TEMA 2
PROGRAMACIÓN

TEMA 2. PROGRAMACIÓN

- 2.1 Algoritmos
- 2.2 Tipos de datos simples y variables
- 2.3 Instrucciones básicas
- 2.4 Estructuras de control
- 2.5 Tipos de datos estructurados: Secuencias
- 2.6 Funciones

2.2 Tipos de datos compuestos

Cadena *str* : secuencia de caracteres

Ejemplo: cadena=’buenos días’

Lista *list*: secuencia de elementos, potencialmente de distinto tipo, a la que se le puede eliminar elementos, añadir nuevos, y modificar valores individuales.

Ejemplo: lista=[’hola’,4, (1,2),{3,4}]

2.5 Tipos de datos estructurados: Secuencias

```
nombre = “ALBERTO”
for letra in nombre:
    print(letra)
```

2.5 Tipos de datos estructurados: Secuencias

- Cadenas ***str***

  Podemos acceder a cada uno de los caracteres de una cadena utilizando un operador de indexación:

  a[i] es el carácter que ocupa la posición i+1 en a

```
>>> a =
"ABCDEFG"
>>> a[0]
'A'
>>> a[1]
'B'
>>> a[-1]
'G'
>>> a[-2]
'F'
```

2.5 Tipos de datos estructurados: Secuencias

- **Subcadenas**
- La expresión a[i:j] significa que se desea obtener la subcadena formada por los caracteres a[i] , a[i+1] , . . . , a[j-1].
- El corte a[:j] es equivalente a a[0,j] y el corte a[i:] equivale a a[i:len(a)]
- a[0:len(a):2] selecciona los caracteres de índice par
- Invertir la cadena: a[::-1]

```
>>> a[2:4]
'CD'

>>> a[:3]
'ABC'

>>> a[2:]
'CDEFG'

>>> a[0:len(a):2]
'ACEG'

>>> a[::-1]
'GFEDCBA'
```

2.5 Tipos de datos estructurados: Secuencias

- Cadenas ***str***
  - **Cuidado:** Las cadenas son inmutables
  -

```
>>> a = "ABCD"

>>> a[1]
'B'

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

2.5 Tipos de datos estructurados: Secuencias

- Listas ***list***

Python nos permite definir secuencias de valores de cualquier tipo. Los valores de una lista deben estar entre corchetes y separados por comas.

```
contactos = ['Maria','Juan','Miguel', 'Sara']

print('Tengo {} amigos'.format(len(contactos)))
for nombre in contactos:
    print(nombre)
```

2.5 Tipos de datos estructurados: Secuencias

- Listas ***list***

Muchos de los operadores y funciones que trabajan sobre cadenas, también lo hacen sobre listas: len, +,*,operador de corte:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]

>>> len(a)
3

>>> a + b
[1, 2, 3, 4, 5, 6]

>>> a * 2
[1, 2, 3, 1, 2, 3]
```

2.5 Tipos de datos estructurados: Secuencias

- Listas ***list***

  Modificación de elementos de listas:

```
>>> lista = [1, 2, 3]
>>> lista
[1, 2, 3]
>>> lista[1] = 4
>>> lista
[1, 4, 3]
```

2.5 Tipos de datos estructurados: Secuencias

- **append(), del(), pop()**

```
>>> a = [1, 2, 3]
>>> a + [4]
[1, 2, 3, 4]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

```
>>> del(a[1])
>>> a
[1, 3, 4]
```

```
>>> a.pop()
4
>>> a
[1, 3]
>>> a.pop(0)
1
```

2.5 Tipos de datos estructurados: Secuencias

- Listas ***list***

Mutabilidad, inmutabilidad y representación de la información en memoria:

- Python procura almacenar en memoria una sola vez cada valor inmutable

```
>>> nombre1 = "Juan"
>>> nombre2 = "Juan"
>>> nombre1 ==
nombre2
True
>>> nombre1 is nombre2
True
```

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
>>> c = a
>>> a is c
True
```

2.5 Tipos de datos estructurados: Secuencias

- Listas ***list***
  - ***split()***
  - ***join()***

```
>>> lista = input("lista: ")
lista: A B C D E F

>>> lista
'A B C D E F'

>>> lista2 = lista.split()
>>> lista2
['A', 'B', 'C', 'D', 'E', 'F']

>>> ','.join(lista2)
'A,B,C,D,E,F'
```

2.5 Tipos de datos estructurados: Secuencias

- **Leyendo listas**

```
>>> n = 5
>>> a = []
>>> for i in range(n):
...     el = input('Elemento: ')
...     a.append(el)
...
Elemento: 1
Elemento: 2
Elemento: 3
Elemento: 4
Elemento: 5
>>> a
[1, 2, 3, 4, 5]
```

```
lista = input('lista: ')
lista: [1,2,3,4,5]

>>> eval(lista)
[1, 2, 3, 4, 5]
```

