

Introduccion a Python



Este notebook incluye una introducción al lenguaje de programación de Python.
 Para un mejor entendimiento, los alumnos deben tener cierta familiarización con Python.
 Estos ejemplos corren en Python 3.12 with Spark 2.1.

Objetivos

- Entender la estructura básica de Python
- Leer y escribir sentencias simples de Python

Historia de Python

Python fue creado a principios de los 90s por Guido Van Rossum en el Stichting MATheMATish Centrum (CWI) en Holanda como sucesor del lenguaje llamado ABC. Guido es el principal autor de Python, sin embargo el lenguaje incluye muchas contribuciones de diferentes autores.

Porque utilizar Python?

Python es fácil de interpretar, orientado a objetos y lenguaje de alto nivel. Hace énfasis en la lectura de la sintaxis mientras usa una biblioteca extendida y mantenida que puede ser distribuida libremente.

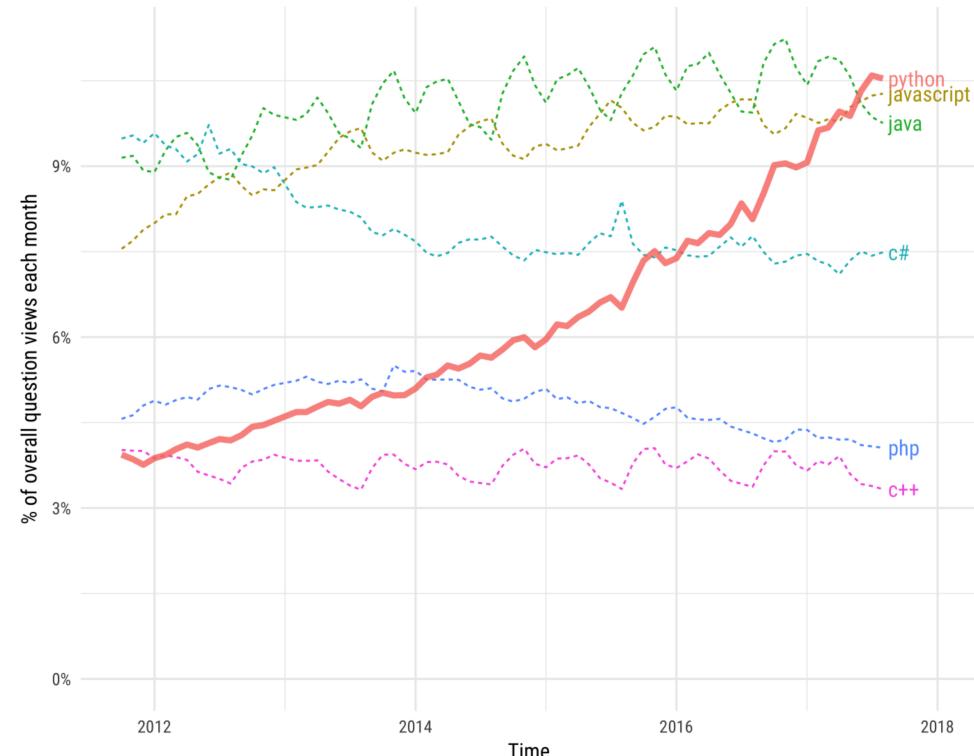
La popularidad de Python para la ciencia de datos se debe a la sólida de sus bibliotecas centrales y en gran medida a su sólida como lenguaje de programación de propósito general. Por lo tanto, al aprovechar las capacidades centrales de Python, es fácil construir scripts de código simples y complejos para, en este caso, resolver tareas manuales y repetitivas en muy poco tiempo.

Esta es una breve lista de los beneficios de Python:

- Alta productividad en creación de prototipos
- Bibliotecas básicas matemáticas y estadísticas con funciones prediseñadas
- Lenguaje de programación para propósito general
- Automatización de trabajos de procesamiento de datos, minería de datos, análisis y modelado de datos, todo en un solo lenguaje.

