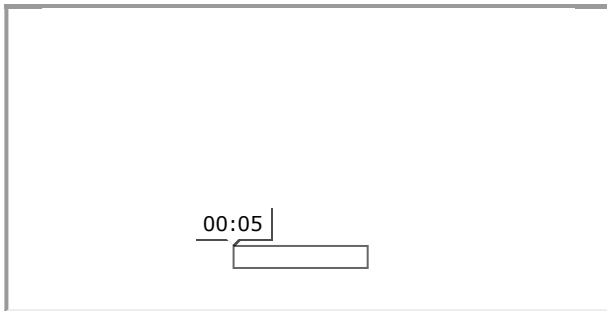


Módulo 2. Variables, operadores y funciones

Video de inmersión



Unidad 1. Variables y operadores

Habiendo explicado en la lectura anterior el contexto del lenguaje de programación Python, y preparado el ambiente de manera correcta para su uso, ya es posible avanzar con la programación propiamente dicha. Para esto, se comenzará con el aspecto esencial de cualquier lenguaje de programación: las variables.

Tema 1. Variables y tipos de datos

Al inicio de la lectura anterior se comentó que una computadora por sí misma no posee ningún tipo de inteligencia, sino que esa inteligencia se la da un programa escrito por un humano, en el que se indica cómo ejecutar ciertas operaciones y procesar los resultados. Para poder realizar esto, en su forma más básica, es necesario almacenar valores en la memoria, realizar las operaciones deseadas entre ellos y, posteriormente, guardar el resultado.

Una variable es un espacio en la memoria que almacena un valor determinado y que puede ser de distintos tipos. Este dato que es almacenado en la variable, será utilizado para participar de operaciones dentro del programa, modificando el valor de otras variables, o incluso de la variable misma sobre la que se está trabajando. Así también, las variables se utilizan para evaluar condiciones y modificar el flujo del programa con base en ello.

Las variables en Python se clasifican en diferentes tipos, tal como lo muestra el siguiente gráfico.

Figura 1. Clasificación de los diferentes tipos en las variables de Python



Fuente: elaboración propia.

Enteros (*int*): como su nombre indica, son números enteros. Es decir, un número sin parte fraccionaria o decimal. Ejemplo: 12.

Flotante (*float*): a diferencia del tipo de dato anterior, un número flotante contiene (o es capaz de contener) una componente fraccionaria. Ejemplo: 3.14.

Cadena de texto (*str*): las cadenas se utilizan cuando se desea o se necesita procesar texto en lugar de números. Siempre se debe usar comillas (ya sea doble o simple) para definir. Ejemplo: “Hola Mundo”.

Booleanos (*bool*): este tipo de dato solo acepta dos valores posibles, “*True*” o “*False*”. Los valores booleanos son útiles principalmente para evaluar si una condición se cumple o no, y actuar en consecuencia al resultado.

Valor nulo (*none*): el valor nulo no es realmente un tipo de dato en sí, pero vale la pena mencionarlo dentro de esta clasificación. Su uso aplica principalmente cuando se desea especificar que una variable no tiene valor, y asignarle el valor 0 o -1 sería algo ambiguo. En esos escenarios, este objetivo se puede cumplir asignando el valor “*None*”.

Definiendo las variables en Python

Una particularidad del lenguaje Python, que facilita en cierta manera su uso, es que las variables pueden tener el tipo de dato que quieran. Esto significa que no se especifica estáticamente el tipo de variable que se está definiendo, como en otros lenguajes de programación, sino que simplemente con el hecho de asignarle un valor a una variable, esta última detectará automáticamente el tipo de dato que le corresponde.

Se ejemplificará lo mencionado anteriormente a continuación, definiendo distintos tipos de variables y usando la función “*type()*” a modo introductorio, para entender cómo Python

está reconociendo el tipo de variables. Esta función al pasarle como parámetro la variable de interés, devolverá el tipo de dato de la misma.

Paso 1. Crearemos un nuevo archivo dentro de Visual Studio Code, y se le asignará un nombre terminando con la extensión “.py” (por ejemplo: datatypes.py).

Figura 2. Paso 1

```
a = 3
print(a)
b = 3.43
print(b)
c = "Esto es un ejemplo"
print(c)
```

Fuente: elaboración propia.

En las líneas anteriores, se procedió a definir las variables a, b y c y se asignó un valor a cada una. También se imprimió por pantalla cada una de ellas mediante la función “print()”. Al ejecutar el **script** haciendo clic en “Run -> Run without debugging”, se verá el siguiente resultado:

Figura 3. Ejecutar script

```
7.3.7 /bin/python3 /Users/amanueli/.vscode/extensions/ms-python.pyt
829 -- /Users/amanueli/Documents/TECLAB/Scripts/datatypes.py
3
3.43
Esto es un ejemplo
AMANUELI-M-77Q3:DevNet amanueli$
```

Fuente: elaboración propia.

