Programación en Python
TEMA 7
Tuplas y conjuntos

Universidad Politécnica de Valencia

2021-2022

Índice

---

1. Las tuplas son inmutables

Una tupla[1] es una secuencia de valores similar a una lista. Los valores guardados en una tupla pueden ser de cualquier tipo, y son indexados por números enteros. La principal diferencia es que las tuplas son *inmutables*. Las tuplas además son *comparables* de modo que las listas de tuplas se pueden ordenar y también usar tuplas como valores para las claves en diccionarios de Python.

Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, es común encerrar las tuplas entre paréntesis para ayudarnos a identificarlas rápidamente cuando revisemos código de Python:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un solo elemento, es necesario incluir una coma al final:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Sin la coma, Python considera `('a')` como una expresión con una cadena entre paréntesis que es evaluada como de tipo cadena (string):

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Otra forma de construir una tupla es utilizando la función interna `tuple`. Sin argumentos, ésta crea una tupla vacía:

```
>>> t = tuple()
>>> print(t)
()
```

Si el argumento es una secuencia (cadena, lista, o tupla), el resultado de la llamada a `tuple` es una tupla con los elementos de la secuencia:

```
>>> t = tuple('altramuces')
>>> print(t)
('a', 'l', 't', 'r', 'a', 'm', 'u', 'c', 'e', 's')
```

Dado que `tuple` es el nombre de un constructor, debería evitarse su uso como nombre de variable.

La mayoría de los operadores de listas también funcionan en tuplas. El operador corchete para la indexación de sus elementos:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Y el operador de concatenación (+) funciona con tuplas:

[1] Dato curioso: La palabra "tuple" proviene de los nombres dados a secuencias de números de distintas longitudes: simple, doble, triple, cuádruple, quíntuple, séxtuple, séptuple, etc.

```
>>> (1,2) + (3,4,5)
  (1, 2, 3, 4, 5)
>>> (2,) + (9,)
  (2, 9)
```

Y el operador de repetición (∗) funciona con tuplas:

```
>>> 3 * (1,)
  (1, 1, 1)
>>> 5 * (1,2,3)
  (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Y el operador de calcular el length (**len**) funciona con tuplas:

```
>>> len ((4,5,5,6,7,8))
  6
>>> len()
  TypeError: len() takes exactly one argument (0 given)
>>> len(())
  0
```

Y el operador de rebanado (slice) selecciona un rango de elementos.

```
>>> print(t[1:3])
('b', 'c')
```

Pero si se intenta modificar uno de los elementos de la tupla, se produce un error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

No se puede modificar los elementos de una tupla, pero sí se puede reemplazar una tupla por otra:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

2. Comparación de tuplas

Los operadores de comparación funcionan con tuplas y otras secuencias. Python comienza comparando el primer elemento de cada secuencia. Si ambos elementos son iguales, pasa al siguiente elemento y así sucesivamente, hasta que encuentra elementos diferentes. Los elementos subsecuentes no son considerados (aunque sean muy grandes).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

La función `sort` funciona de la misma manera. Ordena inicialmente por el primer elemento, pero en el caso de que ambos elementos sean iguales, ordena por el segundo elemento, y así sucesivamente.

3. Patrón de diseño: DSU

La comparación de tuplas da lugar a un patrón de diseño llamado *DSU* que podemos usar para ordenar secuencias de elementos según una determinada propiedad de estos elementos que usamos como índice. DSU consiste en:

Decorate (Decora) una secuencia, construyendo una lista de tuplas con uno o más índices ordenados precediendo los elementos de la secuencia,

Sort (Ordena) la lista de tuplas utilizando la función interna `sort`, y

Undecorate (Quita la decoración) extrayendo los elementos ordenados de la secuencia.

Por ejemplo, suponiendo una lista de palabras que se quieren ordenar de la más larga a la más corta:

```
txt = 'Pero qué luz se deja ver allí'
palabras = txt.split()

#Decorar las palabras de la lista con su len
t = list()
for palabra in palabras:
    t.append((len(palabra), palabra))

#Ordenar
t.sort(reverse=True)

