

**UNIVERSIDAD TECNOLÓGICA DE**

**SAN LUIS RIO COLORADO**

**Trabajo de Investigación**

**MTRO. MIGUEL ANGEL RODRIGUEZ**

**ALUMNO: VICTOR MANUEL GALVAN COVARRUBIAS**

San Luis Rio Colorado, Sonora Enero, 2021

**Tipos de Datos Primitivos Simples**

**Números** (numbers): Secuencia de dígitos (pueden incluir el - para negativos y el. para decimales) que representan números.

Ejemplo. 0, -1, 3.1415.

**Cadenas** (strings): Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles.

Ejemplo. ‘Hola’, “Adiós”.

**Booleanos** (boolean): Contiene únicamente dos elementos True y False que representan los valores lógicos verdadero y falso respectivamente.

Estos datos son inmutables, es decir, su valor es constante y no puede cambiar.

**Tipos de datos primitivos compuestos (contenedores)**

**Listas** (lists): Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. Se representan con corchetes y los elementos se separan por comas.

Ejemplo. [1, “dos”, [3, 4], True].

**Tuplas** (tuples). Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. A diferencia de las listas son inmutables, es decir, que no cambian durante la ejecución. Se representan mediante paréntesis y los elementos se separan por comas.

Ejemplo. (1, ‘dos’, 3)

**Diccionarios** (dictionaries): Colecciones de objetos con una clave asociada. Se representan con llaves, los pares separados por comas y cada par contiene una clave y un objeto asociado separados por dos puntos.

Ejemplo. {‘pi’:3.1416, ‘e’:2.718}.

**Clase de un dato (type())**

La clase a la que pertenece un dato se obtiene con el comando type()

>>> type(1)

<class 'int'>

>>> type("Hola")

<class 'str'>

>>> type([1, "dos", [3, 4], True])

<class 'list'>

>>>type({'pi':3.1416, 'e':2.718})

<class 'dict'>

>>>type((1, 'dos', 3))

<class 'tuple'>

**Números (clases int y float)**

Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números. Pueden ser enteros (int) o reales (float).

>>> type(1)

<class 'int'>

>>> type(-2)

<class 'int'>

>>> type(2.3)

<class 'float'>

**Operadores aritméticos**

Operadores aritméticos: + (suma), - (resta), \* (producto), / (cociente), // (cociente división entera), % (resto división entera), \*\* (potencia).

**Orden de prioridad de evaluación:**

|  |  |
| --- | --- |
| 1 | Funciones predefinidas |
| 2 | Potencias |
| 3 | Productos y cocientes |
| 4 | Sumas y restas |

Se puede saltar el orden de evaluación utilizando paréntesis ( ).

>>> 2+3

5

>>> 5\*-2

-10

>>> 5/2

2.5

>>> 5//2

2

>>> (2+3)\*\*2

25

**Operadores lógicos con números**

Devuelven un valor lógico o booleano.

Operadores lógicos: == (igual que), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), != (distinto de).

>>> 3==3

True

>>> 3.1<=3

False

>>> -1!=1

True

**Cadenas (clase str)**

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas ' o dobles “.

'Python'

"123"

'True'

# Cadena vacía

''

# Cadena con un espacio en blanco

' '

# Cambio de línea

'\n'

# Tabulador

'\t'

**Secuencias de escape en Python**

Las secuencias de escape nos dejan incrustar caracteres en cadenas que ni pueden ser fácilmente ingresadas desde un teclado. El carácter \ y uno o más caracteres seguidos a este son cambiados por un único carácter en la cadena final.

Por ejemplo, aquí tenemos una cadena que tiene dos saltos de línea.

>>> a = 'a\nb\nc'

>>> a

'a\nb\nc'

>>> print(a)

a

b

c

>>>

En este ejemplo hemos usado la secuencia de escape para el salto de línea que viene a ser representada por la barra invertida "\" más el carácter "n" (\n).

Lista de secuencias de escape en Python

**Backslash (guarda solo un \)**

\\

**Comilla simple (guarda ')**

\'

**Comilla doble (guarda ")**

\"

**Bell**

\a

**Retroceso**

\b

>>> a = 'abcd\bef'

>>> print(a)

abcef

**Formfeed**

\f

>>> a = 'abcd\fef'

>>> print(a)

abcd

ef

**Nueva línea**

\n

>>> a = 'abcd\nef'

>>> print(a)

abcd

ef

**Carriage return**

\r

>>> a = 'abcd\ref'

>>> print(a)

efcd

**Tab horizontal**

\t

>>> a = 'abcd\tef'

>>> print(a)

abcd ef

**Tab vertical**

\v

>>> a = 'abcd\vef'

>>> print(a)

