

Curso Intensivo Python - Día 2

Gabriel Valenzuela

@elestudianteclasses

Verano 2021

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Agenda

- Los temas a ver el día de hoy comprenden:
 - Estructuras de decisión
 - Estructuras de repetición
 - Estructuras de datos

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

What if ?...

- Estamos acostumbrados a tomar decisiones en el día a día de nuestras vidas, las cuales tienen una consecuencia asociada

What if ?...

- Estamos acostumbrados a tomar decisiones en el día a día de nuestras vidas, las cuales tienen una consecuencia asociada



What if ?...

- En Python, como en la mayoría de los lenguajes existen variables que se denominan **booleanas**

What if ?...

- En Python, como en la mayoría de los lenguajes existen variables que se denominan **booleanas**
- En Python éstas pueden tomar el valor de **verdad** (True) o **falso** (False)
¡Ojo! Aquí las mayúsculas son importantes.

What if ?...

- En Python, como en la mayoría de los lenguajes existen variables que se denominan **booleanas**
- En Python éstas pueden tomar el valor de **verdad** (True) o **falso** (False)
¡Ojo! Aquí las mayúsculas son importantes.
- Se considera como verdad a toda variable que sea distinto de **0 (cero)**

What if ?...

- El uso de las variables booleanas resultan imprescindibles para las **estructuras de decisión**

What if ?...

- El uso de las variables booleanas resultan imprescindibles para las **estructuras de decisión**
- Se entiende por estructura de decisión o control como un diseño lógico que controla el **orden** en el cuál *un conjunto* de sentencias o instrucciones se ejecutan.

What if ?...

- El uso de las variables booleanas resultan imprescindibles para las **estructuras de decisión**
- Se entiende por estructura de decisión o control como un diseño lógico que controla el **orden** en el cuál *un conjunto* de sentencias o instrucciones se ejecutan.
- Hemos usado hasta ahora el tipo más simple de estructura de control, la estructura de secuencia, la cuál no es nada mas ni nada menos que un conjunto de sentencias que se ejecutan en el orden que aparecen (*De la forma en la cuál el intérprete las lee*)

What if ?...

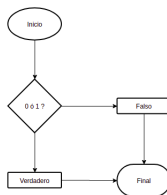
- Aunque esta estructura de secuencia es la que generalmente usamos cuando programamos, no puede manejar todos los tipos de tareas, por ejemplo, elegir que sentencias ejecutar bajo una determinada circunstancia.

What if ?...

- Aunque esta estructura de secuencia es la que generalmente usamos cuando programamos, no puede manejar todos los tipos de tareas, por ejemplo, elegir que sentencias ejecutar bajo una determinada circunstancia.
- Es aquí donde entran las estructuras de decisión que definimos en la slide anterior. Para verlo de una forma gráfica, podemos describir el diagrama de flujo de la lógica que tiene una estructura de decisión simple.

What if ?...

- Aunque esta estructura de secuencia es la que generalmente usamos cuando programamos, no puede manejar todos los tipos de tareas, por ejemplo, elegir que sentencias ejecutar bajo una determinada circunstancia.
- Es aquí donde entran las estructuras de decisión que definimos en la slide anterior. Para verlo de una forma gráfica, podemos describir el diagrama de flujo de la lógica que tiene una estructura de decisión simple.



What if ?...

- El diamante representa la condición a evaluar

What if ?...

- El diamante representa la condición a evaluar
- La sentencia **if** es usada para crear las estructuras de decisión básicas, las cuales permiten a un programa tener más de un camino de ejecución.

What if ?...

- El diamante representa la condición a evaluar

- La sentencia **if** es usada para crear las estructuras de decisión básicas, las cuales permiten a un programa tener más de un camino de ejecución.

- El bloque de sentencias es ejecutado solo cuando la expresión a evaluar del if es **True**.

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

What if ?...

- Cuando se usa `if` se puede probar si una variable es **True** o **False** viendo si es distinta de 0 o *None* (El Null de otros lenguajes)

What if ?...

- Cuando se usa `if` se puede probar si una variable es **True** o **False** viendo si es distinta de 0 o *None* (El Null de otros lenguajes)
- Sin embargo, es posible usar los **operadores de relación** u **operadores lógicos** para determinar si existe una relación de orden entre dos valores.

What if ?...

- Cuando se usa `if` se puede probar si una variable es **True** o **False** viendo si es distinta de 0 o *None* (El Null de otros lenguajes)
- Sin embargo, es posible usar los **operadores de relación** u **operadores lógicos** para determinar si existe una relación de orden entre dos valores.

Operador	Significado
>	Mayor que...
<	Menor que...
>=	Mayor O igual que...
<=	Menor O igual que...
==	Igual que...
!=	No igual que...

What if ?...

- Dijimos que el bloque de sentencias de un **if** se ejecuta solo si la condición es **True**

What if ?...

