Herramientas de programación en Python

2. Tipos de datos estructurados

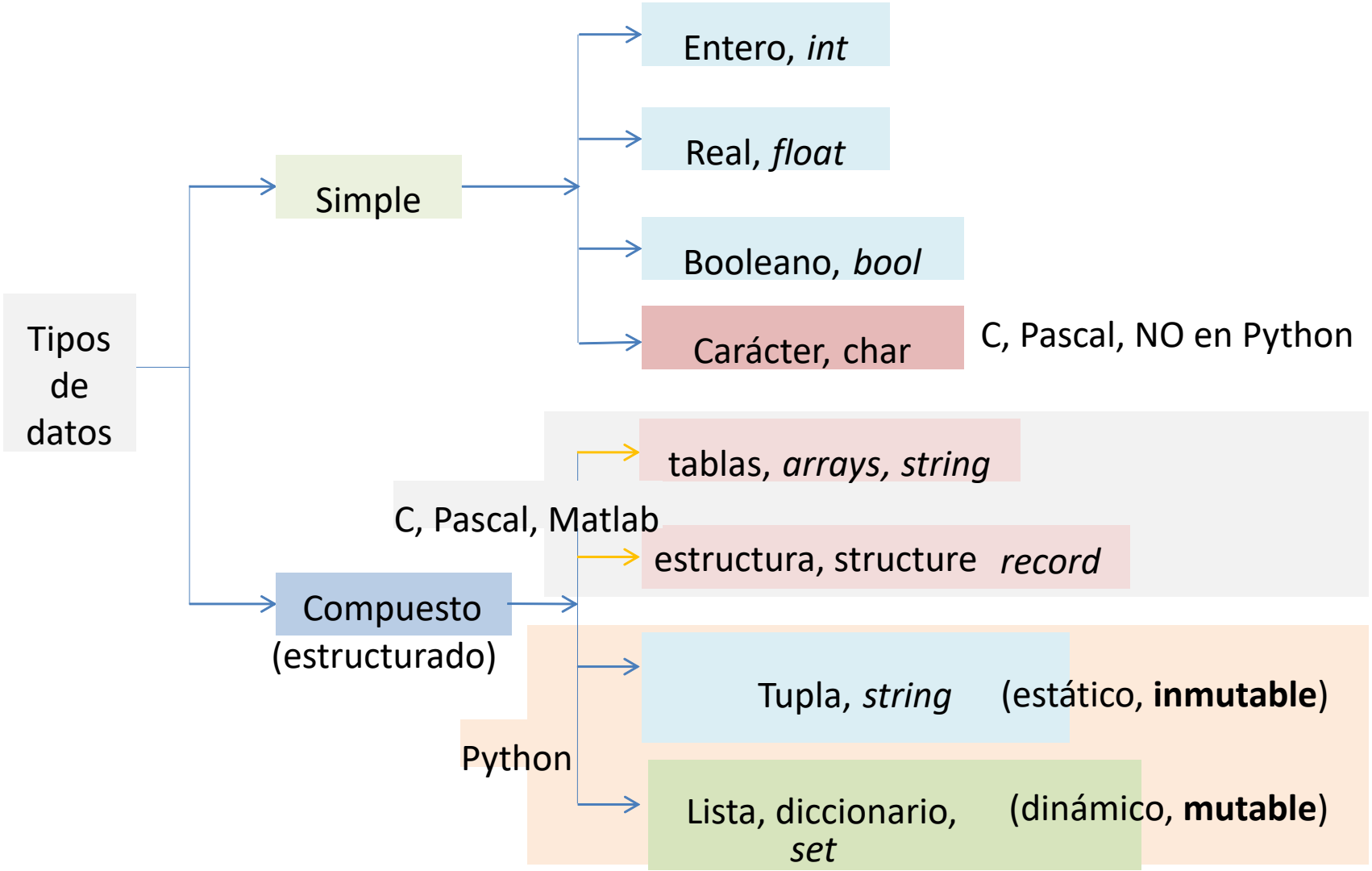
2.1 homogéneos y heterogéneos, **estáticos** y dinámicos

