Fundamentos de Algoritmos

Profs. Agustín Gravano y Hernán Czemerinski

Primer Semestre de 2020

Clase 4: Tipos de datos

Repaso: Tipo de datos Lista

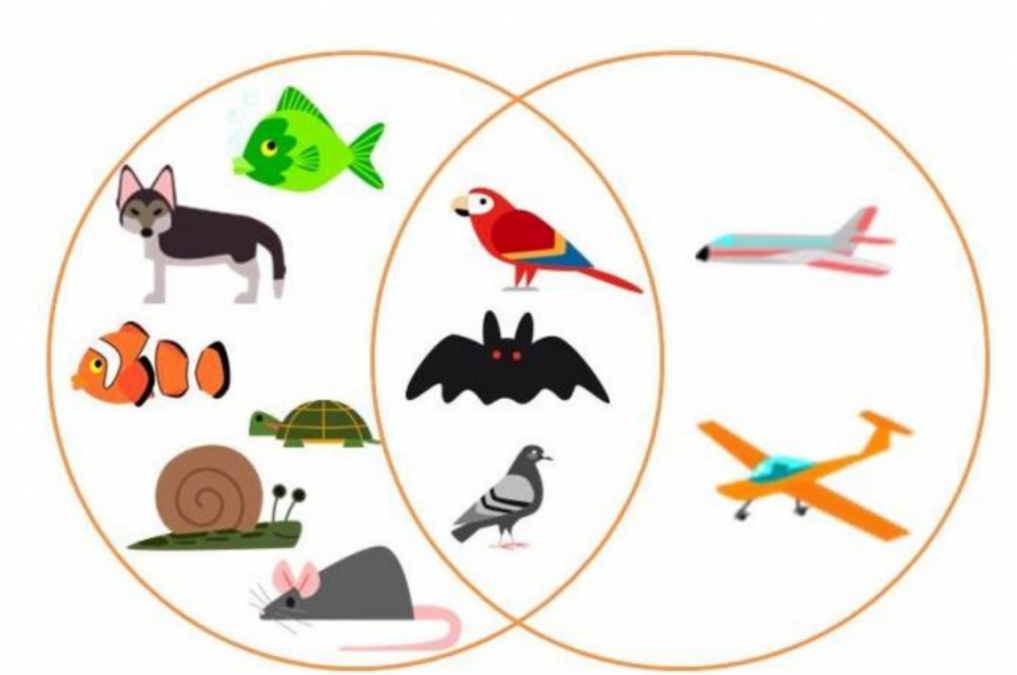
```
a = [11, 12, 13, 14]
len(a)           # cantidad de elementos
a.append(15)     # agregar al final
a[1]             # leer una posición
a[1] = 9999      # sobreescribir una posición
9999 in a        # pertenencia

a = list(range(11, 15))
[x*x for x in a if x%2==0]          # [144,196]

def elevarPares(x):
  if x%2==0:
    return x * x
  else:
    return x

[elevarPares(x) for x in a]         # [11,144,13,196]
```

Tipo de datos Conjunto

Hasta ahora trabajamos con `bool`, `int`, `float`, `string`, `list`.

Los tipos de datos son útiles para representar datos, y también para escribir algoritmos más simples.

¿Cómo hacemos para trabajar con conjuntos?



Ejemplo

Queremos representar conjuntos de especies animales.
Podemos hacerlo con listas, por ejemplo:

```
felinos = ["león", "gato", "tigre", "guepardo", "puma"]
cánidos = ["lobo", "coyote", "chacal", "dingo", "zorro"]
cetáceos = ["delfín", "ballena", "orca", "cachalote"]
```

¿Las listas son una buena forma de representar conjuntos?

- Orden: `["lobo", "coyote"] == ["coyote", "lobo"]` ?
- Repetidos: `["delfín", "delfín"]` ?

Mejor usar un tipo Conjunto que nos evite estos problemas.

Tipo Conjunto (`set` en Python)

Operaciones:

- `c = set()`: Crea un conjunto vacío.
- `c.add(x)`: Agrega el elemento x al conjunto c.
- `x in c`: Dice si el elemento x está en c.
- `c.remove(x)`: Elimina el elemento x de c.
  Precondición: `x in c` (caso contrario, tira error)
- `len(c)`: Devuelve la cantidad de elementos de c.
- `c1==c2`: Dice si los dos conjuntos son iguales.
- `c1.union(c2)`: Devuelve un nuevo conjunto con c1 $\cup$ c2.
- `c1.intersection(c2)`: Devuelve un nuevo conj. con c1 $\cap$ c2.
- `c1.difference(c2)`: Devuelve un nuevo conj. con c1 $\setminus$ c2.
- `list(c)`: Devuelve una lista con los elementos de c.

