tuplas-listas-diccionarios

January 28, 2020

1 Tuplas, Listas y Diccionarios

Hasta ahora hemos trabajado con tipos de data como `str`, `int` y `float`. Muchos problemas comunes son mas facil de resolver cuando ciertos tipos de datos son combinados para crear estructuras de datos mas complejas.

Una **estructura de dato** modela una coleccion de data, como una lista de numeros, una fila de una hoja tabulada o un registro en una base de datos. Modelar la data que el programa utiliza, usando la estructura de dato correcta, es muchas veces la llave para escribir codigo simple y efectivo.

Python tiene tres estructuras de datos incorporadas que son el enfoque de este capitulo: 1. tuplas 2. listas 3. diccionarios

En esta seccion, aprendermos: * Como trabajar con tuplas, listas y diccionarios * Que es la inmutabilidad y porque es importante * Cuando utilizar estructuras de datos distintas

1.1 Las tuplas son secuencias inmutables

Una **secuencia** es una lista ordenada de valores. Cada elemento en una secuencia tiene un numero entero asignado, denominado **indice**, que determina el orden en el cual su valor aparece.

Por ejemplo, las letras del alfabeto son una secuencia que empieza con la letra A y termina con la letra Z. Los strings tambien son secuencias. El string `Panama` tiene 6 elementos, comenzando con `P` en el indice `0` y terminando con `a` en el indice `5`.

Algunas aplicaciones reales de las secuencias incluyen valores emitidos por un sensor cada segundo, la secuencia de las notas estudiantiles, o la secuencia de los valores de las acciones en una empresa sobre un periodo de tiempo.

En esta seccion aprenderemos como utilizar el tipo de dato incorporado `tuple`, para crear una secuencia de valores.

1.1.1 Que es una tupla?

La palabra proviene de la matematica, que es utilizado para describir una secuencia ordenada de valores. Usualmente los matematicos escriben tuplas ingresando cada elemento separado por una coma, dentro de un parentesis, e.g. (1, 2, 3) es una tupla que contiene 3 numeros enteros.

Tuplas son ordenadas porque sus elementos aparecen en orden. El primero elemento de `(1, 2, 3)` es `1`, el segundo elemento es `2` y el tercero es `3`.

Python utiliza el mismo nombre y manera de escribirla (notacion)

1.1.2 Como crear una Tupla

Existen algunas maneras de crear una tupla en Python. Vamos a explicar dos de ellas: 1. Literales de Tupla 2. El incorporado `tuple()`

1.1.3 Literales de Tupla

Asi como el literal string es un string que es explicitamente creado por rodear algo de texto con citas o comillas, un literal de tupla es una tupla que esta explicitamente escrita como una lista de valores separados con coma y rodeados de parentetis.

[2]:
```
primera_tupla = (1, 2, 3)
```

[3]:
```
type(primera_tupla)
```

[3]:
```
tuple
```

[4]:
```
# las tuples pueden guardar cualquier tipo de elemento
segunda_tupla = (1, True, 'hola')
segunda_tupla
```

[4]:
```
(1, True, 'hola')
```

[6]:
```
# tambien podemos tener una tupla vacia
tupla_vacia = ()
tupla_vacia
```

[6]:
```
()
```

[10]:
```
# esto no crea una tupla
x = (1)
type(x)
```

[10]:
```
int
```

[11]:
```
x
```

[11]:
```
1
```

[13]:
```
# necesitamos la coma
x = (1,)
type(x)
```

[13]:
```
tuple
```

[14]:
```
x
```

[14]:
```
(1,)
```

1.1.4 tuple()

[15]:
```
tuple('Panama')
```

[15]:
```
('P', 'a', 'n', 'a', 'm', 'a')
```

[16]:
```
# tuple acepta unicamente un solo parametro
tuple(1, 2, 3)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-16-b13ab3c7d441> in <module>
  1 # tuple acepta unicamente un solo parametro
----> 2 tuple(1, 2, 3)


TypeError: tuple expected at most 1 argument, got 3
```

[17]:
```
tuple(1)  # el parametro tiene que ser iterable / secuencia
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-17-69b6b14cc30a> in <module>
----> 1 tuple(1)


TypeError: 'int' object is not iterable
```

[18]:
```
# esto si sirve
tuple()
```

[18]:
```
()
```

1.1.5 Similitudes entre Tuplas y Strings

Las Tuplas y los Strings tienen mucho en comun: * ambos son tipos de secuencia con finita longitud, * ambos aceptan indexacion y segmentacion, * ambos son inmutables y * ambos pueden ser iterados en un ciclo.