Paso 2. A continuación, se utilizará la función “type()” mencionada anteriormente para corroborar el tipo de dato de cada variable. El objetivo es mostrar el resultado de la misma en la terminal; se pasará la función “type()” como parámetro de una función “print()”.

Figura 4. Paso 2

```
a = 3
print(a)
b = 3.43
print(b)
c = "Esto es un ejemplo"
print(c)

# A continuación se imprime en pantalla el tipo
# de dato de cada variable

print(type(a))
print(type(b))
print(type(c))
```

Fuente: elaboración propia.

Se observa que en el código se introduce un comentario a través del carácter “#”. Esto no tiene ningún tipo de influencia en el código, sino que se emplea de manera informativa para el desarrollador o cualquier persona que esté leyendo el código. Posteriormente, se utiliza tal cual se indicó para la función “print()”, pasando como parámetro la función “type()”, que a su vez lleva como parámetro cada una de las variables a evaluar. Al ejecutar el **script**, se observa el siguiente resultado:

Figura 5. Ejecutar *script*

```
7.3.7/bin/python3 /Users/amanueli/.vscode/extensions/ms-python.py  
339 -- /Users/amanueli/Documents/TECLAB/Scripts/datatypes.py  
3  
3.43  
Esto es un ejemplo  
<class 'int'>  
<class 'float'>  
<class 'str'>  
AMANUELI-M-77Q3:DevNet amanueli$
```

Fuente: elaboración propia.

Luego de haber impreso cada una de las variables en la terminal, se observa que tal como se definió en el código, se imprime el tipo de dato de cada una de ellas. Tal como se esperaba, el resultado muestra que, al definir cada variable, Python considera automáticamente a la variable “a” como un entero, a “b” como un flotante y, por último, a “c” como una cadena de texto.

Ahora, es válido también preguntarse: ¿y si Python toma la variable como un entero, pero yo quiero que sea una cadena de texto?

Para estos casos de uso, existe la posibilidad de realizar conversiones entre distintos tipos de datos.

1. Pregunta de repaso

Al momento de definir una variable en Python, es necesario especificar de manera estática el tipo de dato que dicha variable va a contener.

• Verdadero.

• Falso.

Justificación

Tema 2. Conversión de datos

Existen ocasiones en las que debemos contar con variables del mismo tipo para poder mostrar un resultado de determinada manera, o bien porque la operación requiere que la variable sea de un tipo de dato específico. Para esto, Python permite hacer una conversión entre distintos tipos de datos.

Supongamos que definimos un número del tipo flotante, pero queremos que Python lo interprete como un número entero.

Analicemos el procedimiento.

Figura 6. Número tipo flotante

```
num_a = 3.14
num_b = int(num_a)

print(num_b)
print(type(num_b))
```

Fuente: elaboración propia.

Al ejecutar el *script* anterior, se puede leer lo siguiente en la terminal:

Figura 7. Nueva variable

```
3
<class 'int'>
```

Fuente: elaboración propia.

Se observa que si bien se define la variable “num_a” como un número flotante, en la segunda línea se define una nueva variable que toma el valor de “num_a”, pero como un número entero. Para hacer esto, se pasa la variable que se desea convertir como parámetro dentro de la función “int()”.

De la misma manera, si se quisiera realizar la conversión hacia un formato *string*, la función sería “str()”. Para convertir hacia el tipo flotante, la función sería “float()”.

Se detallará a continuación un ejemplo de los casos previamente mencionados.


Figura 8. Ejemplo

```
a_int = 4
#Se convierte el número entero a string
a_str = str(a_int)
#Ahora se puede concatenar la variable con otra cadena de texto.
print("El número elegido es "+ a_str)
```

Fuente: elaboración propia.

El resultado que se observa en pantalla al ejecutar el *script* anterior es el siguiente:

Figura 9. Resultado

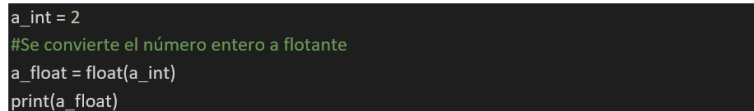


```
El número elegido es 4
```

Fuente: elaboración propia.

De la misma manera, se procederá a realizar la conversión a una variable del tipo flotante y se imprimirá en pantalla.

