Estructuras de datos I

IEEE
UPCT SB

Estructuras en Python

- Las estructuras son simplemente series de elementos ordenados según un criterio.
- En lenguajes de bajo nivel la estructura más simple es el array o arreglo (datos numéricos ordenados consecutivamente).
- En python, de base, contamos con estructuras de más alto nivel. Hoy veremos:
  - Listas
  - Diccionarios
  - Tuplas

Listas

- Una lista es una estructura de datos ordenados.
- Los datos pueden ser de distintos tipos.
- Para crear una lista usamos corchetes.

```
Ej:  a = [1, 2, 3]
     b = [‘hola’, 1, [1,2], a]
```

- Es la estructura básica más utilizada para la iteración.

Listas

- La lista es un tipo de dato **mutable**, es decir, podemos cambiar los elementos que la componen y modificar su estructura.
- Para acceder a los elementos de la lista usamos el operador de acceso []:

```
a = [1, 2, 3]

a[0] = 5

a[-1] = 4

print(a) → [5, 2, 4]
```

Listas multidimensionales

- Un elemento de una lista puede ser otra lista.

  A = [[1, 2, [1, 2]], [“Hola”, “a”, “todos”]]

- Esto permite crear listas multidimensionales. Ejemplo de esto es por ejemplo la estructura de una matriz:

  matriz = [[1, 2, 3], [4, 5, 6]]

Iterar con listas

- Podemos iterar directamente sobre los valores de una lista:

```
a = [1, 2, 3]

for item in a:

    print(item) → 1 => 2 => 3
```

- En cada iteración item copia el valor de cada elemento de a. Si cambiamos el valor de item no cambiará el valor de a.

Iterar con listas

- La lista es muy útil por la posibilidad de iterar sobre ella y cambiar sus valores. Para acceder a los valores usamos el operador de acceso y su índice.
- Para conocer su índice usamos range con la longitu de la lista que extraemos con len().

Ej:

```
a = [1, 2, 3]
for i in range(len(a)):
    a[i] = a[i] * 2
```

Iterar con listas

- Otra forma de iterar con listas es usar la función enumerate, que devuelve en número de iteración además del elemento.

```
a = [1, 2, 3]

for index, item in enumerate(a):
    print(index, item)
```

- Por supuesto podemos iterar sobre listas de más de una dimensión:

```
matriz = [[1,2,3],[4,5,6],[7,8,9]]

for fila in matriz:
    for item in fila:
        print(item, end=‘ ’)
    print()
```

Copiar listas

- Si intentamos copiar una lista usando el operador de asignación nos podemos encontrar en una operación no deseada.

```
a = [1, 2, 3]

b = a

b[0] = 10

print(a) → [10, 2, 3]  # !!!
```

- Para evitar esto usaremos el slicer [:] para acceder a todos los valores a la vez.

```
a = [1, 2, 3]

b = a[:]

b[0] = 10

print(a) → [1, 2, 3]
```

Operaciones con listas

- Las listas son muy flexibles porque permiten operaciones con ellas:

```
a = [1, 2, 3]

b = a + [4, 5] → [1, 2, 3]

c = a*3 → [1, 2, 3, 1, 2, 3]

d = [0] * 100 # 100 elementos 0

4 in a → false
```

- Además contamos con diferentes métodos que nos hacen la vida más fácil.

Métodos de listas

- Algunos de los métodos más utilizados son:
  - append, añade al final:

    a = [1, 2, 3]

    a.append(4) → [1, 2, 3, 4]
  - count, cuenta coincidencias:

    a = [1, 1, 0, 1, 0, 1, 1]

    a.count(1) → 5
  - index, devuelvel el índice de la primera coincidencia:

    a = [1, 2, 6, 3, 1, 6, 8]

    a.index(6) → 2

Tuplas

- Las tuplas funcionan como las listas pero es un tipo de dato inmutable, es decir, no podemos cambiar los elementos que la componen (el operador de acceso [] solo es de lectura) ni cambiar su estructura (nada de append). Se generan con ().

  Ej: a = (1, 2, 3)

  a[0] = 4 → Error!

- Cuando devolvemos más de un argumento en una función estamos devolviendo en realidad una tupla.

```
def foo(a, b):
    return a+b, a-b → tupla
```

Tuplas

- Las tuplas son útiles porque podemos empaquetar los argumentos de una función y luego desempaquetarlos con el operador *:

```
mi_tupla = (3, “Hola”)

def foo(a, b):
    for i in range(a):
        print(b)

foo(*mi_tupla) → “Hola” => “Hola” => “Hola”
```

Conversión

- Para convertir una tupla en lista:

```
a_t = (1, 2, 3)
a_l = list(a_t)
```

- Para convertir una lista en tupla:

```
a_l = [“Hola”, “Mundo”]
a_t = tuple(a_l)
```

Funciones con listas/tuplas

- Si queremos partir un string en función de un separador usamos el método split. split devuelve una lista con los elementos extraídos.

```
palabras = “Hola a todos”.split(‘ ‘)

type(palabras) → list
```

