

CIENCIA DE DATOS CON PYTHON: RECOLECCIÓN, ALMACENAMIENTO Y PROCESO



Fuente: adobestock/84383512



Autor:

Amaury Giovanni Méndez Aguirre

ÍNDICE

Introducción.	3
Ciencia de datos con Python_ recolección, almacenamiento y proceso.	4
¿Qué es Python?	5
Instalación de un intérprete de Python	5
Comenzando a programar	7
Comentar un código.	7
Variables	8
Ejercicios	11
Programación estructurada vs Programación orientada a objetos.	12
Herencia	17
Conclusiones	18
Introducción.	19
Tipos de Datos en Python.	20
Números Enteros (int)	21
Números Decimales (float).	22
Booleanos (bool).	22
Letras y Textos (str)	26
Listas (list)	30
Tuplas (tuple).	33
Diccionarios (dict).	33
Estructuras de control de flujo	34
Condiciones múltiples y anidación	35
Bucles o ciclos.	37
Bucle while.	37
Bucle for	38
Conclusiones	40
Pasos - procesos	41
Bibliografía	45

INTRODUCCIÓN

Python es un lenguaje que se ha popularizado en los últimos años por su sencillez y facilidad tanto en su uso como en su aprendizaje. Por tal motivo vas a encontrar los aspectos relevantes y fundamentales del lenguaje y su forma de programar con base en la metodología orientada a objetos, preparándote el camino para usar las librerías especializadas en este lenguaje para la ciencia de datos. Vas a aprender cómo desarrollar programas en Python, cómo programar orientado a objetos, qué son las comparaciones y condiciones en programación, bucles o ciclos y los tipos de datos más usados como las listas y los diccionarios. A lo largo de este curso, irás encontrando ejemplos de código que podrás ir desarrollando para entrenar tus habilidades de programación, así que empecemos.



Ciencia de datos con Python_ recolección, almacenamiento y proceso



¿Qué es Python?

Como lenguaje de programación, Python tiene una estructura o sintaxis muy sencilla en comparación con otros lenguajes de programación como C++ o Java. Esto no le resta potencia a la hora de ejecutar tareas complejas en varios escenarios con menos líneas de código y en menor tiempo, como en el caso de la Ciencia de Datos, donde es preferido junto al lenguaje R precisamente por esta razón (García, 2018)

Desde un punto de vista más técnico, Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel. Fue creado por Guido van Rossum en 1991, con la idea de ser un lenguaje fácil, intuitivo, y de código abierto. Es uno de los lenguajes que más ha crecido, debido a su compatibilidad con la mayoría de los sistemas operativos y la capacidad de integrarse con otros lenguajes de programación. Aplicaciones como Dropbox, Reddit e Instagram son escritas en Python (Python para Data Science, 2022)

La programación estructurada presenta simplemente una secuencia de instrucciones en donde su potencial se evidencia en bloques de código que pueden reutilizarse indefinidamente, llamados **funciones**. Para la programación orientada a objetos (POO), esto es simplemente una característica de su potencial, puesto que estos “objetos” pueden ser creados y ser independientes de su bloque de código original. Esta diferencia será clave durante todo el módulo y mostrará el potencial de programar en Python.

Instalación de un intérprete de Python

Para empezar a programar en Python, vamos a optar por instalar un entorno de desarrollo conocido como **Jupyter Lab**. Una vez instalado, podremos correr nuestro primer programa (se recomienda que se digite la instrucción completa y no que se emplee la herramienta de copiar y pegar):



Función:

En programación, una función es un bloque de código que puede reutilizarse sin la necesidad de volver a definir su contenido.

Jupyter Lab:

Es un entorno de desarrollo con interfaz web para la programación en Python. El proceso de instalación se puede ver en el siguiente video: <https://youtu.be/tzxcKmFj24A>

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

suma

```
print(2 + 3)  
>> 5
```

multiplicación

```
print(3 * 4)  
>> 12
```

división

```
print(10 / 3)  
>> 3.3333333333333335
```

división parte entera

```
print(10 // 3)  
>> 3
```

resto de la división

```
print(20 % 6)  
>> 2
```

hacer mayúsculas un texto

```
print("programación para ciencia de datos".upper( ))  
>> PROGRAMACIÓN PARA CIENCIA DE DATOS
```

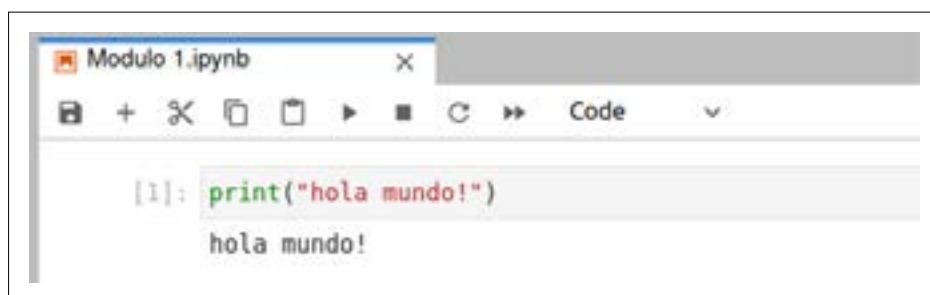
unir textos

```
print("programación" + "en Python")  
>> Programaciónen Python
```

separar palabras en un texto

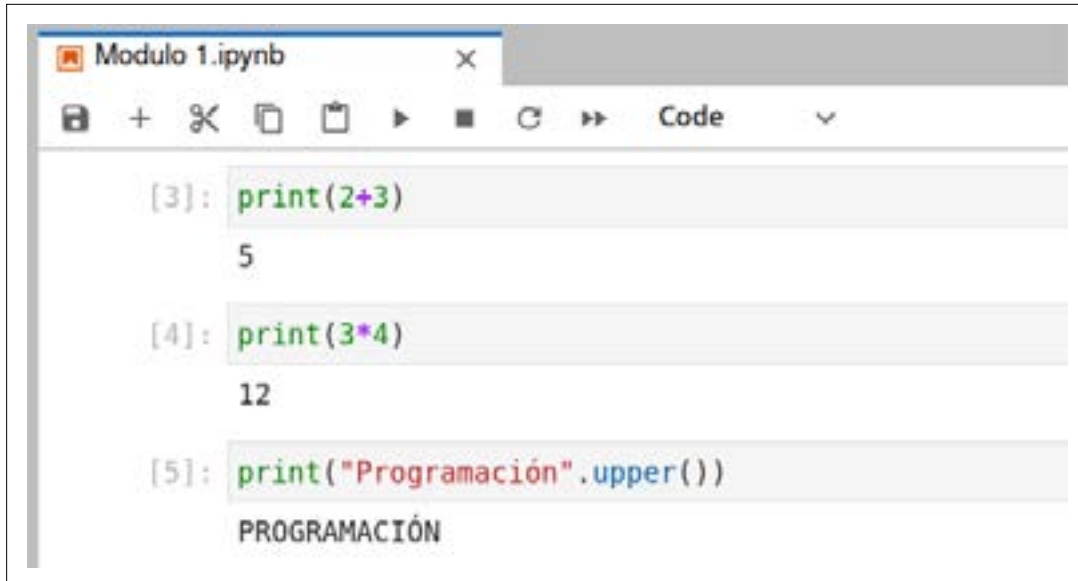
```
print("Ciencia de Datos".split( ))  
>> [ 'Ciencia', 'de', 'Datos' ]
```

Figura 1



Fuente: propia

Figura 2



```
[3]: print(2+3)
5

[4]: print(3*4)
12

[5]: print("Programación".upper())
PROGRAMACIÓN
```

Fuente: propia

La función **print()** recibe como argumento un texto que está delimitado por las comillas dobles y es condición necesaria para que Python lo interprete como un texto. Esta función ya se encuentra precargada al iniciar cualquier intérprete de Python y no es necesario importarla, como veremos más adelante con otras funciones.

Comenzando a programar

Para empezar a programar, debes entender que un lenguaje de programación en esencia puede realizar operaciones matemáticas, pero también puede realizar otro tipo de operaciones o instrucciones con diversos tipos de datos que no solo son numéricos. Veamos algunos ejemplos y ejercicios que puedes ir desarrollando dentro de Jupyter Lab: Observa de estos ejemplos que, así como se pueden usar los símbolos aritméticos para realizar operaciones matemáticas, también se pueden usar símbolos lingüísticos como las comillas, la coma y el punto, y en este último su uso se hace inherente a la POO, como al usar los métodos **.upper()** y **.split()** de la clase *string*, pero este es un tema que trataremos más adelante.

¿Te atreves a realizar operaciones como $(3 + 4 * 7 * (5 - 2) / (10 ** 2))$?

