar entre { }

Métodos

1. El método add() permite agregar elementos al conjunto.
2. Los métodos in y not in nos permite verificar si está o no un valor en el conjunto.
3. El método set() nos permite convertir listas a conjuntos.

OJO: Un conjunto elimina automáticamente elementos duplicados.

Cuando aplicamos la conversión a una cadena, el conjunto resultante toma en cuenta cada letra para hacer el conjunto.

- OJO: Recuerda que los strings son **LISTAS** de caracteres...

```
trabalenguas = 'Parangaricutirimicuaro'
set(trabalenguas)
```

```
{'P', 'a', 'c', 'g', 'i', 'm', 'n', 'o', 'r', 't', 'u'}
```

3.5 DICCIONARIOS

Son colecciones de datos similares a las listas.

Características:

- Elementos definidos por ***key : value***
- Las ***key*** funcionan como índice de cada elemento (no son sólamente nums)
- Se pueden agregar nuevos valores usando la sintaxis ***diccionario[key] = valor***

```
colores = {'rojo': 'red', 'morado': 'purple', 'cafe': 'brown'}
pers = {1: 'Abdo', 2: 'Sabag', 3: 22}
```

```
colores['rosa'] = 'pink'
colores['blanco'] = 'white'

print(colores)
```

```
{'rojo': 'red', 'morado': 'purple', 'cafe': 'brown', 'rosa': 'pink', 'blanco': 'white'}
```

```
ages = {'Abdo': 22, 'Vic': 21, 'Felipe': 20}

print(ages['Vic'])

ages['Abdo'] += 2

print(ages)

print(ages['Vic'] + ages['Felipe'])
```

```
21
{'Abdo': 24, 'Vic': 21, 'Felipe': 20}
41
```

Métodos

1. Usando el bucle for podemos recorrer elementos de un diccionario.
   a. Sintaxis: **for** *varTemporal* **in** *diccionario*: ...

```
# Imprime las keys
for elem in ages:
    print(elem)

# Imprime los valores de las keys
for val in ages:
    print(ages[val])
```

```
Abdo
Vic
Felipe
24
21
20
```

2. El método items() nos devuelve ***key*** y ***value***.
   a. Sintaxis: *diccionario*.items()

```
for x,y in ages.items():
    print(f'Key: {x}, Value: {y}')
```

```
Key: Abdo, Value: 24
Key: Vic, Value: 21
Key: Felipe, Value: 20
```

3. El método del() nos permite eliminar elementos de un diccionario.
   a. Sintaxis: del(*diccionario['key'])*

```
del(colores['rojo'])
print(colores)
```

```
{'morado': 'purple', 'cafe': 'brown'}
```

---

4. SENTENCIAS Y CONTROL

4.1 Sentencias Condicionales (If, Else, Elif)

En Python existen sentencias condicionales if, else, elif (else if), con las cuales podemos dividir el flujo de un programa en caminos distintos. Para esto, definimos un bloque de instrucciones, el cual se ejecutará en caso de que la condición previa se cumpla.

IF

- Sintaxis: if(***condición***):

  bloque de código indentado

  - OJO: Los paréntesis para la condición son opcionales…
- Podemos escribir if's anidados (respetando indentación), es decir, poner uno dentro de otro
- También podemos incluir operadores lógicos (and, or, not) dentro de las sentencias para aumentar eficiencia y legibilidad

ELSE IF Y ELIF

- *Else* se encadena al final del bloque para abrir una nueva lista de instrucciones.
- *Elif* establece una nueva condición que se encadena a un if o a otro elif cuya condición resultó en False. Con esto podemos establecer varias condiciones para lidiar con múltiples posibles entradas que el programa puede recibir.

```python
txt = input('Qué quieres hacer? Saludar o Salir?').upper()

if txt == 'SALUDAR':
    print('Hola!')
elif txt == 'SALIR':
    print('Nos vemos!')
else:
    print('No sé qué es eso, intenta de nuevo porfas...')
```

4.2 Bucles While

Un bucle es una secuencia que ejecuta una acción varias veces hasta que la condición asignada a ese bucle deje de cumplirse. Se le conoce como "*Iteración*" a cada nueva ejecución de la secuencia.

La sentencia *while* repite operaciones mientras la condición que evalúa sea verdadera.