Figura 10. Variable del tipo flotante



```
a_int = 2
#Se convierte el número entero a flotante
a_float = float(a_int)
print(a_float)
```

Fuente: elaboración propia.

El resultado que se observa en pantalla al ejecutar el *script* anterior es el siguiente:

Figura 11. Resultado



```
2.0
```

Fuente: elaboración propia.

Como se observa, si bien la variable definida originalmente era un número entero, al efectuar la conversión hacia tipo flotante, se muestra en pantalla con notación decimal.


Tema 3. Operadores

De poco serviría tener variables si no se pudieran llevar a cabo operaciones con ellas. Conociendo ahora cómo definir variables y efectuar conversiones entre tipos de datos, podemos avanzar con las distintas operaciones.

Operaciones aritméticas con enteros y flotantes

A continuación, se ejemplifica cómo realizar operaciones aritméticas simples con números enteros y flotantes, ya sea asignando el valor directamente en la variable (a) o tomando el valor de una variable anterior y operando sobre el mismo en una nueva variable (b, c y d)

Figura 12. Operaciones simples



```
# Operaciones simples
a = 10 + 5 # Suma
b = a - 6.14 # Resta
c = b * 3 # Multiplicación
d = c / 2 # División
```

Fuente: elaboración propia.

Como se observa en el ejemplo anterior, las variables pueden definirse con base en el valor de otra variable previa. Adicionalmente, también se puede definir un nuevo valor, con base en el valor anterior de esa misma variable. Esto puede ser de utilidad, por ejemplo,

cuando se necesita incrementar el valor de una misma variable en una iteración (tema que se cubrirá en una lectura posterior).

Figura 13. Definición de la variable

```
# Se define la variable
a = 10

# Se opera sobre el valor anterior de la misma
a = a + 3

# Una manera más programática de escribir lo mismo:
a += 3
a *= 3
```

Fuente: elaboración propia.

Un operador adicional que vale la pena mencionar es el operador módulo (%). Este operador devuelve como resultado el resto de la división entre dos números. Por ejemplo, al dividir 5 por 2, el resto es 1. Cuando un número es divisible por otro, el resto es 0.

Figura 14. Operador módulo

```
a = 5%2
print(a)

b = 4%2
print(b)
```

Fuente: elaboración propia.

Al ejecutar el *script* anterior, se obtiene el siguiente resultado en la terminal:

Figura 15. Resultado

```
1
0
```

Fuente: elaboración propia.

Operaciones con cadenas de texto

La versatilidad de Python permite utilizar operadores aritméticos no solo con números, sino también con cadenas de texto. Esto es sumamente útil, por ejemplo, para concatenar variables a la hora de mostrar un resultado en pantalla. De esta manera, no es necesario emplear una función “print()” por cada variable que se desee mostrar, sino que se podrían agrupar distintas variables dentro de una misma función.

Figura 16. Agrupar distintas variables

```
str_a = "Hola "
str_b = "Mundo!"

str_c = str_a + str_b

print(str_c)

#Otra forma de realizar lo mismo, sin crear una nueva variable:
print(str_a + str_b)
```

Fuente: elaboración propia.

Resultado que se muestra en pantalla:

Figura 17. Resultado

```
Hola Mundo!  
Hola Mundo!
```

Fuente: elaboración propia.

Es importante destacar que solamente se pueden concatenar variables del mismo tipo. Veamos qué sucede cuando buscamos concatenar una cadena de texto con un número entero.

Figura 18. Concatenación

```
a = "Cantidad: "  
b = 6  
  
print(a + b)
```

Fuente: elaboración propia.

Al ejecutar el *script* anterior, se observa el siguiente error en pantalla:

Figura 19. Error

```
Traceback (most recent call last):  
  File "/Users/amanueli/Documents/TECLAB/Scripts/datatypes.py", line 58, in <module>  
    print(a + b)  
TypeError: can only concatenate str (not "int") to str
```

Fuente: elaboración propia.

Esto se soluciona implementando la conversión entre tipos de datos. Esto fue explicado anteriormente. Se debe convertir el número entero a una variable del tipo *string*:

Figura 20. Conversión

```
a = "Cantidad: "  
b = 6  
  
print(a + str(b))
```

Fuente: elaboración propia.

De esta manera, se obtiene el siguiente resultado en pantalla:

Figura 21. Resultado

```
Cantidad: 6
```

Fuente: elaboración propia.

2. Actividad de repaso

Analizar la imagen y completar la pieza de código faltante. El resultado de la ejecución de este *script* será una cadena de texto en la pantalla diciendo: "Juan tiene 3 hijos".