Comentar un código

En Python, al igual que en otros lenguajes de programación, puedes realizar pequeños o grandes comentarios para explicar líneas de código, dejar información importante o algún otro uso que el programador desee dejar y para esto se puede realizar bien sea un comentario de una sola línea o comentarios de varias líneas.

Para comentar una sola línea, basta con utilizar el símbolo numeral #, y para comentar varias líneas se suele emplear la comilla simple o las comillas dobles, tres veces al inicio y tres veces al final. Veámos como:

```
#Comentario de una sola línea  
"""  
Comentarios  
de varias líneas  
según requiera o desee el programador  
"""
```

Variables

Una variable es simplemente un espacio en memoria que se usará temporalmente para almacenar algún tipo de dato. Veámos un ejemplo:

```
edad = 17  
  
print( edad )  
  
>> 17
```

Observa que se ha usado el símbolo igual (=) para realizar este almacenamiento temporal del número 17, en una variable creada con el nombre edad. El número 17 resulta ser entonces un tipo de dato que en este caso es un número entero. En este ejemplo, para mostrar el contenido de la variable, se ha usado la función **print()** y como resultado se muestra el número almacenado.



Variable:

Es simplemente un espacio en memoria que se usará temporalmente para almacenar algún tipo de dato.

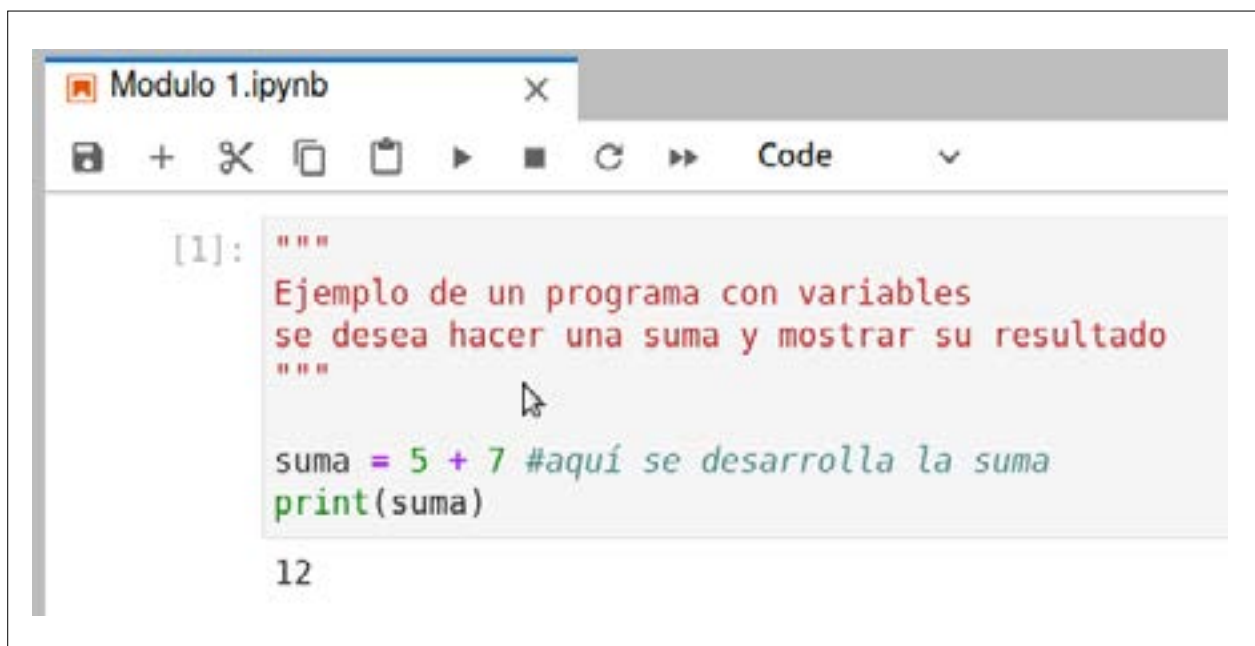
Para indagar en Python por cuál tipo de dato ha sido almacenado en la **variable**, podemos utilizar una función denominada **type()** la cual cumple este propósito:

```
type( edad )  
  
>> int
```


Algo muy importante para resaltar, es que las variables pueden almacenar resultados de distintas operaciones aritméticas o de objetos como se muestra a continuación:

```
suma = 5 + 7  
  
print( suma )  
  
>> 12  
  
nombre = "Ciencia de Datos".upper( )  
  
print( nombre )  
  
>> CIENCIA DE DATOS
```

Figura 3



```
[1]: """  
Ejemplo de un programa con variables  
se desea hacer una suma y mostrar su resultado  
"""  
  
suma = 5 + 7 #aquí se desarrolla la suma  
print(suma)  
  
12
```

Fuente: propia

Creación de variables

Las variables pueden ser creadas bajo ciertas reglas:

Puede empezar por una letra, (a - z, A - Z) o con el guión bajo, ejemplo:

```
ancho = 15  
Altura = 20  
_radio = 3.5
```

Puede contener después otras letras, números y el guión bajo, ejemplo:

```
primer_nombre = "Giovanni"  
_horas_dia2 = 4
```

Los nombres de las variables son case sensitive, lo que quiere decir que la variable altura, será diferente de la variable Altura, o de la variable ALTURA

Debido a que Python emplea algunas palabras para ejecutar instrucciones de código, no todas las palabras pueden ser usadas para crear variables, y a esto se le conoce como palabras reservadas.



Visitar página

Encuentre la lista detallada en el siguiente enlace

https://www.w3schools.com/python/python_ref_keywords.asp



Ejercicios

Resuelve con Python estas operaciones matemáticas

1. $y = x^2 + 4x + 10$ donde $x = 5$
2. $y = 3x^2 + 5x - 50$ donde $x = -15$
3. $y = 2x^3 - 4x + 10 - 6 * 7$ donde $x = 35$
4. $y = ((10 + 30) ** 4) - (600 / 200)$
5. $y = x^m - 7k + 10$ donde $x = 5, m = 3, k = 8$

Un posible código para el ejercicio 1 sería el siguiente:

```
x = 5

y = x**2 + 4*x + 10

print("y =", y)

>> y = 55
```

Ahora continúa con los demás ejercicios.

Programación estructurada vs Programación orientada a objetos

Considera las siguientes líneas de código como un ejemplo de la programación estructurada o secuencial, donde el código se ejecutará línea tras línea

```
edad = 20

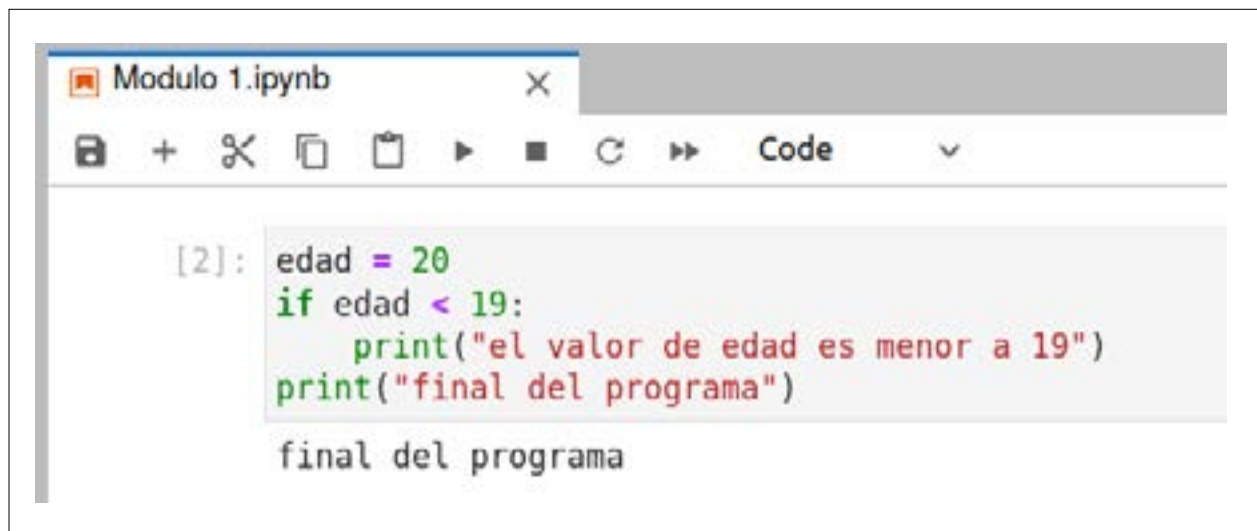
if edad < 19:

    print("el valor de edad es menor a 19")

print("final del programa")

>> final del programa
```

Figura 4



Fuente: propia

En este ejemplo, se ha definido una característica en forma de variable con la palabra edad, y se le ha asignado un valor numérico de tipo entero de 20. El resultado del programa será el mensaje "final del programa" puesto que la condición impuesta de un valor menor al número 19 no se cumple, pero esto lo estudiaremos más a fondo posteriormente. Por ahora considera que un programa estructurado o secuencial simplemente toma instrucciones de forma jerárquica y las va ejecutando una por una.