La diferencia principal entre strings y tuplas es que los elementos de las tuplas pueden ser de cualquier valor, mientras que los elementos de un string solo pueden ser caracteres.

[19]:
```
# Tuplas tienen longitud
numeros = (1, 2, 3)
len(numeros)
```

[19]: 3

[22]:
```
# Tuplas tinen indexacion
valores = (1, 3, 5, 7, 9)
valores[2]
```

[22]: 5

[23]:
```
# Tuplas tienen segmentacion
valores[2:4]
```

[23]: (5, 7)

[24]:
```
# Tuplas son inmutables
valores[0] = 2
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-24-503b63720143> in <module>
  1 # Tuplas son inmutables
----> 2 valores[0] = 2


TypeError: 'tuple' object does not support item assignment
```

[26]:
```
# tuplas son iterables
vocales = ('a', 'e', 'i', 'o', 'u')
for vocal in vocales:
    print(vocal.upper())
```

```
A
E
I
O
U
```

[28]:
```
# empaque y desempaque de Tuplas
coordenadas = 4.21, 9.29
type(coordenadas)
```

[28]:
```
tuple
```

[29]:
```
x, y = coordenadas
print(x, y)
```

```
4.21 9.29
```

[1]:
```
nombre, edad, ocupacion = 'Juan', 30, 'programador'
```

[2]:
```
print(nombre, edad, ocupacion)
```

```
Juan 30 programador
```

Esto funciona porque los valores de la derecha son insertados en una Tupla y luego los valores son desempacados en tres variables: `nombre`, `edad` y `ocupacion`.

1.1.6 Revisando la existencia de valores con `in`

[32]:
```
vocales = ('a', 'e', 'i', 'o', 'u')
```

[33]:
```
'e' in vocales
```

[33]:
```
True
```

[34]:
```
'f' in vocales
```

[34]:
```
False
```

1.1.7 Retornando multiples valores de una funcion

[35]:
```
# uso comun de de tuplas
def agrega_resta(num1, num2):
    return (num1 + num2), num1 - num2
```

[36]:
```
agrega_resta(3, 2)
```

[36]:
```
(5, 1)
```

1.1.8 Ejercicios

1. Crea un literal de tupla llamado `numeros_cardinales` que contiene los strings `primero`, `segundo` y `tercero` en ese orden.
2. Utilizando el indice y `print()`, muestra el string en el indice `1` en `numero_cardinales`.
3. Desempaca los valores en `numeros_cardinales` en tres (3) strings llamados `uno`, `dos`, `tres` en una lina de codigo, luego imprime cada valor en una linea separada.
4. Crea una tupla llamada `mi_nombre` que contiene las letras de su nombre utilizando `tuple()` y un literal de string.
5. Verifique si el caracter `x` esta en `mi_nombre` utilizando `in`.
6. Crea una Tupla que contiene todas excepto la primera letra en `mi_nombre` con el metodo de segmentacion.

1.2 Listas son Secuencias Mutables

La estructura de dato `list` es otro tipo de secuencia en Python. Como los strings y las tuplas, las listan tambien contienen indices que empiezan desde `0`.

Podemos indexar, segmentar, verificar la existencia de un elemento con `in`, y podemos iterar sobre listas con un `for` loop.

A diferencia de las tuplas, sin embargo, las listas son **mutables**, por lo que el valor ubicado enun indice puede ser modificado despues que la lista haya sido creado.

En esta seccion aprenderemos como crear listas y compararlas con las tuplas.

1.3 Creando Listas

Al igual que las tuplas, creamos listas con literales de lista y con el incorporado `list()`.