- Si queremos hacer la operación inversa usamos el método de string join, que une en función del carácter elegido una lista o tupla.

```
frase = ‘-’.join([“Hola”, “a”, “todos”])

print(frase) → Hola-a-todos
```

Funciones con listas/tuplas

- Hay una función muy útil llamada map que permite aplicar una función a todos los elementos de una lista/tupla:

```
lista = [1.354, 4.65, 2.56, 47.53]

for i in map(int, lista): # map(funcion, lista/tupla)
    print(i) → 1, 4, 2, 47

for i in map(lambda x: x*2, lista):
    print(i) → Devuelve el doble de cada valor
```

Si queremos aplicar el cambio a una lista simplemente tendremos que desempaquetar:

```
lista_enteros = [*map(int, lista)]
```

Funciones con listas/tuplas

- map permite funciones con varios argumentos pero necesitarás una lista/tupla por argumento.

```
A = [1, 2, 3]

B = [2, 2, 3]

map(lambda a, b : a*b, A, B) → 2, 4, 9
```

Funciones con listas/tuplas

- * Lambda es una forma corta de hacer funciones de una instrucción, su sintaxis es:

  lambda par1, par2, …, parN : instrucción

  Pueden ser guardadas en varibles y funcionar como una función.

  doble = lambda x: 2*x

  doble(3) → 6

- map permite funciones con varios argumentos pero necesitarás una lista/tupla por argumento.

  A = [1, 2, 3]

  B = [2, 2, 3]

  map(lambda a, b : a*b, A, B) → 2, 4, 9

Code pills

- Más adelante veremos como leer/escribir ficheros correctamente pero por ahora vamos a usar estos trozos de código para obtener y escribir información:
-
```
def leer(nombre: str = 'in.txt') → str:
    with open(nombre, 'r') as file:
        out = file.read()
    return out
```
-
```
def escribir(data: str, nombre: str = 'out.txt') → None:
    with open(nombre, 'r') as file:
        file.write(str)
```

Ejercicio

- Ordena una lista de números enteros por métodos iterativos.
- Sacar la moda de una lista de números mediante métodos iterativos.
- Hacer un módulo que permita realizar las operaciones básicas de matrices (+, -, ·, /, x, .)
- Elimina de una lista los datos atípicos extremos (puedes usar el módulo statistics).

$v < Q_1 - 3 \cdot IQR$ ó $v > Q_3 + 3 \cdot IQR \mid IQR = Q_3 - Q_1$

Ordenar

- Una forma más decente de ordenar una lista es usar la función sorted() o el método sort.

  A = [23, 14, 65, 1]

  sorted(A) → [1, 14, 23, 65]

  A.sort() → A = [1, 14, 23, 65]

  A.sort(reverse=True) → A = [65, 23, 14, 1]

Diccionarios

- Los diccionarios son estructuras de datos formadas por un par clave-valor. Se crean con {}:

```
A = {‘a’: 1, ‘b’: 2}

B = {1: “Hola”, “T”:[1, 2, 3]}

C = dict() # Diccionario vacío.
```

- Las claves deben ser datos inmutables pero los valores pueden ser de cualquier tipo.
- Podemos anidar diccionarios:

```
A = {“A”:{‘a’: 1, ‘b’: 2},
     “B”:{‘a’:[1, 2], ‘b’:[3, 4]}}
```

Diccionarios

- Para acceder a los datos de un diccionario usamos el operador de acceso [].

```
prefj = {“es”:34, “fr”:33, “al”: 49}

prefj[‘es’] → 34
```

- También podemos cambiar el valor de un elemento y aumentar el dicionario:

```
prefj[‘ru’] = 44
```

- Para ver si hay una clave en el diccionario usamos el operador in.

```
if ‘us’ in prefj:
    print(prefj[‘us’])
```

Diccionarios

- Podemos iterar sobre los diccionarios de la siguientes formas:
  - Sobre las claves:

```
for key in diccionario:
  pass
for key in diccionario.keys(): # igual
  pass
```

  - Sobre los valores:

```
for value in diccionario.values():
  pass
```

  - Sobre los pares clave-valor:

```
for key, value in diccionario.items():
  pass
```

Diccionarios

- Encontramos algunos métodos útiles en los diccionarios:
  - get(key, valor_defecto). Permite acceder a los elementos sin que se genere un error, si no encuentra la clave devuelve el valor por defecto (por defecto es None).

    A = {1:'a', 2:'b'}

    A.get(3) → None

    A.get(3, 'nada') → 'nada'

Diccionarios

- update(diccionario). Actualiza el valor de un diccionario con otro diccionario.

  A = {1:’a’, 2:’b’}

  B = {2:’x’, 3:’y’}

  A.update(B) → {1:’a’, 2:’b’, 3:’c’}

- copy(). Hace una copia el diccionario.
- pop(clave). Elimina un valor del diccionario (y lo devuelve).

Ejercicios

Diccionarios.

- Implementación de leer cadena con diccionario.
- Implementación de recursividad con diccionario. [Profe]
- Realiza la tabla de frecuencias de una lista de números.
- Json.

Final del tema.

- Búsqueda binaria.