#Undecorar, quitando el len
res = list()
for longitud, palabra in t:
    res.append(palabra)

print(res)
```

El primer bucle genera una lista de tuplas, donde cada tupla es una palabra precedida por su longitud.

`sort` compara el primer elemento (longitud) primero, y solamente considera el segundo elemento para desempatar. El argumento clave `reverse=True` indica a `sort` que debe ir en orden decreciente.

El segundo bucle recorre la lista de tuplas y construye una lista de palabras en orden descendente según la longitud. Las palabras de cuatro letras están ordenadas en orden alfabético *inverso*, así que "deja" aparece antes que "allí" en la siguiente lista.

La salida del programa es la siguiente:

```
['deja', 'allí', 'Pero', 'ver', 'qué', 'luz', 'se']
```

Por supuesto, la línea pierde mucho de su impacto poético cuando se convierte en una lista de Python y se almacena en orden descendente según la longitud de las palabras.

4. Asignación de tuplas

A menudo es útil intercambiar los valores de dos variables. Con asignaciones convencionales, tienes que utilizar una variable temporal. Por ejemplo, para intercambiar `a` y `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Esta solución es incómoda; la **asignación de tupla** es más elegante:

```
>>> a, b = b, a
```

El lado izquierdo es una tupla de variables; el lado derecho es una tupla de expresiones. Cada valor es asignado a su respectiva variable. Todas las expresiones en el lado derecho son evaluadas antes que cualquiera de las asignaciones.

Ambos lados de la sentencia son tuplas, pero el lado izquierdo es una tupla de variables; el lado derecho es una tupla de expresiones. Cada valor en el lado derecho es asignado a su respectiva variable en el lado izquierdo. Todas las expresiones en el lado derecho son evaluadas antes de realizar cualquier asignación.

El número de variables en el lado izquierdo y el número de valores en el lado derecho deben ser iguales:

```
>>> a, b = 1, 2, 3
  ValueError: too many values to unpack
```

Generalizando más, el lado derecho puede ser cualquier tipo de secuencia (cadena, lista, o tupla). De esta manero tenemos un mecanismo de empaquetado/desempaquetado utilizando tuplas.

Por ejemplo, para dividir una dirección de e-mail en nombre de usuario y dominio, se podría escribir:

```
>>> dir = 'monty@python.org'
>>> nombreus, dominio = dir.split('@')
```

El valor de retorno de `split` es una lista con dos elementos; el primer elemento es asignado a `nombreus`, el segundo a `dominio`.

```
>>> print(nombreus)
  monty
>>> print(dominio)
  python.org
```

En el siguiente ejemplo tenemos una lista de dos elementos (la cual es una secuencia) y asignamos el primer y segundo elementos de la secuencia a las variables `x` y `y` en una única sentencia.

```
>>> m = [ 'pásalo', 'bien' ]
>>> x, y = m
>>> x
  'pásalo'
>>> y
  'bien'
>>>
```

No es magia, Python traduce *aproximadamente* la sintaxis de asignación de la tupla de este modo::[2]

```
>>> m = [ 'pásalo', 'bien' ]
>>> x = m[0]
>>> y = m[1]
>>> x
  'pásalo'
>>> y
  'bien'
>>>
```

Estilísticamente, cuando se utiliza una tupla en el lado izquierdo de la asignación, se omiten los paréntesis, pero lo que se muestra a continuación es una sintaxis igualmente válida:

[2]Python no traduce la sintaxis literalmente. Por ejemplo, si se trata de hacer esto con un diccionario, no va a funcionar como se podría esperar.

