Tecnologías de la Programación

Relación de Ejercicios

y

Sesiones de Prácticas

Luis Daniel Hernández Molinero

Dpto. de Ingeniería de la Información y las Comunicaciones

ldaniel@um.es

Última modificación:
12 de diciembre de 2023

Preámbulo

Este documento consta de una serie de ejercicios de dificultad creciente divididos por semanas y sesiones. Cada sesión consta de una parte de teoría y otra de prácticas (salvo la primera sesión). La parte teórica de todas las sesiones de una semana se habrán tenido que leer antes de la sesión de clase presencial de dicha semana. Al inicio de la clase presencial el profesorado hará un resumen de los aspectos teóricos más relevante dejando los detalles para esa lectura previa y su estudio.

Cada ejercicio consta de un nombre seguido de una caja con la siguiente apariencia.

★☆☆ ✍

Resumen/intención del ejercicio

- En la parte superior aparece una secuencia de 3 estrellas, cuyo significado es el siguiente:

  ★☆☆ Nivel fácil.
  ★★☆ Nivel medio.
  ★★★ Nivel alto.

  El nivel de dificultad se establece siempre en el contexto del tema/tópico correspondiente. Por ejemplo, para el tópico de variables pueden encontrarse los 3 niveles, pero los mismos problemas en el contexto del uso de funciones todos tendrían un nivel fácil pues el uso de variables es un tema fundamental.
- A continuación aparece una mano. Es un icono que no siempre aparece.
  Si aparece el icono de la mano significa que es un ejercicio que: o se explicará en clase u **obligatoriamente deberá ser desarrollado por el estudiante para adquirir las destrezas mínimas necesarias** en la resolución de problemas con programación orientada a objetos e implementación de tipos de datos abstractos.
  Para los que no aparece el icono no significa que el ejercicio no deba de hacerse, todo lo contrario pues son ejercicios de refuerzo y conviene hacerlos o al menos plantearlos.
- En la caja con fondo gris se expresa en una o dos frases la intencionalidad del ejercicio.

El desarrollo de los ejercicios se hará mediante una estrategia de Divide y Vencerás. En concreto están diseñados para realizarse en grupos de 2 miembros y entre ambos deben de realizar los ejercicios propuestos. Cada miembro debería desarrollar un ejercicios en su totalidad y explicarlo al otro miembro del grupo. Un ejemplo más práctico. En una sesión hay que implementar las pilas y las colas y cada uno de estos TDA consta de 6 métodos. Entonces un miembro afronta las pilas y el otro miembro afronta las colas. Se pueden usar otras estrategias pero lo que se recomienda es que cada alumno implemente una serie de métodos y los ponga en común con su compañero/a para resolver el ejercicio.

Índice del Capítulo

Página

1. Python y PyCharman **1**
1.1. Introducción . . . 1
1.2. Instalación de Python . . . 1
1.3. Instalación de Herramientas de Desarrollo . . . 3
1.4. Sesión de trabajo . . . 4
1.4.1. Usando Editores . . . 4
1.4.2. Sesión de Trabajo con PyCharm . . . 5
1.5. Documentar el Código del Proyecto . . . 6
1.6. Generar la Documentación de un Proyecto . . . 6
1.6.1. Sin crear ficheros externos . . . 7
1.6.2. Creando ficheros externos . . . 7
1.6.2.1. Cómo usar `pydoctor` . . . 7
1.6.2.2. Cómo usar `pdoc3` . . . 7
1.6.2.3. Cómo usar `sphinx` . . . 8
1.7. Gestión de Algunos Errores en PyCharm . . . 9
1.7.1. Resolver el problema de Paquetes en PyCharm . . . 9

2. Abstracción **11**
2.A. Conceptos Básicos . . . 11
2.B. Mecanismos de Abstracción . . . 12
2.C. Tipos de Abstracción . . . 13
2.C.1. Abstracción Procedimental . . . 13
2.C.2. Abstracción de Datos . . . 14
2.D. Tipos de Datos Abstractos . . . 14
2.D.1. Especificación de Datos Abstractos . . . 15
2.D.1.1. Especificación Formal . . . 16
2.D.1.2. Especificación Informal . . . 17
2.D.2. Representación de TDAs . . . 18
2.D.3. Implementación de TDAs . . . 19
2.E Python . . . 20
2.E.1 Breve Historia de Python . . . 20
2.E.2 Características y Uso . . . 21
2.E.3 Variables, Expresiones y Conversión de Tipo . . . 22
2.E.4 Programación Estructurada . . . 23
2.E.5 Programación Procedimental . . . 25
2.E.6 Programación Modular . . . 28
2.E.7 Programación Orientada a Objetos . . . 28
2.1. Un clase para números positivos . . . 29
2.2. TDA Fecha Simple . . . 30
2.3. TDA Fecha Simple con Positivo . . . 31
2.4. Un clase para cilindros . . . 31
2.5. TDA Fracción . . . 31
2.6. Adivina el número . . . 32