En el siguiente ejemplo, analiza un problema donde padre e hijo tienen cada uno un auto y se desea describir sus características como se muestra a continuación:

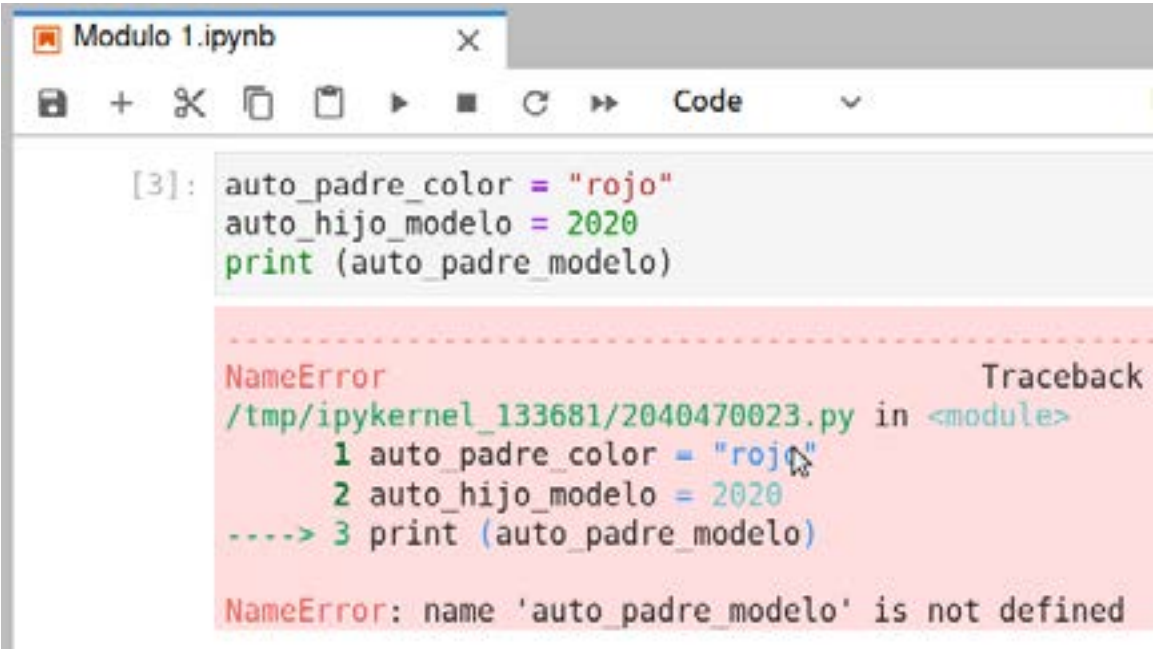
```
auto_padre_color = "rojo"

auto_hijo_modelo = 2020

print (auto_padre_modelo)

>> NameError: name 'auto_padre_modelo' is not defined
```

Figura 5



```
Modulo 1.ipynb x
+ ✂ 📄 ▶ ■ ↺ ⏏ Code ▾

[3]: auto_padre_color = "rojo"
      auto_hijo_modelo = 2020
      print (auto_padre_modelo)

-----
NameError                                Traceback
/tmp/ipykernel_133681/2040470023.py in <module>
      1 auto_padre_color = "rojo"
      2 auto_hijo_modelo = 2020
----> 3 print (auto_padre_modelo)

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

Fuente: propia

En este ejemplo, Python nos mostrará un error en el cual nos dice que la variable **auto_padre_modelo** no ha sido definida, y tiene razón!, puesto que aunque la variable ha sido definida para el **auto_hijo**, no lo ha sido para el auto padre y viceversa con la característica del color. ¿Te imaginas cuántas variables tendríamos que crear para describir las características de cada auto, no solo para este ejemplo, sino por ejemplo, para cada auto nuevo que ingrese a nuestro programa? Y si en algún momento se definiera una nueva característica para los autos, como un nuevo sistema de navegación, ¿Cuántas nuevas variables tendríamos que crear? Bueno, es aquí donde la programación orientada a objetos cobra sentido al ahorrarnos la creación de nuevas variables por medio de la capacidad de crear nuevos objetos a partir de una plantilla que se conoce con el nombre de Clase. De hecho, al crear variables donde definimos nombres, estamos creando nuevos objetos de tipo texto (**str**) y éstos están heredando todos sus atributos y métodos. Veamos un ejemplo:

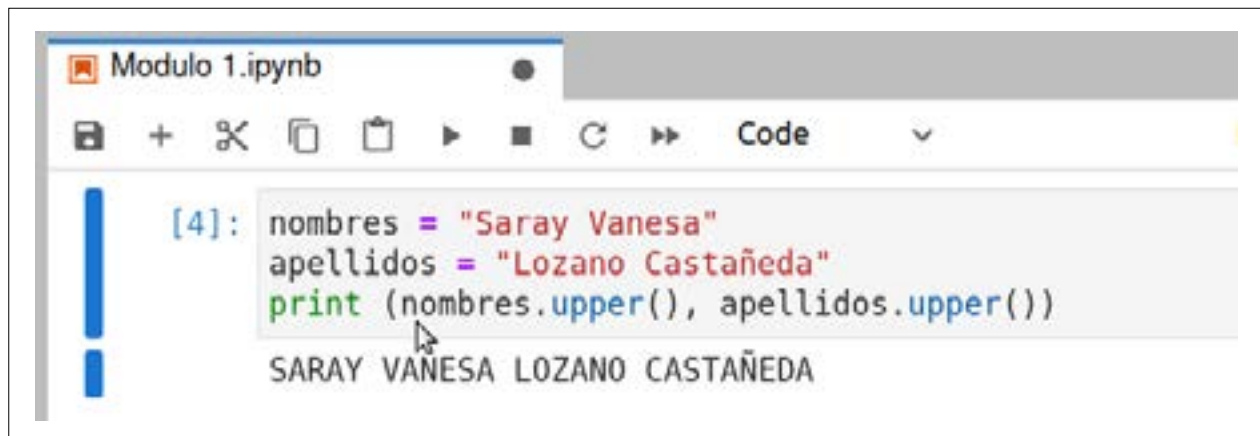
```
nombres = "Saray Vanesa"

apellidos = "Lozano Castañeda"

print ( nombres.upper( ), apellidos.upper( ) )

>> SARAY VANESA LOZANO CASTAÑEDA
```

Figura 6



The screenshot shows a Jupyter Notebook window titled 'Modulo 1.ipynb'. The code cell contains the following Python code:

```
[4]: nombres = "Saray Vanesa"
     apellidos = "Lozano Castañeda"
     print (nombres.upper(), apellidos.upper())
```

The output of the code is displayed below the code cell:

```
SARAY VANESA LOZANO CASTAÑEDA
```

Fuente: propia

Observa que al crear las variables nombres y apellidos, ambas tienen la capacidad de usar el método `.upper()` que transforma el texto en mayúsculas

En código, una "Clase" se crea de la siguiente manera (es muy importante que al digitar el siguiente ejemplo, se respeten los espacios, pues es una característica de Python denominada indentación)

```
class Auto: #definimos el nombre de la clase Auto

    """creamos los atributos de la clase, por defecto
    llevan la palabra reservada self"""

    def __init__(self, modelo, color):

        self.modelo = modelo

        self.color = color

    def ver_modelo(self):

        print(self.modelo)

    def ver_color(self):

        print(self.color)