El programador debe planificar un momento en que la condición anterior sea falsa para que se detenga el bucle. De otra manera se obtiene un bucle infinito...

```python
count = 0

while count <= 5:
    count += 1
    print(f'El contador vale {count}')
```

```
El contador vale 1
El contador vale 2
El contador vale 3
El contador vale 4
El contador vale 5
El contador vale 6
```

Break: Con este comando podemos detener un bucle en cualquier momento.

Continue: Con este comando saltamos a la siguiente iteración sin romper el bucle.

```python
cuenta = 0

while cuenta <= 5:
    cuenta += 1
    if cuenta == 4:
        print(f'Rompemos el bucle cuando Cuenta vale {cuenta}')
        break
    print(f'Cuenta vale {cuenta}')
else:
    print(f'Se iteró {cuenta} veces')
```

```
Cuenta vale 1
Cuenta vale 2
Cuenta vale 3
Rompemos el bucle cuando Cuenta vale 4
```

```python
q = 0

while q <= 5:
    q += 1
    if q == 2 or q == 4:
        print(f'Ya que Q vale {q}, saltamos la iteración y vamos a la siguiente.')
        continue
    print(f'Q vale {q}')
else:
    print(f'Se iteró {q} veces')
```

```
Q vale 1
Ya que Q vale 2, saltamos la iteración y vamos a la siguiente.
Q vale 3
Ya que Q vale 4, saltamos la iteración y vamos a la siguiente.
Q vale 5
Q vale 6
Se iteró 6 veces
```

4.3 Bucles For

Este tipo de bucle nos ayuda a repetir un bloque de código un número determinado de veces (se reitera tantas veces como elementos tenga el objeto iterable que recorre el bucle).

Los iterables pueden ser listas, cadenas de texto, etc.

```
print('Bucle For:')

for element in nums:
    print(element)
```

```
Bucle For:
1
2
3
4
5
```

```
nums = [1, 2, 3, 4, 5]
ID = 0
```

Ya que las listas son mutables, podemos modificar el valor en cada índice usando un bucle.

También podemos iterar a través de cadenas de caracteres.

```
x = 0
nums = [0, 1, 2, 3]

# Por cada elemento (sol
# multiplica por 10 el e
for e in nums:
    nums[x] *= 10
    x += 1

print(nums)
```

```
[0, 10, 20, 30]
```

```
cad = "Hola Mundo"
for car in cad:
    print(car)
```

```
H
o
l
a

M
u
n
d
o
```

Métodos

- enumerate()
  - Permite regresar lecturas secuenciales de obj con key y value. Esta función devuelve el elemento y su índice.

```
for elem, num in enumerate(nums):
    nums[elem] *= 10

print(nums)
```

```
[0, 100, 200, 300]
```

- range()
  - Permite generar un rango de números, con los cuales podemos hacer operaciones e incluso la equivalencia a ciclos "for".

```
rango = range(0, 5, 1)
for i in rango:
    print(i)
```

```
0
1
2
3
4
```

5. FUNCIONES

5.1 Definir Funciones

Las funciones en Python permiten reutilizar código. Podrían considerarse como una variable que encierra un conjunto de instrucciones. Al llamar a una función, lo que estamos llamando es la orden de ejecución de las instrucciones.

Algunas características de funciones en Python son:

- Se pueden crear en cualquier momento del programa
- Su palabra reservada es ***def***
- Seguido de ***def*** va el nombre de la función, y entre ( ) van los argumentos de entrada
- **NO** es obligatorio que la función devuelva un valor, aunque se puede usando la palabra reservada ***return***
- Las variables declaradas dentro de ellas son de ámbito local
- Para declarar variables globales dentro de ellas, podemos usar la palabra reservada ***global***
    - No podemos declarar que la variable es global y asignarle un valor en la misma línea
    - Una vez declarada como global una variable local, no se puede regresar a ser local
    - ***OJO:*** Python siempre dará prioridad a los valores de variables locales por sobre las globales (considerará a ambas como variables diferentes, aunque tengan el mismo nombre)

```
def saludar():
    print('Hola!')

saludar()
```

```
Hola!
```

Scope

En Python, cada nombre (identificadores de variables, funciones, objetos, etc) tiene su ***scope*** (ámbito/alcance). El ***scope*** define el área en donde puedes acceder al nombre.