(strings y tuplas)

Pedro Gomis

pedro.gomis@upc.edu

Tipos de datos



Datos estructurados

for elemento in secuencia:
    Instrucciones

Secuencias típicas en Python:

- **range(n)** , ejemplo: **range(5)** → secuencia: 0,1,2,3,4
- **String,** ejemplo 'Hola' → secuencia 'H','o','l','a'
- **Tupla**, ejemplo (3, 5, True) → secuencia 3, 5, True
- **Listas**, ejemplo [4, 7, 'Hola'] → secuencia 4, 7, 'Hola'

Datos estructurados

- Los **datos compuestos o estructurados** comprenden aquellos datos con elementos de valores de un mismo tipo o de diferentes tipos, que se representan unificados en un **objeto** para ser guardados o procesados.
- Estos tipos de datos ***no son escalares*** pues se pueden dividir en elementos y acceder a ellos, son datos estructurados.
- Los datos estructurados o compuestos contienen elementos, que pueden ser datos **simples** u otros datos **compuestos**.
- Los elementos pueden ser todos del mismo tipo, como los **string** que contiene caracteres, y en este caso se llaman datos estructurados **homogéneos**. Otros lenguajes (C/C++, Matlab, Pascal) disponen de estructuras homogéneas como el ***array*** (arreglo o tabla). **Python** lo tiene en el módulo **numpy**.
- También los elementos pueden ser de distinto tipo. Este tipo se suele llamar datos estructurados **heterogéneos**.
- En **Python** los datos compuestos se pueden clasificar en dos grupos, de acuerdo a la ***característica*** de sus elementos, ***pueden o no ser cambiados***, reducidos o ampliados: datos estructurados **mutables** e **inmutables**.

Datos estructurados inmutables (estáticos)

- Los datos estructurados **inmutables** (llamados también **estáticos** o de **valores/tamaño fijos**), se caracterizan porque los elementos del objeto **no pueden ser cambiados ni eliminados, ni añadirse nuevos elementos**.
- En Python tenemos como datos estructurados inmutables las cadenas de caracteres (**string**) y las **tuplas** (tuple), como **secuencias de datos**, y los conjuntos congelados (**frozenset**).

Cadena de caracteres (string)

- Los **string**, son una secuencia de caracteres de cualquier tipo (letras, números, caracteres especiales; cualquier carácter Unicode*) que forman un **objeto de Python**.

```
>>> s = 'casa de madera'
>>> letra_1 = s[0]
>>> long = len(s)
>>> letra_ultima = s[long-1]    # alternativa: s[-1]
>>> print(letra_1, letra_ultima, long)
c a 14
```

* https://es.wikipedia.org/wiki/Unicode

1. Strings

Indexación o acceso y longitud de las secuencias

- Se pueden acceder a los elementos (caracteres) de los **string** (o de ***cualquier secuencia***) a través de un índice que se pone entre corchetes, [**indice**], y dice en qué posición se encuentra el elemento dentro del string.

```
>>> s = 'casa de madera'
```

	c	a	s	a		d	e		m	a	d	e	r	a
Índice o posición de los elementos →	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Primer elemento con el índice `0` (`letra_1 = s[0]`). En Python el primer elemento de las secuencias está en la posición `0` al accederlo.
- Para calcular el número de elementos, o longitud, de la secuencia de los datos estructurados usamos la función interna `len()`. La cadena tiene 14 elementos y su ***último*** elemento está en la ***posición*** 13 (`len(s)- 1`) o `-1`.
- Se puede crear un string `s` vacío (`len(s)` → `0`):

```
>>> s = ''    # (dos comillas simples o dobles sin espacio)
```

1. Strings

Recorte o rebanado (slicing) de secuencias y otras operaciones

- Para extraer un subconjunto de elementos (o segmento) de un string o de cualquier secuencia se usa el **operador de corte** (***slicing operator***) `[n:m]`, donde `n` es el primer elemento a extraer y `m-1` el último.