Growth of major programming languages

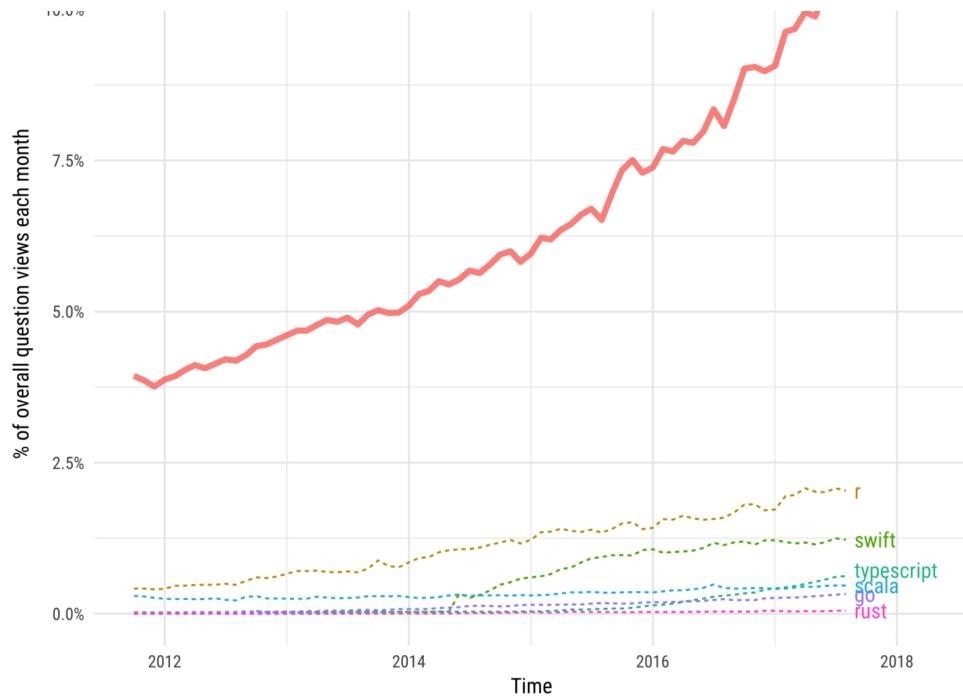
Based on Stack Overflow question views in World Bank high-income countries



Python compared to smaller, growing technologies

Based on question traffic in World Bank high-income countries





```
In [1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Variables

Una pieza del código que puede definir libremente como se puede almacenar una pieza de datos.

Las variables sirven como motor para almacenar datos al crear programas complejos, páginas web, motores de búsqueda, etc.

Algo que debemos tener en cuenta es que **todas las variables distinguen entre mayúsculas y minúsculas**, esto significa que una variable **Apple** no es lo mismo que la variable **apple**.

```
In [2]: esta_es_una_variable = 12
print(esta_es_una_variable)
```

```
In [1]: x = 3
y = 10
xy = x + y
print(xy)
```

13

```
In [20]: enero = 31
febrero = 28
marzo = 31
abril = 30
mayo = 31
junio = 30
julio = 31
agosto = 31
setiembre = 30
octubre = 31
noviembre = 30
diciembre = 31

dia_nac = 6
mes_nac = marzo
año_nac = 1987

dia_hoy = 5
mes_hoy = febrero
año_actual = 2025

print(año_actual-año_nac)
dias_restantes_naci = 365-enero-febrero-dia_nac
print(dias_restantes_naci)
dias_transcurridos_actual = enero+dia_hoy
print(dias_transcurridos_actual)
dias_vividos = 365*(año_actual-año_nac-1)+dias_restantes_naci*dias_transcurridos_actual
print(dias_vividos)
print(dias_vividos//365)
```

```

print(enero+febrero+dia_nac-dias_transcurridos_actual)

38
300
36
13841
37
29

### Reasignar

Podemos reasignar valores a las variables definidas previamente.
En este ejemplo vamos a cambiar el valor de la variable **mi_numero** de 200 a 10.

In [ ]: mi_numero = 200
         print(mi_numero)
         mi_numero = 10
         print(mi_numero)

```

Numeros

Este es el tipo basico de dato en Python. Hay dos tipos de numeros estandar:

- Flotante (float)
- Entero (int)

```

In [2]: a = 10
         print(a)
         print(type(a))

10
<class 'int'>

In [3]: a = 10.0
         print(a)
         print(type(a))

10.0
<class 'float'>

```

En muchas ocasiones, los tipos de dato estan definidos cuando importamos datos de otros recursos. Sin embargo, es sencillo cambiar de tipo de dato, lo cual es muy util, cuando necesitamos cambiar un numero/variable.

```

In [ ]: edad = 14.0
         print(type(edad))
         edad = int(edad)
         print(type(edad))

```

Pero que sucede cuando el numero contiene decimales y cambiamos su tipo de Flotante a Entero? bueno, en este caso el numero es redondeado al entero mas bajo.

```

In [ ]: numero = 15.9
         numero = int(numero)
         print(numero)

```

Libreria Random

La libreria random es muy util cuando necesitamos crear una serie de numeros para trabajar con ellos.

```

In [ ]: import random
         random.randint(0,10)

In [ ]: random.randrange(100)

In [ ]: random.randrange(10,100,20)

```

Operadores

Esta es la lista de operadores aritmeticos usados en Python.