3. Colecciones **33**
3.A. Estructuras de Datos . . . 33
3.B. Secuencias de Python . . . 34
3.C. Colecciones no Secuenciales de Python . . . 39
3.D. Iteración en Colecciones . . . 42
3.E. Construcción de Contenedores por comprehensión . . . 42
3.F. Métodos Mágicos . . . 43
3.G. Arrays en Python . . . 45
3.G.1. La Estructura Array . . . 45
3.G.2. Listas como arrays . . . 46
3.G.3. Array compacto . . . 47

3.G.4. ¿Por qué necesitamos arrays? Implementación en Python . . . 48
3.1. Construcción por comprehensión . . . 49
3.2. Más Ejercicios con Secuencias . . . 49
3.3. Un poco de estadística . . . 50
3.4. TDA Bag . . . 50
3.5. Anagramas . . . 51
3.6. TDA Map . . . 52
3.7. TDA Set . . . 53
3.8. Cursos de un Título . . . 53
3.9. Iterador de listas . . . 54
3.10. TDA Array1D . . . 55
3.11. TDA Array 2D . . . 56
3.12. Acceder a datos indexados . . . 57
3.13. Matrices . . . 58
3.14. Juego de la Vida . . . 58
3.15. Reversi . . . 59

4. Listas **61**
4.A. TDAs Lineales . . . 61
4.B. Implementación de TDAs lineales . . . 62
4.C. TDA Lista . . . 63
4.C.1. Implementación con Arrays . . . 63
4.C.2. Implementación con Estructuras Enlazadas . . . 64
4.D. Búsqueda y Ordenación . . . 67
4.D.1. Búsqueda . . . 68
4.D.1. Ordenación . . . 69
4.1. Listas con Array1D . . . 72
4.2. Lista Simplemente Enlazada . . . 72
4.3. Lista Simplemente Enlazada con Índices . . . 72
4.4. Lista Doblemente Enlazada . . . 73
4.5. Módulo de Búsqueda . . . 73
4.6. Módulo de Ordenación . . . 73
4.7. Ordenar a los alumnos . . . 74

5. Pilas y Colas **75**
5.A. TDA Pila . . . 75
5.B. TDA Cola . . . 76
5.C. TDA Cola de Prioridad . . . 76
5.D. Problemas de Búsqueda . . . 77
5.1. Pilas . . . 79
5.2. Colas . . . 79
5.3. Resolución de Laberintos . . . 79
5.4. Laberintos con Heurística . . . 81
5.5. Resolución de n-Puzzle . . . 81

6. Árboles y Grafos **83**
6.A. Recursividad . . . 83
6.B. TDA Árbol . . . 85
6.B.1. Árbol Binario . . . 89
6.B.3. Árboles binarios de búsqueda . . . 92
6.B.3. Heap . . . 94
6.B.3. Representación de los árboles . . . 95
6.C. TDA Grafo . . . 97
6.C.1. Recorrido en Grafos . . . 99
6.C.2. Representaciones de un Grafo . . . 99
6.1. Árboles . . . 100
6.2. TDA Árbol Binario . . . 100
6.3. Adivinador . . . 100
6.4. Árboles Binarios de Búsqueda . . . 101

6.5. Heap Min 101
6.6. Árbol de partición 101
6.7. Árboles de Codificación 102
6.8. Caminos de coste mínimo 102
6.9. Árboles Generadores 104

Sesión 1

Python y PyCharman

Índice Parcial

	Página
1.1. Introducción	1
1.2. Instalación de Python	1
1.3. Instalación de Herramientas de Desarrollo	3
1.4. Sesión de trabajo	4
1.4.1. Usando Editores	4
1.4.2. Sesión de Trabajo con PyCharm	5
1.5. Documentar el Código del Proyecto	6
1.6. Generar la Documentación de un Proyecto	6
1.6.1. Sin crear ficheros externos	7
1.6.2. Creando ficheros externos	7
1.6.2.1. Cómo usar `pydoctor`	7
1.6.2.2. Cómo usar `pdoc3`	7
1.6.2.3. Cómo usar `sphinx`	8
1.7. Gestión de Algunos Errores en PyCharm	9
1.7.1. Resolver el problema de Paquetes en PyCharm	9

1.1 Introducción

Python es un lenguaje de programación de propósito general escrito sobre el lenguaje C. Se usa en una amplia variedad de disciplinas como biología, finanzas, química, análisis numérico, inteligencia artificial, etc. También se usa como lenguaje script para los administradores de sistemas informáticos.