- Dijimos que el bloque de sentencias de un **if** se ejecuta solo si la condición es **True**
- Pero, hay una forma de ejecutar un bloque de sentencias si la condición es **False** y es a través de la palabra reservada **else**
- Veamos algunos ejemplos

if-else

```

1 temperatura = 10
2 if (temperatura > 30): #Las sentencias if SIEMPRE terminan con :
3     #<-- Aquí hay un indentado, SIEMPRE debe ir después de :
4     print("Que calor que hace en el balc\u00F3n de Paul \U0001F3B5")
5     #Sip podemos imprimir emojis si la consola lo soporta
6     #\u00F3 == ó. Las tildes se imprimen mejor con Unicode.
7 else: #De nuevo, SIEMPRE terminar con :
8     print(";Que hermoso día! \U00002744") #Y no olvidarse el indentado

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

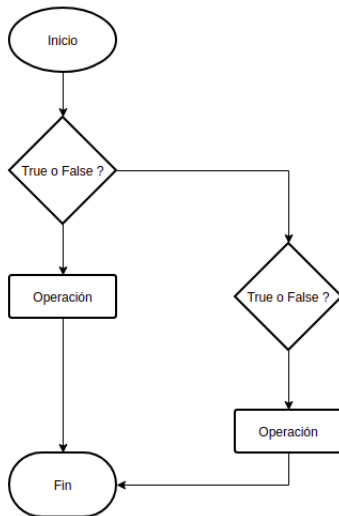
What if ?...

- Existe la posibilidad que si algo es verdad, nos preguntemos si otra condición lo es

What if ?...

- Existe la posibilidad que si algo es verdad, nos preguntemos si otra condición lo es
- Esto se conoce como **condicional anidado** y se puede representar como la siguiente imagen

What if ?...



if-else nested

```

1 temperatura = 30
2 humedad     = 0.1
3 if (temperatura >= 30): #Las sentencias if SIEMPRE terminan con :
4     if (humedad > 0.5):
5         #<-- Indentado del if interno, ojo !
6         print("¡Por qué me persigue la desgracia?!")
7     else:
8         print("Que calor que hace en el balc\u00F3n de Paul \U0001F3B5")
9 else: #De nuevo, SIEMPRE terminar con :
10     if (humedad < 0.2):
11         print("Libre soy, libre soy \U0001F3B5 \U0001F3B5")
12     else:
13         print("¡Que hermoso día! \U00002744") #Y no olvidarse el indentado

```

What if ?...

- Puede tenerse mas anidaciones, y más caminos. Para lo último se usa la palabra reservada **elif**

if-elif-else

```

1 temperatura = 10
2 if (temperatura > 30): #Las sentencias if SIEMPRE terminan con :
3     print("Que calor que hace en el balc\u00F3n de Paul \U0001F3B5")
4 elif (temperatura > 20): #elif DEBE evaluar una condición
5     print("Coffee time !")
6 else: #De nuevo, SIEMPRE terminar con :
7     print("Libre soy, libre soy \U0001F3B5 \U0001F3B5")
    
```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

What if ?...

- **¡Ojo!** Debe tenerse en cuenta, que si la primera condición es **True** el resto de las condiciones **NO** se evalúan.

What if ?...

- **¡Ojo!** Debe tenerse en cuenta, que si la primera condición es **True** el resto de las condiciones **NO** se evalúan.
- Las condiciones que se evalúan pueden hacer eso de los operadores **lógicos**, además de los **relaciones**.

What if ?...

- **¡Ojo!** Debe tenerse en cuenta, que si la primera condición es **True** el resto de las condiciones **NO** se evalúan.
- Las condiciones que se evalúan pueden hacer eso de los operadores **lógicos**, además de los **relaciones**.
- Los operadores lógicos vienen de dos sabores: **and** y **or**. El primero es un *producto*, donde si hacemos **True = 1** y **False = 0** deben cumplirse ambas condiciones si, y solo sí, ambas son verdad.

What if ?...

- **¡Ojo!** Debe tenerse en cuenta, que si la primera condición es **True** el resto de las condiciones **NO** se evalúan.
- Las condiciones que se evalúan pueden hacer eso de los operadores **lógicos**, además de los **relaciones**.
- Los operadores lógicos vienen de dos sabores: **and** y **or**. El primero es un *producto*, donde si hacemos **True = 1** y **False = 0** deben cumplirse ambas condiciones si, y solo sí, ambas son verdad.
- Mientras que el **or**, es una *suma*, donde basta conque una condición sea verdad para que el resultado sea verdad.

What if ?...

- Otro operador útil es el de negación, o **not** que niega una condición. (Si es *True* pasa a ser *False* y viceversa)

What if ?...

- Otro operador útil es el de negación, o **not** que niega una condición. (Si es *True* pasa a ser *False* y viceversa)
- Ambos operadores llevan a cabo una *evaluación de corto-circuito*. En el caso de AND si el lado izquierdo es **False**, el lado derecho se ignora. En cambio en el or, si es **True** el lado derecho se ignora.

Loops...

