Hoja de Referencias. 2022-23
Tecnología de la Programación.

A. Funciones de Orden Superior

A.1. Funciones Lambda o Anónimas

- El λ-cálculo trata las funciones de forma anónima: La escritura anónima de square_sum$(x, y) = x^2 + y^2$ es $(x, y) \mapsto x^2 + y^2$.
- En Python estas expresiones se llaman **Funciones Lambda**.
- Tiene la siguiente expresión: **lambda** argumentos: expresión
  Puede tener cualquier número de argumentos pero tiene una sola expresión.
- Ejemplos:

```
>>> el_doble = lambda x: x * 2
>>> print(el_doble(10))
20
>>> suma = lambda x, y: x + y
>>> print(suma(4, 5))
9
>>> acotado = lambda x: 4 <= x <= 8
>>> print(acotado(5))
True
```

- Son útiles junto con funciones como **map**(), **filter**(), **reduce**(), ... para trabajar con colecciones.

A.2. Funciones de Orden Superior

- Son **funciones de orden superior** (operadores o funcionales), aquellas que cumplen al menos una de las siguientes condiciones:
  - Considera una o más **funciones** como parámetros de **entrada**
  - **Devolver** una **función** como "valor" de retorno
- Ejemplos:

```
>>> def doble(a): # Una función normal.
...     return 2*a
...
>>> def triple(f, x): # Es de orden superior
...     print(3*f(x))
...
>>> def g(): # Es de orden superior
...     return doble
...
>>> triple(doble, 10) # La entrada tiene una función
60
>>> print ( g()(5) )    # g() retorna una función
10
```

A.3. Funciones map(), filter() de Orden Superior

- **map(función, secuencia)**. Aplica a cada elemento la función.
- **filter(función, secuencia)**. Filtra los elementos de la secuencia.

```
>>> print(list(map(lambda x: x**2, range(5))))
[0, 1, 4, 9, 16]
>>> print(list(filter(lambda x: x>2, range(5))))
[3, 4]
```

A.4. Funciones Clausura

- Se llama **clausura** a un **registro** que contiene a: (1) una **función** (2) junto con al **ámbito** en el que esa función se evalúa/ejecuta.
- En una clausura:
  - **la función pueda acceder** a las variables de su ámbito, aún cuando se invoque fuera de él.
  - las variables del ámbito **son solo accesibles** a través de la función.
- **Una forma** de implementar clausuras es **con funciones anidadas**.
  En tiempo de ejecución, cuando se ejecuta la función externa, se forma una clausura, que consta del código de la función interna y las referencias a todas las variables de la función externa que son requeridas por la clausura.
- Se llama **función clausura** a aquella que retorna una función anidada en ella.
- Ejemplo de uso: proteger a una variable "local" que debe actualizarse para alguna acción.

```
>>> def count():
...     x = 0
...     def nested():
...         nonlocal x
...         x = x+1
...         return x
...     return nested
...
>>> contador = count()
>>> contador(); contador()
1
2
```

A.5. Decoradores

- Los **decoradores son funciones matemáticas** que responden al esquema $f_a(f_b) \to f_c$.
  - Un decorador f_a es un función que tiene como argumento una función f_b y retorna otra función f_c. (Son funciones de orden superior)
- **Las funciones clausura definen decoradores** construyendo funciones de orden superior: tanto en la entrada como en la salida.
- Construido el decorador f_a, la decoración de una función de interés consistirá en **anteponer la expresión @f_a** al nombre de la función a decorar.
- En el caso de que la función a decorar deba recibir argumentos deberá usar los parámetros **args** y **kwargs**.
- El **esquema general** de un decorador es el siguiente:

```
def func_a(func_b):
    def func_c(*args, **kwargs): # Parámetros no  requeridos
        # Lo que hacer antes de invocar a la función a decorar
        func_b(*args, **kwargs) # Invocar a la función decorada
        # Lo que hacer después de invocar a la función a decorar
    return func_c

@func_a   # Decorador
def func():  # Función decorada
    pass
```

Ejemplo: ¿Cuánto tarda en ejecutarse una suma?

```
>>> def decorador(function):
...     def wrapper(*args, **kwargs):
...         import time
...         start = time.time()
...         result = function(*args, **kwargs)
...         total = time.time() - start
...         print(total, 'sg.' )
...         return result
...     return wrapper
...
>>> # Se puede hacer así una decoración permanente
>>> suma_decorada = decorador(suma)
>>> print(suma_decorada(10, 20))
6.9141387939453125e-06 sg.
30
>>> # Y se sigue teniendo la suma sin decorar
>>> print(suma(10, 20))
30
>>> # Alternativa como decoración permanente
>>> @decorador
... def suma(a, b):
...     return a + b
...
>>> print(suma(10, 20)) # Muestra el tiempo y los segundos.
6.198883056640625e-06 sg.
30
```

A.6. Generadores

- Un **generador produce** un dato en tiempo de ejecución, sólo cuando se solicita.
- En **Python** son "funciones" que **en lugar de** usar **return** utilizan **yield**.
- En **Python** toda función que utilizan **yield** como instrucción:
  - la función devuelve el control a quién la llamó
  - la función es pausada y el estado (valor de las variables) es guardado
  - en la siguiente invocación continúan desde el punto desde donde estaba
- **La invocación** a la función con **yield**, crea un *generator objet*.