En esta primera sesión práctica se indica cómo llevar a cabo las instalaciones de **Python** y algunas herramientas, cuáles son los pasos usuales de una sesión de trabajon así como los pasos a realizar para documentar nuestros programas.

1.2 Instalación de Python

CPython es la implementación estándar de Python. `https://www.python.org/`

- CPython no traduce el código **Python** a C por sí mismo.
- Ejecuta un intérprete.
- Instalación básica ≈ 120Mb
- Cython es un módulo que compila a código en C o C++ desde Python
  - Permite incluir C en código Python
  - Permite incluir **Python** en código C y ser compilado.

Puede instalar **Python** (**CPython**) sin problema en su Sistema Operativo:

- Windows. Consulte `https://www.python.org/downloads/windows/`. Durante la instalación, verá una ventana de «Setup». Asegúrate de marcar las casillas «Add Python xx to PATH» o «Add Python to your environment variables»
  Para más detalles consultar `https://tutorial.djangogirls.org/es/python_installation/`

- mac OS. Descargue e instale (como en Windows) pero **tenga mucho cuidado si su SO es anterior al 2022**. Algunos macOS llevan Python como parte del sistema operativo. Trabajar directamente sobre Python del SO puede conllevar que éste deje de funcionar si no se sabe exactamente qué se está haciendo.

  Si tiene un Mac OS, abra un terminal y ejecute `python -version`. Si como resultado obtiene `Python 2.xx` entonces debe instalar previamente Homebrew y después Python de Homebrew.

  1. Para instalar Homebrew, abra un terminal y ejecute:
     ```
     ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
     ```
  2. Inserte el directorio de Homebrew al inicio de tu variable de ambiente PATH. Puedes hacer esto agregando la siguiente línea al final de tu archivo `~/.profile` o `~/.bash_profile`

     ```
     export PATH=/usr/local/bin:/usr/local/sbin:$PATH
     ```

  3. Instalar Python 3: `brew install python3`

  Ahora tendrá dos versiones de Python. La del SO y la ubicada en `/usr/local/Cellar`.

  Ahora bien, si al ejecutar `python -version` obtiene un mensaje de error, entonces solo debe instalar la distribución de la página oficial: https://www.python.org/downloads/
- GNU/Linux

  ```
  Debian: sudo apt install python3
  Fedora: sudo dnf install python3
  ```

Como alternativa a CPython (la distribución estándar) puede instalar Python usando alguna suit especializada que lo incluya.

Quizás la más conocida sea **Anaconda Python.** Desarrollada por Anaconda. https://www.anaconda.com/

- Pensada para desarrolladores/empresas de Python que necesiten respaldo.
- Es una distribución pensada para ciencia de los datos y aprendizaje automático que incluye Python, IPython y R entre otros.
- Orientado el trabajo comercial y científico.
- La más completa para ciencias de los datos.
- La edición individual es gratis. Otras superan las $10mil.
- Instalación básica ≈ 4.8Gb. CPython+Editores+IDE+...
- Gestiona paquetes Python y de terceros.

Existe una versión libre y versiones comerciales. Al adaptarse desde estudiantes a profesiones, la distribución es usada por millones de usuarios. Destaca su sistema de gestión de paquetes llamado conda.

Otra suit alternativa es **ActivePython.** Desarrollado por Activestate. https://www.activestate.com/. Tiene la misma filosofía de Anaconda.

Otras alternativas curiosas son:

- **PyPy.** Alternativa a CPython. https://www.pypy.org/
  - Compilador JIT[1] de RPython.
  - 4.2 veces más rápido que CPython.
  - Usa el "núcleo" de CPython.
- **IronPython.** Alternativa a CPython. https://ironpython.net/
  - Implementado en C#.
  - Integra Python en .NET[2].
  - Ejecuta Python en Microsoft DLR (entorno de ejecución)
- **Jython.** Alternativa a CPython. https://www.jython.org/
  - Implementado en Java.
  - En Java se pueden añadir librerías de Jython.
  - En Jython se puede añadir java.
  - Ejecuta Python en JVM.
- Hay más: WinPython, MicroPython, Pycopy, RustPython, MesaPy, ...

[1] Compilación en tiempo de ejecución (JIT) mejora el rendimiento compilando a bytecode y traducir el bytecode a código máquina nativo en tiempo de ejecución.

[2] Plataforma de aplicaciones que permite la creación y ejecución de servicios web y aplicaciones de Internet.

