Tema 1: Conceptos básicos sobre estructuras de datos

Héctor Xavier Limón Riaño

April 3, 2024

Contents

1 Tipos de datos

Existen dos tipos de datos en los lenguajes de programación:

- Datos simples
- Datos estructurados

1.1 Datos simples

- También llamados primitivos
- No se pueden descomponer
- El lenguaje de programación los provee por defecto
- Están estrechamente asociados al procesador
- Cada tipo de dato suele tener un tamaño fijo en bytes
- Hay cuatro tipos tradicionalmente:
  - Números enteros
  - Números flotantes
  - Caracteres
  - Booleanos

1.1.1 Números enteros

- Es probablemente el tipo de dato más general
- Internamente, en muchos lenguajes (como C) los caracteres y booleanos son números enteros
- En algunos lenguajes como C hay varios tipos enteros que varían de a cuerdo a su tamaño en bytes:
  - `short` 2 bytes
  - `int` 4 bytes
  - `long` 8 bytes
- El tamaño en bytes establece el valor máximo que se puede guardar, por ejemplo en 4 bytes cabe hasta 2 elevado a la potencia 32

```
bytes = 4
bits = bytes * 8 # un byte son 8 bits
print(2 ** bits) # 2 porque se trabaja en binario
# En realidad la mitad puesto que hay que considerar negativos
print((2 ** bits)/2)
```

```
4294967296
2147483648.0
```

- El tamaño es importante para el procesador pues las instrucciones necesitan hacer referencia a posiciones de memoria y éstas consideran longitudes
- Si el número no cabe en el tipo de dato el valor se desborda (se toman los primeros N bytes de acuerdo al tamaño del tipo, lo que puede dar valores que parecen extraños)
- En Python, al ser un lenguaje dinámico, el tamaño se adapta, por lo que sólo hay un tipo `int`

```
print(type(5))
print(type(9999 ** 9999)) # un número muy grande, no cabe en long
```

```
<class 'int'>
<class 'int'>
```

- Cuidado: aunque no tengas que declarar el tipo o hacer referencia explícita a él, internamente toda expresión tiene un tipo en Python
- Con la función `type` puede verse el tipo interno

1.1.2 Números flotantes

- Se caracterizan por llevar un .
- El procesador los manipula de forma diferente a los enteros
- Se le suele aplicar truncado cuando hay muchos decimales (posiblemente infinitos)
- En Python sólo hay el tipo `float` dinámico
- En otros lenguajes estáticos com `C` también hay un tipo más grande `double`
- Cuando hay divisiones Python convierte a `float` aunque los operandos sean enteros
- Si hay interacciones aritméticas entre enteros y flotantes, Python convierte a flotante

```
print(type(4.0))
print(1/3) # truncado
print(type(4/2))
print(3 + 3.0)
```

```
<class 'float'>
0.3333333333333333
<class 'float'>
6.0
```

1.1.3 Caracteres

- En lenguajes como `C` los caracteres son en realidad números
- Su tamaño es variable de acuerdo a la codificación, aunque tradicionalmente en `C` es de 1 byte
- En Python no existe este tipo de dato, en caso que lo necesites se pueden crear cadenas de longitud 1

- Más adelante en el curso se hablará sobre codificación de cadenas para entender la relación entre valores numéricos y caracteres

```
print(type('s'))
print(type("s")) # lo mismo
```

```
<class 'str'>
<class 'str'>
```

1.1.4 Booleanos

- Representan valores lógicos verdadero o falso
- En lenguajes como `C` son internamente enteros, `0` para falos y `1` para verdadero
- En Python tenemos los valores literales `True` y `False`
- En Python, al necesitarse una expresión booleana, varias cosas diferentes pueden ser evaluadas como verdadero o falso:
  - 0 evalúa a falso, otro entero cualquiera a verdadero
  - Cadena vacía evalúa a falso, no vacía a verdadero
  - Lista vacía evalúa a falso (en general cosas vacías evalúan a falso)

```
print(type(True))

if 0: # evalúa a falso, cualquier otro int a verdadero
    print('entra al if')
else:
    print('No entra')

lista = [1, 2, 3]
if lista:
    print('la lista no está vacía')
else:
    print('la lista está vacía')
```

```
<class 'bool'>
No entra
la lista no está vacía
```

1.1.5 Conversión entre tipos simples

- Python provee funciones para convertir entre tipos simples (también las hay para tipos complejos)
- Esta conversión se puede hacer si hay compatibilidad

```
print(int('100')) # de cadena a entero
print(int(4.7)) # se trunca decimales
print(int(True)) # válido

print(float('4.33'))
print(float('500'))
print(float(False))
```

```
100
4
1
4.33
500.0
0.0
```

- En la conversión a booleano se consideran cosas vacías o valor de 0

```
print(bool('True'))
print(bool('eueuaeu'))
print(bool('False'))
print(bool(''))
print(bool(0))
print(bool(100))
print(bool(0.0))
print(bool([]))
print(bool([1, 2, 3]))
```

```
True
True
True
False
False
```

```
True
False
False
True
```