```
>>> m = [ 'pásalo', 'bien' ]
>>> (x, y) = m
>>> x
'pásalo'
>>> y
'bien'
>>>
```

5. Tuplas como valores de retorno

Estrictamente hablando, una función puede devolver solo un valor de retorno, pero si el valor es una tupla, el efecto es el mismo que devolver múltiples valores. Por ejemplo, si quieres dividir dos enteros y calcular el cociente y el resto, es ineficiente calcular `x//y` y luego `x%y`. Es mejor calcular ambos al mismo tiempo.

La función incorporada `divmod` toma dos argumentos y devuelve una tupla de dos valores: el cociente y el resto. Puedes almacenar el resultado como una tupla:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

O bien, utilizar asignación de tupla para almacenar los elementos por separado:

```
>>> cociente, resto = divmod(7, 3)
>>> cociente
2
>>> resto
1
```

Aquí hay un ejemplo de una función que devuelve una tupla:

```
def min_max(t):
    return min(t), max(t)
```

`max` y `min` son funciones incorporadas que encuentran los elementos más grande y más pequeño en una secuencia. `min_max` calcula ambos y devuelve una tupla de dos valores.

6. Tuplas de argumentos de longitud variable

Las funciones pueden tomar una cantidad variable de argumentos. Un nombre de parámetro que comienza con `*` hace una **reunión** de argumentos en una tupla. Por ejemplo, `printall` toma cualquier cantidad de argumentos y los imprime:

```
def printall(*args):
    print(args)
```

El parámetro de reunión puede tener cualquier nombre que quieras, pero `args` es convencional. Aquí se muestra cómo opera la función:

```
>>> printall(1, 2.0, '3')
  (1, 2.0, '3')
>>> printall(2.3, '3', 'abc', 34)
  (2.3, '3', 'abc', 34)
```

El complemento de la reunión es la **dispersión**. Si tienes una secuencia de valores y quieres pasarlo a una función como múltiples argumentos, puedes utilizar el operador `*`. Por ejemplo, `divmod` toma exactamente dos argumentos; no funciona con una tupla:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Sin embargo, si dispersas la tupla, funciona:

```
>>> divmod(*t)
(2, 1)
```

Muchas de las funciones incorporadas utilizan tuplas de argumentos de longitud variable. Por ejemplo, `max` y `min` pueden tomar cualquier cantidad de argumentos:

```
>>> max(1, 2, 3)
3
```

Pero `sum` no puede.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

Una función llamada que tome cualquier cantidad de argumentos y devuelva su suma lo podemos escribir nosotros como por ejemplo abajo:

```
def suma_todo(*items):
    if (not items):
        return items

    resultado = items[0]
    for item in items[1:]:
        resultado = resultado + item
    return resultado
```

Lo primero que hacemos es verificar si recibimos algún argumento. Si no, volvemos `items`, una tupla vacía. Esto es necesario porque el resto de la función requiere que sabemos que tenemos un elemento en el índice 0. Observa que no buscamos una tupla vacía comparándola con () o marcando que su longitud es 0. Más bien, decimos `(not items)`, lo que solicita el valor booleano de nuestra tupla. Debido a que una secuencia de Python vacía es falsa en un contexto booleano, obtenemos False si `items` está vacío y True en caso contrario.

```
>>> suma_todo()
()
>>> suma_todo(2)
2
>>> suma_todo(10, 20, 30, 40)
100
>>> suma_todo('a', 'b', 'c', 'd')
```

```
'abcd'
>>> suma_todo([10, 20, 30], [40, 50, 60], [70, 80])
[10, 20, 30, 40, 50, 60, 70, 80]
```

7. Diccionarios y tuplas

Los diccionarios tienen un método llamado `items` que retorna una lista de tuplas, donde cada tupla es un par clave-valor:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

Como sería de esperar en un diccionario, los elementos no tienen ningún orden en particular.

Aun así, puesto que la lista de tuplas es una lista, y las tuplas son comparables, ahora se puede ordenar la lista de tuplas. Convertir un diccionario en una lista de tuplas es una forma de obtener el contenido de un diccionario ordenado según sus claves:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

La nueva lista está ordenada en orden alfabético ascendente de acuerdo al valor de sus claves.

La combinación de `items`, asignación de tuplas, y `for`, produce un buen patrón de diseño de código para recorrer las claves y valores de un diccionario en un único bucle:

```
for clave, valor in list(d.items()):
    print(valor, clave)
```

Este bucle tiene dos *variables de iteración*, debido a que `items` retorna una lista de tuplas y `clave, valor` es una asignación en tupla que itera sucesivamente a través de cada uno de los pares clave-valor del diccionario.

Para cada iteración a través del bucle, tanto `clave` y `valor` van pasando al siguiente par clave-valor del diccionario (todavía en orden de dispersión).

La salida de este bucle es:

```
10 a
1 b
22 c
```

De nuevo, las claves están en orden de dispersión (es decir, ningún orden en particular).

Si se combinan esas dos técnicas, se puede imprimir el contenido de un diccionario ordenado por el *valor* almacenado en cada par clave-valor.

Para hacer esto, primero se crea una lista de tuplas donde cada tupla es `(valor, clave)`. El método `items` dará una lista de tuplas `(clave, valor)`, pero esta vez se pretende ordenar por valor, no por clave.

Una vez que se ha construido la lista con las tuplas clave-valor, es sencillo ordenar la lista en orden inverso e imprimir la nueva lista ordenada.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for clave, valor in d.items() :
...     l.append( (valor, clave) )
...
>>> l
[(10, 'a'), (1, 'b'), (22, 'c')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

Al construir cuidadosamente la lista de tuplas para tener el valor como el primer elemento de cada tupla, es posible ordenar la lista de tuplas y obtener el contenido de un diccionario ordenado por valor.

8. Uso de tuplas como claves en diccionarios

Dado que las tuplas son **dispersables** *(hashable)* y las listas no, si se quiere crear una clave **compuesta** para usar en un diccionario, se debe utilizar una tupla como clave.

Usaríamos por ejemplo una clave compuesta si quisiéramos crear un directorio telefónico que mapea pares appellido, nombre con números telefónicos. Asumiendo que hemos definido las variables `apellido`, `nombre`, y `número`, podríamos escribir una sentencia de asignación de diccionario como sigue:

```
directorio[apellido,nombre] = numero
```

La expresión entre corchetes es una tupla. Podríamos utilizar asignación de tuplas en un bucle `for` para recorrer este diccionario.

```
for apellido, nombre in directorio:
    print(nombre, apellido, directorio[apellido,nombre])
```

Este bucle recorre las claves en `directorio`, las cuales son tuplas. Asigna los elementos de cada tupla a `apellido` y `nombre`, después imprime el nombre y el número telefónico correspondiente.

9. Conjuntos

Un conjunto es una colección no ordenada y mutable de objetos únicos. Los conjuntos son ampliamente utilizados en lógica y matemática. Python tambien provee este tipo de datos al igual que otras más convencionales como las listas, tuplas y diccionarios.

Para crear un conjunto especificamos sus elementos entre llaves:

```
>>> s = {1, 2, 3, 4}
```

Al igual que otras colecciones, sus miembros pueden ser de diversos tipos:

```
>>> s = {True, 3.14, None, False, "Hola mundo", (1, 2)}
```

No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

```
>>> s = {[1, 2]}
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'list'
```

No hay que confundir el diccionario vació (`{}`) con un conjunto vació (`set()`):

```
>>> s = {}
>>> type(s)
  <class 'dict'>
>>> s = set()
>>> type(s)
  <class 'set'>
```

Podemos obtener un conjunto a partir de cualquier objeto iterable:

```
s1 = set([1, 2, 3, 4])
s2 = set(range(10))
```

Un conjunto puede ser convertido a una lista y viceversa. En este último caso, los elementos duplicados son unificados.

```
>>> list({1, 2, 3, 4})
[1, 2, 3, 4]
>>> set([1, 2, 2, 3, 4])
{1, 2, 3, 4}
```

Los conjuntos son objetos mutables. Vía los métodos `add()` y `discard()` podemos añadir y remover un elemento indicándolo como argumento.

```
>>> s = {1, 2, 3, 4}
>>> s.add(5)
>>> s.discard(2)
>>> s
{1, 3, 4, 5}
```

Date cuenta que si el elemento pasado como argumento a `discard()` no está dentro del conjunto es simplemente ignorado. En cambio, el método `remove()` opera de forma similar pero en dicho caso lanza la excepción `KeyError`.

Para determinar si un elemento pertenece a un conjunto, utilizamos la palabra reservada **`in`**.