1.3 Instalación de Herramientas de Desarrollo

Existen varios Entornos de Desarrollo Integrado (Integrated Development Environment, IDE) y editores que ayudan a la programación. En el contexto de **Python** podemos destacar

- PyCharm. Es uno de los IDE de **Python** más completos http://jetbrains.com/pycharm
- PyDev. Un buen IDE con soporte para CPython, Jython e Iron Python. http://pydev.org
- Spyder. IDE de código abierto y totalmente gratuito. Desarrollado principalmente para científicos e ingenieros. Se recomienda usar el que viene con Anaconda https://www.spyder-ide.org/
- Visual Studio Code. Es un editor de código fuente con multitud de plugins. https://code.visualstudio.com/
- Sublime Text. Es un editor que viene con muchas opciones. https://www.sublimetext.com/
- Atom. Otro editor. Es de lo más usados. https://atom.io/

Veamos cómo instalar algunos de ellos.

Instalación de PyCharm

1. Instalar **PyCharm Community** desde https://www.jetbrains.com/es-es/pycharm/download/
2. Para instalar un diccionario español:
   - Descargar en cualquier directorio el diccionario https://github.com/EOM/Spell-Checking-PHPStorm-Spanish-dic-UTF8/blob/master/spanish-utf8.dic.
   - Desde PyCharm seleccionar Preferences » Editor » Natural Languages » Spelling y añadir el diccionario anterior.
3. En PyCharm » Preferences » Plugins instalarás complementos que te pueden ser útiles. En particular, algunos tema para el IDE:
   - Godot Theme (recomendado)
   - Theme Collection
   - Visual Studio Code Dark Plus Theme

   Pulsar Apply
4. En PyCharm » Preferences » Editor » Proofreading
   - Seleccionar +↓
   - Seleccionar Español

   Pulsar Apply

Para el desarrollo de las prácticas de este curso de Tecnología de la Programación es suficiente PyCharm Community, que es la que está instalada en los ordenadores de la universidad. Pero también se puede descargar la versión profesional en el mismo enlace.

Las diferencias entre una y otra versión se pueden ver la final de la página https://www.jetbrains.com/es-es/pycharm/features/. De la versión profesional os puede venir bien la Çapacidad para desarrollo remoto". Esta característica permite que todos los miembros de un grupo puedan desarrollar código a la vez, lo que es ideal para esta primera parte de las prácticas de la asignatura.

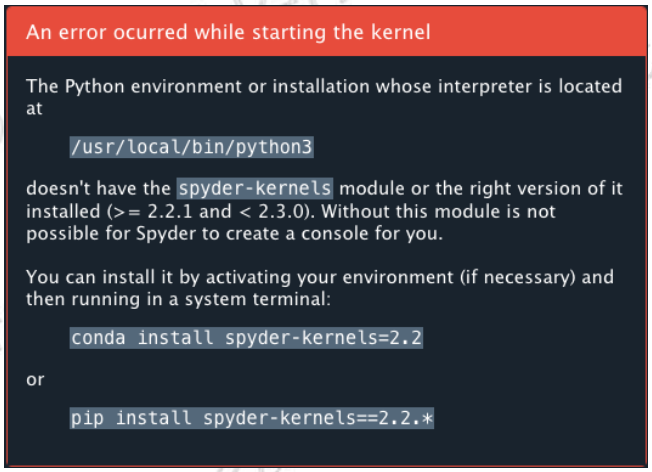
El problema está en que PyCharm Professional vale la friolera de unos cuantos cientos de dólares pero si eres estudiante te lo dan gratis. Solo tienes que seguir los pasos que se indican en https://www.jetbrains.com/es-es/community/education/#students. Si quieres, puedes solicitar la licencia rellenado los datos y pon como año de graduación 3 o 4 más del año actual. Recibirás un correo de "JetBrains account". Sigue las instrucciones del correo y recibirá un correo de "JetBrains sales" para activar tu paquete gratuito donde debes pulsar "click this link". Echo todo esto, solo debes entrar en tu cuenta de JetBrains y descargar PyCharm Professional. Una vez instalado, inicia tu cuenta desde el propio programa para obtener el "code with me"[3].

Instalación de Spyder

1. Puede instalar el IDE desde Anaconda. Siga los pasos para ese entorno.
   - Si usa Linux, la recomendación es usar Anaconda, https://docs.spyder-ide.org/current/installation.html#anaconda-install-ref, aunque puede seguir los pasos que se indican para instalarlo directamente en el sistema.
2. Alternativamente puede instalar el IDE desde la opción Download de https://www.spyder-ide.org/

[3] Estos pasos fueron facilitados por el alumno Pablo Roques Villa durante el curso 2021-2022

- En Windows, si aparece un cuadro de diálogo SmartScreen, haga clic en *More Info* seguido de *Run away* y luego continúe con los pasos del instalador.
- Si al arrancar Spyder obtiene este error:



  deberá de instalar spyder-kernels (p.e. `pip3 install spyder-kernels==2.2.1`). Los kernels instalarán los paquetes ipython (shell python interactivo), jupyter (notebooks), pygments (coloreado de sintaxis), tornado (servidor web) y otros paqeutes que permitirán dar mayor funcionalidad a Spyder.
  Si usa Anaconda, no debería presentar este error, pero si instaló Python en su SO y debe instalarlo, conviene usar un entorno virtual y no trabajar directamente sobre la versión Python del SO. No obstante, esta tarea puede ser muy difícil en algunos sistemas como maxOS (p.e. la versión homebrew no funciona en Mojave).