  Un *generator objet* es un **iterador** (**Iterable** que se puede recorrer con **next()**)

```
>>> def generador():
...     cont = 0
...     while (cont < 3):
...         yield cont
...         cont = cont + 1
...
>>> iter = generador()  # Crea un iterador
>>> print(next(iter), next(iter), next(iter))
0 1 2
```

B. Iterables

B.1. Iterables

- En **Python** algunos contenedores se llaman **iterables**.
- Son iterables los rangos, listas, tuplas, conjuntos y diccionarios.
- Los iterables responden a la abstracción de iteración (Al usar el **for** se crean un iterador sobre el iterable).

B.2. Iteradores

- Un iterador es un objeto que es capaz de recorrer un iterable.
- Se construye de forma explícita mediante `iter(iterable)` o de forma implícita (p.e. por un **for**).

Se pueden construir iteradores propios siguiendo estos pasos:

```
class MiLista:
  def __init__(self, lista):
    self._list = lista
  def __iter__(self): #1. Retornar un objeto iterador
    return _IteradorDeMiLista(self._list)

class _IteradorDeMiLista:
  def __init__(self, refLista):
    self._listRef = refLista  # 2. Referencia al iterable
    self._pos = 0
  def __iter__(self): # 3. Se referencia a sí mismo.
    return self
  def __next__(self): # 4. Sabe como avanzar
    if self._pos < len(self._listRef):
      element = self._listRef[self._pos]
      self._pos = self._pos + 2
      return element
    else:
      raise StopIteration # 5. Lanza excepción al final.
```

B.3. Construcción por Comprehensión

$$S = \{ \underbrace{2 \cdot x}_{\text{Expresión salida}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{Conj. entrada}} , \underbrace{x^2 > 3}_{\text{Predicado}} \}$$

B.3.1. Para listas, tuplas y conjuntos

```
salida = <expresion(x) for x in secuencia [if condicion]>
```

con < >: [ ], para **listas**; ( ), para **tuplas**; { }, para **conjuntos**.

- `expresion(x)` es cualquier expresión sobre `x`. Ejemplos:
  - No alterar el valor de `x`; es decir, `expresion(x)=x`.
  - Aplicar alguna operación o función. P.e. `expresion(x)=x**2`.
  - Usar algún método. P.e. `expresion(x)=x.upper()`.
  - Usar expresiones ternarias. P.e. `expresion(x)=x if x%2==0 else 1000`.
- `for x in secuencia` se puede sustituir por cualquier conjunto de sentencias que definan el valor de `x`. Por ejemplo, un bucle anidado donde aparezca `x`.

```
>>> [2*x for v in range(2) for x in range(5) if x*x > 3]
[4, 6, 8, 4, 6, 8]
```

B.3.2. Para diccionarios

```
salida = {k: v for key, val in iterable [if condicion]}
```

En este caso el iterable tiene que ser un **diccionario** o el obtenido por un **enumerate()**. La función **enumerate(iterable, start=0)** añade un índice al iterable.

C. Programación Orientada a Objetos

C.1. Clases

- Una clase es una estructura de datos que encapsula variables (atributos) y funciones (métodos).
- Una clase no es un TDA, pero un TDA se puede implementar usando clases.
- Una clase es un plantilla (abstracción) de un conjunto de datos (instancias u objetos).
- Todos los atributos deben ser privados o protegidos. Para acceder a ellos se usarán métodos o propiedades.
- Los métodos pueden ser de creación, consulta o modificación.
- Solo los métodos que puedan ser usados por el usuario serán públicos.
- La recomendación es:
  - anteponer doble guión bajo para el modificador privado: **__miembro**
  - anteponer un guión bajo para el modificador interno: **_miembro**
  - no anteponer guiones para el modificador público: **miembro**

C.2. Tipos de Variables

- **De clase o estáticas**. Evalúan atributos propios de clase.
  - Se definen dentro de un clase pero fuera de cualquier método.
  - Acceso: **Clase.variable**
- **De instancia** (u objeto). Definen el estado para un objeto.
  - Se definen en **def __init__(self)**
  - Acceso: **objeto.varible**; pero debe usar un método Get/Set.
- **Locales**. Las auxiliares propias de un método o función.
- **Globales**. Las que son accesibles por cualquier clase o función.