Figura 22. Script

```
a = "Juan"  
print(a + "tiene" + _____ + "hijos")
```

Fuente: elaboración propia.

Resultado en pantalla: "Juan tiene 3 hijos".

Respuestas posibles: "str(3)", "3".

Tema 4. Entradas y salidas simples

Ingreso de datos

En muchas ocasiones, se buscará ingresar datos a medida que se ejecuta el *script*, en lugar de haberlos ingresado manualmente en el código.

Para esto, existe la función "input()".

Utilizando esta función, el *script* detendrá su ejecución esperando el ingreso de un dato por parte del usuario. Una vez que el dato es ingresado, la variable toma ese valor y el programa continúa su ejecución. El valor devuelto por esta función siempre será una cadena de texto, por lo cual, en caso de desear que el mismo sea considerado como el tipo "int()" o "float()", deberá hacerse la conversión correspondiente en el código antes de seguir con el resto del *script*. El parámetro que se debe proveer a la función en su argumento, es el texto a mostrar en pantalla.

A continuación, se demostrará lo comentado anteriormente ingresando un texto y haciendo uso de la concatenación de cadenas para mostrar el resultado.

Figura 23. Código

```
a = input("Ingrese su nombre: ")
print("Usted ingresó: " + a)
```

Fuente: elaboración propia.

El resultado en terminal se ve de la siguiente manera:

Figura 24. Código

```
Ingrese su nombre: Carlos
Usted ingresó: Carlos
```

Fuente: elaboración propia.

Por último, a continuación, se dará un ejemplo del momento en el que se desea ingresar un valor numérico y operar con él. Como se mencionó, antes de seguir trabajando con el dato, se debe realizar la conversión correspondiente.

Figura 25. Ingresar un valor numérico

```
a = int(input("Ingrese un número: "))
b = a + 2

print("El resultado de " + str(a) + " + 2 es = " + str(b))
```

Fuente: elaboración propia.

El resultado que se observa en terminal es el siguiente:

Figura 26. Ingresar un número

```
Ingrese un número: 5
El resultado de 5 + 2 es = 7
```

Fuente: elaboración propia.

Salida de datos

Para mostrar resultados o contenido a través de la terminal, como se pudo observar anteriormente, se hace uso de la función “print()”. El objetivo de esta sección es comentar en detalle las particularidades de esta función para poder implementarla en el código de la manera eficiente.

El uso más simple de la función “print()” es pasar como parámetro la variable a imprimir en pantalla o de manera directa el contenido que se desee mostrar.

Figura 27. Imprimir en pantalla

```
a = "Hola"
print(a)
print("Mundo!")
```

Fuente: elaboración propia.

Resultado en la terminal:

Figura 28. Resultado

```
Hola
Mundo!
```

Fuente: elaboración propia.

Como observamos anteriormente, dentro de la función “print()” se puede, además, concatenar distintas variables como una operación aritmética:

Figura 29. Operación aritmética

```
str_a = "Hola "
str_b = "Mundo!"

str_c = str_a + str_b

print(str_c)
```

Fuente: elaboración propia.

Adicionalmente, existen formas más óptimas de concatenar variables entre sí. A continuación, se demostrarán algunas de ellas.

Figura 30. Concatenación a través de comas

```
#Concatenando a través de comas

a = "Carlos"
b = "34"

print("Su nombre es",a,"y tiene",b,"años")

#Texto resultante: Su nombre es Carlos y tiene 34 años
```

Fuente: elaboración propia.

Otra opción viable es utilizar un método denominado “format()”. Este método consiste en escribir el texto completo y usar “{}” en el lugar de cada variable. Luego, pasar las variables como parámetros de “format()” en la secuencia correcta en que deberían aparecer.

Figura 31. Método “format()”

```
a = "Carlos"
b = "34"

print("Su nombre es {} y tiene {} años".format(a,b))

#Texto resultante: Su nombre es Carlos y tiene 34 años
```

Fuente: elaboración propia.

Existe una manera aún más elegante y simple que las comentadas anteriormente. Esta forma fue introducida con Python 3. La misma consiste en agregar una “f” antes de la cadena de texto que se pasa como parámetro a la función “print()” e incorporar las variables entre llaves “({nombre_variable})” en el sitio donde deberían incluirse.

Figura 32. Agregar “f”

```
a = "Carlos"
b = "34"

print(f"Su nombre es {a} y tiene {b} años")

#Texto resultante: Su nombre es Carlos y tiene 34 años
```

Fuente: elaboración propia.