- Para convertir a cadena se usa la función `str`, cualquier cosa se puede convertir a cadena

```
print(str(44))
print(str(4.22))
print(str(True))
```

```
44
4.22
True
```

1.2 Datos complejos

- También llamados estructurados
- Son tipos de datos que se componen de otros tipos (primitivos u otros datos estructurados)
- A estos tipos también se les suele llamar "Estructuras de datos"
- Las estructuras de datos son centrales en la programación
- Para resolver problemas de programación, la mayoría del tiempo hay que pensar en qué estructura(s) de datos es la más adecuada para resolver el problema
- Por tanto, entender y saber usar estructuras de datos es una habilidad esencial para todo programador
- Las estructuras de datos permiten abstraer los problemas, ayudando al manejo de la complejidad
- Existen estructuras de datos lineales, no lineales y jerárquicas (las cuales se verán más a fondo en todo el curso)

1.2.1 Lineales

- Cada elemento (excepto el primero y último) tienen un (y sólo uno) elemento predecesor y un elemento sucesor
- También se les llama indexados, puesto que se pueden recuperar sus elemento a través de un índice
- Tipos principales de estructuras lineales en Python:
  - Listas
    - Similares a los arreglos de otros lenguajes
    - Son dinámicas (más sobre esto después)
    - Son mutables (más sobre esto después)
  - Tuplas
    - Similares a las listas, pero son no mutables y estáticas
  - Cadenas
    - Internamente son arreglos de caracteres
    - Son no mutables y estáticas

```
l = [1, 2, 3]
print(type(l))
print(l[0])

t = (1, 2, 3)
print(type(t))
print(t[0])

s = "hola"
print(type(s))
print(s[0])
```

```
<class 'list'>
1
<class 'tuple'>
1
<class 'str'>
h
```

1.2.2 No lineales

- No tienen un orden
- No son estructuras de datos pensadas para ser recorridas (aunque se puede hacer)
- En Python se tienen estos tipos principales:
  - Diccionarios
    - En otros lenguajes son conocidos como Hash maps o Hash tables
    - Cada elemento consta de dos partes: una llave y un valor asociado
    - Permiten la recuperación aleatoria de elementos (se puede acceder directamente a posiciones de memoria sin necesidad de visitar otras posiciones antes)
    - La llave es como el índice, pero puede ser de otros tipos además de enteros (más sobre esto en el tema del curso de diccionarios)
  - Conjuntos (sets)
    - Se pueden entender en el sentido matemático
    - Son colecciones de elementos no ordenados donde no importan las repeticiones
    - Muy útiles cuando se necesitan operaciones sobre conjuntos como unión, diferencia e intersección
  - Objetos
    - Son tipos de datos especiales de la programación orientada a objetos (POO)
    - Sirven para encapsular varios valores a través de atributos
    - También pueden tener comportamiento a través de métodos
    - Para crear objetos antes debes definir clases
    - Se verán más adelante en el curso

```
d = {'a': 1, 'b': 2, 'c': 3}
print(type(d))
print(d['a'])

s = {'hola', 'mundo', 'hola', 'mundial'}
```

```
print(type(s))
print(s)
```

```
<class 'dict'>
1
<class 'set'>
{'mundo', 'hola', 'mundial'}
```

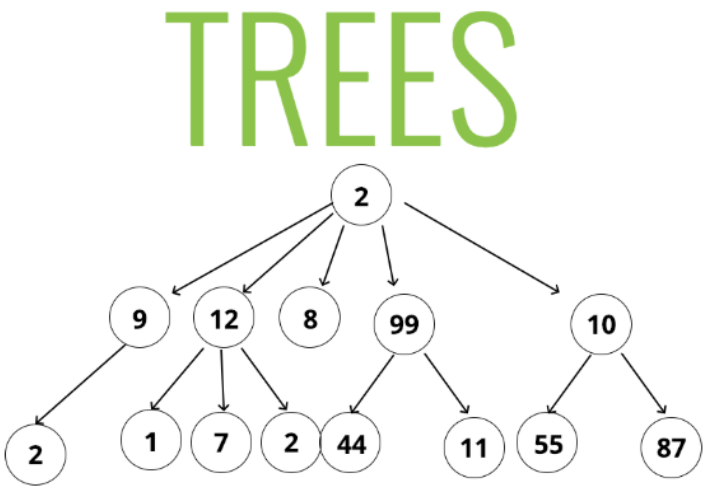
```
class Persona():
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

pepe = Persona('Pepe', 20)
print(pepe.edad)
pepe.pais = 'méxico'
juan = Persona('Juan', 15)
print(juan.nombre)
```

```
20
Juan
```

1.2.3 Jerárquicas

- Los elementos tienen un orden jerárquico, esto es pueden tener ancestros y descendientes
- El tipo principal son los árboles
- Los árboles se componen de nodos
- Python no cuenta por defecto con este tipo de datos, pero es fácil crearlo manualmente a partir de objetos