C.3. Tipos de Métodos

- **De instancia/objeto**.
  - Los asociados a un objeto.
  - Invocación: **objeto.metodo(parámetros)**
- **De clase**.
  - Los asociados a una clase.
  - Tienen el decorador **@classmethod**.
  - Invocación: **Clase.metodo(parámetros)**
  - Ejemplo de uso: **Funciones Factoría**.
- **Estáticos**.
  - No están asociados ni a una clase ni a un objeto.
  - Tienen el decorador **@staticmethod**.
  - Invocación: **Clase.metodo(parámetros)**

C.4. Propiedades

- La interface pública relacionada con el estado se llama Getter/Setter
- Por defecto, los atributos tienen métodos Getter pero no Setter.
- La interface Getter/Setter **tiene que ser usada** por el resto de los métodos, aún cuando tengan acceso a los atributos.
- El *Pythonic way* usa acceso directo **con la notación punto** sobre los atributos.
- La siguiente forma mantiene el atributo totalmente privado

```
x = property(fget=None, fset=None, doc=None)
```

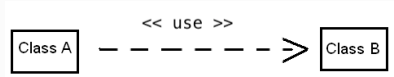
Cambia entonces **None** de `fget` y `fset` por los métodos de instancia que convengan.

```
class Persona:
    def __init__(self, nombre: str):
        self.__nombre = nombre
    def get_n(self):
        return self.__nombre
    def set_n(self, nombre: str):
        self.__nombre = nombre
    nombre = property(fget=get_n, fset=set_n)

p = Persona('L. Daniel')
p.nombre = 'Hernández'
print(p.nombre)
```

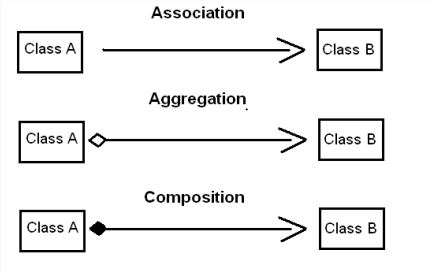
- Otra forma es usando decoradores:
  - Un método **get()** se llamará igual que el atributo att y se decora con **@property**.
  - El método **set()** se le llama igual que al atributo att y se decora con **@att.setter**.
  - Tendremos dos métodos **att(self)** y **att(self, x)** con el mismo nombre y distinto número de parámetros.
  - **att(self)** se invoca escribiendo **objeto.att**.
  - **att(self, x)** se invoca escribiendo **objeto.att = x**.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre # Método set será decorado
    def __str__(self):
        return f'nombre: {self.__nombre}'
    @property                   # La decoración permite
    def nombre(self):           # la expresión
        return self.__nombre    # p.nombre
    @nombre.setter              # La decoración permite
    def nombre(self, nombre):   # la expresión
        self.__nombre = nombre # p.nombre = nombre

p = Persona('L. Daniel')
p.nombre = 'Hernández'  # Por @property
print(f'{p} {p.nombre}') # Por @nombre.setter
```

C.5. Asociación




```
>>> class B:
...     def use(self):
...         print('me utilizan')
...
>>> class A:
...     def metodo(self, b):
...         b.use()
...
>>> a = A(); b = B(); a.metodo(b)
me utilizan
```



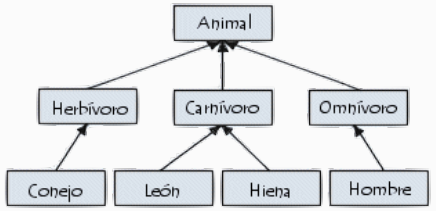

```
>>> class B:
...     ...
...
>>> class A:
...     def __init__(self, b):
...         # A depende de B
...         self.__b = b
...
```

C.6. Delegación

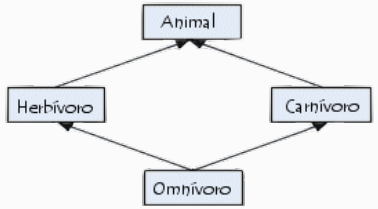
- Cuando una clase contiene una o más instancias de otra clases como atributos, entonces la clase **delega** su funcionalidad a los atributos.
- Cuando un objeto reciba una petición, **delegará** la ejecución del método a sus atributos.
- Ejemplo: trasladar un rectángulo en el plano delega en trasladar su punto origen en el plano.

C.7. Herencia

- **Herencia simple**



- **Herencia múltiple**



- Una subclase debe llamar al **__init__()** del padre con **super()**.
- **class.mro()** retorna el valor del atributo **class.__mro__**, que retorna el orden de las clases padres relativo a la clase **class**.
- Debes saber cómo resolver el problema del diamante usando empaquetamiento.

C.8. Clases Abstractas

- Una **clase abstracta representa** a objetos para los que **conocemos algunas de las variables** que definen su estado y **conocemos algunas de sus funcionalidades pero no sabemos establecerlas** sin concretar el objeto.
- Formalmente, una **clase abstracta** es la que tiene (al menos) un método abstracto (aquel que tiene declaración pero no está definido).
- Una clase abstracta es una clase que (1) **no se puede instanciar** (no representan algo específico), (2) **se usa únicamente para definir subclases**
- Una clase abstracta puede tener constructores; pero no se debe poder construir objetos.