5.2 Retorno de Valores

Para retornar valores dentro de una función podemos usar la palabra reservada ***return***

```
def funcion():
    return 'Hola mundo!'

funcion()
```

```
'Hola mundo!'
```

5.3 Argumentos y Parámetros

Parámetros son los valores que pasamos para usarse en una función.

Argumentos son los tipos de datos que configuramos para su uso en una función.

- Es decir, pasamos parámetros y la función los recibe como argumentos

```
# n1 y n2 son los argumentos
def resta(n1, n2):
    return n1 - n2

# 7 y 3 son los parámetros
resta(7,3)
```

Podemos definir un valor predeterminados para nuestros parámetros

```
def suma(a=None, b=None):
    if(a == None or b == None):
        print('ERROR: Envía 2 nums a la función')
        return
    else:
        return print(a + b)

suma(5,3)
suma()
```

```
8
ERROR: Envía 2 nums a la función
```

Argumentos por Valor

```
def duplica(num):
    return num * 2

n = 10
duplica(n), n
```

```
(20, 10)
```

Argumentos por Referencia

```
def duplica2(nums):
    for i,j in enumerate(nums):
        nums[i] *= 2
    return nums

notasA = list(range(5))
notasB = duplica2(notasA)

# La lista notasA no se pasa como copia en la función, sino que se pasa como
# original y al momento de usarse también se modifican sus valores

notasC = list(range(6))
notasD = duplica2(notasC[:])

# OJO: Para enviar una copia de una lista podemos hacer slicing de esta manera
# nombreLista[:]

print(notasA, notasB)
print(notasC, notasD)
```

```
[0, 2, 4, 6, 8] [0, 2, 4, 6, 8]
[0, 1, 2, 3, 4, 5] [0, 2, 4, 6, 8, 10]
```

5.4 Argumentos Indeterminados

Los argumentos indeterminados representan la forma de pasar una colección de parámetros desconocidos a las funciones.

- Funciona como el operador spread (**...**) en JS
- Pasa el argumento ****argumentos*** y como parámetros podemos pasar los que queramos, del tipo que queramos
- Al pasarle 2 *, le pedimos trabajar con key: value

```python
def argum2(*args):
    for i in args:
        print(i, type(i))

argum2(3.14, ('Hola', 4, False), "Manuel")
```

```
3.14 <class 'float'>
('Hola', 4, False) <class 'tuple'>
Manuel <class 'str'>
```

```python
# Al declarar ambos tipos de argumentos indet. tienes que poner los simples
# primero, luego los key:value

def superNominacion(*args, **keyVal):
    sum = 0
    for e in args:
        sum += e
    print(f'El promedio indet. es {round(sum / len(args))}')

    for key in keyVal:
        print(f'Key: {key}; Valor: {keyVal[key]}')

# Toma como argumentos simples (*args) los primeros 4 indices porque no tienen
# sintaxis key:value; los demás pertenecen a los argumentos **keyVal
superNominacion(1, 2, 3, 4, num = 3.14, tupla = ('Hola', 4, False), nom = "Manuel")
```

```
El promedio indet. es 2
Key: num; Valor: 3.14
Key: tupla; Valor: ('Hola', 4, False)
Key: nom; Valor: Manuel
```

5.5 Funciones Recursivas

Es un tipo de función que se usará a sí misma en el mismo cuerpo de su definición.

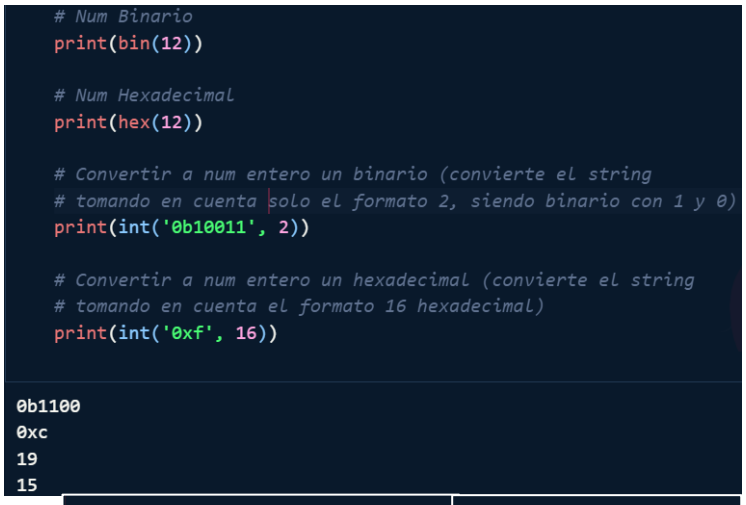
```python
def cuentaAtras(num):
    num -= 1
    if(num > 0):
        print(num)
        cuentaAtras(num)
    else:
        print(num, "BOOM!!!")

    print("Fin de la func", num)

cuentaAtras(5)
```