[38]:
```
# se ve igual que una tupla, excepto se utilizan []
colores = ['rojo', 'amarillo' , 'verde', 'azul']
colores
```

[38]:
```
['rojo', 'amarillo', 'verde', 'azul']
```

[39]:
```
# podemos crear una lista con list()
list('verde')
```

[39]:
```
['v', 'e', 'r', 'd', 'e']
```

[40]:
```
# podemos crear una lista de un tuple
list(tuple('azul'))
```

[40]:
```
['a', 'z', 'u', 'l']
```

[42]:
```
# tambien podemos crear un list desde un string, con str.split()
cosas = 'cereal, avena, queso'
lista_de_cosas = cosas.split(', ')  # ', ' es el separador
lista_de_cosas
```

[42]:
```
['cereal', 'avena', 'queso']
```

1.4 Operaciones basicas

[43]:
```
numeros = [1, 2, 3, 4]
```

[44]:
```
numeros[1]
```

[44]:
```
2
```

[45]:
```
numeros[1:3]
```

[45]:
```
[2, 3]
```

[46]:
```
'7' in numeros
```

[46]:
```
False
```

[48]:
```
3 in numeros
```

[48]:
```
True
```

[55]:
```
for numero in numeros:
    if numero % 2 == 0:
        print(f'{numero} es par')
    else:
        print(f'{numero} es impar')
```

```
1 es impar
2 es par
3 es impar
4 es par
```

1.5 Cambiando elementos en una lista

Las listas son mutables, por tanto podemos alterar el contenido de la misma, cambiando sus elementos.

Podemos pensar es una lista como una secuencia numerada de vacantes. Cada vacante sostiene un valor, y cada vacante debe estar llena en todo momento, pero podemos intercambiar un valor en una vacante por otro valor cuando queremos.

[53]:
```
colores = ['amarillo', 'rojo', 'blanco']
colores[0] = 'azul'
colores
```

[53]:
```
['azul', 'rojo', 'blanco']
```

[54]:
```
colores[1:] = ['naranja', 'negro']
colores
```

[54]:
```
['azul', 'naranja', 'negro']
```

1.6 Metodos para agregar y eliminar elementos

insert()

[56]:
```
# insert ingresa elementos individuales a la lista, tiene 2 parametros
colores = ['rojo', 'azul', 'blanco']
colores.insert(1, 'negro')  # el indice donde se ingresa, y el color
colores
```

[56]:
```
['rojo', 'negro', 'azul', 'blanco']
```

pop()

[57]:
```
# pop elimina el elemento ubicado en el indice dado o en el ultimo
color = colores.pop(1)
```

[58]:
```
color
```

[58]:
```
'negro'
```

[59]:
```
colores
```

[59]:
```
['rojo', 'azul', 'blanco']
```

[60]:
```
# sin parametro, elimina el ultimo elemento
colores.pop()
```

[60]:
```
'blanco'
```

[61]:
```
colores
```

[61]:
```
['rojo', 'azul']
```

append()

[62]:
```
# append() agrega un elemento al final de la lista
colores.append('blanco')
colores
```

[62]:
```
['rojo', 'azul', 'blanco']
```

extend()

[63]:
```
# extend() se usa para agregar varios elementos al final de la lista
colores.extend(['negro', 'gris'])
colores
```

[63]:
```
['rojo', 'azul', 'blanco', 'negro', 'gris']
```

1.7 Lista de numeros

[64]:
```
# obtengamos la suma de la lista
numeros = [1, 2, 3, 4, 5]
total = 0
for numero in numeros:
    total = total + numero
total
```

[64]:
```
15
```

[65]:
```
# podemos utilizar la funcion incorporada sum()
sum(numeros)  # sum() toma un iterable de numeros como parametro
```