Ejemplo de conjuntos

```
felinos1 = set()
felinos1.add("león")
felinos1.add("gato")

felinos2 = set()
felinos2.add("gato")
felinos2.add("león")
felinos2.add("león")

felinos1 == felinos2        # Devuelve True

felinos1.add("tigre")
felinos2.add("puma")
felinos1.intersection(felinos2)
    # Devuelve {'gato', ' león'}
felinos1.union(felinos2)
    # Devuelve {'puma', 'gato', ' león', 'tigre'}
```

Ejercicio. (1) Escribir una función que dada una palabra obtenga el conjunto de letras que la componen. (2) Escribir un programa que dadas dos palabras, liste las letras que tienen en común (sin repetidas).

Tipo Diccionario (`dict`)

Asocia un valor (de algún tipo) a una clave (de algún tipo).

- Ej: palabra/definición, país/población, depto/inquilino.

Operaciones:

- `d = dict()`: Crea un diccionario vacío.
- `d[c] = v`: Asocia el valor v a la clave c en el diccionario d.
- `c in d`: Dice si la clave c está definida en d.
- `d[c]`: Devuelve el valor asociado a la clave c en el dicc. d.
  Precondición: `c in d` (caso contrario, tira error)
- `d.keys()`: Devuelve las claves del diccionario.

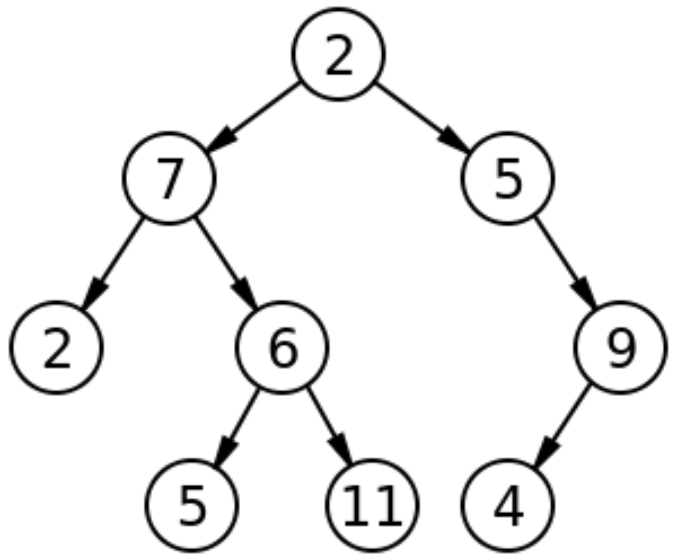
Ejemplo de uso de diccionarios

```
peso_atómico = dict()
peso_atómico['H']  = 1.00797
peso_atómico['He'] = 4.0026
peso_atómico['C']  = 12.0111
peso_atómico['U']  = 238.02891
'Ag' in peso_atómico  # devuelve False
peso_atómico['Ag'] = 107.8683
'Ag' in peso_atómico   # devuelve True
peso_atómico.keys()
```

Ejercicio: Escribir una función que, dado un string que representa un número natural, devuelva un string enumerando sus dígitos. P.ej., para "4137", debe devolver el string "cuatro uno tres siete".

Otros tipos de datos: árbol

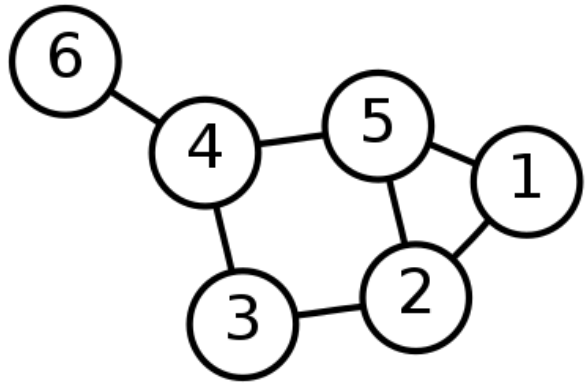
- Conjunto de nodos.
- El nodo de arriba de todo es la raíz del árbol.
- Cada nodo (excepto la raíz) tiene un padre y 0 o más hijos.
- No deben formarse ciclos.