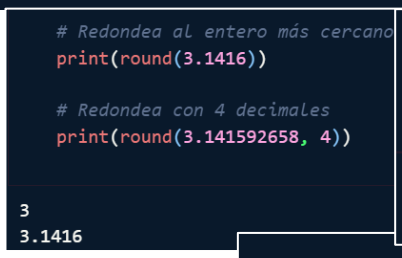
```
4
3
2
1
0 BOOM!!!
Fin de la func 0
Fin de la func 1
Fin de la func 2
Fin de la func 3
Fin de la func 4
```

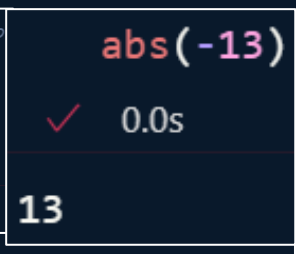
5.6 Funciones Integradas

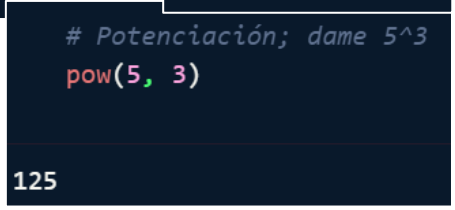
Numéricas

- bin() convierte enteros a números binarios
- hex() convierte hacia números hexadecimales
- Con int(*numBinario,* 2) podemos convertir de binario a entero
- Con int(*numHex,* 16) podemos convertir de hexadecimal a entero
- abs() regresa el valor absoluto de un número
- round(*num, decimalesIncluidos*) redondea al entero más cercano; el segundo argumento es 0 por defecto, aunque puede especificar la cantidad de decimales a querer incluir en el redondeo.
- pow(*num, exponente*) regresa un número exponenciado



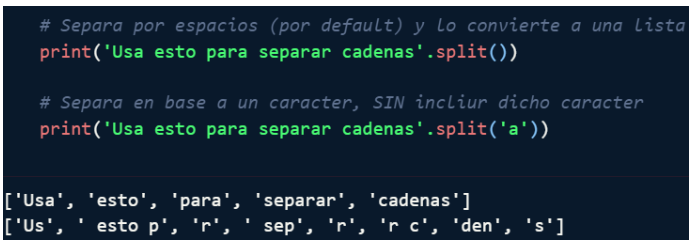


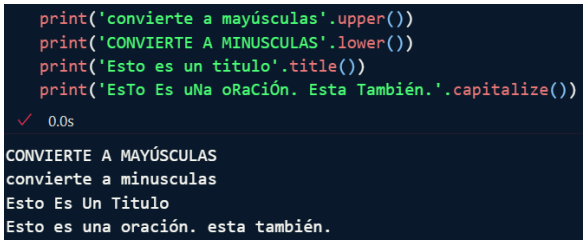




Otras Funciones

- .split(*caracter*) permite separar una cadena de caracteres (usando por defecto los espacios) y crea una nueva lista a partir de esas separaciones
- .upper() convierte todos a mayúsculas
- .lower() convierte todos a minúsculas
- .title() cambia la primera letra de cada palabra en mayúscula
- .capitalize() fotmatea el texto de tal manera de que sólo la primera letra en el string sea mayúscula (como una oración).
- .zip([*elementosList1*], [*elementosLista2*], [*elementosListaN*]) permite crear tuplas bidimensionales, donde cada tupla en la segunda dimensión tiene como igual su índice





```
    # Usado para crear una tupla de 2 dimensiones de 2 listas
    # Toma los elementos por índice
    a, b, c, d = zip(['A', 'B', 'C', 'D'], [11, 12, 13, 14])
    a, b, c, d
```

```
(('A', 11), ('B', 12), ('C', 13), ('D', 14))
```

```
    # Crea un diccionario key:value con un zip
    dict(zip(['A', 'B', 'C', 'D'], [11, 12, 13, 14]))
```

```
{'A': 11, 'B': 12, 'C': 13, 'D': 14}
```