3. Actividad de repaso

Se desea que el usuario ingrese su edad por pantalla. Con base en lo que se observa en la imagen, completar con el nombre de la función de Python que permite realizar esta acción.

Figura 33. Script

```
a = ____("Ingrese su edad: ")
print("La edad ingresada fue: " + a)
```

Fuente: elaboración propia.

La respuesta correcta es *input*.

Unidad 2. Funciones

Tema 1. Funciones en Python

En ocasiones, será necesario reutilizar distintos fragmentos de código repetidas veces dentro de un *script*. Por ejemplo, imprimir un texto en pantalla de una determinada manera y efectuar operaciones entre variables que deban ser repetidas para múltiples variables, entre otras. Podemos identificar prácticamente cualquier proceso que requiera más de una línea o que se utilice para distintas variables dentro de un programa. En esos casos,

copiar y pegar nunca es la solución. Para esto, la manera más óptima es a través de **funciones**.

Una función es un fragmento de código que se definirá de manera genérica en el script y que será ejecutado cada vez que se invoque en el script principal.

Una función en Python se define de la siguiente manera:

Figura 34. Función

```
def funcion_ejemplo (parametro1, parametro2):  
    variable_ejemplo = parametro1 + parametro2  
    print(variable_ejemplo)
```

Fuente: elaboración propia.

Se comienza con la palabra clave “def” y se asigna un nombre representativo de la tarea que realiza la función. Las variables que se pasan como parámetro a la función no deben haberse definido previamente, sino que es un nombre que se asigna con el objetivo de indicarle a la función cómo trabajar con ellas dentro de la misma. Estas variables son genéricas, dado que las variables reales con las que se trabajará serán asignadas al momento de invocar la función en el *script*.

Cabe destacar que la cantidad de variables que se pasan como parámetro a la función son definidas por el usuario. Estas pueden ser tanto dos o tres como también ninguna. Dependerá del objetivo que la función debe cumplir y de qué datos del código necesita para procesar y cumplir su tarea.

Veamos un ejemplo definiendo una función que realice la suma entre dos variables e imprima el resultado en pantalla. Luego de definir la función, se invocará a la misma dos veces en el código, enviando como parámetro distintas variables para comprobar su funcionamiento.

Figura 35. Comprobación

```
# Se define la función  
def suma (num_a, num_b):  
    resultado = num_a + num_b  
    print(f"El resultado de {num_a} + {num_b} es {resultado}")  
  
# Habiendo definido la función, aquí comienza el script.  
a = 3  
b = 5  
c = 2  
  
# Se invoca a la función creada previamente, pasando como  
# parámetro las variables de interés  
suma(a, b)  
suma(b, c)
```

Fuente: elaboración propia.

Al ejecutar el *script* anterior, se observa en pantalla el siguiente resultado:

Figura 36. Resultado

```
El resultado de 3 + 5 es 8  
El resultado de 5 + 2 es 7
```

Fuente: elaboración propia.

4. Pregunta de repaso

No es necesario haber definido previamente en el *script* las variables que se pasan como parámetros al momento de definir una función.

☒ Verdadero.

☐ Falso.

Justificación

Tema 2. Devolver resultado de una función

Si bien en los ejemplos anteriores el objetivo de la función era imprimir un resultado por pantalla, otra de las cosas que se puede realizar es devolver un valor para ser almacenado en una variable. El objetivo de esto es extraer el valor procesado dentro de una función para poder trabajar con el resultado dentro del *script* principal. Esto se lleva a cabo a través del comando *return*.

Veamos esto aplicado en el ejemplo de la función suma:

Figura 37. *Return*

```

# Se define la función
def suma (num_a, num_b):
    resultado = num_a + num_b
    return resultado

# Habiendo definido la función, aquí comienza el script.
a = 3
b = 5

# Se invoca a la función creada previamente, pasando como
# parámetro las variables de interés
# Se almacena el resultado de la función en la variable "res"
res = suma(a, b)

print(f"El resultado de {a} + {b} es {res}")

```

Fuente: elaboración propia.

Es posible también devolver más de un resultado desde una función. Esto se hace separando por comas las variables deseadas luego del comando *return*. Se verá esto en detalle en un módulo posterior, dado el tipo de estructura de datos devuelta por la función.

Tema 3. Funciones integradas

Existen distintos tipos de funciones integradas en el lenguaje Python. A continuación, además de las ya explicadas en módulos anteriores, se mencionan las más utilizadas.