```
>>> 2 in {1, 2, 3}
True
>>> 4 in {1, 2, 3}
False
```

La función `clear()` elimina todos los elementos.

```
>>> s = {1, 2, 3, 4}
>>> s.clear()
>>> s
set()
```

El método `pop()` retorna un elemento en forma aleatoria (no podría ser de otra manera ya que los elementos no están ordenados). Así, el siguiente bucle imprime y remueve uno por uno los miembros de un conjunto.

```
while s:
    print(s.pop())
```

`remove()` y `pop()` lanzan la excepción `KeyError` cuando un elemento no se encuentra en el conjunto o bien éste está vacío, respectivamente.

Para obtener el número de elementos aplicamos la ya conocida función len():

```
>>> len({1, 2, 3, 4})
4
```

9.1. Operaciones principales

Algunas de las propiedades más interesantes de los conjuntos radican en sus operaciones principales: unión, intersección y diferencia.

La unión se realiza con el carácter `|` y retorna un conjunto que contiene los elementos que se encuentran en al menos uno de los dos conjuntos involucrados en la operación.

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a | b
{1, 2, 3, 4, 5, 6}
```

La intersección opera de forma análoga, pero con el operador `&`, y retorna un nuevo conjunto con los elementos que se encuentran en ambos.

```
>>> a & b
{3, 4}
```

La diferencia, por último, retorna un nuevo conjunto que contiene los elementos de a que no están en b.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> a - b
{1, 4}
```

Dos conjuntos son iguales si y solo si contienen los mismos elementos (a esto se lo conoce como principio de extensionalidad):

```
>>> {1, 2, 3} == {3, 2, 1}
True
>>> {1, 2, 3} == {4, 5, 6}
False
```

Se dice que B es un subconjunto de A cuando todos los elementos de aquél pertenecen también a éste. Python puede determinar esta relación vía el método `issubset()`.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> b.issubset(a)
True
```

Inversamente, se dice que A es un superconjunto de B.

```
>>> a.issuperset(b)
True
```

La definición de estas dos relaciones nos lleva a concluir que todo conjunto es al mismo tiempo un subconjunto y un superconjunto de sí mismo.

```
>>> a = {1, 2, 3, 4}
>>> a.issubset(a)
True
>>> a.issuperset(a)
True
```

La diferencia simétrica retorna un nuevo conjunto el cual contiene los elementos que pertenecen a alguno de los dos conjuntos que participan en la operación pero no a ambos. Podría entenderse como una unión exclusiva.

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a.symmetric_difference(b)
{1, 2, 5, 6}
```

Dada esta definición, se infiere que es indistinto el orden de los objetos:

```
>>> b.symmetric_difference(a)
{1, 2, 5, 6}
```

Por último, se dice que un conjunto es disjoint respecto de otro si no comparten elementos entre sí.

```
>>> a = {1, 2, 3}
>>> b = {3, 4, 5}
>>> c = {5, 6, 7}
>>> a.isdisjoint(b)
False  # No son disjoint ya que comparten el elemento 3.
>>> a.isdisjoint(c)
True   # Son disjoint.
```

En otras palabras, dos conjuntos son disjoint si su intersección es el conjunto vacío, por lo que puede ilustrarse de la siguiente forma:

```
>>> def isdisjoint(a, b):
...     return a & b == set()
...
>>> isdisjoint(a, b)
False
>>> isdisjoint(a, c)
True
```

10. Secuencias: cadenas, listas, tuplas, conjuntos - ¡Dios mío!

Muchos ejemplos de este capítulo funcionan también con listas de listas, tuplas de tuplas, y tuplas de listas. Para evitar enumerar todas las combinaciones posibles, a veces es más sencillo hablar de secuencias de secuencias.

En muchos contextos, los diferentes tipos de secuencias (cadenas, listas, tuplas, conjuntos) pueden intercambiarse. Así que, ¿cómo y por qué elegir uno u otro?

Para comenzar con lo más obvio, las cadenas están más limitadas que otras secuencias, debido a que los elementos tienen que ser caracteres. Además, son inmutables. Si necesitas la capacidad de cambiar los caracteres en una cadena (en vez de crear una nueva), quizá prefieras utilizar una lista de caracteres.

También obvio, un conjunto es una colección no ordenada y sus elementos no pueden ser mutables (como listas). Así si necesitas una secuencia que esta ordenado, o con elementos de mutables, los conjuntos no son tu solución.

Las listas son más comunes que las tuplas, principalmente porque son mutables. Pero hay algunos casos donde es preferible utilizar tuplas:

1. En algunos contextos, como una sentencia return, resulta sintácticamente más simple crear una tupla que una lista. En otros contextos, es posible que prefieras una lista.
2. Si quieres utilizar una secuencia como una clave en un diccionario, debes usar un tipo inmutable como una tupla o una cadena.
3. Si estás pasando una secuencia como argumento de una función, el uso de tuplas reduce la posibilidad de comportamientos inesperados debido a la creación de alias.

Dado que las tuplas son inmutables, no proporcionan métodos como `sort` y `reverse`, que modifican listas ya existentes. Sin embargo, Python proporciona las funciones internas `sorted` y `reversed`, que toman una secuencia como parámetro y devuelven una secuencia nueva con los mismos elementos en un orden diferente.

11. Depuración

Las listas, diccionarios, tuplas y conjuntos son conocidas de forma genérica como *estructuras de datos*; en este capítulo estamos comenzando a ver estructuras de datos compuestas, como listas de tuplas, y diccionarios que contienen tuplas como claves y listas como valores. Las estructuras de datos compuestas son útiles, pero también son propensas a lo que yo llamo *errores de modelado*; es decir, errores causados cuando una estructura de datos tiene el tipo, tamaño o composición incorrecto, o quizás al escribir una parte del código se nos olvidó cómo era el modelado de los datos y se introdujo un error. Por ejemplo, si estás esperando una lista con un entero y recibes un entero solamente (no en una lista), no funcionará.

12. Glosario

comparable Un tipo en el cual un valor puede ser revisado para ver si es mayor que, menor que, o igual a otro valor del mismo tipo. Los tipos que son comparables pueden ser puestos en una lista y ordenados.

estructura de datos Una collección de valores relacionados, normalmente organizados en listas, diccionarios, tuplas, etc.

DSU Abreviatura de "decorate-sort-undecorate (decorar-ordenar-quitar la decoración)", un patrón de diseño que implica construir una lista de tuplas, ordenarlas, y extraer parte del resultado.

reunir La operación de tratar una secuencia como una lista de argumentos.

hashable (dispersable) Un tipo que tiene una función de dispersión. Los tipos inmutables, como enteros, flotantes y cadenas son dispersables (hashables); los tipos mutables como listas y diccionarios no lo son.

dispersar La operación de tratar una secuencia como una lista de argumentos.

modelado (de una estructura de datos) Un resumen del tipo, tamaño, y composición de una estructura de datos.

singleton Una lista (u otra secuencia) con un único elemento.

tupla Una secuencia inmutable de elementos.

asignación por tuplas Una asignación con una secuencia en el lado derecho y una tupla de variables en el izquierdo. El lado derecho es evaluado y luego sus elementos son asignados a las variables en el lado izquierdo.

13. Ejercicios

Exercise 1:

Escriba una función `string_list_to_tuple` Python que convierta una cadena en una tupla.

example test runs

```
>>> string_list_to_tuple("")
  ()
>>> string_list_to_tuple("hola")
  ('h', 'o', 'l', 'a')
>>> string_list_to_tuple("hola caracola")
  ('h', 'o', 'l', 'a', ' ', 'c', 'a', 'r', 'a', 'c', 'o', 'l', 'a')
```

Exercise 2:

Escriba una función `suma_tuplas` en Python para calcular la suma de los elementos de 3 tuplas dadas. Puedes testear tu función como sigue:

```
import pytest

x1 = (1,2,3,4)
y1 = (3,5)
z1 = (2,2,3,1,9,10,4)
salida_esperada1 = (6, 9, 6, 5, 9, 10, 4)

x2 = (-2,5,9)
y2 = (1,)
z2 = (2,3,3,1,9,3)
salida_esperada2 = (1, 8, 12, 1, 9, 3)