```

+ suma
- resta
/ division
% modulo
* multiplicacion
// residuo
** potencia

```

```

In [ ]: 1 + 2
In [ ]: 7 - 6
In [ ]: 5 * 5
In [ ]: 10 / 2
In [4]: 15 % 10
Out[4]: 5

```

Este muestra el residuo de una division

```

In [5]: 3 // 2
Out[5]: 1

```

Es el numero entero mas cercano a la division

```

In [6]: 5 ** 3
Out[6]: 125

```

Operadores Relacionales

Esta es la lista de operadores relacionales

```
== Verdadero si es igual a  
!= Verdadero, si no es igual a  
< menor que  
> mayor que  
<= menor o igual a  
>= mayor o igual a
```



```
In [1]: a = 10 # This is actually defining a variable, and not checking if it is equal  
In [7]: a == 10  
Out[7]: True  
In [8]: a < 20  
Out[8]: True  
In [9]: a > 12  
Out[9]: False  
In [10]: a != 9  
Out[10]: True
```

Capturar informacion del usuario

`input()` acepta la informacion escrita por el usuario y la almacena como cadena de texto. Entonces si la informacion escrita es un entero, es necesario convertir el texto a entero y despues continuar con las operaciones del codigo.

```
In [1]: mi_captura = input('Escribe algo aqui: ')  
Escribe algo aqui: adsada  
In [2]: type(mi_captura)  
Out[2]: str  
In [11]: mi_captura2 = int(input('Escribe tu edad: '))  
Escribe tu edad: 18
```

Aqui, se esta cambiando la captura a numero Entero, de lo contrario se puede generar un error si la captura no es un entero.

```
In [12]: type(mi_captura2)  
Out[12]: int  
In [13]: #this will make the input to accept integers or floats  
mi_captura3 = eval(input('Escribe tu edad: '))  
Escribe tu edad: 1231  
In [14]: type(mi_captura3)  
Out[14]: int
```

Importante el metodo `type()` nos regresa el formato o tipo de una variable o numero.

Sentencia Print

La sentencia `print` es uno de las funciones mas utiles en Python. Se pueden crear diferentes salidas con este metodo, como:

- `print("Hola mundo!")`
- `print("Hola mundo!",)`

```
In [ ]: print('Hola mundo!')
```

En Python, una, doble y triple comas son usadas para definir una cadena de texto. La mayoria usa una comilla simple para declarar un solo caracter. Comillas dobles cuando declaran una linea de texto y triple comilla cuando declaran un parrafo de multiples lineas.

```
In [ ]: print('Holis!')  
In [15]: print('''Este es un notebook magico creado con  
el propósito de hacer que este grupo se  
enamore del azombroso y magico Python! :) ''')  
Este es un notebook magico creado con  
el propósito de hacer que este grupo se  
enamore del azombroso y magico Python! :)  
In [17]: mes = 'Junio'  
dia = 7  
año = 1992  
  
print('Yo naci el', dia, 'de', mes, 'del', año)  
Yo naci el 7 de Junio del 1992
```

De manera similar, cuando se utiliza otro tipo de dato, se pueden utilizar las siguientes anotaciones:

- `%s` -> cadena
- `%d` -> Entero
- `%f` -> Flotante
- `%o` -> Octadecimal
- `%x` -> Hexadecimal
- `%e` -> Exponential

```
In [18]: print ("Numero Entero = %d" %29)
print ("Valor Flotante = %f" %29)
print ("Numero Ocadecimal equivalente = %o" %29)
print ("Numero Hexadecimal equivalente = %x" %29)
print ("Numero Exponencial equivalente = %e" %29)
```

```
Numero Entero = 29
Valor Flotante = 29.000000
Numero Ocadecimal equivalente = 35
Numero Hexadecimal equivalente = 1d
Numero Exponencial equivalente = 2.900000e+01
```

```
In [19]: print ('Si!'*10)
Si!Si!Si!Si!Si!Si!Si!Si!Si!
```

Cadenas/String

Otro tipo de dato es son los **Strings** Cadenas, que puede ser expresadas de diferentes maneras: Como se menciono en la sección del metodo print, en Python, una, doble y triple comillas son usadas para denotar a una cadena. As we mentioned it on the Print Statement section, in Python, single, double and triple quotes are used to denote a string.

