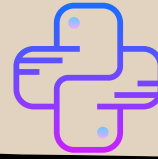


PYTHON CHEAT SHEET IBt



strings

Los strings en Python son un tipo de dato formado por cadenas (o secuencias) de caracteres de cualquier tipo, formando un texto.

print

Declaración que al ejecutarse muestra (o imprime) en pantalla el argumento que se introduce dentro de los paréntesis.

input

Función que permite al usuario introducir información por medio del teclado al ejecutarse, otorgándole una instrucción acerca del ingreso solicitado. El código continuará ejecutándose luego de que el usuario realice la acción.

concatenación

Unificación de cadenas de texto:

```
print("Hola" + " " + "mundo")  
>> Hola mundo
```

caracteres especiales

Indicamos a la consola que el caracter a continuación del símbolo \ debe ser tratado como un caracter especial.

```
\> > Imprime comillas  
\n > Separa texto en una nueva línea  
\t > Imprime un tabulador  
\> > Imprime la barra invertida textualmente
```

mostrar texto

Ingresamos entre comillas simples o dobles los caracteres de texto que deben mostrarse en pantalla.

```
print("Hola mundo")  
>> Hola mundo
```

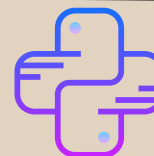
mostrar números

Podemos entregarle a print() el número que debe mostrar, o una operación matemática a resolver. No empleamos comillas en estos casos.

```
print(150 + 50)  
>> 200
```

```
input("Dime tu nombre: ")  
>> Dime tu nombre: |  
                                     ↖ Ingreso por teclado  
  
print("Tu nombre es " + input("Dime tu nombre: "))  
  
>> Dime tu nombre: Federico  
>> Tu nombre es Federico
```

PYTHON CHEAT SHEET IBt



tipos de datos

En Python tenemos varios tipos o estructuras de datos, que son fundamentales en programación ya que almacenan información, y nos permiten manipularla.

variables

Las variables son espacios de memoria que almacenan valores o datos de distintos tipos, y (como su nombre indica) pueden variar. Se crean en el momento que se les asigna un valor, por lo cual en Python no requerimos declararlas previamente.

nombres de las variables

Existen convenciones y buenas prácticas asociadas al nombre de las variables creadas en Python. Las mismas tienen la intención de facilitar la interpretabilidad y mantenimiento del código creado.

texto (str)	números	booleanos	
"Python" "750"	int 250 float 12.50	True False	
estructuras	mutable	ordenado	duplicados
listas []	✓	✓	✓
tuplas ()	✗	✓	✓
sets {}	✓	✗	✗
diccionarios {}	✓	✗*	✗:✓**

*: En Python 3.7+, existen consideraciones **: key es única; value puede repetirse

algunos ejemplos de uso

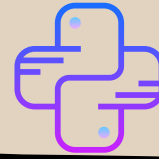
```
pais = "México"
```

```
nombre = input("Escribe tu nombre: ")  
print("Tu nombre es " + nombre)
```

```
num1 = 55  
num2 = 45  
print(num1 + num2)  
>> 100
```

reglas

1. Legible: nombre de la variable es relevante según su contenido
2. Unidad: no existen espacios (puedes incorporar guiones bajos)
3. Hispanismos: omitir signos específicos del idioma español, como tildes o la letra ñ
4. Números: los nombres de las variables no deben empezar por números (aunque pueden contenerlos al final)
5. Signos/símbolos: no se deben incluir: " ', < > / ? | \ () ! @ # \$ % ^ & * - ~ +
6. Palabras clave: no utilizamos palabras reservadas por Python



integers y floats

Existen dos tipos de datos numéricos básicos en Python: int y float. Como toda variable en Python, su tipo queda definido al asignarle un valor a una variable. La función `type()` nos permite obtener el tipo de valor almacenado en una variable.

int

Int, o integer, es un número entero, positivo o negativo, sin decimales, de un largo indeterminado.

```
num1 = 7
print(type(num1))
>> <class 'int'>
```

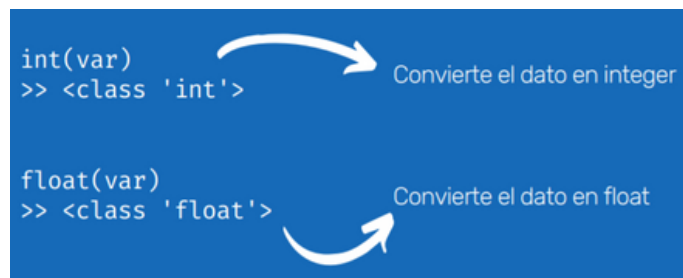
float

Float, o "número de punto flotante" es un número que puede ser positivo o negativo, que a su vez contiene una o más posiciones decimales.

```
num2 = 7.525587
print(type(num2))
>> <class 'float'>
```

conversiones

Python realiza conversiones implícitas de tipos de datos automáticamente para operar con valores numéricos. En otros casos, necesitaremos generar una conversión de manera explícita.