```
from abc import ABC, abstractmethod
class MyClass(ABC):
  @abstractmethod
  def metodo_abstracto(self):
    pass
```

- En la práctica, basta poner `@abstractmethod` en cualquier método, aunque esté bien definido.

C.9. Interfaces

- Una interface (abstracta) es un clase donde todos los métodos de instancia son abstractos y no tienen atributos de objeto.
- Las informales se construyen siguiendo estos pasos:
  1. Se registra la interface, **Interface**.
  2. En un clase, **Clase**, se implementan los métodos registrados en la interface **Interface**.
- Las formales se construyen siguiendo estos pasos:
  1. Se registra la interface, **Interface**, con métodos abstractos.
  2. La clase, **Clase**, **hereda Interface** e implementa los métodos abstractos.

C.10. Registro de Interfaces

```
from abc import ABC, abstractmethod

class Interface(ABC):
    @classmethod
    def __subclasshook__(cls, subclass):
        if cls is Interface:
            return (hasattr(subclass, 'metodo1') and
                    callable(subclass.metodo1) and
                    hasattr(subclass, 'metodo2') and
                    callable(subclass.metodo2))
                    # añadir los métodos necesarios
        return NotImplemented

    """
    # Descomentar si se quiere una interface formal

    @abstractmethod
    def metodo1(self):
        pass
    @abstractmethod
    def metodo2(self):
        pass

    # añadir los métodos necesarios
    """
```

D. Excepciones

D.1. Capturar Excepciones

```
try:
  # Código que se desea ejecutar
except xxxError [as nombre]:
  # Código a ejecutar si se produce un error en try
  # Aquí controla la situación de error xxxError.
  # Puede lanzar otro error.
[except yyyError [as nombre]:] # Opcionales
  # Se pueden poner varios except
  # Aquí controla la situación de error yyyError.
[else:] # Opcional.
  # Bloque a ajecutar si try no tuvo errores
[finally:] # Opcional
  # Código que se ejecutará siempre.
  # Ocurra lo que ocurra en el programa.
  # P.e. cerrar los recursos
```

D.2. Lanzar Excepciones

- El programador puede **lanzar una excepcion** utilizando **raise** o **assert**.
- Ejemplo:

```
assert param > 0, "Deber ser positivo"

if param <= 0:
  raise ValueError("Deber ser positivo")
```

D.3. Definir Excepciones del Usuario

```
>>> class BaseError(Exception):
...   # Lo correcto es definir __init__(self, *args)
...   ...
...
>>> raise BaseError   # Alza un error sin texto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__console__.BaseError
```

Un ejemplo personalizado:

```
>>> class NotInRangeError(Exception):
...   def __init__(self, valor: int,
...                mensaje="No está en (10, 40)"):
...     self._valor = valor
...     self._mensaje = str(valor) + " -> " + mensaje
...     super().__init__(self._mensaje)
...
>>> valor = 50
>>> if valor < 10 or 40 < valor:
...   raise NotInRangeError(valor)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__console__.NotInRangeError: 50 -> No está en (10, 40)
```

E. Entrada y Salida de Datos

E.1. Entrada y Salida Estándar

- **Mostrar objetos en la pantalla**

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
                                   flush=False)
```

  - Todos los argumentos de ***objects** se convierten a cadenas de caracteres.
  - Los objetos se mandan al flujo separados por **sep**.
  - Se termina el envío con **end**.
  - **file** debe ser un objeto que implemente un método **write(string)** (p.e. cualquier fichero de texto). Si no se indica se usará **sys.stdout**. También puede usar **sys.stderr** (donde se envían las indicaciones del intérprete y mensajes de error).
  - **flush** indica si se obliga a vaciar el buffer del stream.
- **Lee datos del teclado.** `input([prompt])`
  - Si se indica **prompt** se mostrará en la salida estándar sin nueva línea.
  - Lee una línea de la entrada **sys.stdin**, la convierte en una cadena (eliminando la nueva línea), y retorna eso.

E.2. Esquema General

1. **Abrir un fichero.** `f = open(file, mode='r')`
   - **file** es un string indicando la ubicación del fichero.
   - **mode** es el modo de E/S. Por defecto es 'rt'.
     - **r**: Abre para **lectura**. El fichero debe existir.
     - **w**: Abre para **escritura**. Sobreescribe el fichero si existe.
     - **a**: Abre para **añadir**. Si no existe el fichero lo crea.
     - **b**: Si se especifica se tratará el fichero como **binario**.
     - **t**: Será una E/S de texto. Valor por defecto.
     - **+**: Se puede usar el fichero para **leer y escribir**.
     - **x**: Abierto para **creación** en exclusiva, falla si el fichero ya existe.
   - Retorna un **file object**, que tiene una interface para tratar archivos (pasos siguientes) y los atributos:
     - **f.closed**: True si el fichero está cerrado. Falso en otro caso.
     - **f.mode**: Indica el modo de acceso de cómo se abrió el fichero.
     - **f.name**: Indica el nombre del fichero.
2. **Leer/Escribir en el fichero.**
   - **f.write(string)**: Escribe un string en el fichero abierto.
   - **f.read([n])**: Vuelca en un string el contenido del fichero abierto. Opcionalmente se puede indicar el número de bytes que se quieran leer.
   - **f.readline()**: Lee una sola línea del archivo. Deja `\n` al final de la cadena.
   - **f.readlines()**: Vuelca en una lista de strings el contenido del fichero
   - **f.writelines(lista)**: Escribe una lista de líneas en el archivo.
   - **f.tell()**: Indica la posición actual del puntero.
   - **f.seek(offset[, from])**: Cambia la posición según offset desde una posición dada. Si `from` es 0 indica el inicio, 1 indica la posición actual y 2 indica la posición final.
3. **Cerrar el fichero.**
   - **f.close()**: Cierra el fichero