Aplicaciones: organigramas, parsing sintáctico, machine learning (decision tree, random forest), algoritmos y estructuras de datos, ...

Otros tipos de datos: Grafo

- Conjunto de nodos conectados por aristas.
- Las aristas pueden ser dirigidas y tener pesos.



Aplicaciones: redes sociales, mapas (tráfico, logística), redes de computadoras, Google PageRank, problemas de matching (asignación de recursos, publicidad online), machine learning (redes bayesianas, procesos de Markov), biología, física, química, ...

Otros tipos de datos: Pila *(Stack)*

Parecido a una lista, pero solo se puede:

- Preguntar si la pila está vacía
- Agregar un elemento arriba: push
- Quitar el elemento de arriba: pop
- Acceder al elemento de arriba: top

Aplicaciones: funcionalidad de *undo*, llamados a funciones, evaluación de expresiones, parsing sintáctico, ... (es muy importante en el diseño e implementación de lenguajes de programación).

Problema: Biblioteca

Queremos programar una biblioteca de libros.
De cada libro nos interesa guardar:

- Título
- Autor
- Año de publicación

¿Cómo representamos los datos de los libros?

- Títulos: lista de strings.
- Autores: lista de strings.
- Años de publicación: lista de enteros.

Tales que la *i*-ésima posición de los 3 listas correspondan al mismo libro. Esta representación *funciona* (cumple el objetivo), pero tiene serios problemas...

Problema: Biblioteca

Mucho mejor sería contar con un tipo Libro, que **encapsule** todos los datos relevantes de un libro para nuestra biblioteca.

Operaciones del Tipo Libro

- `Libro(titulo, autor, anio)`: Crea un nuevo elemento de tipo Libro con el título, autor y año especificados.
- `l.titulo()`: Devuelve el título del libro *l*.
- `l.autor()`: Devuelve el autor del libro *l*.
- `l.anio()`: Devuelve el año de publicación del libro *l*.

Así, podemos representar la biblioteca con un **conjunto de Libros**.

Como usuarios de un tipo, no nos importa cómo está implementado.

Tipo de Datos *ad hoc*: uso. Ejercicio

Completar el código, para imprimir los libros de una bibliteca entre dos años dados. La biblioteca se representa como un conjunto de libros.

```
1  from Libro import *   # Importa el tipo Libro de Libro.py
2
3  def cargar_biblioteca():
4    biblioteca = set()
5    archivo = open("biblioteca.csv")
6    for linea in archivo:
7      [completar]
8    archivo.close()
9    return biblioteca
10
11 def libros_entre_anios(biblioteca, anio1, anio2):
12   libros = set()
13   for libro in biblioteca:
14     [completar]
15   return libros
16
17 # Ejemplo: imprime los libros entre 1800 y 1899
18 biblioteca = cargar_biblioteca()
19 for libro in libros_entre_anios(biblioteca, 1800, 1899):
20     print(libro)
```

Para resolverlo, descargar los archivos `biblioteca.py`, `biblioteca.csv` y `Libro.py` (¡no ver su contenido para no marearse!).

Modularidad y encapsulamiento

Al programar, definimos funciones (ej: esPrimo(n)) para generar código más simple y claro (código *modular*). **Encapsulamos** un algoritmo para poder reusarlo muchas veces.

Generalizamos este concepto, y encapsulamos **datos** (ej: libro) y sus **operaciones** (ej: l.autor()) en Tipos de Datos.

Para **usar** un tipo, el programador solo necesita conocer el nombre del tipo y una descripción de sus operaciones.

Un tipo puede estar implementado de muchas formas. Esto debe ser **transparente** para el usuario.

Tipos de datos: Partes

- Parte pública: Disponible para el usuario del tipo.
  - Nombre del tipo (ej: `Libro`)
  - Operaciones del tipo (ej: `l.titulo()`)

- Parte privada: solo accesible desde dentro del tipo.
  ¡El usuario externo **nunca** debe ver ni meterse con esto!

Tipos de datos: Implementación

Veamos ahora cómo podemos definir un tipo nuevo "Libro".

Primero, hay que definir la estructura de representación del tipo: variables que almacenan los datos de cada elemento.

De cada libro nos interesa guardar **título**, **autor** y **año de publicación**.