formatear

cadena

Para facilitar la concatenación de variables y texto en Python, contamos con dos herramientas que nos evitan manipular las variables, para incorporarlas directamente al texto:

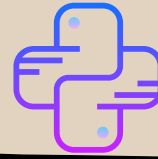
- **Función format:** se encierra las posiciones de las variables entre corchetes {}, y a continuación del string llamamos a las variables con la función format

```
print("Mi auto es {} y de matrícula {}".format(color_auto,matricula))
```

- **Cadenas literales (f-strings):** a partir de Python 3.8, podemos anticipar la concatenación de variables anteponiendo `f` al string

```
print(f"Mi auto es {color_auto} y de matrícula {matricula}")
```

PYTHON CHEAT SHEET IBt



operaciones matemáticas

Veamos cuáles son los operadores matemáticos básicos de Python, que utilizaremos para realizar cálculos:

Suma:	+	
Resta:	-	
Multiplicación:	*	
División:	/	
Cociente (división "al piso"):	//	
Resto (módulo):	%	útil para detectar valores pares :)
Potencia:	**	
Raíz cuadrada:	**0.5	¡es un caso especial de potencia!

redondeo

El redondeo facilita la interpretación de los valores calculados al limitar la cantidad de decimales que se muestran en pantalla. También, nos permite aproximar valores decimales al entero más próximo.

`round(number, ndigits)`

valor a redondear → cantidad de decimales
(si se omite, el resultado es entero)

algunos ejemplos de uso

```
print(round(100/3))  
>> 33  
  
print(round(12/7, 2))  
>> 1.71
```

index

Utilizamos el método `index()` para explorar strings, ya que permite hallar el índice de aparición de un carácter o cadena de caracteres dentro de un texto dado.

`string.index(value, start, end)`

variable que almacena un string → las apariciones antes del índice **start** se ignoran

carácter(es) buscado(s) → las apariciones luego del índice **end** se ignoran

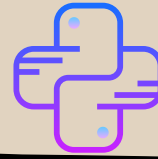
`string.rindex(value, start, end)`

búsqueda en sentido inverso

`string[i]` → devuelve el carácter en el índice **i***

*: En Python, el índice en primera posición es el 0

PYTHON CHEAT SHEET IBt



substrings

Podemos extraer porciones de texto utilizando las herramientas de manipulación de strings conocidas como slicing (rebanar)

Diagrama de slicing: `string[start:stop:step]`

- índice de inicio del sub-string (incluido) → `start`
- índice del final del sub-string (no incluido) → `stop`
- paso → `step`

Ejemplo: `saludo = H o l a _ M u n d o`

Código	Resultado
<code>print(saludo[2:6])</code>	<code>H o l a _ M u n d o</code>
<code>>> la_M</code>	<code>0 1 2 3 4 5 6 7 8 9</code>
<code>print(saludo[3::3])</code>	<code>H o l a _ M u n d o</code>
<code>>> auo</code>	<code>0 1 2 3 4 5 6 7 8 9</code>
<code>print(saludo[::-1])</code>	<code>H o l a _ M u n d o</code>
<code>>> odnuM_aloH</code>	<code>-9 -8 -7 -6 -5 -4 -3 -2 -1 0</code>

strings: métodos de análisis

count(): retorna el número de veces que se repite un conjunto de caracteres especificado.

```
"Hola mundo".count("Hola")
>> 1
```

find() e **index()** retornan la ubicación (comenzando desde el cero) en la que se encuentra el argumento indicado. Difieren en que **index** lanza **ValueError** cuando el argumento no es encontrado, mientras **find** retorna **-1**.

```
"Hola mundo".find("world")
>> -1
```

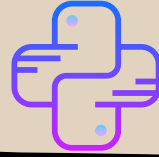
rfind() y **rindex()**. Para buscar un conjunto de caracteres pero desde el final.

```
"C:/python36/python.exe".rfind("/")
>> 11
```

startswith() y **endswith()** indican si la cadena en cuestión comienza o termina con el conjunto de caracteres pasados como argumento, y retornan **True** o **False** en función de ello.

```
"Hola mundo".startswith("Hola")
>> True
```