c	a	s	a		d	e		m	a	d	e	r	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'casa de madera'
>>> segm1 = s[0:3]         # segm1 <- 'cas'
>>> segm1 = s[:3]          # segm1 <- 'cas' , equivale al slice anterior
>>> segm2 = s[8:len(s)]    # segm2 <- 'madera'
>>> segm2 = s[8:]          # segm2 <- 'madera' , equivale al slice anterior
>>> segm3 = s[0:14:2]      # segm3 <- 'cs emdr', slice 0:14 en pasos de 2 en 2
>>> letra_u = s[-1]        # letra_u <- 'a', equivale a acceso último elemento
>>> letra_penu = s[-2]     # letra_penu <- 'r', equivale acceso penúltimo elem
```

- Si se omite el primer índice `[:m]` el recorte comienza desde el primer elemento.
- Si se omite el segundo índice `[n:]` se recorta hasta el final de la secuencia.
- Los ***índice negativos*** son útiles para acceder al último elemento `[-1]` o últimos, sin requerir el uso de la función **`len`**`()`.

`[ini:fin]` desde `ini` a `fin-1`

1. Strings

Operadores de concatenar (+) y repetir (*)

- Los operadores de concatenar (+) o repetir (*) ***strings*** son aplicables a cualquier secuencia de datos:

```
>>> s1='casa'
>>> s2 = 'de'
>>> s3 = 'madera'
>>> s4 = s1 + ' ' + s2 + ' ' + s3
>>> print(s4)
casa de madera
>>> s5 = s1 + ' grande'
>>> s5
'casa grande'
>>> s6 = 3*s1 + '!'
>>> s6
'casacasacasa!'
```

1. Strings

Operadores booleanos **in** y **not in**

- El operador **in** se considera un operador booleano sobre dos ***strings*** y devuelve **True** si el ***string*** de la izquierda es un segmento (o substring) del de la derecha. Si no lo es, devuelve **False**.
- El operador **not in** devuelve el resultado lógico opuesto. Ejemplos:

```
>>> s = 'casa de madera'
>>> 'asa' in s
True
>>> 'casade' in s
False
>>> 'casade' not in s
True
```

1. Strings

Los string son inmutables

- Recordemos que este tipo de dato se considera inmutable porque no podemos cambiar los valores de sus elementos ni cambiar su tamaño.
- Si queremos modificarlos podemos crear un nuevo valor y usarlo luego en la misma u otra variable.

```
>>> s = 'cesa'       # intentamos cambiar la 'e' por 'a'
>>> s[1] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s = s[0] + 'a' + s[2:]    # Solución!
>>> s
'casa'
```

- Si queremos poner en mayúscula la primera letra del ***string*** `s`

```
>>> s = 'casa de madera'
>>> s[0] = 'C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

1. Strings

Métodos de los strings

- Python es un ***lenguaje orientado a objetos*** y los datos en Python están en los **objetos**.
- Los **objetos** tienen asociados **métodos** para que manipulen sus datos. Los métodos *son similares a las funciones*, ya que reciben *argumentos* y devuelven valores (o modifican el objeto en los datos mutables).
- Los ***strings*** tienen métodos que le son propios.
- Por ejemplo, poner en mayúscula la primera letra del string del ejemplo anterior, o poner todas las letras en mayúscula:

```
>>> s = 'casa de madera'
>>> s1 = s.capitalize()
>>> s1
'Casa de madera'
>>> sM = s.upper()
>>> sM
'CASA DE MADERA'
```