#Creamos los objetos a partir de la clase Auto

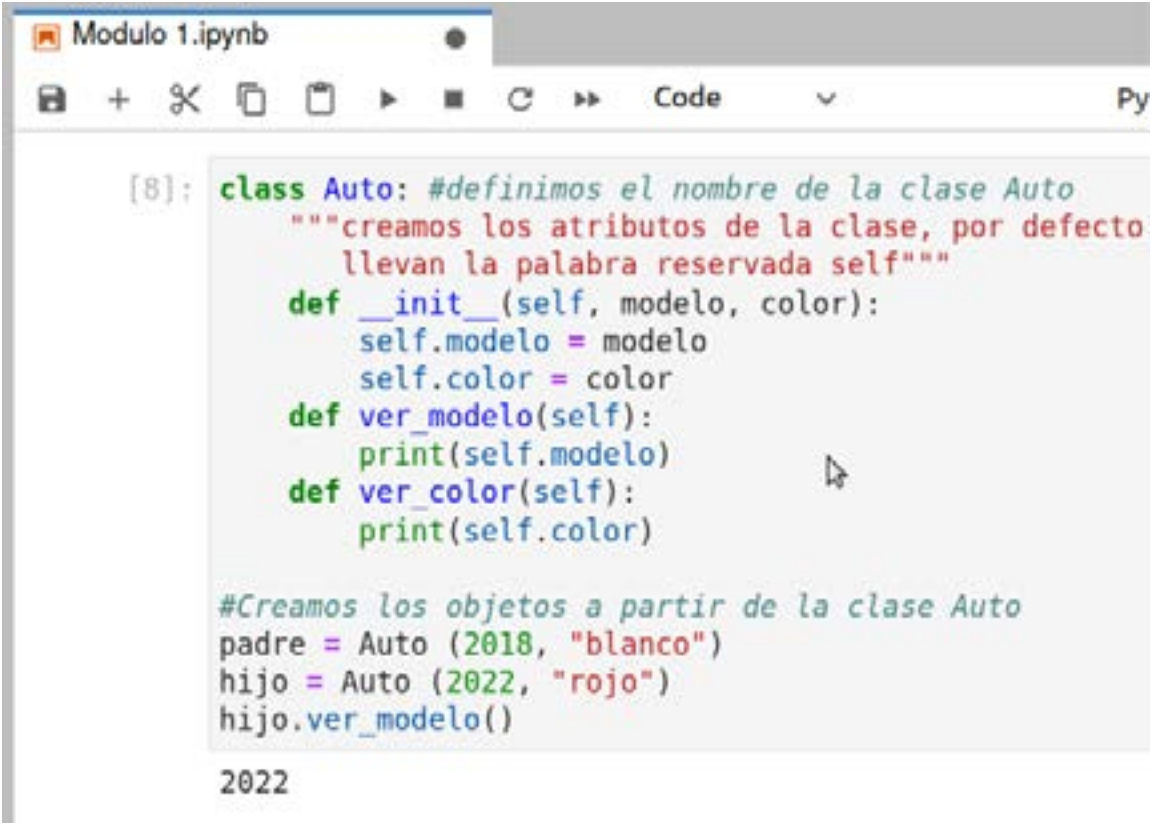
padre = Auto (2018, "blanco")

hijo = Auto (2022, "rojo")

hijo.ver_modelo()

>> 2022
```

Figura 7



```
[8]: class Auto: #definimos el nombre de la clase Auto
    """creamos los atributos de la clase, por defecto
    llevan la palabra reservada self"""
    def __init__(self, modelo, color):
        self.modelo = modelo
        self.color = color
    def ver_modelo(self):
        print(self.modelo)
    def ver_color(self):
        print(self.color)

#Creamos los objetos a partir de la clase Auto
padre = Auto (2018, "blanco")
hijo = Auto (2022, "rojo")
hijo.ver_modelo()

2022
```

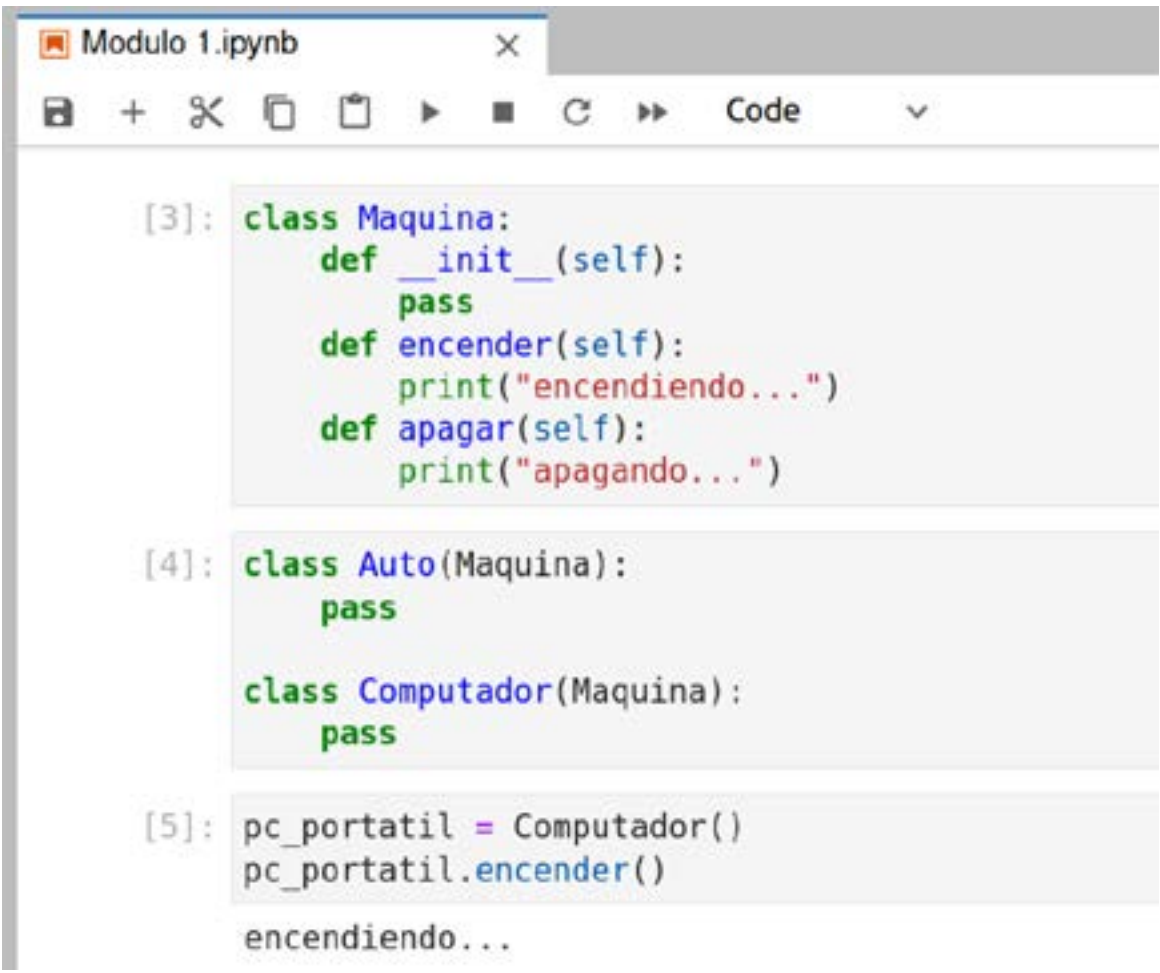
Fuente: propia

Observa que en este momento, la instrucción **hijo.ver_modelo()** da como resultado la impresión del número 2022, puesto que es un método que se definió en la clase Auto. Así mismo, si ahora se ejecutara la instrucción **padre.ver_modelo()**, el resultado debería ser el número 2018. En este código vemos que se ha creado la clase Auto con tres métodos: **__init__(self)** que es un método por defecto, un método que nos sirve para construir inicialmente los objetos de la clase y pasar atributos desde su creación, como sucede al decir que el auto del padre será de modelo 2018 y de color blanco. El método **ver_modelo(self)** que tiene como propósito mostrar como mensaje el valor contenido en el atributo modelo, y el método **ver_color(self)**. En la creación de clases y métodos, la palabra reservada **self** es obligatoria para que Python desarrolle los métodos correctamente. Como sugerencia, piensa en los métodos como acciones que debe realizar el objeto y de ahí que los nombres para mostrar información puedan ser palabras como mostrar o ver o imprimir.

Herencia

En programación, puedes crear clases de objetos que puedan heredar tanto los atributos como los métodos para no tener que definir nuevamente alguno de éstos. En el siguiente ejemplo verás la creación de una clase máquina y a partir de ésta, la creación de otras clases

Figura 8



```
Modulo 1.ipynb X
[3]: class Maquina:
      def __init__(self):
          pass
      def encender(self):
          print("encendiendo...")
      def apagar(self):
          print("apagando...")