```
>>> 'perro loco' # comillas simples
'perro loco'
>>> 'Divo\'s' # usa \' para escapar el caracter de la comilla simple...
"Divo's"
>>> "Divo's" # ...o puedes usar dobles comillas
"Divo's"
>>> """Si," el comentario.
'"Si," el comentario.
>>> """Si,\\" el comentario.
'"Si," el comentario.'
```

```
In [20]: s = 'Esta es la primera linea.\nEsta es la segunda.' # \n significa una nueva linea
print(s)

Esta es la primera linea.
Esta es la segunda.
```

```
In [21]: s = 'Esta es la primera linea.\nEsta es la segunda.'
s # is el metodo print(), \n es incluida en la salida
Out[21]: 'Esta es la primera linea.\nEsta es la segunda.'
```

Si queremos evitar agregar \ que aparece en la salida, puedes agregar r antes de la primera cadena:

```
In [ ]: print('C:\temp\nombre')
print(r'C:\temp\nombre') # note there is an r before the string
print('C:\\temp\\nombre')
```

Las Cadenas/Strings pueden ser concatenadas con el operador de suma +, y repetidas con el operador de multiplicacion *

```
In [22]: 5 * 'Yo estoy +' " " + 'feliz'
Out[22]: 'Yo estoy Yo estoy Yo estoy Yo estoy Yo estoy  feliz'
```

Cuando se necesite dividir cadenas largas de texto, se puede hacer lo siguiente:

```
In [ ]: texto = ('Eso debera ser considerado '
               'como un textoooooooooooo muuyyyyyyy largo')
print(texto)
```

Nota especial : Esto solo funciona con dos cadenas de texto (strings), y no con variables o expresiones, para eso, se debe utilizar el operador de suma +:

```
In [23]: nombre = 'Gilberto'
nombre + ' es un cientifico de datos'
Out[23]: 'Gilberto es un cientifico de datos'
```

Las Cadenas/Strings son indexadas definiendo como 0 al primer caracter:

```
In [1]: texto = 'Monitoreo de estudiantes'
texto[0]
Out[1]: 'M'
```

Tambien se puede calcular el indice iniciando desde la derecha usando numeros negativos:

```
In [25]: texto[-1] # this will bring the last character
Out[25]: 's'
```

Obtener subcadenas / slicing

Mientras el un indice se utiliza para obtener caracteres individuales, El uso de dos indices(slicing) ayuda a obtener una subcadena:

```
In [2]: texto[0:3]
Out[2]: 'Mon'
```

Como usos predeterminados slice tiene diferentes usos; omitir el primer indice el valor predeterminado es 0, omitir el segundo indice el valor predeterminado es el tamaño de la cadena de la que se obtendra la subcadena.

```
In [3]: texto[:2] # Caracter desde el inicio hasta el indice 2(excluido)
Out[3]: 'Mo'
```

```
In [4]: texto[4:] # Caracteres desde el indice 4 (included) hasta el final de la cadena
Out[4]: 'toreo de estudiantes'
```

```
In [27]: # Seleccionando desde La derecha
    texto[-6:]

Out[27]: 'iantes'

In [5]: # toda La cadena
    texto[::-1]

Out[5]: 'Monitoreo de estudiantes'

In [ ]: # Invertir La cadena
    texto[::-1]

La funcion len() nos regresa el valor de la longitud de una cadena/string:
```

```
In [ ]: len(texto)
```

Ejercicio 1

Pregunta al usuario por su comida favorita y escribe el numero de letras que contiene su comida.

```
In [ ]: 
```

Ejercicio 4

Codificar una solucion que escriba el siguiente texto llenando los espacios en blanco con variables que tu definas:

"Mi nombre es _____ y tengo _____ años de edad. Tambien, Siempre he querido ser _____. Gracias por leer me!"

```
In [ ]: 
```

Ejercicio 5

Escribe una solucion que le pregunte al usuario su nombre y apellido, y escribe el texto invertido con un espacio entre ellos.

```
In [ ]: 
```

Crear una solucion que le pregunte al usuario su nombre y edad. Entonces escribir un mensaje que le diga el año en el que el puede llegar a tener 100 años de edad!!

```
In [ ]: 
```

Estructuras de datos

En terminos simples, Es la colección o grupos de datos en una estructura en particular.

Listas

Las listas son la estructura de datos mas común. Piensa como una secuencia de datos que esta definida entre corchetes [], los datos están separados por comas. Se puede acceder a cada uno de ellos utilizando índices.

Las listas son declaradas con el nombre de la variable, simbolo igual y los corchetes [] o la lista con elementos.

```
In [2]: mi_lista = []

In [3]: type(mi_lista)

Out[3]: list
```

O puedes directamente asignar la secuencia de datos a la lista como aqui:

```
In [5]: mi_lista = ['Platanos', 'Naranjas', 'Limones']
```

Indices en las listas

En Python, Los indices inician desde 0, entonces mi_lista que tiene tres elementos, tiene Platanos en el indice 0, Naranjas en el 1 y Limones en el 2.