[65]:
```
15
```

[66]:
```
sum([1, 2, 'letra'])  # no funciona
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-66-dd329d9da579> in <module>
   ----> 1 sum([1, 2, 'letra'])  # no funciona


       TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[67]:
```
# obtener el valor minimo en una lista
min(numeros)
```

[67]:
```
1
```

[69]:
```
# el maximo
max(numeros)
```

[69]:
```
5
```

1.7.1 Ejercicios

1. Crea una lista llamada `comida` con dos elementos `arroz` y `lentejas`.
2. Agrega el string `pollo` a comida utilizando `append()`
3. Agrega el string `pan` y `pizza` a `comida` utilizando `extend()`
4. Imprime los dos primeros elementos de `comida` utilizando `print()` y segmentacion.
5. Imprime el ultimo elemento en la lista `comida` utilizando `print()` y el indice.
6. Crea una lista `desayuno` del string `huevo, fruta, jugo` utilizando el metodo `.split()`.
7. Verifica que `desayuno` tenga tres elementos usando `len()`

1.8 Anidando, Copiando y Ordenando Tuplas y Listas

Ahora que haz aprendido lo que son las tuplas y las listas, como crearlas y como realizar algunas operaciones basicas, vamos a ver otros tres conceptos: 1. Anidar 2. Replicar 3. Ordenar

1.8.1 Anidando listas y tuplas

Listas y tuplas pueden contener valores de cualquier tipo. Es decir, las listas y las tuplas pueden contener listas y tuplas como valores. Una **lista anidada** o **tupla anidada** es una lista o una tupla que esta ubicada entro de otra lista o tupla.

[1]:
```
dos_x_dos = [[1, 2], [3, 4]]
```

[2]:
```
len(dos_x_dos)
```

[2]: 2

[3]:
```
dos_x_dos[0]
```

[3]: [1, 2]

[4]:
```
dos_x_dos[1]
```

[4]: [3, 4]

[5]:
```
# acceder al primer elemento de la primera lista
dos_x_dos[0][1]
```

[5]: 2

1.8.2 Copiando una Lista

A veces tenemos que copiar una lista a otra variable. Sin embargo, no podemos re-asignar una objeto de lista a otro objeto de lista, porque obtendremos este resultado:

[6]:
```
animales = ['perro', 'gato']
```

[7]:
```
animales_de_finca = animales
animales_de_finca.append('caballo')  # agreguemos caballo a la lista
```

[8]:
```
# ahora esta variable tiene 'caballo' tambien. Porque?
animales
```

[8]:
```
['perro', 'gato', 'caballo']
```

Una variable es una referencia a un objeto. No es el objeto, solo una referencia al mismo. Una referencia a la ubicacion exacta en la memoria de la computadora donde la variable reside. En vez de copiar todo el contenido de la lista y crear una nueva lista, `animales_de_finca = animales` solo asigna la referencia del objeto a la nueva variable. Es decir, las dos (2) variables hacen referencia al mismo objeto en memoria y cualquier cambio realizado en uno, afecta el otro.

A fin de obtener una copia independiente e integra de `animales`, podemos segmentar o cortar todo el contenido de la lista e asignarlo a la nueva variable.

[9]:
```
animales = ['perro', 'gato']
animales_de_finca = animales[:]  # de indice 0 al final de la lista
animales_de_finca.append('caballo')
animales_de_finca
```

[9]:
```
['perro', 'gato', 'caballo']
```

[10]:
```
animales
```

[10]:
```
['perro', 'gato']
```

[11]:
```
# tambien podemos copiar una lista de listas
matriz1 = [[1, 2], [3, 4]]
matriz2 = matriz1[:]
```

[12]:
```
matriz2[0] = [5, 6]
```

[13]:
```
matriz2
```

[13]:
```
[[5, 6], [3, 4]]
```

[14]:
```
matriz1
```

[14]:
```
[[1, 2], [3, 4]]
```

[16]:
```
# cambiemos el primer elemento de la segunda lista
matriz2[1][0] = 1
matriz2
```

[16]:
```
[[5, 6], [1, 4]]
```

[17]:
```
# tambien cambió, porque?
matriz1
```

[17]:
```
[[1, 2], [1, 4]]
```

Los elementos de ambas listas cambiaron porque una lista realmente no contiene los objetos mismos, pero referencias a esos objetos en memoria. Cuando copiamos una lista utilizando el metodo de segmentacion `[:]`, una nueva lista es retornada conteniendo las mismas referencias contenidas en la lista original. A esto se le conoce como una **copia simple**.

A fin de realizar una copia de la lista y todos los elementos que contiene, debemos utilizar algo conocido como una **copia profunda**. Este metodo de copiar esta fuera del alcance de esta seccion.

1.8.3 Ordenando listas

Las listas tienen un metodo llamado `.sort()`, que ordena todos los elementos de una lista en orden ascendente.

[18]:
```
colores = ['rojo', 'blanco', 'azul']
colores.sort()  # este metodo cambia el estado de la lista
colores  # ya esta ordenada
```

[18]:
```
['azul', 'blanco', 'rojo']
```

[19]:
```
numeros = [10, 1, 5, 3]
numeros.sort()
numeros
```

[19]:
```
[1, 3, 5, 10]
```

1.8.4 Ejercicios

1. Crea una tupla `data` con dos (2) valores. El primer valor debe sera una tupla `(1, 2)` y el segundo valor debe ser una tupla `(3, 4)`.
2. Escribe un ciclo `for` que cicla sobre `data` e imprime la suma de cada tupla anidada. El dato de salida debe ser algo asi:

```
Fila 1 suma: 3
Fila 2 suma: 7
```

3. Crea una lista `[4, 3, 2, 1]` y asignale una variable `numeros`
4. Crea una copia de la lista `numeros` utilizando el metodo de segmentacion `[:]`
5. Ordena lost `numeros` en la lista en orden numerico utilizando `.sort()`

1.9 Reto: Haz un Poema

Escribe un programa que contiene las siguientes lista de listas:

```
universidades = [
    ['Universidad de Panama', 2000, 35000],
    ['Universidad Tecnologica de Panama', 10000, 40000],
]
```

Defina una funcion `estadistica_de_matriculas` que toma una lista de listas como parametro donde cada lista contiene tres elementos: * nombre * numero total de estudiantes matriculados * tasa anual de de matricula

Defina una funcion `media()` y `mediana()`. Ambas funciones deben tomar la lista como argumento y retornar la media y la mediana de los valores en la lista.

Utilizando `universidades`, las funciones `estadistica_de_matriculas()`, `media()`, y `mediana()` debe calcular: * el numero total de estudiantes * la matricula total * la media * la mediana del numero de estudiantes * la media y mediana de los valores de matricula

Un ejemplo:

```
Total estudiantes: 12,000
Total matricula: 80,000

Media estudiante: 11, 040
Mediana estudiante: 10, 566

Media matricula: 38,800
Mediana matricula: 39,850
```

1.10 Guardando Relaciones en Diccionarios

Una las estructuras de dato mas utiles en Python es el **diccionario**.

En esta seccion aprenderemos lo que es un diccionario, que los distingue de las listas y tuplas, como definirlas y como utilizarlas.

1.10.1 Que es un diccionario?

Un diccionario en Python, como las listas y las tuplas, guardan colecciones de objetos. Sin embargo, en vez de guardar objetos en secuencia, los diccionarios guardan la informacion en una pareja de datos llamada **llave-valor**. Es decir, cada objeto en un diccionario tiene dos partes: una **llave** y un **valor**.

La **llave** en un par **llave-valor** es un nombre unico que identifica el **valor** de la pareja. Comparando esto a un diccionario comun, la llave es la palabra que se define y el valor es la definicion.

Por ejemplo, podemos utilizar un diccionaro para guardar nombres de provincias y su capital:

```
Chiriqui: David
Cocle: Anton
```

Es como un mapa en donde la **llave** contiene el **valor**. Si sabes el nombre de la llave, puedes obtener el valor de la misma.

[20]:
```
capitales = {
    'Panama': 'Panama',
    'Costa Rica': 'San Jose',
    'Colombia': 'Bogota'
}
```

[22]:
```
# tambien podemos crearlo asi
pareja_llave_valor = (
    ('Panama', 'Panama'),
    ('Costa Rica', 'San Jose'),
    ('Colombia', 'Bogota')
)