[4]: class Auto(Maquina):
      pass

      class Computador(Maquina):
          pass

[5]: pc_portatil = Computador()
      pc_portatil.encender()

      encendiendo...
```

Fuente: propia

En este ejemplo, se destacan algunas cosas:

1. La palabra reservada **pass** se usa para dejar parte del código sin definir, como sucede en el método **`__init__(self)`**.
2. Al heredar, se usa el nombre de la clase de la cual se va a heredar, como sucede en el caso de la clase **`class Computador (Maquina)`** donde le estamos programando a la clase **`class Computador`**, todos los atributos y métodos que tenga la clase **`class Maquina`**, y así la clase **`Computador`** hereda el método **`encender`** de **`Maquina`** y por eso al crear el objeto **`pc_portatil`**, éste puede hacer uso de ese método por medio de la instrucción **`pc_portatil.encender()`**

Conclusiones

Python es un lenguaje orientado a objetos, por lo cual debe entenderse que al programar, se debe pensar en que cada instrucción probablemente tenga algún tipo de comportamiento especial que puede ser reutilizado, como por ejemplo, la acción de encendido o apagado de una máquina como un auto, o como una estufa, una nevera, un televisor, entre otros, o el abrir y cerrar de una puerta o ventana. Estas acciones son conocidas como métodos, y las características que puedan tener, como el material, tamaño o color, son conocidos como atributos.

Las variables cumplen condiciones para ser declaradas o creadas y pueden contener cualquier tipo de dato existente en el lenguaje Python. Estas reglas son:

- a. Pueden comenzar por una letra (a-z) (A-Z)
- b. Pueden comenzar con un guión bajo. ejemplo: **`_nombre`**
- c. Son case sensitive, lo cual indica que **`manzana`** es diferente de **`Manzana`** o **`MANZANA`** y todas sus posibles combinaciones de mayúsculas y minúsculas
- d. No pueden emplearse las palabras reservadas de Python como variables



Lectura recomendada

Para finalizar, te invito a realizar la siguiente lectura complementaria:

[¿Qué hay de nuevo en Python?: I](#)

INTRODUCCIÓN

Ahora es momento de conocer los tipos de datos que se manejan en Python, desde la metodología orientada a objetos e identificar las estructuras de control de flujo para que nuestros programas tomen decisiones. Esto se logrará reconociendo las diferentes formas de realizar comparaciones booleanas en programación y reconociendo los métodos más importantes de los tipos de datos.



Tipos de Datos en Python





Visitar página

Python maneja diversos tipos de datos como números, textos y conjuntos de datos especiales. Para un detalle completo de todos los tipos de datos que posee este lenguaje de programación, puede visitar el siguiente enlace web:

https://www.w3schools.com/python/python_datatypes.asp

Números Enteros (int)

Son todos los números tanto positivos como negativos que solo tienen una parte entera. Veamos algunos ejemplos:

```
a = 50
b = 3700
suma = a + b #la operación es 50 + 3700
print ( suma )
>> 3750
c = -5
d = -20
r = d - c #la operación es -20 - (-5)
print ( r )
>> -15
```

Números Decimales (float)

Son todos aquellos que tienen una parte entera y una decimal y pueden ser creados desde su declaración o asignación a una variable, o como resultado de una operación aritmética:

```
temperatura = 36.7 #creado al declarar esta variable
print ( temperatura )
>> 36.7

a = 10
b = 4

r = a / b #creado como resultado de una operación
aritmética
print ( r )
>> 2.5
```

Booleanos (bool)

Es un tipo de dato empleado para conocer si una condición es verdadera o falsa. Las variantes de este dato son dos y son las palabras True y False. Este tipo de dato también puede ser creado al asignarlo como valor a una variable o como resultado de una expresión aritmética:

```
#creado por asignación
umbral = True
print( umbral )
>> True

#creado como resultado de una expresión aritmética
umbral = 16 > 20
print( umbral )
>> False
```

Para este momento, es imprescindible mostrar que este tipo de dato puede ser el resultado de una expresión aritmética donde se pueden comparar datos. Por tanto voy a realizar el siguiente ejercicio para evidenciar el resultado de una comparación donde tengo 10 cuadernos y 30 maletas, y simplemente quiero que Python me diga si es cierto o falso si hay más cuadernos que maletas.

```
cuadernos = 10
maletas = 30
comparar = cuadernos > maletas
print(comparar)
>> False
```

Observa que el resultado es evidentemente falso, la expresión que mostraría un valor de True sería la siguiente

```
comparar = cuadernos < maletas
print(comparar)
>> True
```

Aquí se pueden utilizar los siguientes operadores de comparación, al comparar cantidades numéricas:

igual que ==

(note los dos símbolos del signo igual)

```
10 == 10
>> True
"hola" == "hola"
>> True
m = 30 == 40
print(m)
>> False
"hola" == "Hola"
>> False
```

Observa que en el caso de los textos, si hay diferencia de un hola con la h minúscula a un Hola con la H mayúscula.

mayor que >

```
10 > 10
>> False
m = 40 > 30
print(m)
>> True
```

mayor que o igual que >=

**(note que primero va el signo mayor que
y luego el signo igual que)**

```
10 >= 10
>> True
m = 30 >= 40
print(m)
>> False
```

menor que o igual que <=

```
10 <= 10
>> True
m = 40 <= 30
print(m)
>> False
```

menor que <

```
10 < 10
>> False
m = 30 <= 40
print(m)
>> True
```

diferente de !=

```
10 != 10
>> False
m = 30 != 40
print(m)
>> True
```


Condiciones Booleanas

También existen las comparaciones booleanas and, or y not donde se pueden comparar dos o más condiciones:

Condición or

Dará como resultado un valor True con que una sola comparación sea verdadera. Solo dará un valor de False si todas las comparaciones son falsas:

```
10 == 40 or 3 == 3 or "hola" == "Hola"
>> True #da True porque al menos 3 == 3 es True
10 == 40 or 3 <= 5
>> False #da False porque las dos comparaciones son Falsas
```

Condición and

Dará como resultado un valor de True sólo si todas las comparaciones son verdaderas, de lo contrario dará un valor de False

```
10 == 40 and 3 == 3
>> False
10 < 40 and 3 == 3 and "hola" != "Hola"
```

>> True #da True porque todas las comparaciones son verdaderas

Condición not (negación)

Existe una última condición conocida como la negación (not) la cual simplemente cambia el valor de True a False y viceversa:

```
100 > 90  
  
>>True  
  
not 100>90  
  
>>False
```

En este último ejemplo, es evidente que 100 es mayor que 90 y por eso el programa responde con un valor de **True**, pero al usar la condición not, el valor pasa a tomar el valor de **False**.

Letras y Textos (str)

Hasta este punto simplemente hemos visto tipos de datos que tienen la mayoría de los lenguajes de programación. A continuación, veremos tipos de datos que son propios de la programación orientada a objetos con Python y destacaremos que tienen algo denominado atributos y métodos, que palabras más, palabras menos son características y acciones.



str:

Tipo de dato en Python que maneja tanto caracteres como letras o símbolos como cadenas de texto. Todo lo que se encuentre entre comillas simples o dobles, será considerado de tipo texto. Ejemplo: "Hola mundo", 'clave 123', "#4567"

```
nombre = "Ciencia de Datos"
```

La anterior creación de la variable nombre tiene como tipo de dato un texto, que en Python es conocido como un String (**str**). Si utilizamos la función **type(nombre)** lo podemos comprobar:

```
nombre = "Ciencia de Datos"  
print( type ( nombre ) )  
  
>> "class str"
```

En Python, estos datos son objetos, y como todo objeto, se puede describir y puede realizar ciertas acciones. En este ejemplo vemos que el resultado de la función `print` son las palabras `'class str'` que quiere decir que la variable nombre contiene un dato que hereda como objeto de la clase String. En un ejemplo anterior se mostró el método `.upper()` y `.split()`. Los métodos que utiliza la clase String pueden ser consultados al emplear la función `dir()` de la siguiente forma:

```
print(dir(str))

>>['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Todos los métodos que tienen dos líneas al principio y al final, son heredados de un objeto principal, como el caso del método `__len__` empleado para conocer el tamaño de un objeto. Para usarlos, debe ponerse un paréntesis, por ejemplo, si deseamos saber cuántas letras tiene un texto en una variable, podemos hacer lo siguiente:

```
nombre = "Saray" #creamos la variable de tipo texto (str)
nombre.__len__() #usamos un método de la clase str para
contar sus letras

>> 5
```

Observe que el método `len` aparece en la lista como `'__len__'`, pero al usarlo, se emplea `.__len__()` donde se le ha puesto un punto después de la variable y un paréntesis al final.

De ahora en adelante, al presentar los métodos de una clase, omitiremos los métodos heredados y mostraremos los métodos que sólo pertenecen a dicha clase, como por ejemplo el método `isnumeric()`:

```
nombre.isnumeric()

>> False
```

En ese orden de ideas, los métodos propios de la clase `str` serían los siguientes:

```
'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill'
```

Si desea ver la ayuda de todos los métodos asociados, puede emplear la instrucción `help()` de la siguiente manera:

```
help( str )
>> Help on class str in module builtins:
class str(object)
|   str(object='') -> str
|   str(bytes_or_buffer[, encoding[, errors]]) -> str
|
|   Create a new string object from the given object.
If encoding or
|   errors is specified, then the object must expose a
data buffer
|   that will be decoded using the given encoding and
error handler.
|   Otherwise, returns the result of object.__str__()
(if defined)
|   or repr(object).
|   encoding defaults to sys.getdefaultencoding().
|   errors defaults to 'strict'.
|
|   Methods defined here:
|
|   __add__(self, value, /)
|       Return self+value.
|
|   __contains__(self, key, /)
|       Return key in self.
|
|   __eq__(self, value, /)
|       Return self==value.
```

No se presenta todo el código, pero puede ejecutar la línea de código para ver el resultado completo