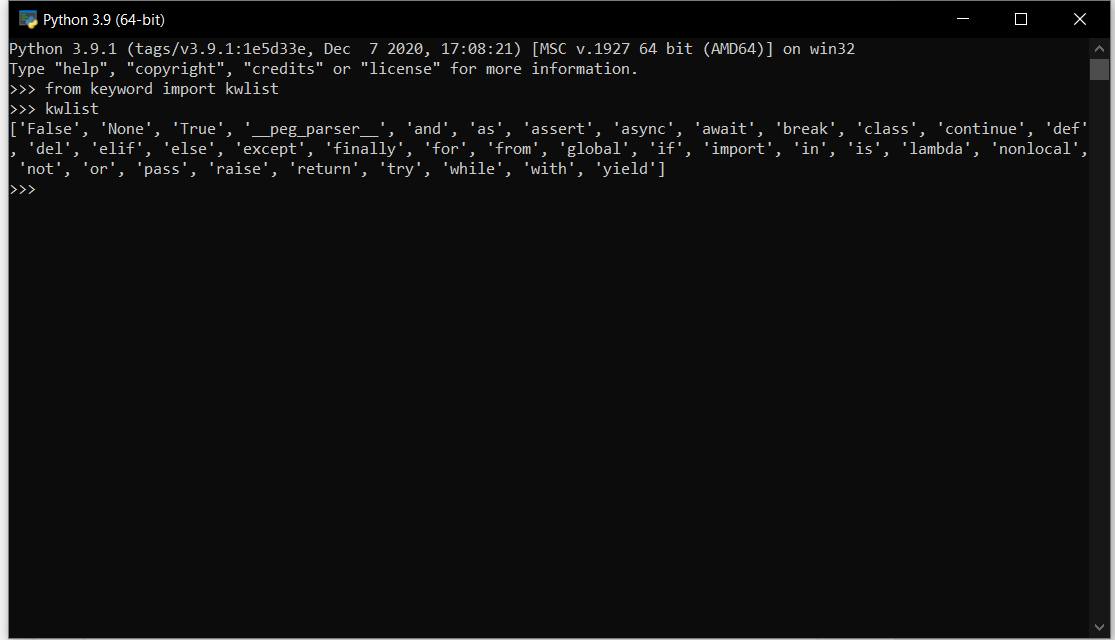
abcd

ef

**Estos son las secuencias más comunes que podemos encontrar en Python.**

**Palabras reservadas en Python**

**Para mostrar las palabras reservadas solo debemos ingresar las instrucciones mostradas en la imagen de abajo.**



|  |  |  |
| --- | --- | --- |
| **False** | **Para indicar que algo es verdadero.** |  |
| **True** | **Para indicar que algo es falso.** |  |
| **None** | **Es utilizado para indicar la ausencia de un valor en un objeto determinado.** |  |
| **and / or** | **Representaciones lógicas de «y» y «ó», respectivamente.** | **>>> True and False**  **False**  **>>> True and True**  **True**  **>>> False and False**  **False**  **>>> True or False**  **True**  **>>> True or True**  **True**  **>>> False or False**  **False** |
| **as** | **Su función es dual. Puede utilizarse al momento de importar un determinado objeto de un módulo, para asignar un nombre diferente.** | **from keyword import kwlist as keyword\_list** |
| **assert** | **Durante la depuración (si \_\_debug\_\_ es True), assert permite especificar una expresión que lanzará AssertionError en caso de ser evaluada como falsa.** | **>>> assert 1 == 2**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in <module>**  **AssertionError**  **>>> assert 1 == 1** |
| **async** | **Se emplea para definir una función como asincrónica.** | **async def f():**  **pass** |
| **await** | **Solo puede ser invocado desde una función asincrónica y sobre otra función de esta índole. Pausa la ejecución de aquella hasta que el resultado de esta esté disponible.** | **import asyncio**  **async def f():**  **await asyncio.sleep(1)** |
| **continue** | **Detener completamente un bucle (break) o detener únicamente la iteración actual y saltar a la siguiente (continúe).** |  |
| **break** |  |
| **class** | **Define una clase.** | **>>> class C:**  **... def \_\_init\_\_(self):**  **... print("Hello world!")**  **...**  **>>> c = C()**  **Hello world!** |
| **def** | **Define una función.** | **>>> def func(a):**  **... print(a)**  **...**  **>>> func("Hello world!")**  **Hello world!** |
| **del** | **En caso de utilizarse seguido de un objeto, elimina su referencia.** | **>>> a = 1**  **>>> a**  **1**  **>>> del a**  **>>> a**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in <module>**  **NameError: name 'a' is not defined** |
| **En diccionarios, remueve el elemento del mismo. En listas, además, reordena el conjunto de elementos para llenar el espacio vacío.** | **>>> d = {"Yes": 1, "No": 2}**  **>>> d["Yes"]**  **1**  **>>> del d["Yes"]**  **>>> d["Yes"]**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in <module>**  **KeyError: 'Yes'**  **>>> d**  **{'No': 2}** |
| **if / elif / else** | **Instrucciones permiten condicionar la ejecución de uno o varios bloques de sentencias al cumplimiento de una o varias condiciones.** | **if a:**  **# bool(a) es True**  **print("a")**  **elif b:**  **# bool(a) es False**  **# bool(b) es True**  **print("b")**  **elif c:**  **# bool(a) es False**  **# bool(b) es False**  **# bool(c) es True**  **print("c")**  **else:**  **# Todas las anteriores son falsas.**  **print("Ninguna de las anteriores.")** |
| **try / except / else / finally** | **Las cláusula try / except permite capturar una excepción dentro de una determinada porción de código.** | **try:**  **func()**  **except Exception:**  **print("Ha ocurrido un error.")** |
| **En caso de no haberse propagado ninguna excepción, el flujo del programa seguirá luego del término else, en caso de encontrarse, al finalizar la ejecución de func()** | **else:**  **print("Ejecutado exitosamente.")** |
| **Por último, el flujo será enviado luego del término finally al finalizar la ejecución del código anterior, independientemente si han surgido errores.** | **finally:**  **cleanup()** |