capitales2 = dict(pareja_llave_valor)
```

[23]:
```
capitales
```

[23]:
```
{'Panama': 'Panama', 'Costa Rica': 'San Jose', 'Colombia': 'Bogota'}
```

[24]:
```
capitales2
```

[24]:
```
{'Panama': 'Panama', 'Costa Rica': 'San Jose', 'Colombia': 'Bogota'}
```

1.10.2 Accediento a los valores de un diccionario

[25]:
```
capitales['Panama']  # se accede llamando la llave
```

[25]:
```
'Panama'
```

[26]:
```
capitales['Colombia']  # no es un indice, es una palabra llave
```

[26]:
```
'Bogota'
```

1.10.3 Agregando y eliminando valores de un diccionario

[27]:
```
capitales['Brasil'] = 'Brasilia'
```

[28]:
```
capitales
```

[28]:
```
{'Panama': 'Panama',
 'Costa Rica': 'San Jose',
 'Colombia': 'Bogota',
 'Brasil': 'Brasilia'}
```

[29]:
```
# eliminando un valor
del capitales['Costa Rica']
```

[30]:
```
capitales  # ya no exista Costa Rica
```

[30]:
```
{'Panama': 'Panama', 'Colombia': 'Bogota', 'Brasil': 'Brasilia'}
```

[31]:
```
# intentemos obtener Costa Rica
capitales['Costa Rica']  # error de llave KeyError
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-31-1f9705304578> in <module>
  1 # intentemos obtener Costa Rica
----> 2 capitales['Costa Rica']


KeyError: 'Costa Rica'
```

[41]:
```
# revisemos si una llave existe en el diccionario
'Panama' in capitales
```

[41]: True

[42]:
```
'Argentina' in capitales
```

[42]: False

[48]:
```
# revisemos si existe, luego hacemos algo
if 'Panama' in capitales:
    print(f"La capital de Panama es {capitales['Panama']}")
```

```
La capital de Panama es Panama
```

1.10.4 Iterando sobre diccionarios

[49]:
```
for llave in capitales:
    print(llave)
```

```
Panama
Colombia
Brasil
```

[50]:
```
for pais in capitales:
    print(f"La capital de {pais} es {capitales[pais]}")
```

```
La capital de Panama es Panama
La capital de Colombia es Bogota
La capital de Brasil es Brasilia
```

[52]:
```
# hay una forma mas facil de hacerlo
capitales.items()  # este metodo retorna algo parecido a una lista de objetos
```

[52]:
```
dict_items([('Panama', 'Panama'), ('Colombia', 'Bogota'), ('Brasil',
'Brasilia')])
```

[54]:
```
# no es una lista
type(capitales.items())  # pero retorna la llave y su valor a la misma vez
```

[54]:
```
dict_items
```

[56]:
```
for pais, capital in capitales.items():  # cada iteracion produce llave, valor␣
↪y se asigna
    print(f"La capital de {pais} es {capital}")
```

```
La capital de Panama es Panama
La capital de Colombia es Bogota
La capital de Brasil es Brasilia
```

[57]:
```
b# las llaves del diccionario pueden ser de cualquier tipo inmutable
capitales[10] = 'Managua'
capitales
```

[57]:
```
{'Panama': 'Panama', 'Colombia': 'Bogota', 'Brasil': 'Brasilia', 10: 'Managua'}
```

[59]:
```
# una lista es mutable, no puede ser una llave
capitales[[1, 2, 3]] = 'invalido'
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-59-330792078af7> in <module>
         1 # una lista es mutable, no puede ser una llave
    ----> 2 capitales[[1, 2, 3]] = 'invalido'


       TypeError: unhashable type: 'list'
```

1.11 Diccionarios Anidados

Podemos anidar diccionarios en diccionarios, de la misma forma que anidamos tuplas en tuplas o listas en listas.

[60]:
```
paises = {
    'Panama': {
        'capital': 'Panama',
```