Tabla 1. Funciones integrales

Función	Descripción	Ejemplo	Resultado en pantalla
len()	Determina la longitud de una cadena.	len("Hola Mundo")	10
upper()	Convierte una cadena en mayúsculas	nombre = "Carlos" print(nombre.upper())	CARLOS
lower()	Convierte una cadena en minúsculas	nombre = "Carlos" print(nombre.lower())	Carlos
replace()	Reemplaza una cadena por otra	Texto = "Hoy es lunes" print(texto.replace("lunes", "martes"))	Hoy es martes
max()	Determina el máximo entre un grupo de números	lista_nros = [0,6,8] print(max(lista_nros))	8
min()	Determina el	lista_nros = [0,6,8]	0

	mínimo entre un grupo de números	<code>print(min(lista_nros))</code>	
<code>sum()</code>	Suma una lista de números	<code>lista_nros = [0,6,8]</code> <code>print(sum(lista_nros))</code>	14
<code>range()</code>	Crea un rango de números	<code>A = range(4)</code> <code>print(list(A))</code>	[0, 1, 2, 3]

Fuente: elaboración propia.

Tema 4. Instalación y uso de librerías externas

A lo largo de los temas anteriores, se fueron usando distintas funciones nativas del lenguaje Python tales como “`print()`”, “`type()`” e “`input()`”. Python facilita muchas tareas a través de este tipo de funciones que cumplen con un objetivo que no conseguiríamos lograr a través de una sintaxis tradicional o, bien, que se podrían lograr escribiendo muchas más líneas de código.

El objetivo de esta sección es introducir las funciones integradas más utilizadas de Python, así como también algunas de los módulos o librerías externas que se pueden instalar para poder cumplir con funciones adicionales.

Librerías externas

Además de las funciones que Python tiene integradas, existe la posibilidad de instalar librerías externas para expandir aún más las funcionalidades y herramientas de este lenguaje de programación. Las librerías consisten de módulos que no están integrados a Python de manera nativa, sino que para poder hacer uso de ellas es necesario descargarlas, instalarlas e importarlas al código que se está desarrollando. Antes de avanzar en la explicación de las más populares, desarrollaremos cómo instalarlas.

Para poder proceder con la instalación de una librería, se debe contar con el módulo “`pip`”. Este módulo se instala por defecto junto con el intérprete de Python que fue instalado previamente.

Para asegurarse que “`pip`” se encuentra instalado, a través de la terminal de Windows se debe ejecutar el comando “`py -m pip --version`”.

En caso de que no se encuentre instalado, se debe guardar **este archivo** (<https://bootstrap.pypa.io/get-pip.py>) con el nombre “`get-pip.py`” y ejecutarlo a través del

comando “python get-pip.py”.

Habiendo instalado el módulo “pip”, la manera de instalar la última versión de una librería externa en Python es ejecutando el siguiente comando (en el ejemplo se utilizará como referencia una librería llamada TEST):

Figura 38. Comando

```
py -m pip install TEST
```

Fuente: elaboración propia.

Ya conociendo cómo instalar una librería, procederemos a comentar aquellas más populares, junto con alguna funcionalidad elemental de cada una.

Invitamos al alumno a explorar la documentación de cada una de ellas para poder avanzar con casos de uso más complejos y aprovechar al máximo el potencial de cada librería.

1. *NumPy*.
2. *Requests*.
3. *Netmiko*.

NumPy

Numpy es la librería fundamental para una persona que trabaja en ciencia de datos. Su nombre deriva de “Numerical Python” y se trata de una herramienta que facilita el trabajo con cálculos matemáticos y arreglos de datos, ya sea tanto por simplicidad en el código como por la velocidad de ejecución.

Se la puede instalar a través de la ejecución del siguiente comando en terminal:

Figura 39. NumPy

```
py -m pip install numpy
```

Fuente: elaboración propia.

Una vez que la librería se encuentra instalada, a diferencia de una función integrada en Python, para poder utilizar la misma en un *script*, primero se debe importar. Esto se hace a través del comando *import*. A continuación, se ejemplificará cómo importarla y cómo asignarle el alias “np”, simplemente, para poder referirse a ella a través de esa denominación posteriormente.

Figura 40. Alias

```
import numpy as np
```

Fuente: elaboración propia.

A través de la librería NumPy se pueden crear vectores (*arrays*): consiste en un grupo de elementos del mismo tipo y se trabaja de manera muy rápida con ellos en comparación

con las listas (una estructura que será explicada en un módulo posterior).

Figura 41. Vectores

```
import numpy as np

x = [2, 3, 4, 5]
nums = np.array(x)

print(nums)
```

Fuente: elaboración propia.

