Estructura de datos

PYTHON

¿QUE SON?

Las **estructuras de datos en programación** son un modo de representar información en una computadora, aunque, además, cuentan con un comportamiento interno. ¿Qué significa? Que se rige por determinadas reglas/restricciones que han sido dadas por la forma en que está construida internamente. En el ámbito de la informática, las **estructuras de datos** son aquellas que nos permiten, como desarrolladores, organizar la información de manera eficiente, y en definitiva diseñar la solución correcta para un determinado problema. Ya sean las más utilizadas comúnmente -como las **variables**, **arrays**, **conjuntos** o **clases**- o las diseñadas para un propósito específico -**árboles**, **grafos**, **tablas**, etc.-, una **estructura de datos** nos permite trabajar en un algo nivel de abstracción almacenando información para luego acceder a ella, modificarla y manipularla

TIPOS DE ESTRUCTURAS DE DATOS

Primero, debemos diferenciar entre **estructura de dato estática** y **estructura de dato dinámica.**

Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa, mientras que una **estructura de datos dinámica** es aquella en la que el tamaño ocupado en memoria puede modificarse durante la ejecución del programa.

Una típica dentro de las **estructuras de datos estáticas** son los **arrays**:

PYTHON

En términos técnicos, Python es un lenguaje de programación de alto nivel, orientado a objetos, con una semántica dinámica integrada, principalmente para el desarrollo web y de aplicaciones informáticas.

Es muy atractivo en el campo del Desarrollo Rápido de Aplicaciones (RAD) porque ofrece tipificación dinámica y opciones de encuadernación dinámicas.

Python es relativamente simple, por lo que es fácil de aprender, ya que requiere una sintaxis única que se

Arrays: Un **array** es un tipo de **dato estructurado** que permite almacenar un conjunto de datos homogéneo y ordenado, es decir, todos ellos del mismo tipo y relacionados. Su condición de *homogéneo*, indica que sus elementos están compuestos por el mismo tipo de dato, y su condición de *ordenado* hace que se pueda identificar del primer al último elemento que lo compone. Por otro lado, vimos que en programación existen **estructuras de datos dinámicas**, es decir, una colección de elementos -nodos- que normalmente se utilizan para dejar asentados registros. A diferencia de un **array** que contiene espacio para almacenar un número fijo de elementos, una **estructura dinámica de datos** se amplía y contrae durante la ejecución del programa. Veamos algunos casos:

ESTRUCTURA DE DATOS LINEALES

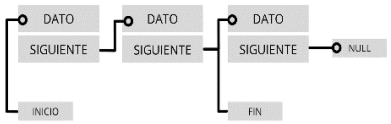
Las **estructuras de datos lineales** son aquellas en las que los elementos ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un único sucesor y predecesor, es decir, sus elementos están ubicados uno al lado del otro relacionados en forma lineal.

Hay tres tipos de **estructuras de datos lineales**:

- Listas enlazadas
- Pilas
- Colas

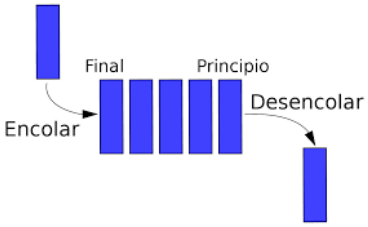
1. Listas enlazadas

En las **estructuras de datos**, las listas enlazadas se construyen con elementos que están ubicados en una secuencia. Aquí, cada elemento se conecta con el siguiente a través de un enlace que contiene la posición del siguiente elemento. De este modo, teniendo la referencia del principio de la lista podemos acceder a todos los elementos de esta.



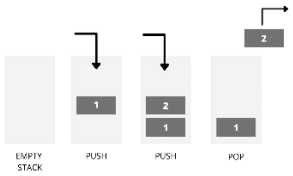
2. Cola

Una cola (también llamada fila) es un tipo de dato abstracto, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pull por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir**.**



3. Pila

La pila es un tipo especial de **lista lineal** dentro de las **estructuras de datos dinámicas** que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés *Last In, First Out*, es decir, *último en entrar, primero en salir*). ¿Cómo funciona? A través de dos operaciones básicas: apilar (push), que coloca un objeto en la pila, y su operación inversa, desapilar (pop), que retira el último elemento apilado.



Estructura de datos no lineales

Las **estructuras de datos no lineales**, también llamadas multienlazadas, son aquellas en las que cada elemento puede estar enlazado a cualquier otro componente. Es decir, cada elemento puede tener varios sucesores o varios predecesores.

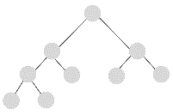
Existen dos tipos:

- Árboles
- Grafos

1. Árboles

En **estructura de datos**, los árboles consisten en una **estructura no lineal** que se utiliza para representar datos con una relación jerárquica en la que cada elemento tiene un único antecesor y puede tener varios sucesores.

Los mismos se encuentran clasificados en: **árbol general**, un árbol donde cada elemento puede tener un número ilimitado de subárboles y **árboles binarios**, que son una estructura de datos homogénea, dinámica y no lineal en donde a cada elemento le pueden seguir como máximo dos nodos.



2. Grafos

Otro tipo de **no lineal** de **estructura de datos en programación**, son los **grafos**. Se trata de una estructura matemática formada por un conjunto de puntos —una estructura de datos— y un conjunto de líneas, cada una de las cuales une un punto a otro. Los puntos se llaman nodos o vértices del grafo y las líneas se llaman aristas o arcos.