```
        'slogan': 'pro mundo beneficio',
        'flor': 'espiritu santo'
    }
}
```

[64]:
```
paises['Panama']
```

[64]:
```
{'capital': 'Panama',
 'slogan': 'pro mundo beneficio',
 'flor': 'espiritu santo'}
```

[61]:
```
paises['Panama']['capital']
```

[61]:
```
'Panama'
```

[62]:
```
paises['Panama']['slogan']
```

[62]:
```
'pro mundo beneficio'
```

[63]:
```
paises['Panama']['flor']
```

[63]:
```
'espiritu santo'
```

1.12 Ejercicios

1. Crea un diccionario vacio llamado `provincias`
2. Ingresa el nombre de la provincia como llave y su ciudad principal como valor
3. Crea un `for` loop para imprimir la provincia y la capital: `La capital de Cocle es Anton`
4. Elimina una llave-valor del diccionario

1.13 Reto: Ciclando sobre Provincias

Con un diccionario de las provincias de Panama con su principal ciudad, vamos a crear un programa que: 1. Escoge una provincia al azar (`random`) 2. Solicita al usuario que ingrese su capital 3. Si la capital es incorrecta debe seguir solicitando la capital de la provincia hasta que se ingrese la correcta. 4. Si el usuario escribe `salir`, el programa termina 5. Si el usuario escribe la capital correcta, imprima `Correcto` 6. Si el usuario sale del programa sin adivinar correctamente, imprima `Hasta luego`

1.14 Como escoger entre listas, tuplas o diccionarios

Usa una lista cuando: 1. Los datos tienen un orden 2. Será necesario actualizar o alterar los datos durante el programa 3. El propósito principal de la estructura de datos es la iteración

Usa una tupla cuando: 1. Los datos tiene un orden 2. No será necesario actualizar o alterar los datos durante el programa 3. El proposito principal de la estructura de datos es la iteración

Usa un diccionario cuando: 1. La data no está ordenada o el orden no importa 2. No será necesario actualizar o alterar los datos durante el programa 3. El propósito principal de la estructura de datos es buscar valores

1.15 Resumen

Aprendimos sobre tres estructuras de datos: 1. Listas 2. Tuplas 3. Diccionarios

La listas, como `[1, 2, 3, 4]` son secuencias mutables de objetos. Podemos interactuar con listas utilizando varios metodos de listas, como `.append()`, `.remove()`, and `.extend()`. Las listas se pueden ordenar utilizando `.sort()`. Podemos acceder elementos individuales de una lista por indice `lista[indice]`. Podemos tambien obtener un segmento de una lista utilizando `lista[indice:indice]`.

Las tuplas, como las listas, son secuencias de objetos. La diferencia primordial entre listas y tuplas es que las tuplas son inmutables. Una vez hemos creado una tupla, no podemos cambiarla. Al igual que las listas, podemos acceder a elementos por indice y podemos obtener un segmento de la tupla utilizando el concepto de segmentacion.

Los diccionarios guardan datos en pareja llave-valor. No son secuencias, por tanto no podemos acceder sus elementos por indice, toda vez que se acceden por su llave. Los diccionarios son utilies para guardar relaciones o cuando necesitamos acceder a algun dato rapidamente. Como las listas, los diccionaros son mutables.

Las listas, tuplas y diccionarios son todos iterables, por tanto podemos ciclar sobre las mismas.

1.16 Prueba de conocimiento

[2]:
```
# tomando esto en consideracion:
letras = ['a', 'b', 'c', 'd']
```

[ ]:
```
letras[2]  # cual es el resultado?
```

[ ]:
```
'e' in letras  # cual es el resultado?
```

[3]:
```
# tomando esto en consideracion:
x = [10, [3.141, 20, [30, 'hola', 2.718]], '507']
```

[ ]:
```
# Como retornamos la "o" de "hola". Escriba el codigo abajo
x =  # respuesta
```

Cual es la principal diferencia entre listas y tuplas? 1. las listas son mutables y las tuplas inmutables 2. las listas son inmutables y las tuplas mutables 3. las listas son mas rapidas y las tuplas mas lentas 4. las listas pueden contener cualquier tipo de datos y las tuplas solo `int` y `str`

[ ]:
```
# Como asignamos una tupla de un valor a la variable "a"? Escriba el codigo␣
↪abajo
a =   # respuesta
```

- Los diccionarios pueden tener objetos de cualquier tipo excepto otros diccionarios?

- Los diccionarios pueden tener cualquier cantidad de objetos anidados?
- Las llaves de un diccionario deben ser del mismot tipo?
- Los elementos de un diccionario se acceden mediante indice o llave?
- Los diccionarios son mutables o inmutables?

[4]:
```
empleado = {
  "id": 1001,
  "name": "Jaime Diaz",
  "email": "jaimito@diaz.com",
  "cargo": "Gerente"
}
```

[ ]:
```
# como obtenemos el email de Jaime? Escriba el codigo abajo.
```