Instalación de Visual Studio Code y Plugins

1. Instalar el editor desde `https://code.visualstudio.com/`
2. En el menú de la izquierda seleccionar Extensions y a continuación buscar la extensión **Python**.
3. Instalar el extensión **Python de Microsoft** y a continuación pulsar Reload.
4. Instale también la extensión **Pylance** para mejorar la extensión anterior. Vuelva a recargar.
5. Abrir la paleta de comandos ⌘+⇧+P o Ctrl+⇧+P y teclear **Shell Command: Install 'code' command in PATH** con lo que tendremos el comando nuevo **code** en nuestra sistema operativo.
6. En la parte inferior izquierda aparece la versión del interprete **Python** que se está utilizando. Se puede cambiar haciendo click para a continuación seleccionar la versión de **Python** deseada.
7. Como plugins complementarios, instalar:
   - **vscode-icons** de VSCode Icons Team (recomendado)
   - **Python DocString Generator** de Nils Werner
   - **AI Python Docstring Generator** Tae Hwan Jung
   - **Python Preview** de dongli
   - **PlantUML** de jebbs
   - **Markdown Preview Enhanced** de Yiyi Wang
   - **Visual Studio IntelliCode** de Microsoft
   - **VSC-Prolog** de arthurwang

1.4 Sesión de trabajo

1.4.1. Usando Editores

Siga siempre los siguientes pasos para trabajar un proyecto desde su inicio:

1. **Creación del directorio principal del proyecto.** Esto se hace una vez.
   Construya una carpeta donde vaya a guardar todo los referente a su proyecto **Python**. Aquí se guardará tanto los ficheros fuentes como la documentación del proyecto. Llamaremos 🖿`Proyecto` a esta carpeta pero en la práctica pon un nombre significativo al proyecto que vayas a desarrollar.
2. **Creación del entorno virtual.** Esto se hace una vez.
   Con **Python 3** se suministra el módulo **venv** que permite crear entornos virtuales con la siguiente instrucción (se hace una vez):

```
$ python3 -m venv venv
```

Con lo que tendremos esta nueva estructura de directorios 🗀 `Proyecto` > `venv`

3. **Activación del entorno virtual.**
   Para poder usar el ambiente hay que activarlo. La instrucción es levemente diferente dependiendo del sistema operativo

   - UN*X

     ```
     $ source venv/bin/activate
     ```

   - Windows

     ```
     $ source venv/Scripts/activate  % Si usa bash
     $ venv/Scripts/activate  % Si usa cmd
     ```

   Entonces el nombre del ambiente virtual aparecerá en el prompt.
4. **Trabajar sobre el proyecto.**
   Activado el entorno virtual puede trabajar exactamente igual que si estuviera en el entorno general. Podrá utilizar un editor de código cualquiera con el que escribir sus programas y ejecutarlos

   ```
   $ python3 Proyecto/main.py
   ```

   También podrá instalar tantos paquetes como requiera su proyecto

   ```
   $ pip3 install paquete
   ```

   todos se guardarán en 🗀 `Proyecto` > `venv` y no le afectará a otras librerías de otros proyectos.
5. **Desactivar el ambiente virtual.**
   Cuando finalice, vuelva al ambiente **Python** general que tenga por defecto su sistema. Para ello desactive el ambiente actual.

   ```
   $ deactivate
   ```

1.4.2. Sesión de Trabajo con PyCharm

Cuando se trabaja con ciertos IDEs los pasos anteriores se realizan con unos pocos clicks de ratón.

1. **Creación del directorio principal del proyecto, creación del directorio del proyecto fuente y activación del entorno virtual**

   - Seleccionar File > New Project
   - En la nueva ventana, en la opción Location pinchar sobre el icono 🗀
     - Seleccionar la carpeta donde crearás todos tus proyectos.
       En su caso selecciona New Folder.
     - Cuando termine seleccionar Open

     Supondremos que la carpeta se llama 🗀 `Proyecto`.
   - En la misma ventana considere la opción New environment
     - Seleccione el valor **Virtualenv**
     - En la opción Location
       - Pinchar sobre el icono 🗀
       - Selecciona New Folder y crea el directorio para el entorno virtual.
         Supongamos que se llama 🗀 `venv`
       - Seleccionar Open
   - En la misma ventana, asegúrese que está activa la opción Create a main.py.
   - Seleccionar Create.
   - En su caso, seleccionar Open Proyect con New Windows.

