

# Tuplas

# Qué son las tuplas?

`('carro', 'casa', 'edificio', 'moto')`

# Qué son las tuplas?

`('carro', 'casa', 'edificio', 'moto')`

**Una colección secuencial de datos!**

**Perdón?, y las listas?**

# Tuplas

- Las tuplas son muy similares a las listas pero tiene algunas diferencias:
  - Su definición se hace por medio de paréntesis.

```
t = (1, 2, True, "python")

# create a tuple with some items and display with a for loop
inventory = ("sword",
            "armor",
            "shield",
            "healing potion")
print("Your items:")
for item in inventory:
    print(item)

input("\nPress the enter key to continue.")
```

- Las tuplas no poseen funciones de modificación, es decir, son inmutables.

# Creando tuplas

- Mediante parentesis y separados por comas:

```
t = (1, 'a', [6, 3.14])  
(1, 'a', [6, 3.14])
```

- Utilizando el *keyword* tuple y pasando una lista de datos.

```
l = [1, 'a', [6, 3.14]]  
t = tuple(l)  
(1, 'a', [6, 3.14])
```

- Separado unicamente por comas.

```
t = 'A', 'tuple', 'needs', 'no', 'parens'  
( 'A', 'tuple', 'needs', 'no', 'parens' )
```

# Asignación Múltiple

- Desempacando una tupla:

```
t = 'A', 'tuple', 'needs', 'no', 'parens'  
article, noun, verb, adjective, direct_object = t  
print(verb)
```

Que imprime?

- Empacando una tupla:

# Asignación Múltiple

- Desempacando una tupla:

```
t = 'A', 'tuple', 'needs', 'no', 'parens'  
article, noun, verb, adjective, direct_object = t  
print(verb)
```

needs

- Empacando una tupla:

# Asignación Múltiple

- Desempacando una tupla:

```
t = 'A', 'tuple', 'needs', 'no', 'parens'  
article, noun, verb, adjective, direct_object = t  
print(verb)
```

needs

- Empacando una tupla:

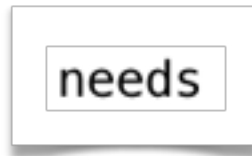
Como seria el proceso de “empacar”?



# Asignación Múltiple

- Desempacando una tupla:

```
t = 'A', 'tuple', 'needs', 'no', 'parens'  
article, noun, verb, adjective, direct_object = t  
print(verb)
```



needs

- Empacando una tupla:

```
t2 = noun, verb, adjective  
print(t2)  
( 'tuple', 'needs', 'no' )
```

# Tuplas

- El operador `in` se utiliza para verificar si un determinado elemento es parte de la colección.

```
# test for membership with in
if "healing potion" in inventory:
    print("You will live to fight another day.")
```

- Para acceder a un determinado elemento en una tupla se especifica el número correspondiente a su posición entre corchetes.

0	1	2	3
"sword"	"armor"	"shield"	"healing potion"
-4	-3	-2	-1

```
# display one item through an index
index = int(input("\nEnter the index number for an item in inventory: "))
print("At index", index, "is", inventory[index])
```

# Tuplas

- Para obtener un fragmento de una tupla se utiliza la especificación de un rango entre corchetes:

```
>>> x = ('abc', 73, 5.28, 'rs', 5)
```

```
>>> x[1:4]  
(73, 5.28, 'rs')
```

Componentes desde 1 hasta el 3 (celdas 1 a 3)  
(el rango no incluye el extremo final)

```
>>> x[2:]  
(5.28, 'rs', 5)
```

Componentes 2 hasta el final (celda 2 hasta el final)  
Los resultados también son tuplas

# Tuplas

- Las tuplas no pueden ser modificadas.

```
>>> inventory = ("sword", "armor", "shield", "healing potion")
>>> print(inventory)
('sword', 'armor', 'shield', 'healing potion')
>>> inventory[0] = "battleax"
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in ?
    inventory[0] = "battleax"
TypeError: object doesn't support item assignment
```

- Sin embargo, si se puede crear nuevas tuplas a partir de unas existentes.