E.3. Sentencia with

```
with open('fichero', 'modo') as f:
  # Bloque que manipula f
```

F.1.1 (TDA Array 1D) *Un array 1-dimensional es una colección de elementos contiguos, todos del mismo tipo y donde cada elemento está identificado por un único entero no nulo. Una vez creado el array su tamaño no puede cambiarse pero su elementos sí.*

- ***Array1D(size) : Array1D***. *Crea un array 1-dimensional que constará de **size**-elementos que se inicializarán a **None**. Se requiere que **size**> 0.*
- ***length() : int***. *Retorna la longitud o número de elementos en el array.*
- ***getItem(index) : value***. *Retorna el elemento o valor almacenado en la posición **index**. El argumento para **index** debe tomar valores entre 0 y **length()-1**.*
  *Este método, en **Python** será invocado usando indexación. Ver ejercicio **??**.*
- ***setItem(index, value) : None***. *Sustituye el el **index**-ésimo valor del array por **value**. El argumento para **index** debe tomar valores entre 0 y **length()-1**.*
  *Este método, en **Python** será invocado usando indexación. Ver ejercicio **??**.*
- ***clear(value) : None***. *Limpia el array asignando a todas las posiciones el valor **value**.*
- ***iterator() : IteratorArray1D***: *Crea y retorna un iterador para el array.*
  *Este método, en **Python** será definiendo el método **__iter__()**. Ver ejercicio **??***

F.1.2 (TDA Array 2D) *Un array 2-dimensional es una colección de elemento contiguos cuyos elementos están identificados por dos enteros únicos. El primer índice se llama índice fila y el segundo índice columna y para ambos su primer valor es el 0. Una vez creado el array, su tamaño no puede cambiarse.*

- ***Array2D(ncols1, ncols2, ...)***: *Crea un array 2-dimensional que constará de **nrows**-índices fila dado por el número de argumentos dados y **ncols1**-índices columna para la primera fila, **ncols2**-índices columna para la segunda fila, etc. Todos los datos se inicializarán a **None**. Se requiere que cada valor **ncols**> 0.*
- ***numRows()***: *Retorna el número de filas del array.*
- ***numCols(row)***: *Retorna el número de columnas del array para la fila **row**.*
- ***getItem(i, j)***: *Retorna el valor almacenado en la posición **[i, j]**. Los argumentos de los índices debe tomar valores dentro de sus rangos válidos.*
- ***setItem(i, j, value)***: *Sustituye el elemento **[i, j]** del array por **value**. Los argumentos de los índices debe tomar valores dentro de sus rangos válidos.*
- ***clear(value)***: *Limpia el array asignando a todas las posiciones el valor **value**.*
- ***iterator()***: *Crea y retorna un iterador para el array.*

Tenga en cuenta en la implementación las particularidades de Python.

F.1.3 (Bag) *Un bolso (Bag) representa a una colección de elementos que pueden aparecer repetidos y que no tienen un orden en particular de aparición.*

- ***Bag() : Bag***. *Crea un nuevo bag, inicialmente vacío.*
- ***len() : int***. *Retorna el número de elementos en el bag. Para una bag cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.*
- ***contains(item) : bool***. *Indica si el elemento **item** se encuentra en el bag. Retorna **True** si está contenido y **False** si no está contenido.*
- ***remove(item) : None***. *Elimina y retorna la ocurrencia **item** del bag. Lanza un error si el elemento no existe.*
- ***add(item) : None***. *Modifica el bag añadiendo el elemento **item** al bag.*
- ***iterator() : IteratorBag***: *Crea y retorna un iterador para el bag.*

F.1.4 (Lista) *Una lista representa una secuencia de elementos indexados que pueden aparecer repetidos.*

- ***List() : Lista***. *Crea una nueva lista, inicialmente vacía.*
- ***len() : int***. *Retorna la longitud o número de elementos en la lista.*
  Para una lista cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.
- ***contains(value) : bool***. *Indica si el valor **value** se encuentra na la lista.*
- ***getItem(pos) : value***. *Retorna el elemento o valor almacenado en la posición **pos**.*
  Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ retornará el valor a_{pos}.
- ***setItem(pos, value) : None***. *Sustituye el **pos**-ésimo valor de la lista por **value**.*
  Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, \ldots, a_{n-1} >$.
- ***insertItem(pos, value) : None***. *Inserta el **pos**-ésimo valor de la lista por **value**.*
  Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, a_{pos}, \ldots, a_{n-1} >$.
- ***removeItem(pos) : None***. *Elimina el elemento de la posición dada, si existe en la lista.*
  Dada una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, a_{pos-1}, a_{pos+1}, \ldots, a_{n-1} >$.
- ***clear() : None***. *Limpia la lista. Se convierte en una lista vacía.*
  Modifica cualquier lista a la lista <>.
- ***isEmpty() : bool***. *Indica si la lista está vacía o no.*
- ***first() : pos***. *Retorna la posición del primer elemento de la lista. Si la lista está vacía retornará **last()***
  Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará la posición donde se localiza a_0.
- ***last() : pos***. *Retorna la posición del último elemento de la lista. Si la lista está vacía retornará **last()***
  Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará la posición donde se localiza a_{n-1}.
- ***next(pos) : pos***. *Retorna la posición del elemento siguiente al elemento de la posición dada.*
  Dada una lista retornará la posición donde se localiza el elemento a_{pos+1}.
- ***previous(pos) : pos***. *Retorna la posición del elemento anterior al elemento de la posición dada.*
  Dada una lista retornará la posición donde se localiza el elemento a_{pos-1}.
- ***iterator() : Lista***: *Crea y retorna un iterador para la lista.*