```
In [6]: mi_lista[0]

Out[6]: 'Platanos'
```

EL indexado puede ser utilizado en orden inverso. Ya que el ultimo elemento puede ser accedido primero. De esa manera el indexado inicia de -1. Así que el indice -1 nos devolvería Limones, Naranjas sería encontrado en el indice -2.

```
In [7]: mi_lista[-2]

Out[7]: 'Naranjas'
```

Tal vez ya descubriste pero hay dos formas de acceder a los elementos de la lista, x[0] = x[-2], x[1] = x[-1]. Este concepto puede usarse en listas con muchos mas elementos.

Ahora, Definimos dos listas mi_lista y tu_lista contenido sus propios datos. Entonces estas dos listas pueden ser puestas en otra lista listas_anidadas. Esto puede ser llamado listas_anidadas o lista_de_listas.

```
In [9]: mi_lista = ['Platanos', 'Naranjas', 'Limones']
tu_lista = ['Asombroso!', 'Increíble!', 'Fuera de este mundo!']

In [10]: listas_anidadas = [mi_lista,tu_lista]
print(listas_anidadas)
listas_anidadas

Out[10]: [['Platanos', 'Naranjas', 'Limones'], ['Asombroso!', 'Increíble!', 'Fuera de este mundo!']]
[[['Platanos', 'Naranjas', 'Limones'], ['Asombroso!', 'Increíble!', 'Fuera de este mundo!']]
```

El indexado en listas anidadas o una lista de listas puede ser, a veces un poco confuso, si no entiendes como el indexado funciona en python.

Así que analicémoslo y luego llegaremos a una conclusión.

Vamos a acceder al dato 'Platanos' en la siguiente lista anidada. Primero en el índice 0 esta la lista ['Platanos', 'Naranjas', 'Limones'] y en el índice 1 la otra lista ['Asombroso!', 'Increible!', 'Fuera de este mundo!']

Por lo tanto, `listas_anidadas[0]` debería darnos la primera lista que contiene 'Platanos'.

```
In [11]: lista_anidada_copia = listas_anidadas[0]
print (lista_anidada_copia)
```

['Platanos', 'Naranjas', 'Limones']

Ahora observe que `lista_anidada_copia` no es en absoluto una lista anidada, para acceder a 'Platanos', `lista_anidada_copia` debe indexarse en 0.

```
In [12]: lista_anidada_copia[0]
Out[12]: 'Platanos'
```

En lugar de hacer lo anterior, puede acceder a 'Platanos' simplemente escribiendo los valores del índice cada vez uno al lado del otro.

```
In [13]: listas_anidadas[0][0]
Out[13]: 'Platanos'
```

Si había una lista dentro de una lista dentro de otra lista, entonces se puede acceder al valor más interno ejecutando `listas_anidadas[][][]`

Obtener SubListas/List Slicing

La indexación solo se limitaba a acceder a un solo elemento; por otro lado, una sección de una lista es acceder a una secuencia de datos dentro de la lista. En otras palabras, "cortar" la lista.

La división se realiza definiendo los valores de índice del primer elemento y el último elemento de la lista principal que se requiere en la lista dividida. Está escrito como lista de padres [a: b] donde a, b son los valores de índice de la lista de padres. Si a o b no está definido, entonces el valor del índice se considera el primer valor de a si a no está definido y el último valor de b cuando b no está definido.

```
In [14]: num = [0,1,2,3,4,5,6,7,8,9]
```

```
In [15]: print (num[0:6])
print (num[4:])

```

[0, 1, 2, 3, 4, 5]
[4, 5, 6, 7, 8, 9]

También puede dividir una lista principal definiendo el tamaño del salto.

```
In [16]: num[::-3]
Out[16]: [0, 3, 6, 9]
```

Uso de funciones integradas en Listas

Para encontrar la longitud de la lista o el número de elementos en una lista, se utiliza `len()`.

```
In [17]: len(num)
Out[17]: 10
```

Si la lista consta de todos los elementos enteros, entonces `min()` y `max()` dan el valor mínimo y máximo en la lista.

```
In [18]: min(num)
```

Out[18]: 0

```
In [19]: max(num)
```

Out[19]: 9

Las listas se pueden concatenar agregándoles '+'. La lista resultante contendrá todos los elementos de las listas que se agregaron. La lista resultante no será una lista anidada.

```
In [20]: [1,2,3] + [4,5,6]
Out[20]: [1, 2, 3, 4, 5, 6]
```

Puede surgir un requisito en el que deba verificar si un elemento en particular está en una lista predefinida. Considere la siguiente lista.

```
In [21]: nombres = ['Paco', 'Fernando', 'Gilberto', 'Cynthia']
```

Para comprobar si 'Paco' y 'Jamie' están presentes en los nombres de la lista. Podemos usar el concepto 'a en b' que devolvería 'Verdadero' si a está presente en b y 'Falso' si no.