Si se desea ver una ayuda en concreto, puede utilizar la instrucción completa del método que desea ver, por ejemplo, si deseamos ver la ayuda del método **.upper()** podemos hacer la siguiente línea de código:

```
help(str.upper)

>>Help on method_descriptor:

upper(self, /)

    Return a copy of the string converted to uppercase.
```

Listas (list)

Las listas son un tipo de datos de colección ordenada y son el equivalente a los llamados arrays o vectores que existen en otros lenguajes de programación (Gonzalez, n.d) Son arreglos de datos que pueden contener otros datos como números, texto, e incluso otras listas. Estos se crean encerrando los datos entre paréntesis cuadrados. Veamos un ejemplo:

```
datos = [ 10, 11, 13, "Ciencia", [ 23, 11 ] ]
```



list:

Tipo de dato en Python que maneja datos de forma organizada por medio de índices. Su similitud con otros lenguajes de programación son los vectores o filas de datos. en Python, este tipo de dato se debe encontrar entre paréntesis cuadrado. Ejemplo: [34,67,33,"dia","fibra"]

Para acceder a los datos de una lista, se puede hacer uso de los índices, teniendo en cuenta que éstos empiezan desde el número cero. Si queremos, por ejemplo, que se imprima la palabra Ciencia, el código sería el siguiente:

```
print ( datos[ 3 ] )

>> Ciencia
```

Pero debemos tener mucho cuidado de no utilizar un índice que no existe, por ejemplo, si utilizamos el índice con valor 20, obtendremos un error:

```
print ( datos[ 20 ] )  
  
>> IndexError: list index out of range
```

De igual forma sucederá si intentamos agregar un nuevo elemento, pues los índices en este caso sólo servirán si ya existen. De lo contrario, tendremos que usar métodos especiales para agregar más datos.

```
datos = [ 10, 11, 13, "Ciencia", [ 23, 11 ] ]  
datos[0] = 17  
print( datos )  
  
>>[17, 11, 13, 'Ciencia', [23, 11]]
```

Como vemos, aunque el dato de índice cero contenía originalmente el valor numérico 10, pudo ser reemplazado por el valor numérico 17. Si lo que deseamos es agregar un elemento nuevo sin que se pierdan los anteriores, puede emplearse el método `.append()` como se muestra a continuación:

```
datos = [ 10, 11, 13, "Ciencia", [ 23, 11 ] ]  
datos.append( 17 )  
print( datos )  
  
>>[10, 11, 13, 'Ciencia', [23, 11], 17]
```

Esto hará que los datos nuevos que se agreguen, queden al final de la lista. Estos datos también soportan el doble indexado, por ejemplo, si deseamos imprimir el número 23 el indexado sería el siguiente

```
print( datos[4][0] )  
>> 23
```

Los métodos que podemos emplear con las listas son los siguientes:

```
'append', 'clear', 'copy', 'count',  
'extend', 'index', 'insert', 'pop',  
'remove', 'reverse', 'sort'
```

Como ejemplo, observe el uso del método sort:

```
lista = [4,5,12,3,9,30,11]    #se crea una lista  
lista.sort()                 #se emplea el método sort para ordenarla  
print(lista)  
>> [3, 4, 5, 9, 11, 12, 30]
```


Tuplas (tuple)

Las tuplas son similares a las listas, se crean con los paréntesis curvos, pero con la excepción que al ser creadas, sus valores iniciales no pueden ser cambiados por indexación:

```
datos = ( 15, 16, 17 )  
datos[2] = 20  
>>TypeError: 'tuple' object does not support item assignment
```

Debido a la característica de no poder cambiar sus valores iniciales, los métodos que se emplean con las tuplas son los siguientes:

```
'count', 'index'
```

Diccionarios (dict)

Los diccionarios tienen la capacidad de utilizar índices en forma de texto, llamados claves o llaves y funcionan como un par { "clave" : valor }. Veamos:

```
datos = { "enero":31, "febrero":28,  
"marzo":31 }  
print( datos[ "marzo" ] )  
>> 31
```



dict:

Tipo de dato en Python que permite la organización de los datos a partir de un índice de tipo texto al cual se le llama clave. Se debe encontrar entre corchetes. Ejemplo: {"OS": "Linux_UbuntuMate"}

Estos datos, también soportan múltiples datos como otros números en su valor, tuplas, listas, y otros diccionarios. Si desea agregar nuevos datos o cambiar los existentes, se hace por medio de los índices de la siguiente forma:

```
datos["abril"] = 30  
  
print(datos)  
  
>> "enero":31, "febrero":28, "marzo":31, "abril":30
```

Observa que he agregado el valor de 30 al índice "abril" del diccionario datos. Esta característica impide que los diccionarios pudiesen tener índices duplicados, es algo que no va a pasar.

Los métodos que se emplean en los diccionarios son los siguientes:

```
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',  
'popitem', 'setdefault', 'update', 'values'
```

Estructuras de control de flujo

¡Es el momento de ver cómo toman decisiones los programas!

Si por ejemplo, queremos validar un mensaje para un número que no debe ser ingresado, por ejemplo, que la edad de un cliente no pueda ser menor al número 18, sino que por el contrario siempre tiene que ser mayor o igual que esta cifra, una posible solución en alguna parte de nuestro código sería la siguiente. Haz la prueba:

```
edad_cliente = 20  
edad_permitida = 18  
  
if edad_cliente >= edad_permitida:  
    print("permitido... puede continuar")  
else:  
    print("No permitido... ERROR..")
```

Aquí, la palabra reservada **if** nos permite decirle al programa que tome una decisión con base en una operación de comparación, que de dar como resultado un valor de **True**, emitirá el mensaje "permitido... puede continuar". Pero si por el contrario, la comparación da un valor de **False**, se ejecutará la parte del programa que se encuentre después de la palabra reservada **else**. Te invito a experimentar con diferentes valores numéricos, comparaciones e incluso tipos de datos.



if:

Palabra reservada en Python para crear bloques de código que se ejecutarán o no de acuerdo al resultado de comparaciones o condiciones booleanas

Condiciones múltiples y anidación

Considera el caso en el que necesitas validar más de una opción, sea con la misma variable o con diferentes variables. Veamos cómo podríamos solucionar un ingreso de documento y contraseña para ingresar a un sistema. Presentaré a manera de ejemplo, el cual te invito a desarrollar, una parte del código:

```
#creamos las variables necesarias para nuestro programa
doc_cliente = 1234
clave_cliente = "holamundo"
doc_ingresado = 1234
clave_ingresada = "secret"

#creamos la estructura de control con dos instrucciones if
#Esto se conoce como anidación
if doc_cliente == doc_ingresado:
    if clave_cliente == clave_ingresada:
        print("acceso permitido... puede continuar")
    else:
        print("El documento o la contraseña son incorrectos")
```

Mira que, para este ejemplo, primero hemos creado las variables para realizar las comparaciones respectivas, y luego, nuestra estructura de control validará si el documento del cliente es igual al documento ingresado, y acto seguido validará si la clave del cliente es igual que la clave ingresada, y en este ejemplo, al ser diferentes, lo que ocurrirá es que el mensaje mostrado será el de "El documento o la contraseña son incorrectos" ¿Te atreves a mejorar el programa usando las condiciones booleanas?

Ahora considera que tenemos que validar ciertos rangos de números, por ejemplo, que el programa pueda clasificar los años y decir que la cifra 1995 pertenece a los 90, que la cifra 1987, pertenece a los 80's y así sucesivamente. Una posible solución son los múltiples if (**elif**)

```
fecha = 1995

if fecha >= 1970 and fecha < 1980:
    print(fecha, "pertenece a los 70's")
elif fecha >= 1980 and fecha < 1990:
    print(fecha, "pertenece a los 80's")
elif fecha >= 1990 and fecha < 2000:
    print(fecha, "pertenece a los 90's")
else:
    print("Aún no programado...")

>>1995 pertenece a los 90's
```

Aquí, se va evaluando la condición hasta encontrar una coincidencia **True**, por lo que no todas las comparaciones serán evaluadas. Esto hará que pueda ejecutarse un programa con mayor o menor velocidad, puesto que depende del momento en que encuentre una comparación verdadera ¿Te atreves a hacer un programa que clasifique las notas obtenidas por un estudiante de la siguiente manera?

Si el estudiante obtiene una nota entre 1.0 y 2.9 el programa muestra el mensaje de reprobado.

Si el estudiante obtiene una nota entre 3.0 y 4.5 el programa muestra el mensaje de aprobado.

Si el estudiante obtiene una nota entre 4.6 y 4.9 el programa muestra el mensaje de aprobado con mérito.

Si el estudiante obtiene una nota de 5.0 el programa muestra el mensaje de aprobado con honores.

Bucles o ciclos

Los bucles o ciclos son estructuras de control cuando se desea que cierta parte de nuestro código se repita ya sea un número limitado de veces, o hasta que se cumpla cierta condición o comparación, o simplemente de forma ilimitada. Para esto también se emplean las comparaciones y en Python se realiza de la siguiente manera:



while:

Palabra reservada en Python para crear bloques de código que se ejecutarán o no y repetirán de acuerdo a comparaciones o condiciones booleanas.