Entonces, el tipo Libro podría tener esta estructura de representación:

- título: string
- autor: string
- año: entero

Tipos de datos: Implementación

Segundo, definimos el Tipo Libro y una operación para crear nuevos elementos del tipo.

```
class Libro:

  def __init__(self, tit, aut, anio):
    self._titulo = tit
    self._autor = aut
    self._anio = anio
```

En Python:

- **class** es una palabra reservada para definir un nuevo tipo;
- **__init__** es la función que se invoca cuando se crea un elemento;
- **self** es una palabra reservada para acceder a las variables privadas; y
- la convención es usar **guión bajo** (_) para las variables privadas.

Tipos de datos: Implementación

Segundo, definimos el Tipo Libro y una operación para crear nuevos elementos del tipo.

```
class Libro:

  def __init__(self, tit, aut, anio):
    self._titulo = tit
    self._autor = aut
    self._anio = anio
```

Ahora, ya podemos podemos crear elementos de tipo Libro:

```
el_aleph = Libro("El Aleph", "Borges", 1949)
```

Tipos de datos: Implementación

también definimos operaciones para usar los elementos de tipo Libro. P.ej., podemos querer **inspeccionar** sus datos.

```
def titulo(self):
  return self._titulo

def autor(self):
  return self._autor

def anio(self):
  return self._anio
```

```
print(el_aleph.titulo())    # Imprime "El Aleph"
print(el_aleph.autor())     # Imprime "Borges"
print(el_aleph.anio())      # Imprime "1949"
```

Tipos de datos: Implementación

Por último, definimos algunas operaciones muy prácticas para tipos en general: cómo imprimir y comparar Libros.[1]

```
def __repr__(self):
  return "<" + self._titulo + ":" + self._autor + ":" +
      self._anio + ">"

def __eq__(self, other):
  return self._titulo == other._titulo

def __lt__(self, other):
  return self._titulo < other._titulo
```

```
print(el_aleph)    # Imprime "<El Aleph:Borges:1949>"
ulises = Libro("Ulises", "Joyce", 1922)
el_aleph == ulises       # devuelve False
el_aleph < ulises        # devuelve True
```

[1] Ver https://docs.python.org/3/reference/datamodel.html

Tipo de datos Libro: Implementación

```
1  # Implementación en Python del Tipo Libro.
2  class Libro:
3    def __init__(self, tit, aut, anio):
4      self._titulo = tit
5      self._autor = aut
6      self._anio = anio
7
8      def titulo(self):
9        return self._titulo
10     def autor(self):
11       return self._autor
12     def anio(self):
13       return self._anio
14
15     def __repr__(self):
16       return "<" + self._titulo + ":" + self._autor + ":"
             + self._anio + ">"
17     def __eq__(self, other):
18       return self._titulo == other._titulo
19     def __lt__(self, other):
20       return self._titulo < other._titulo
21   (...)
```

Ejercicio

Completar el tipo Agenda que se muestra a continuación.

```
1  class Agenda:
2    # Crea una Agenda a partir de "contactos.csv",
3    # que asocia nombres con mails.
4    def __init__(self):
5      archivo = open("contactos.csv")
6      [completar]
7      archivo.close()
8
9    # Devuelve el mail del contacto 'nombre'.
10   def mail(self, nombre):
11     [completar]
12
13   # Devuelve el string que representa una agenda,
14   # mostrando un contacto por línea.
15   # Formato de cada línea: [nombre]: [email]
16   def __str__(self):
17     [completar]
```

Ejercicio (cont.)

Ejemplo de código cliente y resultado esperado:

```
1  agenda = Agenda()
2  for contacto in ["Alicia", "Juan"]:
3    print(contacto + ":", agenda.mail(contacto))
4
5  print()
6  print(agenda)
```

```
1  Alicia:  alicia@yahoo.com
2  Juan:  juan@gmail.com
3
4  Agustín:  agustin@utdt.edu
5  Alicia:  alicia@yahoo.com
6  Analía:  analia@yahoo.com.ar
7  Juan:  juan@gmail.com
8  Luz:  luz@gmail.com
9  Martina:  martina@hotmail.com
```

Repaso de la clase de hoy

- Conjunto, diccionario, pila, árbol, grafo.
- Tipos de datos. Parte pública y parte privada.
- Implementación de un tipo de datos: estructura de representación.

Con lo visto, ya pueden resolver la sección 5 de la guía de ejercicios.