PYTHON CHEAT SHEET IBt



isdigit(): determina si todos los caracteres de la cadena son **dígitos**, o **pueden formar números**, incluidos aquellos correspondientes a lenguas orientales.

```
"abc123".isdigit()  
>> False
```

isnumeric(): determina si todos los caracteres de la cadena son **números**, incluye **también caracteres de connotación numérica** que no necesariamente son dígitos (por ejemplo, una fracción).

```
"1234".isnumeric()  
>> True
```

isdecimal(): determina si todos los caracteres de la cadena son **decimales**; esto es, formados por dígitos **del 0 al 9**.

```
"1234".isdecimal()  
>> True
```

isalnum(): determina si todos los caracteres de la cadena son **alfanuméricos**.

```
"abc123".isalnum()  
>> True
```

isalpha(): determina si todos los caracteres de la cadena son **alfabéticos**.

```
"abc123".isalpha()  
>> False
```

islower(): determina si todos los caracteres de la cadena son **minúsculas**.

```
"abcdef".islower()  
>> True
```

isupper(): determina si todos los caracteres de la cadena son **mayúsculas**.

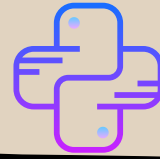
```
"ABCDEF".isupper()  
>> True
```

isprintable(): determina si todos los caracteres de la cadena son **imprimibles** (es decir, no son caracteres especiales indicados por \...).

```
"Hola \t mundo!".isprintable()  
>> False
```

isspace(): determina si todos los caracteres de la cadena son **espacios**.

```
"Hola mundo".isspace()  
>> False
```



strings: métodos de transformación

En realidad los strings son inmutables; por ende, todos los métodos a continuación no actúan sobre el objeto original sino que retornan uno nuevo.

capitalize() retorna la cadena con su primera letra en mayúscula.

```
"hola mundo".capitalize()  
>> 'Hola mundo'
```

encode() codifica la cadena con el mapa de caracteres especificado y retorna una instancia del tipo bytes.

```
"Hola mundo".encode("utf-8")  
>> b'Hola mundo'
```

replace() reemplaza una cadena por otra.

```
"Hola mundo".replace("mundo", "world")  
>> 'Hola world'
```

lower() retorna una copia de la cadena con todas sus letras en minúsculas.

```
"Hola Mundo!".lower()  
>> 'hola mundo!'
```

upper() retorna una copia de la cadena con todas sus letras en mayúsculas.

```
"Hola Mundo!".upper()  
>> 'HOLA MUNDO!'
```

swapcase() cambia las mayúsculas por minúsculas y viceversa.

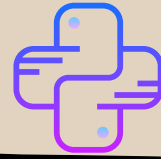
```
"Hola Mundo!".swapcase()  
>> 'hOLA mUNDO!'
```

strip(), **lstrip()** y **rstrip()** remueven los espacios en blanco que preceden y/o suceden a la cadena.

```
"  Hola mundo! ".strip()  
>> 'Hola mundo!'
```

Los métodos **center()**, **ljust()** y **rjust()** alinean una cadena en el centro, la izquierda o la derecha. Un segundo argumento indica con qué carácter se deben llenar los espacios vacíos (por defecto un espacio en blanco).

```
"Hola".center(10, "*")  
>> '***Hola***'
```



strings: métodos de separación y unión

split() divide una cadena según un caracter separador (por defecto son espacios en blanco). Un segundo argumento en **split()** indica cuál es el máximo de divisiones que puede tener lugar (-1 por defecto para representar una cantidad ilimitada).

```
"Hola mundo!\nHello world!".split()
>> ['Hola', 'mundo!', 'Hello', 'world!']
```

splitlines() divide una cadena con cada aparición de un salto de línea.

```
"Hola mundo!\nHello world!".splitlines()
>> ['Hola mundo!', 'Hello world!']
```

partition() retorna una tupla de tres elementos: el bloque de caracteres anterior a la primera ocurrencia del separador, el separador mismo, y el bloque posterior.