```
for (int i = 0; i < 3; i++)
```

i = 0

i = 1

i = 2

i = 3



Loops...

- Los desarrolladores generalmente escriben código que lleva a cabo una acción un número limitado (o ilimitado) de veces.

Loops...

- Los desarrolladores generalmente escriben código que lleva a cabo una acción un número limitado (o ilimitado) de veces.
- Para evitar el *código spaghetti* los lenguajes (en su mayoría) incorporan **estructuras de repetición**, las cuales hacen que una sentencia o conjunto de sentencias se ejecuten de forma repetida.

Loops...

- Los desarrolladores generalmente escriben código que lleva a cabo una acción un número limitado (o ilimitado) de veces.
- Para evitar el *código spaghetti* los lenguajes (en su mayoría) incorporan **estructuras de repetición**, las cuales hacen que una sentencia o conjunto de sentencias se ejecuten de forma repetida.
- Dichas estructuras vienen en dos sabores en Python: **controladas por condición** (Condition controlled) y **controladas por contador** (Counter-controlled).

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Loops...

- Un bucle controlado por condición ejecuta una sentencia o conjunto de sentencias mientras que la condición sea **True**. En Python, este bucle se utiliza con la palabra reservada **while**.

Loops...

- Un bucle controlado por condición ejecuta una sentencia o conjunto de sentencias mientras que la condición sea **True**. En Python, este bucle se utiliza con la palabra reservada **while**.

- El ciclo while consiste de dos partes:
 - La condición a evaluar
 - El conjunto de sentencias a ejecutar

Loops...

- Un bucle controlado por condición ejecuta una sentencia o conjunto de sentencias mientras que la condición sea **True**. En Python, este bucle se utiliza con la palabra reservada **while**.
- El ciclo while consiste de dos partes:
 - La condición a evaluar
 - El conjunto de sentencias a ejecutar
- Es un **pre-tested loop**, lo cual quiere decir, que evalúa la condición antes de ejecutar la/s sentencia/s.

Loops...

- Puede darse el caso que la condición sea siempre **True**, en tal caso se tiene un bucle infinito ∞

Loops...

- Puede darse el caso que la condición sea siempre **True**, en tal caso se tiene un bucle infinito ∞
- Es una situación a evitar *excepto* que se tenga control y conocimiento de la misma.

Loops...

- Puede darse el caso que la condición sea siempre **True**, en tal caso se tiene un bucle infinito ∞
- Es una situación a evitar *excepto* que se tenga control y conocimiento de la misma.
- Veamos un ejemplo

while ∞

```

1 condicion = 0
2 while (condicion < 10): # Aquí también debemos usar :
3     print('Cuenta: {}'.format(condicion))
4     #¿Hay error aquí?
5 print('Fin while')
```

while fixed

```

1 condicion = 0
2 while (condicion < 10): # Aquí también debemos usar :
3     print('Cuenta: {}'.format(condicion))
4     condicion += 1 #Forma corta de:
5     #condicion = condicion + 1

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Loops...

- El bucle controlado por contador en Python es el **for**

Loops...

- El bucle controlado por contador en Python es el **for**
- Su ejecución está dada por un número limitado de veces, sin lugar a generar loops infinitos.

Loops...

- El bucle controlado por contador en Python es el **for**
- Su ejecución está dada por un número limitado de veces, sin lugar a generar loops infinitos.
- En su ejecución, el bucle for asigna a una variable temporal los valores (Comenzado por el primero) de una secuencia de datos.

Loops...

- El bucle controlado por contador en Python es el **for**
- Su ejecución está dada por un número limitado de veces, sin lugar a generar loops infinitos.
- En su ejecución, el bucle for asigna a una variable temporal los valores (Comenzado por el primero) de una secuencia de datos.
- Veamos un ejemplo

for

```

1 datos = [1,2,3,4,5]
2 for dato in datos: #dato es la variable temporal. IN siempre se usa
3     print(dato)
4 print('Fin de for')
```

for con range

```

1 rango_v1 = range(5) #0,1,2,3,4
2 rango_v2 = range(-5,5) #-5,-4,-3,-2,-1,0,1,2,3,4
3 rango_v3 = range(-5,5,2) #-5,-3,-1,1,3
4 #range(inicio,final,salto) es una función definida en Python
5 #Viene de 3 sabores:
6 # -> range(final):
7 # genera una secuencia de datos desde 0 hasta final - 1
8 # -> range(inicio,final):
9 # genera una secuencia de datos desde inicio hasta final - 1
10 # -> range(inicio,final,salto):
11 # genera una secuencia de datos desde inicio hasta final - 1, cada salto valor
12 for valor in rango_v1:
13     print(valor,end='\t')
14 print('\nFin for v1')
15 for valor in rango_v2:
16     print(valor,end='\t')
17 print('\nFin for v2')
18 for valor in rango_v3:
19     print(valor,end='\t')
20 print('\nFin for v3')

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Loops...

- Los bucles son imprescindibles cuando se tienen que hacer:
 - Acumuladores (Sumadores)
 - Centinelas: Variables que controlan la ejecución hasta que se da una condición
 - Validadores de entrada: Solo se avanza cuando el usuario introduce un dato válido (Para el programador)

Loops...

- Los bucles son imprescindibles cuando se tienen que hacer:
 - Acumuladores (Sumadores)
 - Centinelas: Variables que controlan la ejecución hasta que se da una condición
 - Validadores de entrada: Solo se avanza cuando el usuario introduce un dato válido (Para el programador)

- Además para los menues de consola interactivos son muy útiles.

Loops...

- Los bucles son imprescindibles cuando se tienen que hacer:
 - Acumuladores (Sumadores)
 - Centinelas: Variables que controlan la ejecución hasta que se da una condición
 - Validadores de entrada: Solo se avanza cuando el usuario introduce un dato válido (Para el programador)
- Además para los menues de consola interactivos son muy útiles.
- Existen dos palabras reservadas para controlar la ejecución de un bucle:
 - break** la cual provoca que se termine la ejecución de forma inmediata y
 - continue** la cual provoca que se comience una iteración nueva de forma inmediata (iteración == nueva ejecución desde el comienzo del bucle)

Loops...

- Los bucles son imprescindibles cuando se tienen que hacer:
 - Acumuladores (Sumadores)
 - Centinelas: Variables que controlan la ejecución hasta que se da una condición
 - Validadores de entrada: Solo se avanza cuando el usuario introduce un dato válido (Para el programador)

- Además para los menues de consola interactivos son muy útiles.

- Existen dos palabras reservadas para controlar la ejecución de un bucle:
 - break** la cual provoca que se termine la ejecución de forma inmediata y
 - continue** la cual provoca que se comience una iteración nueva de forma inmediata (iteración == nueva ejecución desde el comienzo del bucle)

- Como los condicionales, los bucles se pueden anidar para tener un bucle dentro de otro bucle, independiente su tipo. Veamos algunos ejemplos

break