# Operaciones con tuplas

- La función `len()` sirve para obtener el número de elementos en la tupla.

```
# create a tuple with some items and display with a for loop
inventory = ("sword",
            "armor",
            "shield",
            "healing potion")

print("Your items:")
for item in inventory:
    print(item)

input("\nPress the enter key to continue.")

# get the length of a tuple
print("You have", len(inventory), "items in your possession.")

input("\nPress the enter key to continue.")
```

# Operaciones con tuplas

- Para concatenar tuplas, se las junta por medio del operador +

```
inventory = ("sword", "armor", "shield", "healing potion")
print(inventory)

# concatenate two tuples
chest = ("gold", "gems")
print("You find a chest. It contains:")
print(chest)

print("You add the contents of the chest to your inventory.")
inventory += chest
print("Your inventory is now:")
print(inventory)

input("\n\nPress the enter key to exit.")
```

- Note que no se modificó la tupla inventory original, sino que se creó una nueva con los elementos de inventory y chest.

# Listas vs. Tuplas

- Las listas pueden hacer todo lo que las tuplas y más. Sin embargo, es recomendable usar tuplas en los siguientes escenarios:
  - Las tuplas son más rápidas que las listas.
  - La inmutabilidad de las tuplas las hace apropiadas para la creación de constantes en vista que estas no pueden cambiar.
  - En ocasiones, Python requiere valores inmutables. Por ejemplo, los diccionarios requieren tipos inmutables, por lo tanto las tuplas son esenciales en la creación de este tipo de escenarios.

# Colecciones anidadas

- Se puede crear listas o tuplas anidadas.

```
>>> nested = ["first", ("second", "third"), ["fourth", "fifth", "sixth"]]
>>> print(nested)
['first', ('second', 'third'), ['fourth', 'fifth', 'sixth']]
```

- En este caso, `nested` solo tiene 3 elementos: una cadena, una tupla y una lista.
- No obstante, las colecciones anidadas usualmente obedecen a un patrón:

```
>>> scores = [("Moe", 1000), ("Larry", 1500), ("Curly", 3000)]
>>> print(scores)
[('Moe', 1000), ('Larry', 1500), ('Curly', 3000)]
```



# Colecciones anidadas

- Para acceder a los elementos de una colección anidada se lo hace por medio de su índice:

```
>>> scores = [("Moe", 1000), ("Larry", 1500), ("Curly", 3000)]
>>> print(scores[0])
('Moe', 1000)
>>> print(scores[1])
('Larry', 1500)
>>> print(scores[2])
('Curly', 3000)
```

- Si se desea obtener un valor dentro de una de las colecciones anidadas se lo puede hacer de la siguiente manera:

```
>>> print(scores[2][0])
Curly
```

# Colecciones anidadas

- Si se conoce de antemano el número de elementos en una secuencia, es posible asignar cada uno de estos a una variable:

```
>>> name, score = ("Shemp", 175)
>>> print(name)
Shemp
>>> print(score)
175
```

# Tuplas y funciones

- Cuando en una función retornamos mas de una valor, lo que en realidad estamos retornando es una tupla. Por ejem:

```
def func(x,y):  
    # code to compute x and y  
    return x,y
```

```
t = func(1, 2)
```

# Tuplas como argumentos

- Una función puede recibir un numero variable de argumentos, agregando “\*” al frente de un argumento y esta se traducirá en una tupla dentro de la función. Por ejem:

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

```
printall(1, 2, 3, 4)
```

```
(1, 2, 3, 4)
```

# Comparando tuplas

- Los operadores de comparación en las tuplas funcionan de la siguiente manera: Python empieza comparando los primeros elementos, si estos resultan ser iguales, compara los siguientes hasta que encuentre los elementos que difieran.

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

```
True
```

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

```
True
```

# Ejercicio 1

- Escriba una función sumaTotal que reciba como argumento un numero variable de enteros y retorne la suma de los mismos.

# Ejercicio 2

- El tiempo como tuplas:
  1. Proponer una representación con tuplas para representar el tiempo.
  2. Escribir una función sumaTiempo que reciba dos tiempos dados y devuelva su suma.