Bucle while

Esta estructura de control permite que una parte del código se repita o de forma ilimitada o hasta que una comparación de como resultado un valor de False

#No usar este ejemplo

```
while True:
    print("Enviando mensajes por siempre")
```

Esta forma de emplear el ciclo es un ejemplo que se repetirá por siempre y tiene sus ventajas y desventajas. En este caso será una desventaja porque consumirá todos los recursos de cómputo de la máquina. Una forma segura de usarlo sería la siguiente:

```
import time as t
while True:
    print("mensaje")
    t.sleep(1)
```

La primera línea, **import time as t**, trae a nuestro programa una clase llamada **time** y de esta empleamos el método **sleep**, como se muestra en el código, para simplemente controlar la velocidad y no saturar los recursos de cómputo. El programa deberá detenerse manualmente con el botón de detener. La opción **as t** es para hacer un

sobrenombre o alias de la clase **time** y llamarla simplemente **t**

Esto se verá durante el resto del curso al emplear librerías como numpy o pandas.

Otra opción sin usar librerías es la siguiente:

```
a=0
while a<5:
    print("Enviando mensajes por siempre")
    a=a+1
```

En este ejemplo, el programa inicia con el valor de *a* igual a cero y la estructura *while* simplemente se ejecutará hasta que la comparación se vuelva *False*, en otras palabras, se ejecutará mientras la comparación sea verdadera y eso es mientras el valor de *a* sea menor que 10

Bucle for

Esta estructura de control sirve para que los bloques de código se repitan un número limitado de veces. Veamos cómo:



for:

Palabra reservada en Python para crear bloques de código que se ejecutarán una determinada cantidad de veces

Figura 1

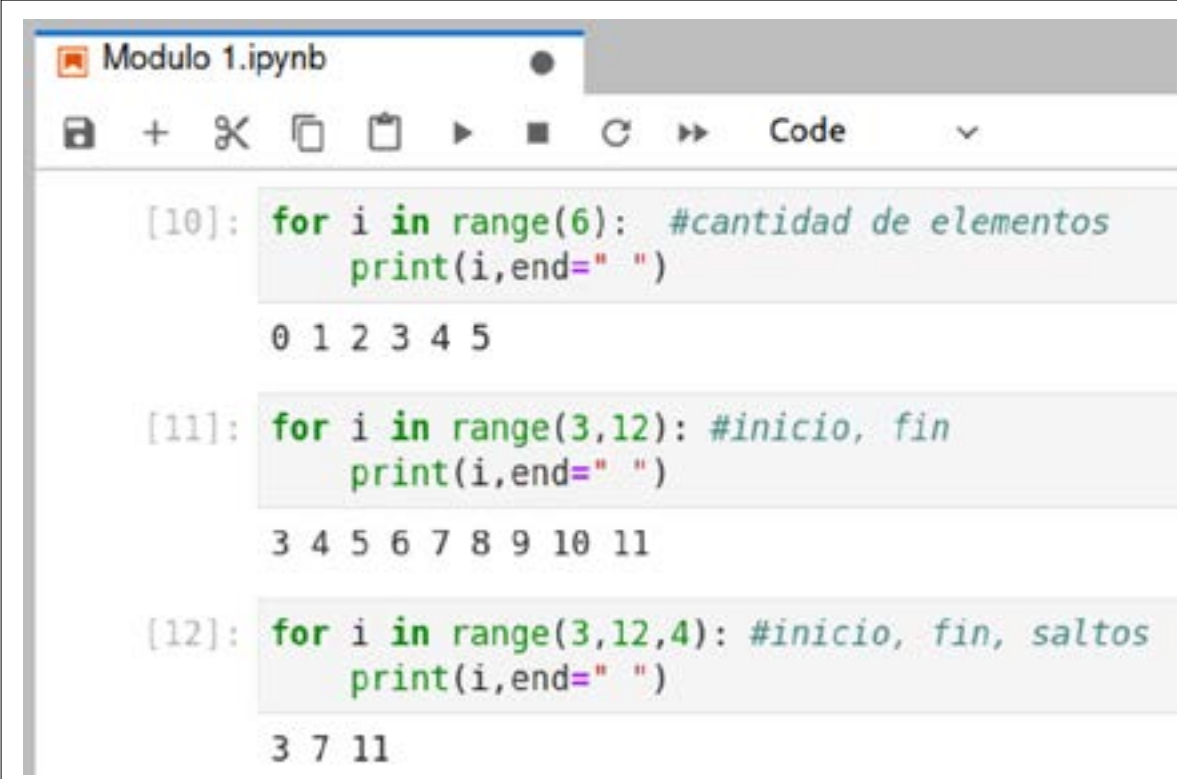
```
Modulo 1.ipynb
[1]: datos = [0,1,2,3,4,5,6,7,8,9]
    for i in datos:
        print(i)
0
1
2
3
4
5
6
7
8
9
```

Fuente: propia

Aquí vemos algo interesante. La letra *i* sirve como una variable temporal dentro del ciclo que permite almacenar valores momentáneamente. Por otro lado, con la palabra reservada **in**, le decimos a Python que itere en el objeto datos por lo cual ese objeto debe ser iterable, es decir, debe ser un conjunto de datos como una lista, una tupla o un diccionario

Existe un tipo de dato que se emplea muy a menudo y es el objeto **range()**, el cual puede crear un conjunto de datos con ciertas condiciones específicas. Veamos ejemplos:

Figura 10



```
Modulo 1.ipynb

[10]: for i in range(6): #cantidad de elementos
      print(i,end=" ")
      0 1 2 3 4 5

[11]: for i in range(3,12): #inicio, fin
      print(i,end=" ")
      3 4 5 6 7 8 9 10 11

[12]: for i in range(3,12,4): #inicio, fin, saltos
      print(i,end=" ")
      3 7 11
```

Fuente: propia

Por simplicidad, hemos puesto a la función **print()**, un argumento **end = " "** para que pueda mostrar los elementos de forma horizontal.

Observa que cuando el objeto **range()** tiene un solo argumento, en este caso el número 6, crea 6 números que son los que se imprimen dentro del ciclo. Cuando tiene dos argumentos, Python lo toma como el inicio y el final de una serie de números y cuando tiene tres números, lo toma como el inicio, el final, y la cantidad de saltos que debe dar para el siguiente número.

Conclusiones

Python implementa tipos de datos que son conjuntos de números como listas, tuplas y diccionarios, los cuales son los más empleados a la hora de compartir información entre aplicaciones y similitud con un formato conocido como JSON.

Recordemos que las listas y los diccionarios son indexadas y pueden cambiar sus valores iniciales, mientras que las tuplas, aunque son indexadas, no pueden cambiar sus valores iniciales. En los diccionarios, los índices son conocidos como claves y tienen la característica de referenciar un valor por cada clave, el cual puede ser cualquier otro tipo de dato existente en Python.

Las estructuras de control son fundamentales para que el programa tome decisiones con base en comparaciones que retornan siempre valores booleanos de verdadero o falso (True, False) lo cual permitirá diseñar la lógica de cualquier programa, también conocido como algoritmo.



Lectura recomendada

Para finalizar esta unidad te invito a realizar la siguiente lectura:

- Modelos de datos en Python:
<https://docs.python.org/es/3/reference/datamodel.html>
- Tutorial de Python:
<https://docs.python.org/es/3.8/tutorial/index.html>

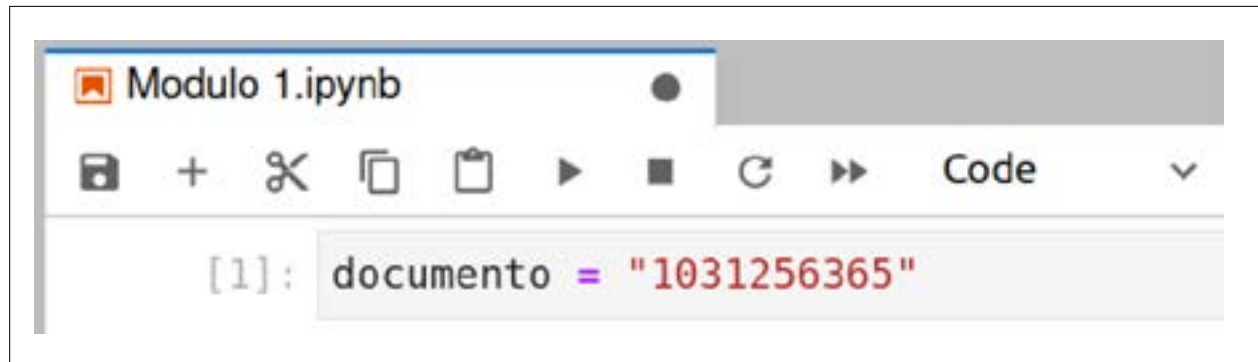


Pasos - procesos



Usualmente los datos se almacenan temporalmente en variables cuando se carga un programa, un ejemplo de esto se puede ver a continuación:

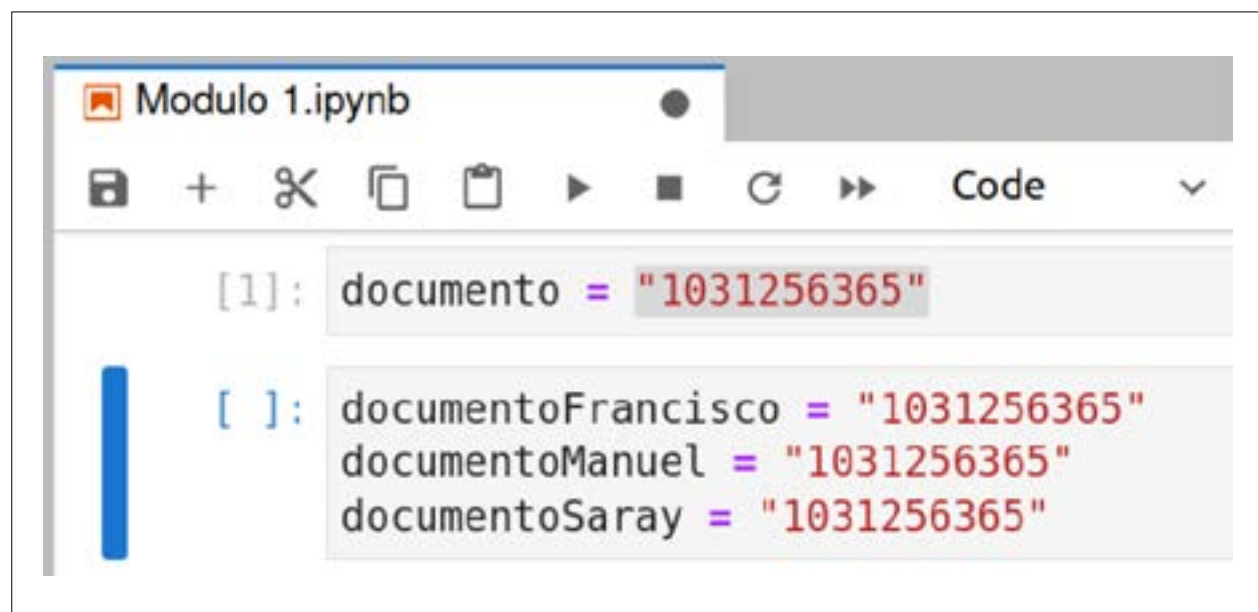
Figura 1



The screenshot shows a Jupyter Notebook interface with a tab labeled 'Modulo 1.ipynb'. Below the tab is a toolbar with icons for saving, adding, deleting, copying, pasting, running, and other actions. The main area contains a single code cell with the following text: `[1]: documento = "1031256365"`. The text is in a monospaced font, with the string value in red.

Ahora imagina que cada vez que se necesite asignar un documento a un usuario, tendríamos que crear nuevas variables:

Figura 2



The screenshot shows a Jupyter Notebook interface with a tab labeled 'Modulo 1.ipynb'. Below the tab is a toolbar with icons for saving, adding, deleting, copying, pasting, running, and other actions. The main area contains a code cell with the following text: `[1]: documento = "1031256365"` followed by a blank line and then `[]: documentoFrancisco = "1031256365"`, `documentoManuel = "1031256365"`, and `documentoSaray = "1031256365"`. The text is in a monospaced font, with the string values in red. A blue vertical bar is visible on the left side of the code cell.

Y si existiesen varios usuarios con nombres iguales pero diferentes apellidos, y se crearán miles de usuarios, ¿Qué podríamos hacer? Además, si en algún momento se generara un nuevo campo como por ejemplo, un número de celular que antes no existía como dato, tendrías que crear esas variables. Tal vez en este punto podrías pensar en una solución más óptima usando listas y diccionarios o una combinación de ambos. Lo importante aquí es reconocer cómo podría Saray resolver este problema y para eso ella recurrirá a la programación orientada a objetos

```
class persona:
    def __init__(self,nombre,id):
        self.nombre=nombre.upper()
        self.id=id
        self.rol=""
    def verDatos(self):
        pass
    def actualizarDatos(self):
        pass

class estudiante(persona):
    self.rol="estudiante"

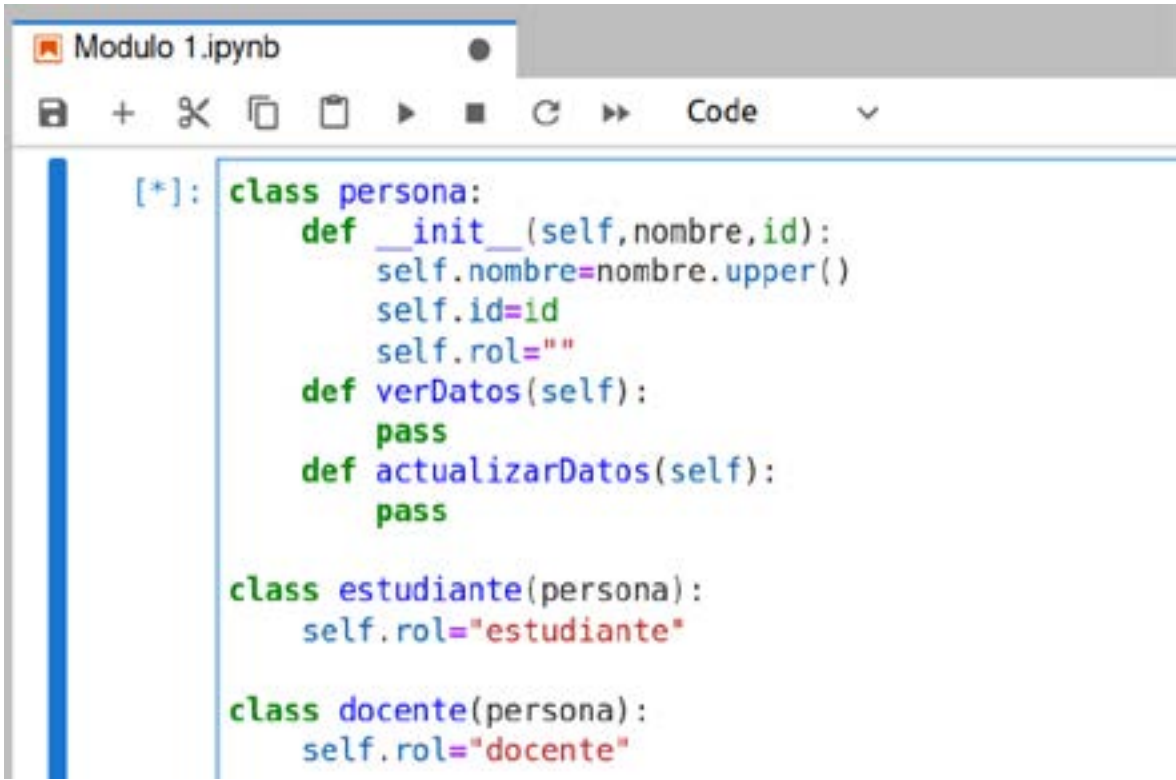
class docente(persona):
    self.rol="docente"

usuarios = []

usuarios.append(estudiante("gabriel","1031998656"))
usuarios.append(estudiante("Felipe", "1013568989"))
usuarios.append(docente("Andrés", "196589"))

print(usuarios)
```

Figura 3



```
[*]: class persona:
    def __init__(self,nombre,id):
        self.nombre=nombre.upper()
        self.id=id
        self.rol=""
    def verDatos(self):
        pass
    def actualizarDatos(self):
        pass

class estudiante(persona):
    self.rol="estudiante"

class docente(persona):
    self.rol="docente"
```

Analiza que cuando se vayan a crear los objetos, éstos heredarán los atributos y métodos de la clase padre o principal que en este caso es la clase persona. Así podrás plantear lo que los datos de todos los usuarios deberían tener y posteriormente definir los métodos para actualizar los datos o verlos o cualquier otro comportamiento.

BIBLIOGRAFÍA

García, J. (2018). Ciencia De Datos. Técnicas Analíticas Y Aprendizaje Estadístico. Un Enfoque Práctico - Jesús García.pdf [6ng22yvvd2lv]. Idoc. pub. Recuperado de <https://idoc.pub/documents/idocpub-6ng22yvvd2lv>.

Python Tutorial. W3schools.com. (2022). Retrieved 31 August 2022, from <https://www.w3schools.com/python/default.asp>.