```

1 import random as rd
2
3 aleatorio = 0
4 intentos = 0
5
6 ACIERTO = 7 #Declaracion de valor constante (Mayúsculas)
7
8 while(True): #bucle infinito...
9     aleatorio = rd.randint(0,10)
10    if(aleatorio == ACIERTO):
11        break #Termina ejecución de bucle y las
12        #sentencias posteriores NO se ejecutan
13    else:
14        intentos += 1
15 print('Se necesitaron {0} intentos'.format(intentos))

```

continue

```

1 import random as rd
2
3 aleatorio = 0
4 intentos = 0
5
6 ACIERTO = 7 #Declaracion de valor constante (Mayúsculas)
7
8 while(True): #bucle infinito...
9     aleatorio = rd.randint(0,10)
10    if(aleatorio == ACIERTO):
11        break #Termina ejecución de bucle y las
12        #sentencias posteriores NO se ejecutan
13    else:
14        continue #Se vuelve al comienzo del loop
15        #¿Qué produce esto?
16        intentos += 1
17 print('Se necesitaron {0} intentos'.format(intentos))

```

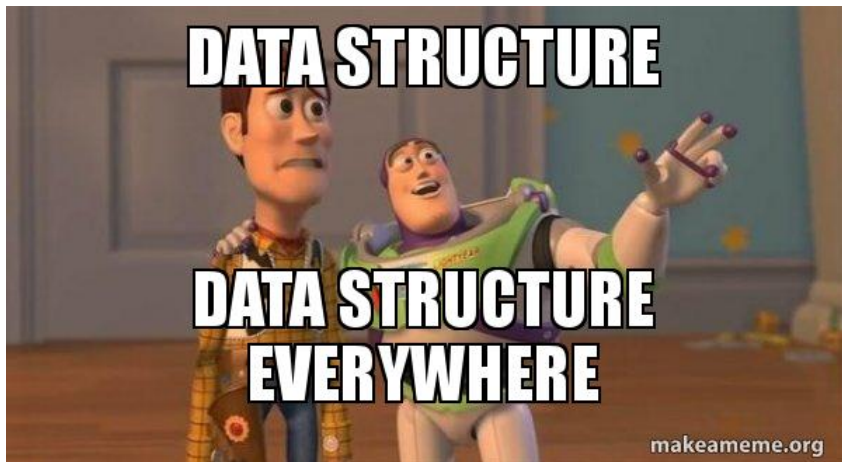
nested loops