# Diccionarios



# Diccionarios

- Los diccionarios son colecciones de datos que permiten definir y acceder a los componentes mediante una clave.
- Cada componente de un diccionario es un par clave:valor

```
{clave: valor, clave: valor, clave: valor, ..., clave: valor}
```

```
>>> x={123: 'Algebra', 325: 'Física', 215: 'Química'}
```

'Algebra'	'Física'	'Química'
123	325	215

# Diccionarios

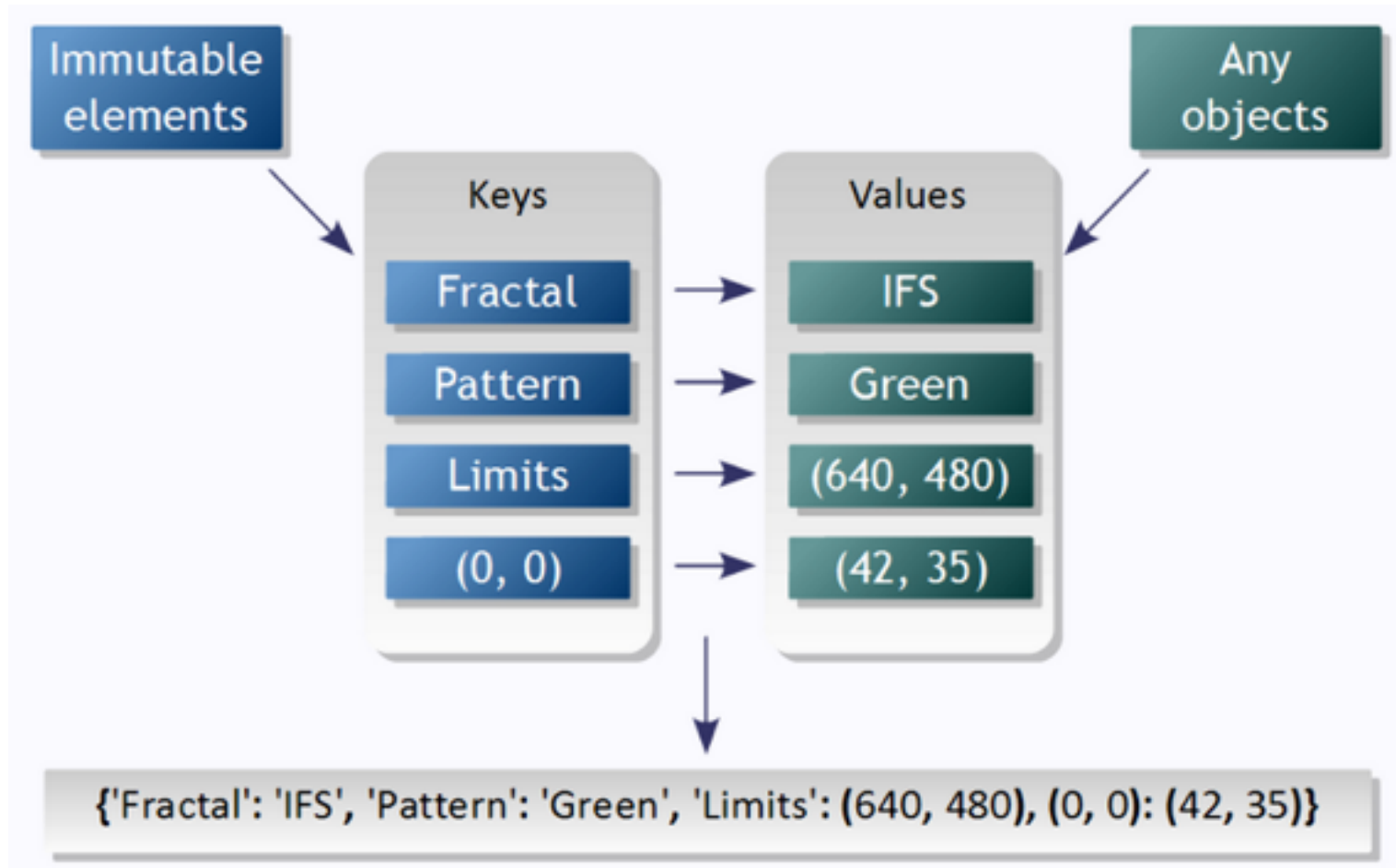
- No se pueden modificar las claves pero si los valores asociados a las mismas.
- Las claves y los valores pueden ser de cualquier tipo.

```
# Geek Translator
```

```
# Demonstrates using dictionaries
```

```
geek = {"404": "clueless. From the web error message 404, meaning page not found.",
        "Googling": "searching the Internet for background information on a person.",
        "Keyboard Plaque" : "the collection of debris found in computer keyboards.",
        "Link Rot" : "the process by which web page links become obsolete.",
        "Percussive Maintenance" : "the act of striking an electronic device to make ↩
↩it work.",
        "Uninstalled" : "being fired. Especially popular during the dot-bomb era."}
```