1. Strings

Métodos de los strings

```
>>> s = 'casa de madera'
>>> sM = s.upper()                  # convierte las letras en mayúsculas
>>> sM
'CASA DE MADERA'
>>> sM.lower()                      # convierte las letras en minúsculas
'casa de madera'
>>> s.capitalize()                  # primera letra del string en mayúscula
'Casa de madera'
>>> s.title()               # primera letra de cada palabra del string en mayúscula
'Casa De Madera'
>>> i = s.find('e')         # busca el índice (posición) del primer string 'e'
>>> i                               # si no encontrara el string devuelve -1
6
>>> s.count('a')              # cuenta las veces que aparece el elemento o string
4
>>> s.count('de')
2
>>> s.replace('a','e')              # remplaza el primer string por el segundo
'cese de medere'
>>> s.split(' ')               # divide s usando el string ' ' produciendo lista
['casa', 'de', 'madera']
>>> s.split()                       # equivale a s.split(' ') o s.split(sep=None)
['casa', 'de', 'madera']
```

1. Strings

Ejemplos Tratamiento de strings

```
def contar_a(s):
    """Contar letra a en un string

    """
    n = 0
    for c in s:
        if c == 'a':
            n = n + 1
    return n
p = input('Entra texto: ')
print('El número de "a" es',contar_a(p))
```

Opción con método `s.count()`

```
def contar_a(s):
    return s.count('a')
```

Ejemplos Tratamiento de strings

```
def contar(s,c1):
    """Contar letra c1 en string s
    Ejemplos:
    >>> contar('patata','a')
    3
    >>> contar('patata caliente','a')
    4
    >>> contar('patata caliente','e')
    2
    >>> contar('patata','u')
    0
    """
    n = 0
    for c in s:
        if c == c1:
            n = n + 1
    return n
print(contar('Barcelona','a'))
print(contar('casablanca','a'))
```

Típico esquema de recorrido

Ejemplos Tratamiento de strings

```
def buscar(s,c1):
    """buscar letra c1 en string s
    Ejemplos:
    >>> buscar('patata','a')
    True
    >>> buscar('patata caliente','a')
    True
    >>> buscar('patata','u')
    False
    """
    for c in s:
        if c == c1:
            return True
    return False

print(buscar('Barcelona','a'))
print(buscar('casablanca','e'))
```

Típico esquema de búsqueda

```
def cambia(s,c1,c2):
    """Cambia caracter c1 por c2 (si existe c1) opción 1,
       recorriendo el string s
    Hay que crear otro string!
    Ejemplos:
    >>> cambia('patata','a','e')
    'petete'
    >>> cambia('camion','o','ó')
    'camión'
    >>> cambia('patata','e','i')
    'patata'
    """
    sc = ''  # string cambiado
    for c in s:
        if c == c1:
            sc = sc + c2
        else:
            sc = sc + c
    return sc
print(cambia('Barcelona','a','e'))
print(cambia('barco','o','a'))
```

Ejemplos Tratamiento de strings

Ejemplos Tratamiento de strings

```
def cambia(s,c1,c2):
    """Cambia c1 por c2 en string s (si existe c1)
    Hay que crear otro string!
    """
    n = len(s)
    sc = ''
    for i in range(n):
        if s[i] == c1:
            sc = sc + c2
        else:
            sc = sc + s[i]
    return sc

p = input('Entra texto: ')
t1 = input('cambiar?: ')
t2 = input('por: ')
print(p, 'cambiado',t1, 'por',t2, 'es:')
print(cambia(p,t1,t2))
```

Opción usando el índice del string (general para otros lenguajes)

```
Entra texto: Cesado
cambiar?: e
por: a
Cesado cambiando e por a es:
Casado
>>>
```

Ejemplos Tratamiento de strings

Opción con método s.replace()

```
def cambia(s,c1,c2):
    """Cambia caracter c1 por c2 (si existe c1) opción 2,
       recorriendo el string s
    Hay que crear otro string!
    Ejemplos:
    >>> cambia('patata','a','e')
    'petete'
    >>> cambia('camion','o','ó')
    'camión'
    >>> cambia('patata','e','i')
    'patata'
    """
    return s.replace(c1,c2)
```