```

1 import random as rd
2
3 MAX_CANTIDAD_ALUMNOS      = 9
4 MAX_CANTIDAD_EXAMENES    = 3
5 cantidadAlumnos          = 0
6 total                    = 0
7 promedio                  = 0
8
9 while(cantidadAlumnos < MAX_CANTIDAD_ALUMNOS):
10     print('Alumno N°{0}'.format(cantidadAlumnos+1))
11     print('-----')
12     for nota in range(MAX_CANTIDAD_EXAMENES):
13         total += rd.randint(2,11) #2 - 10 valores
14     promedio = total/MAX_CANTIDAD_EXAMENES
15     print('Promedio obtenido {0:.2f}'.format(promedio))
16     print('-----')
17     total = 0
18     promedio = 0
19     cantidadAlumnos += 1
20 print('Fin programa')

```

Data Structures



Hagamos una analogía...



Data structures

- Si imaginamos la memoria como los casilleros, vemos que los valores ocupan un determinado lugar

Data structures

- Si imaginamos la memoria como los casilleros, vemos que los valores ocupan un determinado lugar
- El número de casillero, representa el *índice* del dato y el casillero el *dato* propiamente dicho

Data structures

- Si imaginamos la memoria como los casilleros, vemos que los valores ocupan un determinado lugar
- El número de casillero, representa el *índice* del dato y el casillero el *dato* propiamente dicho
- Cuando programamos, podemos necesitar hacer referencia a un conjunto de valores a través de una sola variable, y es por tal motivo que existen estas estructuras de datos.

Data structures

- Si imaginamos la memoria como los casilleros, vemos que los valores ocupan un determinado lugar
- El número de casillero, representa el *índice* del dato y el casillero el *dato* propiamente dicho
- Cuando programamos, podemos necesitar hacer referencia a un conjunto de valores a través de una sola variable, y es por tal motivo que existen estas estructuras de datos.
- Podemos tener múltiples valores, ordenarlos, buscar valores determinados, etc. Python ofrece estructuras de datos de diversos sabores y con diversas aplicaciones, cada una tiene sus fortalezas y debilidades

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Data structures

- La **lista** es la más conocida y posiblemente mas utilizada

Data structures

- La **lista** es la más conocida y posiblemente mas utilizada
- Consiste en una secuencia de valores, de **diversos tipos**, **muteable** y **dinámica**

Data structures

- La **lista** es la más conocida y posiblemente mas utilizada
- Consiste en una secuencia de valores, de **diversos tipos, mutable y dinámica**
- Es decir, una lista puede contener datos enteros, flotantes, cadenas u otra lista conviviendo sin problemas

Data structures

- La **lista** es la más conocida y posiblemente mas utilizada
- Consiste en una secuencia de valores, de **diversos tipos, mutable y dinámica**
- Es decir, una lista puede contener datos enteros, flotantes, cadenas u otra lista conviviendo sin problemas
- Al ser mutable, estos datos pueden modificarse si es necesario y que sea dinámica quiere decir que podemos solicitar mas espacio para almacenar datos cuando querramos.

Data structures

- La lista se declara con []

Data structures

- La lista se declara con []

- Podemos darle una representación de forma gráfica:

Data structures

- La lista se declara con []
- Podemos darle una representación de forma gráfica:



Data structures

- Es posible definir una lista mediante el operador de repetición

Data structures

- Es posible definir una lista mediante el operador de repetición
- Este operador es el `*` y se utiliza como

Data structures

- Es posible definir una lista mediante el operador de repetición
- Este operador es el `*` y se utiliza como

`[lista] * numeroRepeticiones`

Definición de lista

```

1 listaVacía = [] #Se declara una lista vacía, sin elementos
2 listaConValores = [1,'a',3.14,[8,9]] #Valores declarados por el programador
3 listaConRepetición = [0]*10 #Se crea una lista con 10 elementos 0s.
4 print(listaVacía)
5 print(listaConValores)
6 print(listaConRepetición)

```

Data structures

- Los elementos de una lista se pueden acceder mediante un ciclo for, debido a que es una secuencia de datos

Data structures

- Los elementos de una lista se pueden acceder mediante un ciclo for, debido a que es una secuencia de datos
- También mediante el operador de **indexación** (`[]`), con el cuál accedemos mediante el índice, es decir, la posición del dato

Data structures

- Los elementos de una lista se pueden acceder mediante un ciclo for, debido a que es una secuencia de datos
- También mediante el operador de **indexación** (`[]`), con el cuál accedemos mediante el índice, es decir, la posición del dato
- **¡Importante!** En la mayoría de los lenguajes, excepto Matlab, los índices arrancan en **0** y terminan en **LongitudDato - 1**

Datos en lista

```

1 import random as rd
2
3 lista = list(range(20))
4 #Range devuelve una secuencia de 0 a 19
5 #Esa secuencia se pasa a la función list que la transforma
6 #en una lista
7 print('Lista desde for')
8 for dato in lista:
9     print('El dato es {}'.format(dato))
10 #El resultado sería igual a
11 print('\nLista desde for accedido por indice')
12 for indice in range(len(lista)): #len(secuencia) nos devuelve la longitud
13     #de una secuencia, al ser un valor entero, range lo toma y calcula la
14     #secuencia desde 0 hasta longitudSecuencia - 1
15     print('El dato es {}'.format(lista[indice])) #El operador de indexación es
16 print('-----')
17 #A través de dicho operador se puede llevar a cabo las modificaciones
18 indiceAleatorio = rd.randint(0,len(lista)-1) #0,19
19 numeroAleatorio = rd.randint(-len(lista),len(lista)) #-20,20
20 print('Lista antes de modificarse: {}'.format(lista))
21 lista[indiceAleatorio] = numeroAleatorio
22 print('Lista despues de modificarse: {} en {} con el valor {}'.format
23 (lista,indiceAleatorio,numeroAleatorio))

```

Data structures

- Las listas se pueden concatenar (unir o join) mediante el operador `+`

Data structures

- Las listas se pueden concatenar (unir o join) mediante el operador `+`
- El operador de repetición internamente hace eso: Crea N copias de listas y las concatena en una lista final

Data structures

- Las listas se pueden concatenar (unir o join) mediante el operador `+`

- El operador de repetición internamente hace eso: Crea N copias de listas y las concatena en una lista final

- Además de concatenarse, pueden copiarse, pero... **¡Cuidado!**

Concatenacion y copia

```

1 listaPares = list(range(0,10,2))
2 listaImpares = list(range(1,9,2))
3 print('Lista pares {0} \nE impares {1}'.format(listaPares,listaImpares))
4 listaTotal = listaPares + listaImpares
5 print(listaTotal)
6 print('\n-----\n')
7 #Creo una copia
8 copiaLista = listaTotal
9 print('Lista original {0} \nCopia: {1}'.format(listaTotal,copiaLista))
10 print('-----')
11 listaTotal[0] = 10
12 print('Lista original {0} \nCopia: {1}'.format(listaTotal,copiaLista))
13 #¿Qué pasó?