5.7 Funciones Lambda

AKA Funciones Anónimas, representan la manera de crear funciones sin usar la palabra reservada ***def*** ni asignar un nombre a la función creada. La utilidad de estas funciones viene cuando las asignas a una variable.

```
    doblar = lambda num: num * 2
    doblar(5)
```

```
10
```

```
    suma = lambda x,y: x + y
    suma(3, 4)
```

```
7
```

5.8 Funciones Open

Función de permite leer o escribir archivos externos especificando la URL donde se encuentra el archivo y el modo de acceso dependiendo de la acción que se quiere hacer. Algunas opciones son:

- w: Acceso de escritura a un archivo nuevo
- r: Acceso de lectura a un archivo existente
- a: Acceso de escritura para agregar más elementos a archivos existentes
    - Este último siempre mandará al cursor al final del archivo, después del último carácter

```
# El 'with open' te permite abrir temporalmente el archivo
with open(r'C:\                              \archivo.txt',
          'w', encoding='utf-8') as f:
    # Abre el archivo y escribe sobre él
    f.write('Escritura de Ficheros')
    f.write('\nOtra Línea ')
    f.write('Estando en la misma línea')
```

```
with open(r'C:\                                    \archivo.txt',
          'r', encoding='utf-8') as f:
    print(f.read())
```

```
Escritura de Ficheros
Otra Línea Estando en la misma línea
```

```
with open(r'C:\                                    \archivo.txt',
          'r', encoding='utf-8') as f:
    print(f.readlines())
```

```
['Escritura de Ficheros\n', 'Otra Línea Estando en la misma línea']
```

---

6. MANEJO DE EXCEPCIONES

6.1 Errores y Excepciones

ERRORES: Al encontrarse con ellos tras correr un programa, detienen el flujo del código.

- Syntax Error: Surge por error en sintaxis o escritura del código

```
print('Hola'    "(" was not closed
```

```
  Cell In[1], line 1
    print('Hola'
                 ^
SyntaxError: incomplete input
```

- Name Error: Surge cuando nos equivocamos al declarar el nombre de un obj.

```
pint('Hola')    "pint" is not defined
```

```
---------------------------------------
NameError
Cell In[2], line 1
----> 1 pint('Hola')

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

- Index Error: Son usualmente identificados por el editor de código antes de ser ejecutado, pero pueden pasar desapercibidos.

```
    lis = []
    lis.pop()


---------------------------------
IndexError
Cell In[3], line 2
      1 lis = []
----> 2 lis.pop()

IndexError: pop from empty list
```

- Type Error: Surge cuando el tipo de dato no es compatible para realizar una operación

```
    num = input('Dame un num: ')
    'El num {} dividido da {}'.format(num, num/num)


------------------------------------------------------------
TypeError                                 Traceback (most rec
Cell In[4], line 2
      1 num = input('Dame un num: ')
----> 2 'El num {} dividido da {}'.format(num, num/num)

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

- Zero Division Error: Surge cuando se intenta dividir entre 0.

```
    num = int(input('Dame un num: '))
    'El num {} dividido da {}'.format(num, num/num)


------------------------------------------------------------
ZeroDivisionError                         Traceback (mos
Cell In[5], line 2
      1 num = int(input('Dame un num: '))
----> 2 'El num {} dividido da {}'.format(num, num/num)

ZeroDivisionError: division by zero
```

EXCEPCIONES: Bloques de código que permiten continuar con la ejecución del código aun cuando presentan errores.

- Try y Except (“*except*” siendo el equivalente al “*catch*” en JavaScript)

```
while(True):
    try:
        num = int(input('Dame el num'))
        print(f'El num {num} dividido da {num/num}')
        break
    except:
        print('Intente de nuevo')
```

✓ 8.4s

```
Intente de nuevo
El num 2 dividido da 1.0
```

- Else y Finally