2.5 Tipos de datos estructurados: Secuencias

```
# Reading an mxn matrix
m = input("Number of rows?" )
n = input("Number of columns?" )
M = [[None] * n] * m
for i in range(m):
    for i in range(n):
        M[i,j]= input("Element ({},{})".format(i,j))
```

```
>>> M = input('M= ')
M= [[1,2,3],[4,5,6],[7,8,9]]

>>> eval(M)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

2.2 Tipos de datos compuestos

- **Conjunto *set*:** colección de elementos (sin orden y no repetidos), potencialmente de distinto tipo simple al que se le pueden añadir nuevos elementos o eliminar existentes.
  - Ejemplo: conj1={'infinito',1,0,5,('a',1)}
- **Conjunto congelado *frozenset*** : colección de elementos, potencialmente de distinto tipo (mientras estén entre los simples e inmutables), no repetidos y sin orden entre sí
  - Ejemplo: congelado=frozenset({3,5,6.1})
- **Tupla *tuple***: secuencia de 0, 1 o n elementos, potencialmente de distinto tipo

  Ejemplo: t1=(1,'a',3,3)
- **Diccionario *dict*:** colección (conjunto) de pares de clave-valor. La clave es de cualquier tipo inmutable. Los valores pueden ser de cualquier tipo.

  Ejemplo: ingredientes={'tomate':(1,'Kg'), 'pepino':2,'sal':'1 cucharilla', 'aceite':.1, 'oregano':'1 pizca'}

Tuplas

- Es una secuencia de valores agrupados, sirve para agrupar varios valores que deben ir juntos (como si fueran un único valor)
- Diferencia con una lista es que los elementos de las tuplas no se pueden modificar
- ***Creación de tupla:*** se ponen los valores separados por comas y entre paréntesis

```
>>> persona = ('Miguel','Moraga')
>>> persona
('Miguel', 'Moraga')
>>> nombre,apellido = persona
>>> apellido
'Moraga'
```

Tuplas

- **Desempaquetado de tuplas**: los valores de las tuplas pueden ser recuperados asignando el valor a alguna variable.

```
('Miguel', 'Moraga')
>>> nombre,apellido = persona
>>> apellido
'Moraga'
```

- Tambien se puede extraer los valores usando el indice (como en listas)
- >>> persona[1]
- ‘Moraga’

Procesando secuencias

```
team = [‘juan’,’alberto’, ‘nuria’]

for name in team:
    print(name)


for i in range(len(team))
    print(“{}.- {}”.format(i, team[i]))


for i, name in enumerate(team):
    print(“{}.- {}”.format(i, name))
```

Tuplas

- De lista a tupla: tuple (l)
- De tupla a lista: list (t)

```
>>> a = (1, 2, 3)
>>> b = [4, 5, 6]
>>> list(a)
[1, 2, 3]
>>> tuple(b)
(4, 5, 6)
```

Usos de Tuplas

- Se usan siempre que es necesario agrupar valores

```
partido1 = ('Milan', 'Bayern')
```

- La funcion distancia toma 2 puntos en el plano (x,y) y retorna la distancia que existe entre ellos

```
def distancia(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    dx = x2 - x1
    dy = y2 - y1
    return (dx ** 2 + dy ** 2) ** 0.5
```

```
>>> a = (2, 3)
>>> b = (7, 15)
>>> distancia(a, b)
13.0
```

Diccionarios

- Es un tipo de dato que asocia pares de objetos
- Es una colección de llaves las cuales tienen asociadas un valor. Las llaves no están ordenadas y no hay llaves repetidas.
- Para obtener un valor se debe hacerlo a través de su llave
- X = { llave : valor }

- **Crear Diccionarios**
- Se crean usando llaves { }. La llave y el valor se separan por dos puntos
- Diccionario vacio: {} o dict()

Uso de diccionarios

```
alberto = {
   ‘nombre’: "Alberto",
   ‘edad’: 25,
   ‘ciudad’: "Pamplona" }
juan = {
   ‘nombre’: "Ana",
   ‘edad’: 28,
   ‘ciudad’: "Madrid" }
contactos = [alberto, juan]
```

Uso de diccionarios

- Si se asigna un valor a una llave ya asignada en el diccionario, se sobreescribe el valor.
-

```
alberto['ciudad'] = 'Madrid'
```

- Borrar una llave
-

```
del(alberto['ciudad'])
```

Uso de diccionarios

- alberto.keys()
- alberto.values()
- alberto.items()

Conjuntos

- Colección desordenada de valores no repetidos
- Son analogos a los conjuntos matematicos, y se representan por set.
- set es mutable, una vez creado el conjunto, puede ser modificado
- **Creacion de Conjuntos**

  - **Literal (entre llaves):**
  - pares = {2,4,6,8,10}
  - **Funcion set**
  - **set([1,2,2,3,4,4,5])**

Conjuntos

- Conjunto vacío: set ( )
- Los elementos de un conjuntos deben ser inmutables (no se puede crear un conjunto de listas, si de tuplas)
- Un conjunto **NO** es ordenado
- Es iterable

Conjuntos

```
impares = { 1, 3, 5, 7 }
primos = { 2, 3, 5, 7 }
pares = {2,4,6,8}
```

- membership ∈ Python: **`4 in primos`** ⇒ False
- union ∪ Python: **`impares | primos`** ⇒ { 1, 2, 3, 5 }
- intersection ∩ Python: **`impares & primos`** ⇒ { 3, 5 }
- difference \ or - Python: impares **`- primos`** ⇒ { 1 }

- impares.add(11)
- impares.remove(3)