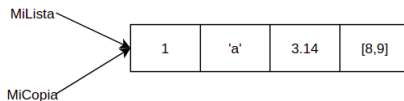
```

Data structures

- Asignar una lista a una nueva lista hace que ambas variables hagan referencia a la misma posición de memoria, es decir, a la misma lista.

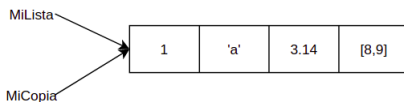
Data structures

- Asignar una lista a una nueva lista hace que ambas variables hagan referencia a la misma posición de memoria, es decir, a la misma lista.



Data structures

- Asignar una lista a una nueva lista hace que ambas variables hagan referencia a la misma posición de memoria, es decir, a la misma lista.



- Para evitar esto, debemos hacer una copia elemento a elemento de la lista original a la nueva lista. O concatenar la lista original con una **lista vacía** y asignar esa concatenación a la nueva lista.

Data structures

- A partir de una lista original, además de copiar los elementos, es posible generar una *sublista* de los elementos a través del **slicing**

Data structures

- A partir de una lista original, además de copiar los elementos, es posible generar una *sublista* de los elementos a través del **slicing**
- La expresión de slicing selecciona un rango de elementos de una secuencia, un **slice** es un generador de elementos que son tomados de una secuencia

Slicing

```

1 lista = list(range(10))
2 print(lista) #Lista original
3 primerosTres = lista[0:3] #[0,1,2]
4 #El slicing se compone de la forma:
5 # lista[inicio : fin]
6 #Y toma los elementos desde inicio, hasta
7 # (fin-1)
8 ultimosTres = lista[-3:]
9 #La ultima posición es -1, y desde esa posición
10 #hacia adelante se decrementa en 1
11 print(primerosTres)
12 print(ultimosTres)
13 tomaDeADos = lista[:2] #Si no se establece
14 #el inicio o fin, se toma la primera y ultima
15 #posición de la lista
16 print(tomaDeADos)

```

Data structures

- El uso de índices inválidos durante el slicing no **genera una excepción**

Data structures

- El uso de índices inválidos durante el slicing no **genera una excepción**

- Sino que:
 - Si la posición final está más allá del final de la lista, Python usará la longitud de la lista
 - Si la posición inicial está antes de la posición inicial de la lista, Python usará la posición 0
 - Si el inicio es mayor al final, el slicing retornará una lista vacía.

Data structures

- Es posible buscar por un elemento en una lista haciendo uso del operador **in**, o **not in** para buscar un elemento que *no* está en la lista.

Data structures

- Es posible buscar por un elemento en una lista haciendo uso del operador **in**, o **not in** para buscar un elemento que *no* está en la lista.
- Python provee también métodos para trabajar con listas

Búsqueda

```

1 dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
2 buscarDia = input('Ingrese día: ')
3 if buscarDia in dias:
4     print('El día existe')
5 else:
6     print('404 - Calendario not found')

```

Métodos

```

1 lista = list(range(5))
2 print('Lista original: {0}'.format(lista))
3 print('Método append')
4 #append(x) añade un elemento al final de la lista
5 lista.append(5)
6 print('Lista modificada: {0}'.format(lista))
7 print('Método insert')
8 #insert(i,x) inserta un elemento en una posición
9 #determinada. Si existe elemento en esa posición,
10 #se desplazan los otros elementos
11 lista.insert(6,6)
12 print('Lista modificada: {0}'.format(lista))
13 print('Método remove')
14 #remove(x) remueve el PRIMER elemento cuyo valor
15 #es igual a x

```

Métodos

```

16 lista.insert(6,6)
17 print('Lista modificada: {0}'.format(lista))
18 lista.remove(6)
19 print('Lista modificada: {0}'.format(lista))
20 print('Método count')
21 #Cuenta la cantidad de veces que x aparece
22 print('Cuenta {0}'.format(lista.count(6)))
23 print('Lista modificada: {0}'.format(lista))
24 print('Método index')
25 #Retorna el índice del primer elemento cuyo valor
26 #es igual x. PUEDE GENERAR UN ValueError sino
27 #se encuentra
28 print('Valor en índice {0}: {1}'.format(6,lista.index(6)))
29 print('Método clear')
30 #Borra todos los elementos de la lista
31 lista.clear()
32 print('Lista modificada: {0}'.format(lista))
33 print('-----Fin programa-----')

```

Lista anidada

```

1 import random as rd
2
3 FILAS      = 4
4 COLUMNAS   = 3
5
6 matrix = [[0]*COLUMNAS]*FILAS
7 print('Matrix vacia {0}'.format(matrix))
8
9 for i in range(FILAS):
10     for j in range(COLUMNAS):
11         matrix[i][j] = rd.randint(0,(FILAS*COLUMNAS))
12         #Para matrices se usa el primer
13         #[] para fila y el segundo [] la columna
14 print('Matrix modificada {0}'.format(matrix))

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Data structures

- Las tuplas no difieren de las listas, una **tupla** es una secuencia *immutable*, lo cual significa que su contenido no puede modificarse

Data structures

- Las tuplas no difieren de las listas, una **tupla** es una secuencia *immutable*, lo cual significa que su contenido no puede modificarse
- Se pueden crear iniciandolas o usando la función **tuple()** que toma una secuencia y la transforma en tupla.