F.1.5 (Pila) *Una pila representa una lista de elementos que se rigen por el criterio LIFO.*

- ***Stack() : Stack***. *Crea una nueva pila, inicialmente vacía.*
- ***peek() : value***. *Retorna el valor del tope. También se suele usar la signatura* ***top() : value***.
  Para una pila $< a_0, a_1, \ldots, >$ *retornará el valor* a_0.
- ***pop() : value***. *Retorna el valor del tope y además borra el primer nodo de la pila.*
  Para una pila $< a_0, a_1, a_2, \ldots, >$ *retornará el valor* a_0 *y la nueva pila es* $< a_1, a_2, \ldots, >$.
- ***push(value) : None***. *Inserta un nuevo nodo en el tope de la lista.*
  Para una pila $< a_0, a_1, a_2, \ldots, >$ *y un valor* $value$ *la pila se modifica para conseguir la pila* $< value, a_0, a_1, a_2, \ldots, >$.
- ***len() : int***. *Retorna el número de elementos de la pila.*
- ***isEmpty() : Bool***. *Indica si la pila está vacía o no.*
- ***clear() : None***. *Limpia la pila y la deja sin elementos.*

F.1.6 (Cola) *Una cola representa una lista de elementos que se rigen por el criterio FIFO.*

- ***Queue() : Queue***. *Crea una nueva cola, inicialmente vacía.*
- ***peek() : value***. *Retorna el valor del primer elemento de la lista, pero no lo borra. También es usual esta signatura* ***front() : value***.
- ***dequeue() : value***. *Retorna el primer elemento de la cola borrandolo de la cola. También es usual esta signatura* ***top() : value***.
  Para la cola $< a_0, a_1, \ldots, >$ *retornará el valor* a_0. *Se genera un error si la cola está vacía.*
- ***enqueue(value) : None***. *Añade un nuevo elemento al final de la cola*
  Para la cola $< a_0, a_1, \ldots, a_{n-1} >$ *se modificará a la lista* $< a_0, a_1, \ldots, a_{n-1}, value >$
- ***len() : int***. *Retorna el número de elementos de la cola.*
- ***isEmpty() : Bool***. *Indica si la cola está vacía o no.*
- ***clear() : None***. *Limpia la cola y la deja sin elementos.*

F.1.7 (Cola de Prioridad) *Una cola representa es una lista de elementos similar a una cola en la que los elementos tienen adicionalmente, una prioridad asignada. Suponiendo que la prioridad es ser clave-mínima, la definición es:*

- ***PriorityQueue() : PriorityQueue***. *Crea una nueva cola de prioridad.*
- ***add(k ,v)***. *Añade un nuevo elemento* ***(k ,v)***
- ***min()***. *Retorna la pareja* ***(k ,v)*** *siendo* ***k*** *la clave mínima; pero no borrar el item. Se muestra un error si la cola de prioridad está vacía. Si la cola de prioridad tuviera varias entradas con claves equivalente seleccionará uno arbitrario si hay varios con clave mínima.*
- ***pop() : (key, value)***. *Retorna la pareja* ***(k ,v)*** *siendo* ***k*** *la clave mínima. Borra también el item. Muestra un error si la cola de prioridad está vacía. Si la cola de prioridad tuviera varias entradas con claves equivalente seleccionará uno arbitrario si hay varios con clave mínima.*
- ***es_vacia() : bool***. *Indica si la pila está vacía (retorna* ***True****) o no (retorna* ***False****).*
- ***len() : int***. *Retorna el número de items existentes en la cola de prioridad.*

Se puede implementar mediante una lista, pero es más eficiente usar un **montículo** *(ver árboles binarios).*

F.1.8 (Map) *Un Map representa a una colección de registros no repetidos donde cada uno consta de una clave y un valor. La clave debe ser comparable.*

- ***Map() : Map***. *Crea un nuevo map vacío.*
- ***len() : int***. *Retorna el número de registros clave/valor que existen en el map..*
- ***contains(key) : bool***. *Indica si la clave* ***key*** *se encuentra en el contenedor. Retorna* ***True*** *si la clave está contenida y* ***False*** *si no está contenida.*
- ***remove(key) : None***. *Elimina el registro que tiene como clave el valor* ***key****. Lanza un error si el elemento no existe.*
- ***add(key, value) : None***. *Modifica el map añadiendo el par* ***key/value*** *al contenedor. Si existiera un registro con la clave* ***key*** *se sustituye el par* ***key/value*** *existente por el nuevo par* ***key/value****. Retorna* ***True*** *si la clave es nueva y* ***False*** *si se realiza una sustitución.*
- ***valueOf(key) : TipoValor***. *Retorna el valor asociado a la clave dada. La clave debe de existir en el Map.*
- ***iterator() : IteratorMap***: *Crea y retorna un iterador para el conjunto.*