# Diccionarios



- Fuente: [http://nbviewer.ipython.org/github/ricardoduarte/python-for-developers/blob/master/Chapter5/Chapter5\\_Types.ipynb](http://nbviewer.ipython.org/github/ricardoduarte/python-for-developers/blob/master/Chapter5/Chapter5_Types.ipynb)

# Diccionarios

- Si se desea acceder, por medio de una clave, a un valor del diccionario basta con colocar la clave entre corchetes luego del nombre del diccionario.

```
>>> geek["404"]  
'clueless. From the web error message 404, meaning page not found. '  
>>> geek["Link Rot"]  
'the process by which web page links become obsolete. '
```

- Si la llave no existe se generará un error.

```
>>> geek["Dancing Baloney"]  
Traceback (most recent call last):  
File "<pyshell#1>", line 1, in <module>  
geek["Dancing Baloney"]  
KeyError: 'Dancing Baloney'
```

# Diccionarios

- Se puede usar el operador `in` con diccionarios tal y como se lo hace con listas y tuplas.

```
>>> if "Dancing Baloney" in geek:
    print("I know what Dancing Baloney is. ")
else:
    print("I have no idea what Dancing Baloney is. ")

I have no idea what Dancing Baloney is.
```

- También se puede acceder un valor por medio del método `get()`.

```
>>> print(geek.get("Dancing Baloney", "I have no idea. "))
I have no idea.
```

- Si no se especifica un valor predeterminado y la clave no existe, retornará `None`.

```
>>> print(geek.get("Dancing Baloney"))
None
```

# Operaciones con diccionarios

- Agregando un par clave-valor

```
# add a term-definition pair
elif choice == "2":
    term = input("What term do you want me to add?: ")
    if term not in geek:
        definition = input("\nWhat's the definition?: ")
        geek[term] = definition
        print("\n", term, "has been added.")
    else:
        print("\nThat term already exists! Try redefining it.")
```

# Operaciones con diccionarios

- Reemplazando un par clave-valor

```
# redefine an existing term
elif choice == "3":
    term = input("What term do you want me to redefine?: ")
    if term in geek:
        definition = input("What's the new definition?: ")
        geek[term] = definition
        print("\n", term, "has been redefined.")
    else:
        print("\nThat term doesn't exist! Try adding it.")
```

# Operaciones con diccionarios

- Eliminando un par clave-valor

```
# delete a term-definition pair
elif choice == "4":
    term = input("What term do you want me to delete?: ")
    if term in geek:
        del geek[term]
        print("\nOkay, I deleted", term)
    else:
        print("\nI can't do that!", term, "doesn't exist in the dictionary.")
```



# Operaciones con diccionarios

TABLE 5.2 SELECTED DICTIONARY METHODS

Method	Description
<code>get(key, [default])</code>	Returns the value of <i>key</i> . If <i>key</i> doesn't exist, then the optional <i>default</i> is returned. If <i>key</i> doesn't exist and <i>default</i> isn't specified, then <i>None</i> is returned.
<code>keys()</code>	Returns a view of all the keys in a dictionary.
<code>values()</code>	Returns a view of all the values in a dictionary.
<code>items()</code>	Returns a view of all the items in a dictionary. Each item is a two-element tuple, where the first element is a key and the second element is the key's value.