Data structures

- Las tuplas no difieren de las listas, una **tupla** es una secuencia *immutable*, lo cual significa que su contenido no puede modificarse
- Se pueden crear iniciandolas o usando la función **tuple()** que toma una secuencia y la transforma en tupla.
- Las tuplas soportan todas las operaciones de las lista, excepto aquellas que cambian su contenido. Es decir:
 - Acceso por indexación
 - Métodos como **index**
 - Built-in como **len**, **min** y **max**
 - Slicing
 - Operador **in**
 - Operador **+** y *****

Data structures

- El porqué de las tuplas es que tienen una mejor performance que las listas debido a su característica de inmutabilidad. Otra razón es que son *seguras*, debido a que no permiten cambiar los datos.

Tupla

```

1 tupla = tuple(range(10))
2 print('Tupla {0}'.format(tupla))
3 tuplaVacia = ()
4 print('Tupla vacía {0}'.format(tuplaVacia))
5 generador = (0,) * 10
6 print('Generador {0}'.format(generador))
7 concatenacion = generador + tupla
8 print('Concatenacion {0}'.format(concatenacion))
9 print(concatenacion[18]) #Acceso elemento
10 print(tupla[1:5]) #Slicing

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

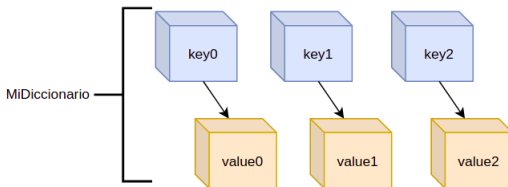
- Listas
- Tuplas
- Diccionarios
- Conjunto

Data structures

- Los diccionarios forman es una estructura de datos del tipo mapa, donde una llave está relacionada con un valor, en una relación 1 a 1

Data structures

- Los diccionarios forman es una estructura de datos del tipo mapa, donde una llave está relacionada con un valor, en una relación 1 a 1



Data structures

- La llave se usa para acceder al valor

Data structures

- La llave se usa para acceder al valor
- Para crear un diccionario se deben encerrar los valores entre `{}`, separando los valores por `,` y escribiendo **llave:valor**

Data structures

- La llave se usa para acceder al valor

- Para crear un diccionario se deben encerrar los valores entre `{}`, separando los valores por `,` y escribiendo **llave:valor**

- Las llaves dentro del diccionario deben ser objetos inmutables, ejemplo: cadenas, enteros, numeros flotantes, tuplas pero no listas

Diccionario

```

1 agenda = {'Batman': '555-1111', 'Iron Man': '555-2222', 'Wonder Woman': '555-3333'}
2 print(agenda)
3 #Para acceder a los datos del diccionario se usa el operador
4 #[]. SI LA LLAVE NO EXISTE GENERA UN KEYERROR
5 print('El teléfono de Batman es {0}'.format(agenda['Batman']))
6 #Para añadir un dato, hacemos uso del mismo operador
7 agenda['Superman'] = '555-5555'
8 print(agenda)
9 #Nota: Las comparaciones de cadenas son sensibles a mayúsculas
10 #Puede usarse in y not in para prevenir los KeyError
11 if 'Guason' in agenda:
12     print('HAAAAAAAAHAHAHA')
13 else:
14     print('Falta the joker')
15 #Para eliminar un elemento hacemos uso de la funcion del
16 del agenda['Iron Man']
17 print(agenda)

```

Diccionario

```

18 #La funcion len nos dice cuantos elementos tiene el diccionario
19 print('Diccionario de {0} elementos'.format(len(agenda)))
20 #Los diccionarios pueden tener multiples valores
21 dicc = {'A':1, 'B':3.14, 'C':[0,1,2], 'D':'E'}
22 print(dicc)
23 #Un diccionario puede crearse vacío
24 diccionarioVacio = {}
25 #O con la funcion dict()
26 diccionarioDict = dict(m=8, n=9)
27 #Algunos métodos son
28 dicc.clear() #Elimina todos los elementos
29 dicc.get('Batman', 'No key') #Obtiene un valor por la llave
30 #Y sino existe devuelve el valor por defecto sin generar
31 #excepción
32 dicc.items() #Retorna el conjunto de llave-valor
33 dicc.keys() #Retorna todas las llaves del diccionario en una secuencia
34 dicc.pop('Batman', 'No key') #Similar a get para remover elemento
35 dicc.values() #Retorna todas los valores del diccionario en una secuencia

```

1 Agenda

2 Curso Intensivo de Python - Día 2

3 Condicionales

- Operadores de comparación
- Condicionales anidados
- Operadores lógicos

4 Estructuras de repetición

- Bucle while
- Bucle for
- Control de bucle y anidamiento

5 Estructuras de datos

- Listas
- Tuplas
- Diccionarios
- Conjunto

Data structures

- Un **set** contiene una colección de valores únicos y trabaja de forma similar a un conjunto matemático

Data structures

- Un **set** contiene una colección de valores únicos y trabaja de forma similar a un conjunto matemático
- Los datos que almacenan deben ser **únicos**, de forma **no ordenada** y pueden ser **de diferentes tipos**

Conjuntos