Se puede ver en el ejemplo anterior, que luego de importar la librería NumPy, se define una lista “x”. Esa lista se convierte en un vector a través del *método* “array”. Un método, dicho de manera simple, está compuesto por funciones incorporadas dentro de la librería y que son invocadas de la manera especificada en el *script* anterior.

Supongamos que se desea operar con cada uno de los números del vector. NumPy permite realizarlo de manera muy sencilla. A continuación, se hará una operación de multiplicación y de normalización (se divide cada número por el máximo del grupo para que el rango de números varíe entre 0 y 1).

Figura 42. Operación de normalización

```
import numpy as np

x = [2, 3, 4, 5]
nums = np.array(x)

#Vector multiplicado por 2
print(nums*2)
#Vector normalizado
print(nums/(np.max(nums)))
```

Fuente: elaboración propia.

En la terminal se puede ver el siguiente resultado:

Figura 43. Resultado

```
[ 4  6  8 10]
[0.4 0.6 0.8 1. ]
```

Fuente: elaboración propia.

Adicionalmente, NumPy permite trabajar con vectores de más de una dimensión, pasando como parámetro más de una lista.

Figura 44. Parámetros

```
import numpy as np

nums = np.array([[2,2,3], [8,11,9]])
print(nums)
```

Fuente: elaboración propia.

Se puede ver el siguiente resultado en pantalla:

Figura 45. Resultado

```
[[ 2  2  3]
 [ 8 11  9]]
```

Fuente: elaboración propia.

NumPy provee una extensa variedad de métodos para poder operar con vectores, así como también para generar y trabajar con matrices. El objetivo de este módulo es simplemente introducir la librería, pero se invita a explorarla más en detalle para cada caso de uso que sea necesario trabajar.

Requests

La librería *requests* permitirá interactuar a Python con distintas API (interfaces de programación de aplicaciones). Esto es, poder actualizar información o bien consultar datos de una aplicación que tenga disponible una API para interactuar con ella.

Esta librería se instala a través del siguiente comando:

Figura 46. Requests

```
py -m pip install requests
```

Fuente: elaboración propia.

Python realizará un “pedido” (*request*) a la API y esta devolverá una respuesta. La respuesta incluirá un código indicando si el pedido se ejecutó correctamente, así como también un cuerpo de mensaje con el contenido solicitado.

Figura 47. Respuesta



Fuente: elaboración propia.

En esta sección, se ejemplificará el uso principal de la librería. Los fundamentos detrás de los *requests* y *responses*, y los modos existentes para procesar los datos recibidos, serán cubiertos en otra materia especializada en el tema.

En el siguiente ejemplo, se importará la librería *requests* y se realizará un pedido del tipo GET hacia la API de GitHub, solo con el objetivo de identificar si está activa.

Figura 48. Request

```
import requests

response = requests.get('https://api.github.com')

print(response.status_code)
```

Fuente: elaboración propia.

Como resultado de este *script*, se observa en pantalla el número 200. Esto significa que el *request* fue ejecutado correctamente.

A continuación, se dará como ejemplo un *request* en el cual se muestra el contenido del mensaje. Para esto, se imprimirá en pantalla una estructura de dato denominada JSON. Si bien la misma está fuera del alcance de este curso, lo exponemos para dar una demostración de cómo Python puede consultar información de un recurso externo. La API utilizada en este ejemplo es de un sitio web de chistes. Al hacer un *request* del tipo GET, la misma devolverá un chiste como respuesta, estructurado en el formato JSON.

Figura 49. Respuesta

```
import requests
import json

url = "https://sv443.net/jokeapi/v2/joke/Pun?type=twopart"

joke = requests.get(url=url).json()

print(json.dumps(joke,indent=2))
```

Fuente: elaboración propia.

El resultado que se muestra en la terminal es:

Figura 50. Resultado

```
{
  "error": false,
  "category": "Pun",
  "type": "twopart",
  "setup": "Did you hear about the claustrophobic astronaut?",
  "delivery": "He just needed a little space.",
  "lang": "en"
}
```

Fuente: elaboración propia.

Netmiko

Python puede utilizarse para enviar comandos a distintos dispositivos de red, tales como

routers y *switches* (explicados en la materia Fundamentos de Redes).