# Características de los diccionarios

- Un diccionario no puede contener múltiples elementos con la misma clave.
- Una clave es inmutable. Puede ser una cadena, un número o una tupla.
- Los valores no tienen que ser únicos.

# Ejercicio 1

- Escriba una función que reciba un numero  $n$  y retorne un diccionario con los números desde 1 hasta  $n$  como claves y el cuadrado de los mismos como valores.

# Ejercicio 2

- Escriba un programa que le pida 10 palabra a un usuario y le muestre el listado de las palabras ingresadas con la frecuencia de las mismas.

# Conjuntos

# Conjuntos

- Los conjuntos se construyen como una lista de valores, **no ordenados** ni repetidos, encerrados entre llaves.

```
>>> u={4,6,7,3,8,6}  
>>> u  
{8, 3, 4, 6, 7}
```

Definición directa de un conjunto

```
>>> r=set([7,3,8,6,9])  
>>> r  
{8, 9, 3, 6, 7}
```

Definición de un conjunto desde una lista

- También se pueden definir conjuntos con la instrucción `set(c)`, en donde `c` representa cualquier objeto que se pueda indexar, como listas, cadenas o tuplas.

# Add, update, copy

- Add

```
>>> s = set([12, 26, 54])
>>> s.add(32)
>>> s
set([32, 26, 12, 54])
```

- Update

```
>>> s.update([26, 12, 9, 14])
>>> s
set([32, 9, 12, 14, 54, 26])
```

- Copy

```
>>> s2 = s.copy()
>>> s2
set([32, 9, 12, 14, 54, 26])
```

# Operador IN

- Asimismo, podemos utilizar el operador “in” para saber si un elemento esta en el set.

```
>>> 32 in s
```

```
True
```

```
>>> 6 in s
```

```
False
```

```
>>> 6 not in s
```

```
True
```



# Subset

- Una función útil dentro de los conjuntos es saber si un conjunto es subconjunto de otro.

```
>>> s = set([32, 9, 12, 14, 54, 26])
>>> s.issubset(set([32, 8, 9, 12, 14, -4, 54, 26, 19]))
True
>>> s.issuperset(set([9, 12]))
True
```

# Removiendo elementos

- Pop: remueve un elemento cualquiera.

```
>>> s = set([1,2,3,4,5,6])
```

```
>>> s.pop()
```

```
1
```

```
>>> s
```

```
set([2,3,4,5,6])
```

- Remove: remueve el elemento especificado, si no existe lanza un error.

```
>>> s.remove(3)
```

```
>>> s
```

```
set([2,4,5,6])
```

- Discard: lo mismo que el “remove” pero sin lanzar un error.

```
>>> s.discard(3)
```

```
>>> s
```

```
set([2,4,5,6])
```

# Iterando sobre los elementos

- Es posible iterar sobre un conjunto, sin embargo, hay que recordar que los elementos en el set **no están ordenados**.

```
>>> s = set("blerg")
>>> for n in s:
...     print n,
...
r b e l g
```

# Operaciones: Unión

- union: combina dos conjuntos. Por ejem:

```
>>> s1 = set([4, 6, 9])
```

```
>>> s2 = set([1, 6, 8])
```

```
>>> s1.union(s2)
```

```
set([1, 4, 6, 8, 9])
```

```
>>> s1 | s2
```

```
set([1, 4, 6, 8, 9])
```

# Operaciones: Intersección

- intersección: obtiene los elementos que están entre dos conjuntos. Por ejem:

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.intersection(s2)
set([6])
>>> s1 & s2
set([6])
```

# Operaciones: Diferencia

- diferencia: obtiene los elementos que están en s1 pero no en s2. Por ejem:

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.difference(s2)
set([9, 4])
>>> s1 - s2
set([9, 4])
```

# Operaciones: Diferencia Simétrica

- diferencia simétrica: obtiene los elementos que están en s1 o s2 pero no en ambos. Por ejem:

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.symmetric_difference(s2)
set([8, 1, 4, 9])
>>> s1 ^ s2
set([8, 1, 4, 9])
```

# Próxima sesión

- Arreglos