   Cuando finalice estos pasos se creará un nuevo proyecto donde verá el siguiente contenido:

   🗀 `Proyecto` ▸ `venv`
   🗀 `Proyecto` ▸ `main.py`
2. **Trabajar sobre el proyecto.**

En 🗀 Proyecto deberá crear todo su código fuente. De todos los ficheros habrá uno que se llamará 🗀 main.py que será el módulo principal - lo construyó antes ¿recuerda? Lo verá también porque en la barra superior aparece precisamente el nombre main▽

Para ejecutar el proyecto puede usar estas alternativas.

- Pinche sobre ▶ en la barra superior
- Pinche sobre ▶ en el editor, en la linea donde aparezca `if __name__ == '__main__':`
- Seleccione Run > Run 'main'

3. **Desactivar el ambiente virtual.**
   La versión más sencilla de esta acción es cerrar el proyecto.

1.5 Documentar el Código del Proyecto

Documentar un programa es escribir las especificaciones de las abstraciones implementadas. Consiste en añadir información en el código fuente. Se realiza en forma de comentarios (docstrings) para explicar qué es lo que hace para que los usuarios programadores de su programa entiendan su funcionamiento. Además, como ayuda a su entendimiento permite detectar errores y extender el programa para tenga nuevas funcionalidades. Todo programa tiene errores (es cuestión de tiempo) y todo programa que tenga éxito será modificado en el futuro, por lo que la documentación es un paso fundamental.

Los programas en Python se documentan con tres comillas dobles o con la almohadilla. Las tres comillas se usan como docstring (para las definiciones de las clases y las especificaciones de los métodos junto con las aclaraciones sobres los mismos). La # se usa para hacer comentarios entre líneas del código no trivial (no para comentar todo lo que se hace).

Para documentar el proyecto con PyCharm con docstrings debe seguir los siguientes pasos

1. Determinar cuál será el formato del Docstring.
   - PyCharm > Preferences > Tools > Python Integrated Tools
   - Seleccionar en Docstring el formato reStructuredText.
2. Construir los docstring de cada función.
   El docstring es una especificación informal de la función. Consta de un pequeño resumen del objetivo de la función, una breve descripción de los parámetros de entrada y de salida. Para construirlos en PyCharm basta empezar una nueva línea con 3 dobles comillas `"""` justo después de la signatura de la función, pulsar ⏎ y completar la información.
   Un ejemplo de docstring es el siguiente:

```
def check_fin(mensaje):
    """
    Realiza una pregunta de confirmación.
    Se dan dos opciones 1 y 2 en el mensaje de entrada.

    :param mensaje: El mensaje de la pregunta
    :return: True si se eligió 1, False en otro caso.
    """
    return '1' == input(f"{mensaje}\n1. sí\n2. no\n")
```

En el caso de querer indicar también el tipo de dato de los parámetros y retorno, deberá seguir los siguientes pasos:

1. Ir a la opción PyCharm > Preferences > Editor > General > Smart Keys > Python
2. Seleccionar Insert type placeholders in ...
3. Confirmar OK

Con esto es suficiente para que, durante el proceso de programación, pueda consultar la especificación de la función/método sin necesidad de ir a la parte del módulo donde escribió el docstring. Si escribe una función documentada y coloca el cursor del ratón sobre el nombre de la función recién escrito verá su docstring.

1.6 Generar la Documentación de un Proyecto

En ocasiones nos interesa tener un documento completo con todas las especificaciones de todas las funciones para poder hacer un estudio más concienzudo de las funcionalidades. Comentemos varias formas para hacer esto.

1.6.1. Sin crear ficheros externos

Python incorpora pydoc para visualizar todos las funciones y clases de su proyecto. Teclee lo siguiente en el terminal.

```
$ cd Proyecto   # Confirme que se encuentra en el proyecto con ficheros .py
$ python3 -m pydoc -p 2334
$ Server commands: [b]rowser, [q]uit
server> b
```

Se abrirá un navegador y podrá consultar la documentación. Para terminar escriba q.

```
$ Server commands: [b]rowser, [q]uit
server> b
server> q
```

1.6.2. Creando ficheros externos

En este caso es mejor usar otras herramientas como pydoctor o sphinx. En ambos casos tendrá que realizar los siguientes pasos

1. Instalar los nuevos módulos en el entorno virtual.
   Desde el IDE lo puede hacer en PyCharm > Preferences > Project: Nombre Proyecto > Python Interpreter
   - Asegurése de que se tiene seleccionado el intérprete del entorno virtual, en Python Interpreter
   - Seleccionar +, buscar pydoctor, pdoc3 y Sphinx e instalar los paquetes.