```

1 myConjunto = set() #Crea un conjunto vacío, puede tomarse
2 #un objeto iterable (lista, tupla, cadena)
3 myConjunto = set('aaabc')
4 print(myConjunto)
5 myConjuntoDos = set(['a', 'b', 'c'])
6 myConjuntoTres = set(['a', 'd', 'e'])
7 print('Longitud del conjunto {0}'.format(len(myConjunto)))
8 myConjunto.add('d') #add(x) añade el elemnto x al conjunto
9 print(myConjunto)
10 myConjunto.update(['abc', 'def']) #update(iterable) añade una secuencia de valores
11 print(myConjunto)
12 myConjunto.discard('d') #discard(x) remueve x del conjunto
13 print(myConjunto)

```


Conjuntos

```

14 #Para iterar se puede usar un ciclo for
15 for dato in myConjunto:
16     print(dato)
17 #Y in o not in para buscar un valor
18 if 'abc' in myConjunto:
19     print('abc!')
20 #Los conjuntos se pueden "concatenar" mediante la union
21 print(myConjuntoDos.union(myConjuntoTres))
22 #Los elementos son todos los del conjunto uno y dos
23 #Se puede hacer una intersección también
24 print(myConjuntoDos.intersection(myConjuntoTres))
25 #Solo tienen los valores que tienen en común ambos conjuntos
26 #La diferencia retorna los elementos que están en el primer
27 #conjunto y no en el segundo
28 print(myConjuntoDos.difference(myConjuntoTres))
29 #La diferencia simétrica devuelve los elementos que
30 #se encuentran en el primer o segundo conjunto pero no
31 #en ambos
32 print(myConjuntoDos.symmetric_difference(myConjuntoTres))
33 #Se puede saber si un subconjunto está dentro de un conjunto
34 #O superconjunto
35 print(myConjuntoDos.issubset(myConjuntoTres))
36 print(myConjuntoDos.issuperset(myConjuntoTres))

```

Unicode


Emoji Charts

Emoji List, v13.1


[Index & Help](#) | [Issues & Rights](#) | [Serc](#) | [Processing Additions](#)

This chart provides a list of the Unicode emoji characters and sequences, with single image and annotations. Clicking on a Sample goes to the emoji in the [full list](#). The ordering of the emoji and the annotations are based on [Unicode CLDR data](#). Emoji sequences have more than one code point in the **Code** column. [Recently-added emoji](#) are marked by a @ in the name and outlined images.








Emoji with skin-tones are not listed here: see [Full Skin Tone List](#).

For counts of emoji, see [Emoji Counts](#).

While these charts use a particular version of the [Unicode Emoji data files](#), the images and format may be updated at any time. For any production usage, consult those data files. For information about the contents of each column, such as the **CLDR Short Name**, click on the column header. For further information, see [Index & Help](#).



Adopt a Character
TM
Anna GanderFinger

Smileys & Emotion				
face-smiling				
No	Code	Sample	CLDR Short Name	Other Keywords
1	U+1F600		grinning face	face grin grinning face
2	U+1F603		grinning face with big eyes	face grinning face with big eyes mouth open smile
3	U+1F604		grinning face with smiling eyes	eye face grinning face with smiling eyes mouth open smile
4	U+1F605		beaming face with smiling eyes	beaming face with smiling eyes eye face grin smile
5	U+1F606		grinning squinting face	face grinning squinting face laugh mouth satisfied smile
6	U+1F607		grinning face with sweat	cold face grinning face with sweat open smile sweat
7	U+1F60A		rolling on the floor laughing	face floor laugh roll rolling rolling on the floor laughing rotfl

Disponible aquí

Unicode

Unicode Character Table 🔍 🇬🇧 English

Unicode Emoji Sets Tools Alphabets HTML Entities Alt codes Holidays

Popular character sets See all >

→
Arrows
”
Quotation Marks
❤️
Hearts
★
Stars
🎭
Fancy Letters
Ⓐ
Symbols for Nickname
😂
Top 50 Emoji
Σ
Math Symbols

0 1 2 3 4 5 6 7 8 9 A B C D E F 🌐 📱 📺 2.7%

0000
0010
0020		!	"	#	\$	%	&	'	()	*	+	,	-	.
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n
0070	o	p	q	r	s	t	u	v	w	x	y	z	{		}
0080	~														

<https://unicode-table.com/en/sets/fancy-letters/>

Basic Latin ▾

[Open in an individual page](#)

Range: 0000–007F

Q: [Click to highlight range](#)

T: [Click to highlight range](#)

Languages: english, german, french, italian, polish

Disponible aquí

¿Preguntas?