Esto es sumamente útil para automatizar tareas repetitivas de configuración. Para poder lograr esto, se emplea una librería externa llamada “Netmiko”. Esta se instala a través del siguiente comando:

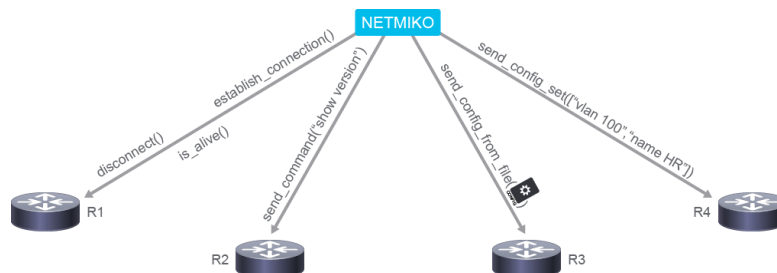
Figura 51. Comando de instalación

```
py -m pip install netmiko
```

Fuente: elaboración propia.

En la siguiente imagen, pueden observarse los métodos principales de Netmiko:

Figura 52. Métodos principales de Netmiko



Fuente: [Imagen sin título sobre Netmiko]. (s. f.). Recuperado de: <https://network-tic.com/script-python-con-netmiko-sabe-mejor/>

ConnectHandler()

Inicia una conexión con el dispositivo.

is_alive()

Determina si la conexión con el dispositivo está activa y devuelve *true* o *false* como respuesta.

send_command(<comando válido>)

Este método se usa para enviar comandos del tipo “*show*” hacia el dispositivo. Debe ser una sola línea que contenga un comando válido.

send_config_set([<comando_1>, <comando_2>])

Este método puede utilizarse para enviar una serie de comandos de configuración hacia el dispositivo. La lista de comandos debe estar en el orden correcto.

A continuación, exponemos un ejemplo enviando un comando del tipo “*show*” hacia un *router* virtual que forma parte de un laboratorio. Se invita al alumno a probar el mismo fragmento de código.

Figura 53. Comandos al router

```
#Enviar comandos al router a través de Netmiko

from netmiko import ConnectHandler

#Se definen los parámetros del router
host= "sandbox-iosxe-latest-1.cisco.com"
username= "developer"
password= "C1sco12345"
device_type= "cisco_xe"
ssh_port= 22

#Se realiza la conexión de Netmiko

router = ConnectHandler(host=host, username=username, password=password,
device_type=device_type, port= ssh_port)

#Se envía comando para mostrar las interfaces
response = router.send_command("show ip int brief")

print(response)
```

Fuente: elaboración propia.

Se puede observar la siguiente respuesta en la terminal: se trata exactamente del mismo texto que mostraría el dispositivo de red si se ejecutara el comando de manera local.

Figura 54. Respuesta

Interface	IP-Address	OK?	Method	Status	Protocol
GigabitEthernet1	10.10.20.48	YES	NVRAM	up	up
GigabitEthernet2	unassigned	YES	NVRAM	administratively down	down
GigabitEthernet3	unassigned	YES	NVRAM	administratively down	down
Loopback1	2.2.2.2	YES	manual	up	up
Loopback2	unassigned	YES	unset	up	up
Loopback10	10.10.10.10	YES	other	up	up

Fuente: elaboración propia.

5. Pregunta de repaso

El comando “pip install” es útil para instalar librerías internas, tales como “print()” e “input()”.

• Verdadero.

• Falso.

Justificación

Video de habilidades



Pregunta de habilidades

1. Es posible concatenar a través del operador una variable del tipo cadena de texto con una variable del tipo entero.

Verdadero.

Falso.

Justificación

2. Para utilizar la función `type()`, es necesario pasarle como parámetro el nombre de la variable y no el valor

directamente.

Verdadero.

Falso.

Justificación

3. ¿Qué función integrada se podría utilizar para ingresar datos por terminal en lugar de definirlos de manera estática en el código?

print()

insert()

input()

write()

Justificación

4. ¿Cuál es la palabra clave utilizada para definir una función en Python?

def

fun

function

code

Justificación

5. Una función sólo puede ser ejecutada en el código una sola vez, al inicio del programa que es donde se define.

Verdadero.

Falso.

Justificación

Cierre

En esta lectura, se comenzaron a incorporar conocimientos prácticos del lenguaje de programación Python. Comenzando desde la definición de variables y los diferentes tipos de datos disponibles, así como la conversión entre datos y la entrada y salida de los mismos a través de la terminal. Se continuó con el desarrollo del tema funciones y del uso de librerías internas y externas para expandir las herramientas disponibles dentro de Python, facilitando su uso en la operatoria diaria. Habiendo cubierto las estructuras de datos básicas y cómo trabajar con ellas, en el siguiente módulo se trabajará con estructuras complejas de datos que permiten almacenar grupos de información de manera estructurada.

Glosario



Referencias

[Imagen sin título sobre Netmiko]. (s. f.). Recuperado de: <https://network-tic.com/script-python-con-netmiko-sabe-mejor/>