F.1.9 (Set) *Un conjunto (set) representa a una colección de elementos no repetidos que no tienen un orden en particular.*

- ***Set() : Set***. *Crea un nuevo conjunto, inicialmente vacío.*
- ***len() : int***. *Retorna el número de elementos en el conjunto.*
  Para una lista cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ *retornará el valor n.*
- ***contains(element) : bool***. *Indica si el elemento* ***element*** *se encuentra en el conjunto. Retorna* ***True*** *si está contenido y* ***False*** *si no está contenido.*
- ***remove(element) : None***. *Elimina el elemento* ***element*** *del conjunto. Lanza un error si el elemento no existe.*
- ***add(element) : None***. *Modifica el conjunto añadiendo el elemento* ***element*** *al conjunto.*
- ***equal(setB) : bool***. *Determina si el conjunto es igual al conjunto dado. Dos conjuntos son iguales si ambos contienen el mismo número de elementos y todos los elementos del conjunto está en el conjunto B. Si ambos están vacíos entonces son iguales.*
- ***isSubsetOf(setB) : bool***. *Determina si un conjunto es subconjunto del conjunto dado.*
  Un conjunto A es subconjunto de B si todos los elementos de A están en B.
- ***union(setB) : Set***. *Retorna un nuevo conjunto que es la unión del conjunto con el conjunto dado.*
  La unión del conjunto A con el conjunto de B es un nuevo conjunto que está formado por todos los elementos de A y todos los elementos de B que no están en A.
- ***difference(setB) : Set***. *Retorna un nuevo conjunto que es la diferencia del conjunto con el conjunto dado.*
  La diferencia del conjunto A con el conjunto de B es un nuevo conjunto que está formado por todos los elementos de A que no están en B.
- ***intersect() : Set***. *Retorna un nuevo conjunto que es la intersección del conjunto con el conjunto dado.*
  La intersección del conjunto A con el conjunto de B es un nuevo conjunto que está formado por todos los elementos que están en A y también en B.
- ***iterator() : IteratorSet***: *Crea y retorna un iterador para el conjunto.*

F.2.1 (TDA Árbol) *Un árbol responde a la siguiente definición recursiva:*

- **Caso base:** *El árbol vacío (sin elementos)*
- **Caso recursivo:** *Es una colección de datos que está formada por un dato y una lista de 0 o más árboles.*

Las especificaciones informales de sus métodos son:

- ***Tree() : Tree***. *Crea un nuevo árbol*
- ***append(value, node=pos) : None***. *Añade un nuevo nodo al árbol con el valor* ***value***. *Si se especifica el segundo parámetro, el nuevo nodo será hijo de* ***pos***.
- ***value(pos) : type_value***. *Retorna el contenido del nodo (posición)* ***pos***.
- ***replace(pos, value) : value***. *Cambia el valor del nodo de la posición* ***pos*** *por un nuevo valor (el de entrada) y retorna el antiguo valor.*
- ***remove(pos) : None***. *Elimina el nodo de la posición* ***pos***.
- ***parent(pos) : pos***. *Retorna la posición del padre para la posición* ***pos***. *Será* ***None*** *si la posición de entrada se corresponde con la raíz.*
- ***children(pos) : container***. *Retorna un contenedor iterable con todos los hijos de* ***pos***.
- ***positions() : container***. *Retorna un contenedor iterable con todos los nodos del árbol.*
- ***elements() : container***. *Retorna un contenedor iterable con todos los valores del árbol.*
- ***num_children(pos) : int***. *Indica el número de hijos que tiene el nodo* ***pos***.
- ***len() : int***. *Retorna el número de elementos que tiene el grafo*
- ***depth(pos) : int***. *Retorna la profundidad del nodo* ***pos***.
- ***height(pos) : int***. *Retorna la altura del nodo* ***pos***.
- ***root() : pos***. *Retorna la posición del nodo raíz del árbol.*
- ***isRoot(pos) : Bool***. *Retorna* `True` *si la posición* ***pos*** *es el nodo raíz del árbol. Retorna* `False` *en otro caso.*
- ***isInternal(pos) : Bool***. *Retorna* `True` *si la posición* ***pos*** *es el de un nodo interno. Retorna* `False` *en otro caso.*
- ***isLeaf(pos) : Bool***. *Retorna* `True` *si* ***pos*** *es un nodo hoja del árbol. Retorna* `False` *en otro caso.*
- ***isEmpty() : Bool***. *Indica si el árbol está vacío o no.*

Se pueden ampliar con una gran cantidad de métodos como, por ejemplo:

- **`borrar(): None`**. Borrar los nodos empezando por el nivel más profundo.
- **`copiar() : TreeBinary`**. Copiar un árbol empezando por la raíz.
- **`contar(criterio) : int`**. Contar el número de elementos que cumplan cierto criterio.
- **`buscar(valor) : bool`**. Buscar un elemento en el árbol.
- **`comparar(arbol) : bool`**. Indica si el árbol dado coincide con el actual.
- **`altura() : int`**. Calcula la altura del árbol.
- **`hojas() : int`**. Calcula el número de hojas del árbol.