   Alternativamente puede abrir un terminal y con el entorno virtual activo, ejecutar

   ```
   $ pip3 install pydoctor
   ```

   o bien

   ```
   $ pip3 install pdoc3
   ```

   o bien

   ```
   $ pip3 install Sphinx
   ```

2. Asegúrese de que tienen esta estructura de directorios

   🗀 Proyecto, lugar donde está sus programas .py comentados con sus especificaciones.
   🗀 Proyecto ▸ docs, lugar donde estará toda la documentación referente a la resolución del problema (enunciados, ayudas, etc).
   🗀 Proyecto ▸ docs ▸ api, el lugar donde se generará la documentación del programa. Se asume que no contiene nada.

1.6.2.1. Cómo usar pydoctor

Siga los siguientes pasos:

1. Teclee en el terminal

   ```
   $ cd Proyecto   # Asegúrese que está en el directorio del proyecto
   $ touch programa/__init__.py  # Convertir el programa en módulo (importante)
   $ pydoctor --make-html --html-output=docs/api --docformat=restructuredtext programa
   ```

2. Visualizar la documentación.
   Con un navegador web abra el fichero 🗀 Proyecto ▸ docs ▸ api ▸ index.html

1.6.2.2. Cómo usar pdoc3

Sigue los pasos de pydoctor. Solo tienes que cambiar la línea de comandos para generar la documentación.

```
$  pdoc3 --html --output-dir html programa
```

1.6.2.3. Cómo usar sphinx

Se presenta a continuación un resumen de los que puede encontrar en el siguiente vídeotutorial:

https://youtube.com/playlist?list=PL__ig9hYf07AdLfTOMrWwSvYeUdWUY3Zt

También dispones del paquete `documentationPythonProject.zip` en el aula virtual para que sigas los pasos del tutorial.

Los pasos a seguir son:

1. Teclee en el terminal

   ```
   $ cd Proyecto/doc/api   # Asegúrese que está en el directorio doc/api del proyecto
   $ sphinx-quickstart
   ```

   Para Mac: Si al ejecutar `sphinx-quickstart` diera problemas, deberá seguir los siguientes pasos:

   ```
   $ open /usr/local/bin/sphinx-quickstart -a textedit
   cambiar la primera línea por #!/usr/local/bin/python3
   Archivo > Guardar
   Archivo > Cerrar
   ```

2. Introducir los siguientes datos:
   - **Separar directorios fuente y compilado (y/n) [n]:** Enter
   - **Nombre del proyecto:** Pon un nombre adecuado Enter
   - **Autor(es):** Pon tu nombre Enter
   - **Liberación del proyecto []:** Enter
   - **Lenguaje del proyecto [en]:** `es` Enter

   Se habrá generado el siguiente contenido en 🗀 `Proyecto > doc > api`:

   ```
   Makefile
   _build
   _static
   _templates
   conf.py
   index.rst
   make.bat
   ```

3. Editar el fichero `conf.py`
   - Descomentar las siguientes líneas:

     ```
     # import os
     # import sys
     # sys.path.insert(0, os.path.abspath('.'))
     ```

     y de la tercera, cambiar `'.'` por `'../..'` para referenciar a 🗀 `Proyecto` (lugar donde están los fuentes `.py`).
   - Además, añadir la siguiente extensión (con comillas simples):

     ```
     extensions = ['sphinx.ext.autodoc']
     ```

4. Editar el fichero `index.rest` y adaptarlo a tu gusto. Lo más básico sería poner `modules`:

   ```
   .. toctree::
      :maxdepth: 2
      :caption: Contents:

      modules
   ```

   Puede cambiar los valores de `caption` y `maxdepth` si lo desea.

5. Ejecutar en el directorio `docs/api` la instruccion `sphinx-apidoc -o . ../..`

   Lo que indica esta instrucción es que busque de forma recursiva en `../..` (directorio 🗀 `Proyecto`) todos los módulos y paquetes de **Python** y cree un archivo reST por cada uno de ellos en el directorio `.` (directorio 🗀 `Proyecto▸doc▸api`).

6. Por último generamos la documentación. En UN*X

   ```
   make html
   ```

   Para Windows use `make.bat`

7. Visualizar la documentación.
   Con un navegador web abra el fichero 🗀 `Proyecto▸docs▸api▸_build▸_static▸index.html`

Opcionalmente puede cambiar la estética de la documentación.