ESTRUCTURAS DE DATOS EN PYTHON

1. Listas

El tipo de dato lista tiene algunos métodos más. Aquí están todos los métodos de los objetos lista:

list.**append**(*x*)
Agrega un ítem al final de la lista. Equivale a a[len(a):] = [x].

list.**extend**(*iterable*)
Extiende la lista agregándole todos los ítems del iterable. Equivale a a[len(a):] = iterable.

list.**insert**(*i*, *x*)
Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto a.insert(0, x) inserta al principio de la lista y a.insert(len(a), x) equivale a a.append(x).

list.**remove**(*x*)
Quita el primer ítem de la lista cuyo valor sea *x*. Lanza un ValueError si no existe tal ítem.

Un ejemplo que usa la mayoría de los métodos de la lista:

```
>>>
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

2. La instrucción del

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción del. Esta es diferente del método pop(), el cual retorna un valor. La instrucción del también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la sección). Por ejemplo:

```
>>>
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del puede usarse también para eliminar variables:

```
>>>
>>> del a
```

Hacer referencia al nombre a de aquí en más es un error (al menos hasta que se le asigne otro valor). Veremos otros usos para del más adelante.

3. Tuplas y secuencias

Vimos que las listas y cadenas tienen propiedades en común, como el indizado y las operaciones de seccionado. Estas son dos ejemplos de datos de tipo *secuencia* (ver Tipos secuencia — list, tuple, range). Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la *tupla*.

Una tupla consiste de un número de valores separados por comas, por ejemplo:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Como puedes ver, en la salida las tuplas siempre se encierran entre paréntesis, para que las tuplas anidadas puedan interpretarse correctamente; pueden ingresarse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una expresión más grande). No es posible asignar a los ítems individuales de una tupla, pero sin embargo sí se puede crear tuplas que contengan objetos mutables, como las listas.

A pesar de que las tuplas puedan parecerse a las listas, frecuentemente se utilizan en distintas situaciones y para distintos propósitos. Las tuplas son immutable y normalmente contienen una secuencia heterogénea de elementos que son accedidos al desempaquetar (ver más adelante en esta sección) o indizar (o incluso acceder por atributo en el caso de las namedtuples). Las listas son mutable, y sus elementos son normalmente homogéneos y se acceden iterando a la lista.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis). Feo, pero efectivo. Por ejemplo:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

La declaración `t = 12345, 54321, 'hola!'` es un ejemplo de *empaquetado de tuplas*: los valores `12345`, `54321` y `'hola!'` se empaquetan juntos en una tupla. La operación inversa también es posible:

```
>>> x, y, z = t
```

Esto se llama, apropiadamente, *desempaquetado de secuencias*, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la secuencia. Notá que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaquetado de secuencias.

4. Conjuntos

Python también incluye un tipo de dato para *conjuntos*. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Las llaves o la función set() pueden usarse para crear conjuntos. Notá que para crear un conjunto vacío tenés que usar set(), no {}; esto último crea un diccionario vacío, una estructura de datos que discutiremos en la sección siguiente.

Una pequeña demostración:

>>>

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

De forma similar a las comprensiones de listas, está también soportada la comprensión de conjuntos:

>>>

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5. Diccionarios

Otro tipo de dato útil incluido en Python es el *diccionario* (ver Tipos Mapa — dict). Los diccionarios se encuentran a veces en otros lenguajes como «memorias asociativas» o «arreglos asociativos». A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con *claves*, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse como clave. No podés usar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como append() y extend().

Es mejor pensar en un diccionario como un conjunto de pares *clave:valor* con el requerimiento de que las claves sean únicas (dentro de un diccionario). Un par de llaves crean un diccionario vacío: {}. Colocar una lista de pares clave:valor separada por comas dentro de las llaves agrega, de inicio, pares clave:valor al diccionario; esta es, también, la forma en que los diccionarios se muestran en la salida.

Las operaciones principales sobre un diccionario son guardar un valor con una clave y extraer ese valor dada la clave. También es posible borrar un par clave:valor con del. Si usás una clave que ya está en uso para guardar un valor, el valor que estaba asociado con esa clave se pierde. Es un error extraer un valor usando una clave no existente.

Ejecutando list(d) en un diccionario retornará una lista con todas las claves usadas en el diccionario, en el orden de inserción (si deseas que esté ordenada simplemente usa sorted(d) en su lugar). Para comprobar si una clave está en el diccionario usa la palabra clave in.

Un pequeño ejemplo de uso de un diccionario:

>>>

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
```

```
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

El constructor dict() crea un diccionario directamente desde secuencias de pares clave-valor:

```
>>>
```

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Además, las comprensiones de diccionarios se pueden usar para crear diccionarios desde expresiones arbitrarias de clave y valor:

```
>>>
```

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Cuando las claves son cadenas simples, a veces resulta más fácil especificar los pares usando argumentos por palabra clave:

```
>>>
```

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

6. Técnicas de iteración

Cuando iteramos sobre diccionarios, se pueden obtener al mismo tiempo la clave y su valor correspondiente usando el método items().

```
>>>
```

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función enumerate().

```
>>>
```

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función zip().

```
>>>
```

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Para iterar sobre una secuencia en orden inverso, se especifica primero la secuencia al derecho y luego se llama a la función reversed().

```
>>>
```

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Para iterar sobre una secuencia ordenada, se utiliza la función sorted() la cual retorna una nueva lista ordenada dejando a la original intacta.

```
>>>
```

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

El uso de set() en una secuencia elimina los elementos duplicados. El uso de sorted() en combinación con set() sobre una secuencia es una forma idiomática de recorrer elementos únicos de la secuencia en orden ordenado.

```
>>>
```

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

A veces uno intenta cambiar una lista mientras la está iterando; sin embargo, a menudo es más simple y seguro crear una nueva lista:

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```