```
In [22]: 'Paco' in nombres
```

Out[22]: True

```
In [23]: 'Jamie' in nombres
```

Out[23]: False

Si desea encontrar el elemento en función de la longitud de la cadena/string, se declara otro parámetro 'key=len' dentro de la función `max()` y `min()`.

```
In [24]: print (max(nombres, key = len))
print (min(nombres, key = len))
```

Fernando
Paco

Una cadena se puede convertir en una lista usando la función `list()`.

```
In [25]: list('Hello')
Out[25]: ['H', 'e', 'l', 'l', 'o']
```

`append()` se utiliza para agregar un elemento al final de la lista.

```
In [26]: lista1 = [5,4,2,8]
```

```
In [27]: lista1.append(9)
print(lista1)
```

```
[5, 4, 2, 8, 9]
```

`count()` se utiliza para contar el número de un elemento particular que está presente en la lista.

```
In [28]: lista1.count(8)
```

```
Out[28]: 1
```

`index()` se utiliza para encontrar el valor del índice de un elemento en particular. Tenga en cuenta que si hay varios elementos con el mismo valor, se devuelve el primer valor de índice de ese elemento.

```
In [29]: lista1.index(8)
```

```
Out[29]: 3
```

`insert()` se utiliza para insertar un elemento y en un valor de índice especificado x. La función `append()` solo permitía insertar al final.

```
In [ ]: lista1.insert(5,'Paco')
print(lista1)
```

También se puede eliminar un elemento especificando el elemento en sí usando la función `remove()`.

```
In [30]: nombres.remove('Paco')
print(nombres)
```

```
['Fernando', 'Gilberto', 'Cynthia']
```

También puede reordenar la lista con la función `reverse()`.

```
In [31]: lista1.reverse()
print(lista1)
```

```
[9, 8, 2, 4, 5]
```

Python ofrece la operación integrada `sort()` para organizar los elementos en orden ascendente.

```
In [32]: lista1.sort()
print(lista1)
```

```
[2, 4, 5, 8, 9]
```

Para el orden descendente, de forma predeterminada la condición inversa será `False` para la inversa. Por lo tanto, cambiarlo a Verdadero organizaría los elementos en orden descendente.

```
In [33]: lista1.sort(reverse=True)
print (lista1)
```

```
[9, 8, 5, 4, 2]
```

Copiar una lista

Este es uno de los errores más comunes cometidos en Python:

```
In [34]: lista_a = [1,2,3,4,5]
```

```
In [35]: #Incorrect
lista_b = lista_a
print(lista_b)
```

```
[1, 2, 3, 4, 5]
```

Aquí hemos declarado una lista, `list_a = [1,2,3,4,5]`. Esta lista se copia en `list_b`. Ahora realizamos algunas operaciones aleatorias en `list_a`.

```
In [36]: lista_a.remove(5)
print(lista_a)
lista_a.append('Optimization')
print(lista_a)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4, 'Optimization']
```

```
In [37]: print(lista_b)
```

```
[1, 2, 3, 4, 'Optimization']
```

`List_b` también ha cambiado incluso si no realizamos ninguna operación en él. La razón detrás de esto es que asignamos el mismo espacio de memoria a `list_a` que a `list_b`. Imaginemos que es como crear un espejo, lo que pase con el lado original afectará al lado copiado.

Hay varias formas de evitar esto, esta es una forma sencilla de hacerlo:

```
In [38]: lista_b = lista_a[:]
print(lista_b)
```

```
[1, 2, 3, 4, 'Optimization']
```

```
In [39]: lista_a.remove('Optimization')
print(lista_a)
```

```
[1, 2, 3, 4]
```

```
In [40]: print(lista_b)
```

```
[1, 2, 3, 4, 'Optimization']
```

Tuplas

Las tuplas son similares a las listas, pero la única gran diferencia es que los elementos dentro de Tuple no se pueden cambiar. Para definir una tupla, se

asigna una variable a paréntesis () o tupla () .

```
In [42]: tup1 = (100)
tup = tuple()
```

Se pueden asignar valores al declarar una tupla. Toma una lista como entrada y la convierte en una tupla o toma una cadena y la convierte en una tupla.

```
In [43]: tup3 = tuple([1,2,3])
print (tup3)
tup4 = tuple('Optimization')
print (tup4)

(1, 2, 3)
('o', 'p', 't', 'i', 'm', 'i', 'z', 'a', 't', 'i', 'o', 'n')
```

Sigue la misma indexación y división que las Listas.