F.2.2 (TDA Árbol Binario) *Un* **árbol binario** *es un árbol que* **siempre** *tiene dos subárboles que reciben el nombre de hijo (o subárbol) izquierdo e hijo (o subárbol) derecho. En un árbol binario los subárboles pueden ser vacíos.*

Las especificaciones informales del constructor y algunos métodos recursivos son:

- ***TreeBinary() : TreeBinary***. *Crea un nuevo árbol binario.*
- ***mostrar(tipo="tipo") : None***. *Muestra el contenido del árbol según el tipo de recorrido: prefijo, infijo, postfijo.*
  - *Si tipo="infijo". Mostrar los elementos de un árbol como si fuera una expresión aritmética.*
  - *Si tipo="prefijo". Mostrar los elementos de un árbol como si fuera una estructura de directorio.*
- *Algunos de los métodos indicados en la definición de Árbol se pueden optimizar para este tipo de árboles.*

F.2.3 (TDA Árbol Binario de Búsqueda) *Un* **árbol binario de búsqueda** *es un árbol binario que cumple la siguientes 2 propiedades:*

- *El valor del hijo izquierdo es menor que el valor de la raíz.*
- *El valor del hijo derecho es mayor que el valor de la raíz.*

Por recursividad, se cumplirá que

- *todos los descendiente de la rama izquierda son menores que el valor de la raíz.*
- *todos los descendiente de la rama derecha son mayores que el valor de la raíz.*

Las especificaciones de algunos métodos propios de este TDA son:

- ***TreeBinarySearch() : TreeBinarySearch***. *Crea un nuevo árbol binario de búsqueda.*
- ***search(valor) : bool***. *Indica si el valor se encuentra en el árbol.*
- ***add(valor) : None***. *Añadir un elemento en el árbol. Los nuevos elemento se añaden siempre como nodos hojas.*
- ***min() : valor***. *Busca el valor mínimo de un árbol.*
- ***max() : valor***. *Busca el valor máximo de un árbol.*
- ***remove(valor) : None***. *Elimina el nodo que tiene un valor. El árbol resultante tiene que seguir siendo un árbol binario de búsqueda.*

Mira en la teoría cómo se deben implementar cada uno de los métodos.

F.2.4 (TDA Heap) *El término heap se puede traducir por* **montículo**, *cúmulo, montón o pila.*

Un **heap** *es un árbol donde cada nodo consta de una pareja* $(clave, valor)$ *y representa a un conjunto ordenado; es decir, existe una ordenación entre los nodos del árbol.*

$$(clave, valor) < (clave', valor') \Leftrightarrow clave < clave'$$

Los montículos máximos tienen la característica de que cada nodo padre tiene una clave mayor que el de cualquiera de sus nodos hijos, mientras que en los montículos mínimos, el valor de la clave del nodo padre es siempre menor al de sus nodos hijos.

Las operaciones básicas sobre heap son:

- *Crear un Heap a partir de una lista*
- *Añadir un nuevo elemento. Se debe seguir manteniendo la propiedad de orden del Heap.*
- *Retornar el valor del nodo raíz del heap y quitarlos del heap. Se debe reordenar el heap para que siga cumpliendo la propiedad de Heap-Mín/Máx.*

Un Heap se puede codificar mediante un **árbol binario completo**, *que además tiene la ventaja de que se puede hacer mediente un TDA lineal (array o lista), localizando la posición del hijo izquierdo por el valor* $2n + 1$ *siendo* n *la posición del nodo padre.*

F.2.1 (Grafo) *Un grafo (Graph) representa a un par* (V, R) *siendo* V *un conjunto de elementos y* $R \subset V \times V$. *Si* $(u, v) \in R$ *se dice que hay un arco de* u *a* v. *Un arco puede tener asociado un coste* $c_{u,v} \in \mathbb{R}$.

- ***Graph() : Graph***. *Crea un nuevo grafo, inicialmente vacío.*
- ***appendVertex(v)***. *Añade un nuevo vértice al grafo,.*
- ***appendArc(u, v, c)***. *Añade un nuevo arco* (u, v) *con un peso* $c = c_{u,v}$. *Si el arco ya existe, modifica su coste actual por el valor* c.
- ***removeVertex(v) : None***. *Elimina el vérice* ***v*** *del grafo, junto con todos los arcos que lo contengan. Lanza un error si el elemento no existe.*
- ***removeArc(u, v) : None***. *Elimina el arco* (u, v) *del grafo, junto con todos los arcos que lo contengan. Lanza un error si el elemento no existe.*
- ***len() : int***. *Retorna el número de vértices del grafo.*
- ***contains(element) : bool***. *Indica si el elemento* ***element*** *se encuentra en el conjunto de vértices. Retorna* ***True*** *si está contenido y* ***False*** *si no está contenido.*