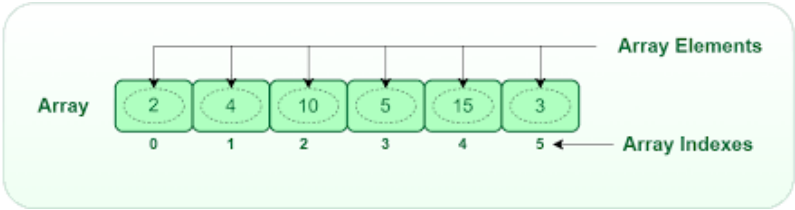
2 Representación estática y dinámica

- Los datos estructurados pueden ser estáticos o dinámicos
- Cuando son estáticos se debe definir su tamaño puesto que en memoria se reserva el tamaño necesario
- El tamano de las estructuras estaticas no puede ser cambiado en tiempo de ejecucion (cuando el programa ya esta corriendo)
- En cambio, las estructuras dinamicas pueden crearse y destruirse en tiempo de ejecucion, variando su tamano a conveniencia
- La ventaja de las estructuras estaticas es que son mas eficientes puesto que suelen utilizar espacios de memoria contiguos (caso de los arreglos), lo que permite el acceso aleatorio a elementos
- En cambio las estructuras dinámicas suelen estar dispersas en memoria, lo que hace más costoso el acceso a elementos
- En Python la mayoría de estructuras de datos son dinámicas, aunque se intenta mantener espacios contiguos en memoria

- Es común que en lenguajes interpretados se tengan estructuras dinámicas por defecto, mientras que en los lenguajes compilados estructuras estáticas
- En lenguajes compilados también se pueden definir estructuras dinámicas usando apuntadores (de forma directa o indirecta en caso de que el lenguaje no permita manipular manualmente apuntadores)
- Los apuntadores son variables especiales (muy importantes en C y C++) que almacenan direcciones de memoria
- Python no permite manejar apuntadores pero existen de forma interna

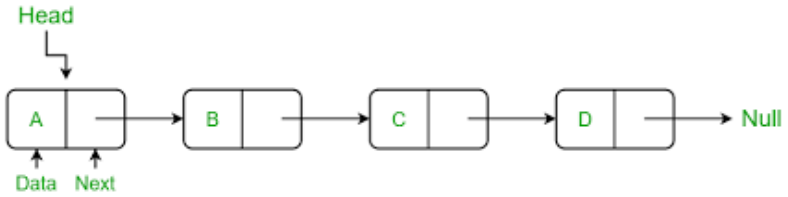
2.1 Ejemplos de estructuras estáticas

- Arreglos
- Cadenas
- Hashmaps (hasta cierto punto, dependiendo de la implementación)



2.2 Ejemplos de estructuras dinámicas

- Listas ligadas
- Árboles
- Objetos



3 Mutabilidad y no mutabilidad

- Es una propiedad de las estructuras de datos que se refiere a si los valores de los elementos de la estructura pueden cambiar (mutable) o no (no mutable)
- Esencialmente si la estructura es no mutable, la memoria asociada está "protegida" contra escrituras, sólo se pueden hacer lecturas
- Si una estructura es no mutable, también es estática (ya no puedes agregar ni quitar elementos)
- Una estructura mutable puede ser tanto estática como dinámica
- Por ejemplo, un arreglo en `C` es estático pero mutable, ya que puedes cambiar los valores de las casillas de memoria
- La no mutabilidad es muy importante en el paradigma de programación funcional (visto en un curso posterior)
- La no mutabilidad también es muy útil en programación concurrente y paralela (vista en otros cursos)
- Por otro lado, la mutabilidad es peligrosa y es una de las mayores fuentes de bugs en los programas, dado que propician la aparición de efectos colaterales (visto en un tema del segundo parcial) y condiciones de carrera (visto en cursos más avanzados)

3.1 Ejemplos de estructuras mutables

- Listas
- Diccionarios
- Objetos
- Árboles
- Sets (conjuntos)

```
lista = [1, 2, 3]
x = lista[0] # lectura, OK
lista[1] = 22 # escritura, OK
```

3.2 Ejemplos de estructuras no mutables

- Tuplas
- Cadenas

```
  tupla = (1, 2, 3)
  x = tupla[0] # lectura, OK
  #tupla[1] = 22 # escritura, error no es mutable

  # mismo caso con cadenas
  cadena = 'hola'
  x = cadena[0] # lectura, OK
#  cadena[1] = 'a' # escritura, error

  cadena1 = 'hola'
  cadena2 = ' mundo'
  cadena3 = cadena1 + cadena2 # no hay problema, se genera nueva memoria
```

```
def multi_valor():
    return 1, True
v, o = multi_valor()
print(v)
print(o)
print(type(multi_valor()))
```

```
1
True
<class 'tuple'>
```