```
In [44]: print (tup3[1])
tup5 = tup4[:3]
print (tup5)

('o', 'p', 't')
```

Conjuntos/Sets

Los conjuntos se utilizan principalmente para eliminar números repetidos en una secuencia/lista. También se utiliza para realizar algunas operaciones de conjuntos estándar.

Los conjuntos se declaran como set() que inicializará un conjunto vacío

```
In [45]: set1 = set()
print (type(set1))

<class 'set'>
```

```
In [46]: set0 = set([1,1,1,1,1,1,1,9,7,2,2,3,3,4])
print (set0)

{1, 2, 3, 4, 7, 9}
```

Diccionario

Los diccionarios en Python son una estructura de datos que permite almacenar su contenido en forma de llave y valor. Los diccionarios se pueden crear con paréntesis {} separando con una coma cada par key: value. En el siguiente ejemplo tenemos tres keys que son el nombre, la edad y el documento.

```
In [47]: d1 = {
    "Nombre": "Sara",
    "Edad": 27,
    "Documento": 1003882
}
print(d1)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}

{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Otra forma equivalente de crear un diccionario en Python es usando dict() e introduciendo los pares key: value entre paréntesis.

```
In [27]: d2 = dict([
    ('Nombre', 'Sara'),
    ('Edad', 27),
    ('Documento', 1003882),
])
print(d2)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}

{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

También es posible usar el constructor dict() para crear un diccionario.

```
In [26]: d3 = dict(Nombre='Sara',
                    Edad=27,
                    Documento=1003882)
print(d3)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}

{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Algunas propiedades de los diccionarios en Python son las siguientes:

Son dinámicos, pueden crecer o decrecer, se pueden añadir o eliminar elementos. Son indexados, los elementos del diccionario son accesibles a través del key. Y son anidados, un diccionario puede contener a otro diccionario en su campo value.

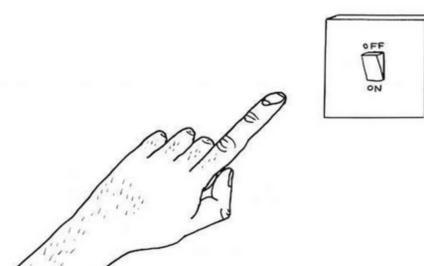
Booleanos/Booleans

Booleano es uno de los diferentes tipos de datos dentro de Python.

De manera sencilla, un booleano es una expresión que arroja un resultado en forma de Verdadero o Falso.

Imaginemos que un booleano es un interruptor de luz, por lo que solo hay dos resultados: encendido y apagado.

Bueno, en booleanos los nombres son: Verdadero y Falso.





```
In [25]: mi_numero = 10  
mi_float = 8.9  
mi_boolean = True
```

Declaraciones de flujo de control

Esta es una de las declaraciones más útiles dentro del lenguaje Python.

if

Básicamente, estamos creando una declaración de condición que realizará una acción cuando se cumpla la condición dada.

```
In [24]: x = 'Andres'  
if 'Andres' == x:  
    print('Hola Andres!')  
  
Hola Andres!
```

if - else

Si no se cumple una condición determinada, podríamos especificar qué sucede a continuación.

```
In [23]: x = 'Andres'  
if 'Paco' == x:  
    print('Hola Paco!')  
else:  
    print('Creí que eras Paco!')  
  
Creí que eras Paco!
```

if - elif

Si no se cumple la condición inicial, podemos especificar un número infinito de condiciones.

```
In [21]: x = 100  
y = 90  
if x > y:  
    print('X es mayor que Y')  
elif x < y:  
    print('X es menor que Y')  
else:  
    print('X es igual a Y')  
  
X es mayor que Y
```

Si una declaración IF está dentro de una declaración If, esto se llama declaración if anidada.

```
In [22]: if x < y:  
    print ('X es menor que Y')  
elif x > y:  
    print ('X es mayor que Y')  
    if x==100:  
        print ("x=100")  
    else:  
        print ("invalido")  
else:  
    print ('X es igual a Y')  
  
X es mayor que Y  
x=100
```

Match

El Match es una herramienta que nos permite ejecutar diferentes secciones de código dependiendo de una condición.

Ejercicio 5 - ¿Impar o par?

Solicite un número al usuario y luego valide si el número es **par** o **ímpar**. Termine con un mensaje que le permita al usuario saber qué tipo de número es.