| **En caso de ocurrir un error en las cláusulas except o else, cleanup() será llamada antes de propagarse la excepción, para permitir realizar una limpieza, principal diferencia con el siguiente código.** | **try:**  **func()**  **except Exception:**  **print("Ha ocurrido un error.")**  **else:**  **print("Ejecutado exitosamente.")**  **cleanup()** |
| **for** | **Permite recorrer los elementos de un objeto iterable (listas, tuplas, diccionarios, etc.).** | **>>> for i in (1, 2, 3):**  **... print(i)**  **...**  **1**  **2**  **3** |
| **from** | **Importa uno o más objetos de un módulo, prescindiendo del nombre de éste como prefijo al ser utilizado.** | **>>> from keyword import kwlist, iskeyword**  **>>> iskeyword("assert")**  **True** |
| **global** | **Permite modificar el valor de un objeto global desde un ámbito con menor alcance (una función, por ejemplo).** | **>>> a = 1**  **>>> def f():**  **... global a**  **... a = 2**  **...**  **>>> f()**  **>>> a**  **2** |
| **import** | **Importa un módulo, o bien un objeto de éste si es utilizado junto a from.** | **>>> import keyword**  **>>> keyword.iskeyword("assert")**  **True** |
| **in** | **Determina la existencia de un determinado valor dentro de una lista, tupla, diccionario o cualquier objeto iterable.** | **>>> 2 in (1, 2, 3)**  **True**  **>>> "Hello" in "Hello, world!"**  **True**  **>>> "Si" in {"Yes": 1, "No": 2}**  **False** |
| **is** | **Determina si dos objetos son iguales, a diferencia de los signos de equivalencia (==), que determinan si ambos tienen el mismo valor.** | **>>> class Number:**  **... def \_\_init\_\_(self, number):**  **... self.number = number**  **... def \_\_eq\_\_(self, other):**  **... return self.number == other**  **...**  **>>> a = Number(1)**  **>>> b = Number(1)**  **>>> a == b**  **True**  **>>> a is b**  **False**  **>>> c = b**  **>>> b is c**  **True** |
| **lambda** | **Se trata de crear funciones de manera rápida, just in time, sobre la marcha, para prototipos ligeros que requieren únicamente de una pequeña operación o comprobación. Por lo tanto, toda función lambda también puede expresarse como una convencional (pero no viceversa).** | Función lambda |
| **nonlocal** | **Permite modificar el valor de un objeto que ha sido creado en un ámbito anterior.** | **>>> def a():**  **... i = 1**  **... def b():**  **... nonlocal i**  **... i = 2**  **... b()**  **... print(i)**  **...**  **>>> a()**  **2** |
| **not** | **Operador lógico «no».** | **>>> not True**  **False**  **>>> not False**  **True** |
| **pass** | **Esta palabra reservada carece de función alguna. Es utilizada para rellenar espacios requeridos por Python para evitar errores de sintaxis.** | **>>> def f():**  **... pass**  **...**  **>>> f()** |
| **raise** | **Lanza una excepción.** | **>>> raise NameError**  **Traceback (most recent call last):**  **File "<stdin>", line 1, in <module>**  **NameError** |
| **return** | **Dentro de una función, especifica el valor de retorno.** | **>>> def f():**  **... return 1**  **...**  **>>> f()**  **1** |
| **while** | **Ejecuta un bloque de código mientras que la expresión sea verdadera.** | **>>> i = 0**  **>>> while i < 2:**  **... print(i)**  **... i = i + 1**  **...**  **0**  **1** |
| **with** | **Permite encapsular la ejecución de un bloque de código, de modo que la inicialización y finalización de un objeto es realizada automáticamente por Python, utilizando las funciones \_\_enter\_\_ y \_\_exit\_\_.** | **with open("info.txt"):**  **raise Exception** |
| **yield** | **Se comporta al igual que return, con la diferencia que en lugar de retornar un único valor, retorna elementos que conforman un generador (un objeto iterable que puede recorrerse una vez, ya que el contenido no es almacenado en la memoria), por lo que puede emplearse múltiples veces en una misma función.** | **>>> def f():**  **... yield 1**  **... yield 2**  **... yield 3**  **...**  **>>> g = f()**  **>>> for i in g:**  **... print(i)**  **...**  **1**  **2**  **3**  **>>> for i in g:**  **... print(i)**  **...** |