2. Tuplas

- Las **tuplas** son una secuencia de elementos ordenados en un objeto de Python. SON INMUTABLES (como los strings)
- A diferencia de los ***strings*** (elementos son caracteres) las **tuplas** pueden contener elementos de cualquier tipo, incluso elementos de diferente tipo.
- Los elementos se indexan y rebanan igual que los ***strings***, a través de un número entero.
- La sintaxis de las **tuplas** es una secuencia de valores separados por comas. Se suelen encerrar entre paréntesis (aunque no es necesario):

```
# Ejemplo de tuplas
>>> a = 1, 2, 3
>>> a
(1, 2, 3)
>>> b = (3, 4, 5, 'a')
>>> b
(3, 4, 5, 'a')
>>> type(a)
<class 'tuple'>
>>> type(b)
<class 'tuple'>
```

2. Tuplas

- Lo importante es incluir las comas entre los elementos. Por ejemplo,

```
>>> t = 'k',
>>> t
('k',)
>>> type(t)
<class 'tuple'>
>>> t2 = 'k'
>>> t2
'k'
>>> type(t2)
<class 'str'>
```

- la función interna **tuple**() convierte una secuencia iterable, como un ***string*** o una **lista**, a **tupla**, o puede crear una tupla vacía:

```
>>> tuple('Hola')
('H', 'o', 'l', 'a')
>>> tuple([1, 2])
(1, 2)
>>> tuple()
()
```

2. Tuplas

Indexación, recorte y otras operaciones de tuplas

- El acceso a los elementos de las tuplas, la extracción de elementos y las operaciones se realizan de forma análoga a los strings:

```
>>> b = (3, 4, 5, 'a')
>>> b[0]
3
>>> b[-1]
'a'
>>> b[0:3]
(3, 4, 5)
>>> t = ('las','tuplas','son','inmutables')
>>> t[0]
'las'
>>> t[1] = 'listas'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

2. Tuplas

Indexación, recorte y otras operaciones de tuplas

- Podemos incluir tuplas dentro de las tuplas, concatenarlas y repetirlas, como los string:

```
>>> b = (3, 4, 5, 'a')
>>> c = (b, 2)
>>> b + c
(3, 4, 5, 'a', (3, 4, 5, 'a'), 2)
>>> 3*b
(3, 4, 5, 'a', 3, 4, 5, 'a', 3, 4, 5, 'a')
```

- La composición iterativa **for** - **in** de Python puede usar cualquier secuencia iterable, incluyendo las **tuplas**:

```
>>> juegos = ('tennis','baseball','football','voleyball','natación')
>>> for deporte in juegos:
...     print(deporte)
tennis
baseball
football
voleyball
natación
```

2. Tuplas

Asignaciones múltiples

- Python permite ***asignaciones múltiples*** mediante asignaciones con ***tuplas***.
- Estas acciones permiten que a una **tupla** de variables a la izquierda de una asignación le sea asignada una tupla de valores a la derecha de ésta.
- La condición a cumplir es que el número de variables de la tupla de variables sea igual al número de elementos de la tupla de valores.
- El objeto a asignar de forma múltiple a la **tupla** de variables puede ser un **string** o una **lista**, siempre que el número de caracteres o elementos sea igual al número de variables de la **tupla**.

```
>>> a,b,c = (1,2,3)
>>> a
1
>>> type(a)
<class 'int'>
>>> d,e,f = 'xyz'
>>> d
'x'
>>> type(d)
<class 'str'>
```

2. Tuplas

Asignaciones múltiples

- Esta característica de la asignación con tuplas resuelve el problema de ***intercambio*** de valores entre dos variables, que en otros lenguajes requiere de **una variable auxiliar**.

Sin usar tuplas, estamos obligados a ejecutar lo siguiente:

```
>>> a = 10
>>> b = 20
>>> tmp = a
>>> a = b
>>> b = tmp
>>> a
20
>>> b
10
```

Usando **tuplas** sería:

```
>>> a = 10
>>> b = 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

2. Tuplas

Funciones con devoluciones múltiples

- Las funciones pueden devolver múltiples resultados que pueden ser asignados a múltiples variables con el uso de **tuplas**.
- Siendo **estrictos**, **las funciones solo devuelven un resultado**. Pero si ese valor es una tupla, entonces ésta se puede asignar a una tupla de variables.
- Se requiere que el número de elementos coincida.

```
def  miFuncion(x):
    """
     Devuelve 2 valores: x incrementado y decrecido en 1
    """
    return x + 1, x - 1
a, b = miFuncion(10)
print(a, b)
print(miFuncion(20))
```

```
>>>
11 9
(21, 19)
```

2. Tuplas

Funciones con número arbitrario de parámetros, usando tuplas

- Existe la opción de definir una función con un número arbitrario (variable) de parámetros usando el operador * antes del nombre del parámetro.
- * se conoce, en este caso, como el operador de desempacar (*unpacking*). Desempaca una tupla

```
def media(*par):
    suma = 0
    for elem in par:
        suma = suma + elem
    return suma/len(par)
```

```
# opción con función sum()
def media(*par):
    suma = sum(par)
    return suma/len(par)
```

```
print(media(3, 4))
print(media(10.2, 14, 12, 9.5, 13.4, 8, 9.2))
print(media(2))
```

```
>>>
3.5
10.9
2.0
```

2. Tuplas

Métodos de las tuplas

- Al igual que en los string, existen métodos asociados a los objetos tipo tupla.
- Pero solo los métodos: `s.index(x)` y `s.count(x)`.
- También se pueden usar las funciones internas **`max`** y **`min`**, cuando las tuplas sean de valores ***numéricos***. Si los elementos son ***strings***, calcula el mayor o menor elemento, de acuerdo a la posición en la tabla ASCII del primer carácter.

```
a = (2, 3, 4, 5, 79, -8, 5, -4)
>>> a.index(5)   # índice de la primera ocurrencia de 5 en a
3
>>> a.count(5)   # ocurrencias totales de 5 en a
2
>>> max(a)
79
>>> min(a)
-8
>>> b = ('az','b','x')
>>> max(b)
'x'
>>> min(b)
'az'
```

2. Tuplas

Iterador zip (del inglés zipper –cremallera)

- Función zip: Es un iterador que opera sobre varios iterables, y crea tuplas agregando elementos de las secuencias iterables (por ejemplo, strings).
- Aplicación de zip para recorrer 2 secuencias en paralelo.

```
def CuentaElemsMismaPosición(s1,s2):
    """ Dice cuántas letras iguales están en la misma posición en 2
        palabras s1 y s2. Se pueden usar listas o tuplas
    >>> CuentaElemsMismaPosición('Hola', 'Colado')
    3
    """
    contador = 0
    for c1, c2, in zip(s1,s2):
        if c1 == c2:
            contador += 1
    return contador
```

```
>>> CuentaElemsMismaPosición('Hola', 'Adios')
0
>>> CuentaElemsMismaPosición('Hola', 'Colado')
3
```

Conjuntos congelados - Frozenset

- En Python se dispone de otro grupo de datos estructurados heterogéneos que tratan de guardar cierta relación con la teoría de conjuntos son los conjuntos **Set** y los **Frozenset**.
- Estos datos. Los primeros los presentamos en la siguiente sección de datos estructurados mutables o dinámicos.
- Un conjunto congelado (**Frozenset**) es una colección de elementos no ordenados que sean únicos e inmutables. Es decir, puede contener números, string, tuplas, pero no listas. Que sean elementos únicos quiere decir que no estén repetidos. Los Set y Frozenset no son secuencias de datos.
- Los conjuntos congelados son inmutables porque no se pueden cambiar, ni quitar o añadir elementos. Ejemplos de datos congelados:

```
>>> FS1 = frozenset({25, 4, 'a',2, 25, 'casa', 'a'})
>>> FS1
frozenset({2, 'a', 4, 'casa', 25})
>>> type(FS1)
<class 'frozenset'>
>>> len(FS1)
5
```