Bucle for

Tiene la capacidad de iterar sobre los elementos de cualquier secuencia, como una lista o una cadena. Para esto, midamos alguna cadena de una lista:

```
In [18]: cosas_felices = ['Perros','Helado de Chocolate','Amanecer', 'Viajar']  
for i in cosas_felices:  
    print(i,len(i))  
  
Perros 6  
Helado de Chocolate 19  
Amanecer 8  
Viajar 6
```

```
In [19]: contador = 0  
for i in cosas_felices:  
    contador += 1  
    print(contador)  
  
1  
2  
3  
4
```

La función range()

Esta función es útil cuando necesita iterar sobre una serie de números. Veamos este ejemplo:

```
In [15]: for i in range(10):
    print(i)

0
1
2
3
4
5
6
7
8
9
```

Es posible dejar que el rango comience en otro número o especificar un incremento diferente, esto generalmente se llama **paso**:

```
In [14]: for i in range(5,10):
    print(i)

5
6
7
8
9
```

```
In [13]: for i in range(0,10,2):
    print(i)

0
2
4
6
8
```

```
In [12]: for i in range(-10,-100,-20):
    print(i)

-10
-30
-50
-70
-90
```

Hemos visto que la declaración `for` es uno de esos iteradores. La función `list()` es otra; crea listas a partir de iterables:

```
In [11]: x = list(range(0,100,20))
print(x)

[0, 20, 40, 60, 80]
```

Sentencias Break y Continue

La declaración `break` se utiliza para romper el bucle `for` o `while` más interno.

Esta declaración está tomada de C.

Los bucles podrían tener una sección `else`, que se ejecutará cuando el bucle termine por agotamiento.

Esto también se aplica a los bucles `While`, cuando se cambia a `False`.

Veamos este ejemplo:

```
In [10]: for n in range(2, 120):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'es igual a', x, '*', n//x)
            break
    else:
        # el bucle o loop falla cuando no encuentra un rango
        print(n, 'es un numero primo')
```

```
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
10 es igual a 2 * 5
11 es un numero primo
12 es igual a 2 * 6
13 es un numero primo
14 es igual a 2 * 7
15 es igual a 3 * 5
16 es igual a 2 * 8
17 es un numero primo
18 es igual a 2 * 9
19 es un numero primo
20 es igual a 2 * 10
```

Por otro lado, las sentencias `continue` (también tomadas de C) continúan con la siguiente iteración del bucle:

```
In [9]: for num in range(2, 10):
    if num % 2 == 0:
        print("Numero Par", num)
        continue
    print("Numero Impar", num)

Numero Par 2
Numero Impar 3
Numero Par 4
Numero Impar 5
Numero Par 6
Numero Impar 7
Numero Par 8
Numero Impar 9
```

Ejercicio 6

Dadas las siguientes lista
listas_anidadas = [['Electrico', 'Gasolina', 'Diesel'],['Tesla', 'Porsche', 'Mercedes']]
Desarrollar una solucion que me diga cual es el indice de la palabra 'Porche' utilizando el bucle for

In []:

Bucle while

Una declaración de bucle while en el lenguaje de programación Python ejecuta repetidamente una declaración de destino siempre que una condición dada sea verdadera.

Cuando la condición se vuelve falsa, el control del programa pasa a la línea que sigue inmediatamente al ciclo.
Creemos una miniserie de Fibonacci para comprender cómo funcionan los bucles While:

```
In [8]: a, b = 0, 1
while b < 10:
    print(b)
    a, b = b, a+b
```

1
1
2
3
5
8

```
In [7]: value = True
while value:
    print('Todo esta bien!')
    break
```

Todo esta bien!

```
In [6]: value = True
while value:
    print('Todo esta bien!')
    value = False
```

Todo esta bien!

```
In [5]: value = True
while value == True:
    name = 'Paco'
    x = input('Cual es mi nombre? ')
    if x != name:
        while value == True:
            x = input('Sigue intentando! Cual es mi nombre? ')
            if x != name:
                pass
            else:
                print('Adivinaste, Mi nombre es '+name)
                value= False
    else:
        print('Adivinaste, Mi nombre es '+name)
        value = False
```

Cual es mi nombre? Juan
Sigue intentando! Cual es mi nombre? Manuel
Sigue intentandol Cual es mi nombre? Paco
Adivinaste, Mi nombre es Paco

Ejercicio 7 - Piedra, Papel y Tijeras!

Crea un juego de piedra, papel y tijera para dos jugadores.

¡Recibe el usuario de cada jugador, compáralo y hazles saber quién ganó!
Si quieres mejorarlo, intenta agregar una puntuación para ver quién gana.

In []:

Aprendizaje Continuo

La mayoría de lo que se puede aprender de Python puede ser encontrado en internet, toneladas de sitios mostrando tutoriales, ejercicios y foros que te pueden ayudar a mejorar tus habilidades de programación.

Sin embargo, aquí puedes encontrar algunos enlaces útiles para mejorar tus habilidades:

[Applied Data Science with Python](#)

[Python Org Website](#)

[Stack Overflow](#)

[Machine Learning with Python](#)

[Google's Python Class](#)