@pytest.mark.parametrize("testcase, x, y, z, salida_esperada",[
(1, x1, y1, z1, salida_esperada1),
(2, x2, y2, z2, salida_esperada2),
])

def test_suma_tuplas(testcase, x, y, z, salida_esperada):
    assert suma_tuplas(x,y,z) == salida_esperada,\
           "caso {0}".format(testcase)
```

Exercise 3:

Escriba una función `todos_equal_tupla` en Python que devuelve `True` cuando todos los elementos de la tupla son iguales y `False` s no lo son.

example test runs

```
>>> todos_equal_tupla((3,4,5,6))
False
>>> todos_equal_tupla((3,3,3,3,3,3))
True
```

Exercise 4:

Escribe una función `suma_resta` que tome una lista o tupla de números. Devuelve el resultado de alternativamente sumar y restar números entre sí.

Por ejemplo: `suma_resta([10, 20, 30, 40, 50, 60])` devuelve el resultado de 10+20-30+40-50+60, es 50.

example test runs

```
>>> suma_resta((3,4,5,6))
  8
>>> suma_resta((13,13,13,13,13,13))
  6
```

Exercise 5:

Escribe un programa que lee un archivo y calcula las frecuencias en que aparecen las *letras*. El programa debe convertir todas las entradas a minúsculas y contar solamente las letras a-z. El programa no debe contar espacios, dígitos, signos de puntuación, o cualquier cosa que no sean las letras a-z.

Encuentra ejemplos de texto en idiomas diferentes (español, ingles, alemán, holandés, francés) y observa cómo la frecuencia de letras es diferente en cada idioma.

Compara tus resultados con las tablas en https://en.wikipedia.org/wiki/Letter_frequency.

Exercise 6:

Escribe una función `mapear_anagrams` que lea una lista de palabras desde un archivo `words.txt` que esta en Poliformat y los guarda en un diccionario de conjuntos de palabras que son anagramas. Hay que construir un diccionario que mapee de una colección de letras a un conjunto de palabras que se puedan escribir con esas letras. La pregunta es, ¿cómo puedes representar la colección de letras de manera que se pueda utilizar como clave?

Despues, escribe otra funcion `imprimir_anagrams` que recibe el diccionario que resulta de `mapear_anagrams` y imprime los conjuntos de palabras que son anagramas. Aquí hay un ejemplo de cómo se vería la salida cuando hacemos tests en el shell:

```
>>> dicc_map = mapear_anagrams("words.txt")
>>> imprimir_anagrams(dicc_map)
{'aa'}
{'aah', 'aha'}
{'aahed', 'ahead'}
{'aal', 'ala'}
{'alas', 'aals'}
....
```

Exercise 7:

Modifica el programa anterior para que imprima la lista de anagramas más grande primero, seguido de la segunda más grande, y así sucesivamente. Recuerda el patron de diseño DSU.

```
>>> dicc_map = mapear_anagrams("words.txt")
>>> imprimir_anagrams_descending(dicc_map)
{'alerts', 'laster', 'salter', 'stelar', 'talers', 'estral', 'alters', 'ratels', 'staler', 'slater', 'artels'}
{'pears', 'presa', 'prase', 'apers', 'spare', 'reaps', 'rapes', 'spear', 'pares', 'asper', 'parse'}
{'tesla', 'slate', 'setal', 'steal', 'teals', 'tales', 'taels', 'stela', 'stale', 'least'}
{'escarp', 'recaps', 'secpar', 'capers', 'crapes', 'spacer', 'pacers', 'parsec', 'scrape'}
{'insert', 'inters', 'trines', 'estrin', 'niters', 'inerts', 'nitres', 'triens', 'sinter'}
{'scare', 'acres', 'carse', 'serac', 'acers', 'escar', 'races', 'cares'}
{'retsina', 'anestri', 'stainer', 'nastier', 'stearin', 'retains', 'retinas', 'ratines'}
```