**Manipular de caracteres en Python**

**Concatenar**

Este término significa juntar cadenas de caracteres. El proceso de concatenación se realiza mediante el operador de suma (+). Ten en cuenta que debes marcar explícitamente dónde quieres los espacios en blanco y colocarlos entre comillas.

En este ejemplo, la cadena de caracteres “mensaje1” tiene el contenido “Hola Mundo”

mensaje1 = 'Hola' + ' ' + 'Mundo'

print(mensaje1)

-> Hola Mundo

**Multiplicar**

Si quieres varias copias de una cadena de caracteres utiliza el operador de multiplicación (\*). En este ejemplo, la cadena de caracteres mensaje2a lleva el contenido “Hola” tres veces, mientras que la cadena de caracteres mensaje2b tiene el contenido “Mundo”. Ordenemos imprimir las dos cadenas.

mensaje2a = 'Hola ' \* 3

mensaje2b = 'Mundo'

print(mensaje2a + mensaje2b)

-> Hola Hola Hola Mundo

**Añadir**

¿Qué pasa si quieres añadir material de manera sucesiva al final de una cadena de caracteres? El operador especial para ello es compuesto (+=).

mensaje3 = 'Hola'

mensaje3 += ' '

mensaje3 += 'Mundo'

print(mensaje3)

-> Hola Mundo

**Extensión**

Puedes determinar el número de caracteres en una cadena utilizando el método len. Acuérdate que los espacios en blanco cuentan como un carácter.

mensaje4 = 'hola' + ' ' + 'mundo'

print(len(mensaje4))

-> 10

**Encontrar**

Puedes buscar una sub-cadena en una cadena de caracteres utilizando el método find y tu programa te indicará el índice de inicio de la misma. Esto es muy útil para procesos que veremos más adelante. Ten en mente que los índices están numerados de izquierda a derecha y que el número en el que se comienza a contar la posición es el 0, no el 1.

mensaje5 = "Hola Mundo"

mensaje5a = mensaje5.find("Mundo")

print(mensaje5a)

-> 5

Si la sub-cadena no está presente el programa imprimirá el valor -1.

mensaje6 = "Hola Mundo"

mensaje6a = mensaje6.find("ardilla")

print(mensaje6a)

-> -1

**Minúsculas**

A veces es útil convertir una cadena de caracteres a minúsculas. Para ello se utiliza el método lower. Por ejemplo, al uniformar los caracteres permitimos que la computadora reconozca fácilmente que “Algunas Veces” y “algunas veces” son la misma frase.

mensaje7 = "HOLA MUNDO"

mensaje7a = mensaje7.lower()

print(mensaje7a)

-> hola mundo

Convertir las minúsculas en mayúsculas se logra cambiando. lower() por upper().

**Reemplazar**

Si necesitas cambiar una sub-cadena de una cadena se puede utilizar el método replace.

mensaje8 = "HOLA MUNDO"

mensaje8a = mensaje7.replace("L", "pizza")

print(mensaje8a)

-> HOpizzaA MUNDO

**Cortar**

Si quieres cortar partes que no quieras del principio o del final de la cadena de caracteres, lo puedes hacer creando una sub-cadena. El mismo tipo de técnica te permite separar una cadena muy larga en componentes más manejables.

mensaje9 = "Hola Mundo"

mensaje9a = mensaje9[1:8]

print(mensaje9a)

-> ola Mun

**Bibliografía**

* [Tipos de datos primitivos en Python](https://aprendeconalf.es/docencia/python/manual/tipos-datos-simples/)
* [Secuencias de escape en Python](https://tutz.tv/python/secuencias-de-escape/)
* [Palabras reservadas en Python](https://recursospython.com/guias-y-manuales/palabras-reservadas-del-lenguaje/)
* [Manejo de Caracteres en Python](https://programminghistorian.org/es/lecciones/manipular-cadenas-de-caracteres-en-python)