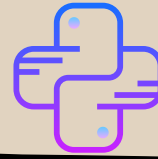
```
"Hola mundo. Hello world!".partition(" ")
>> ('Hola', ' ', 'mundo. Hello world!')
```

rpartition() opera del mismo modo que el anterior, pero partiendo de derecha a izquierda.

```
"Hola mundo. Hello world!".rpartition(" ")
>> ('Hola mundo. Hello', ' ', 'world!')
```

join() debe ser llamado desde una cadena que actúa como separador para unir dentro de una misma cadena resultante los elementos de una lista.

```
", ".join(["C", "C++", "Python", "Java"])
>> 'C, C++, Python, Java'
```

strings: propiedades

- **son inmutables:** una vez creados, no pueden modificarse sus partes, pero sí pueden reasignarse los valores de las variables a través de métodos de strings
- **concatenable:** es posible unir strings con el símbolo +
- **multiplicable:** es posible concatenar repeticiones de un string con el símbolo *
- **multilineales:** pueden escribirse en varias líneas al encerrarse entre triples comillas simples (""" """) o dobles (""" """)
- **determinar su longitud:** a través de la función `len(mi_string)`
- **verificar su contenido:** a través de las palabras clave `in` y `not in`. El resultado de esta verificación es un booleano (`True` / `False`).

listas

Las listas son secuencias ordenadas de objetos. Estos objetos pueden ser datos de cualquier tipo: strings, integers, floats, booleanos, listas, entre otros. Son tipos de datos mutables.

mutable ✓ ordenado ✓ admite duplicados ✓

```
lista_1 = ["C", "C++", "Python", "Java"]  
lista_2 = ["PHP", "SQL", "Visual Basic"]
```

indexado: podemos acceder a los elementos de una lista a través de sus índices [inicio:fin:paso]

```
print(lista_1[1:3])  
>> ["C++", "Python"]
```

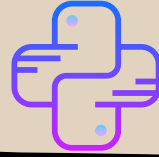
cantidad de elementos: a través de la propiedad `len()`

```
print(len(lista_1))  
>> 4
```

concatenación: sumamos los elementos de varias listas con el símbolo +

```
print(lista_1 + lista_2)  
>> ['C', 'C++', 'Python', 'Java', 'PHP', 'SQL', 'Visual Basic']
```

PYTHON CHEAT SHEET IBt



```
lista_1 = ["C", "C++", "Python", "Java"]
lista_2 = ["PHP", "SQL", "Visual Basic"]
lista_3 = ["d", "a", "c", "b", "e"]
lista_4 = [5, 4, 7, 1, 9]
```

función append(): agrega un elemento a una lista *en el lugar*

```
lista_1.append("R")
print(lista_1)
>> ["C", "C++", "Python", "Java", "R"]
```

función pop(): elimina un elemento de la lista dado el índice, y *devuelve* el valor quitado

```
print(lista_1.pop(4))
>> "R"
```

función sort(): ordena los elementos de la lista *en el lugar*

```
lista_3.sort()
print(lista_3)
>> ['a', 'b', 'c', 'd', 'e']
```

función reverse(): invierte el orden de los elementos *en el lugar*

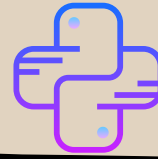
```
lista_4.reverse()
print(lista_4)
>> [9, 1, 7, 4, 5]
```

➔ reverse no es lo opuesto a sort

diccionarios

Los diccionarios son estructuras de datos que almacenan información en pares clave:valor. Son especialmente útiles para guardar y recuperar información a partir de los nombres de sus claves (no utilizan índices).

PYTHON CHEAT SHEET IBt



mutable ✓

ordenado ✗*

admite duplicados ✗: ✓
valor
clave

```
mi_diccionario = {"curso": "Python TOTAL", "clase": "Diccionarios"}
```

agregar nuevos datos, o modificarlos

```
mi_diccionario["recursos"] = ["notas", "ejercicios"]
```

acceso a valores a través del nombre de las claves

```
print(mi_diccionario["recursos"][1])  
>> "ejercicios"
```

métodos para listar los nombres de las claves, valores, y pares clave:valor

keys() ←

↓ values()

↓ items()

**: A partir de Python 3.7+, los diccionarios son tipos de datos ordenados, en el sentido que dicho orden se mantiene según su orden de inserción para aumentar la eficiencia en el uso de la memoria.*

tuples

Los tuples o tuplas, son estructuras de datos que almacenan múltiples elementos en una única variable. Se caracterizan por ser ordenadas e inmutables. Esta característica las hace más eficientes en memoria y a prueba de daños.

mutable ✗

ordenado ✓

admite duplicados ✓

```
mi_tuple = (1, "dos", [3.33, "cuatro"], (5.0, 6))
```

indexado (acceso a datos)

```
print(mi_tuple[3][0])  
>> 5.0
```

casting (conversión de tipos de datos)

```
lista_1 = list(mi_tuple)  
print(lista_1)  
>> [1, "dos", [3.33, "cuatro"], (5.0, 6)] ↗  
ahora es una estructura mutable
```

unpacking (extracción de datos)

```
a, b, c, d = mi_tuple  
print(c)  
>> [3.33, "cuatro"]
```