1. En su entorno virtual ejecute `pip3 install sphinx_rtd_theme` para instalar uno de los muchos temas disponibles.
2. Edite `conf.py`
3. Cambie la entrada `html_theme = 'sphinx_rtd_theme'`
4. Genere de nuevo la documentación.

Tiene más temas en https://pypi.org/search/?q=sphinx+theme&o= pero no pierda el tiempo en la estética.

Ejercicio 1.1

- Modifique el formato del Docstring.
  - PyCharm > Preferences > Tools > Python Integrated Tools
  - Seleccionar en Docstring el formato Google.
- Borre el docstring de una función que tenga parámetros y valor de retorno.
- Vuelva a escribirlo con 3 comillas dobles `"""` justo después de la signatura de la función y pulse ↵.
  Verá que el formato Google es diferente al formato reStructuredText.
- Vuelva a generar la documentación con `pydoctor`.

El formato Google es más legible en una una documentación pero también ocupa más líneas de texto en el código lo que para funciones cortas de pocas líneas puede ser un inconveniente.

1.7 Gestión de Algunos Errores en PyCharm

1.7.1. Resolver el problema de Paquetes en PyCharm

A algunos de vosotros os ocurre que algún módulo (fichero .py) no reconoce algún otro módulo del proyecto. Para solucionar esto sigue los siguientes pasos.

Memoriza la teoría. En las transparencias se indica que se debe tener un fichero vacío `__init__.py` para poder importar todos los módulos de una carpeta. Es decir, hay que construir un paquete con sus módulos. La construcción de paquetes desde PyCharm se hace seleccionando File > New > Python Package. Simplemente crea un carpeta con el fichero `__init__.py`. Observa que todas las carpetas menos venv están en el mismo color. Si además tienen un punto sobre la carpeta, entonces esa carpeta es un paquete.

En principio, solo con lo anterior deberías poder importar los módulos de los paquetes. Pero si te siguiera saliendo el mensaje `Unresolved Reference XXXX` entonces sigue los siguientes pasos:

- Marca el paquete como una fuente raíz.
  Pon el ratón sobre la carpeta y pulsa botón derecho del ratón. Selecciona Mark Directory as > Sources Root. La carpeta cambiará de color.
- Añadir la nueva al `PYTHONPATH`.
  Selecciona Preferences > Build, Execution, Deployment > Console > Python Console y asegurarte de marcar las dos opciones de `Add content` y `Add source`.
- Limpia la caché y reiniciar PyCharm. Selecciona File > Invalide Cache > Restart.

Ya no deberías tener más problemas de reconocimiento de los módulos de los paquetes.

IMPORTANTE:

Algunos, para solucionar el problema habrán seleccionado Install and Import package en vez de hacer la importación como se indica aquí. Si tras realizar esa acción PyCharm ha dejado de quejarse, **tienes un problema**. Habrán instalado unos paquetes extras en venv y ahora tu program entiende que debe importar esos paquetes en vez de los que tú has desarrollado. Obviamente tendrás que quitar esos paquetes extras que no os sirven para nada en la resolución del ejercicio.

Posiblemente alguno de ellos son: `Point`, `Agent`, `Vector`, `Vector2`, `state`, ... La forma de comprobar que has instalado unos paquetes que NO deberías de haber instalado es seleccionar Preferences > Project > Python Interpeter para ver el listado de los paquetes que has instalado. Si hay alguno cuyo nombre coincide con algunos de los nombre que te indico o con el nombre de algún módulo/clase que tú hayas construido tendrás que eliminar el paquete porque muy probablemente tu programa estará usando ese paquete en vez del tuyo.

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Sesión 2

Abstracción

Índice Parcial **Página**

2.A. Conceptos Básicos . . . 11
2.B. Mecanismos de Abstracción . . . 12
2.C. Tipos de Abstracción . . . 13
2.C.1. Abstracción Procedimental . . . 13
2.C.2. Abstracción de Datos . . . 14
2.D. Tipos de Datos Abstractos . . . 14
2.D.1. Especificación de Datos Abstractos . . . 15
2.D.1.1. Especificación Formal . . . 16
2.D.1.2. Especificación Informal . . . 17
2.D.2. Representación de TDAs . . . 18
2.D.3. Implementación de TDAs . . . 19
2.E Python . . . 20
2.E.1 Breve Historia de Python . . . 20
2.E.2 Características y Uso . . . 21
2.E.3 Variables, Expresiones y Conversión de Tipo . . . 22
2.E.4 Programación Estructurada . . . 23
2.E.5 Programación Procedimental . . . 25
2.E.6 Programación Modular . . . 28
2.E.7 Programación Orientada a Objetos . . . 28
2.1. Un clase para números positivos . . . 29
2.2. TDA Fecha Simple . . . 30
2.3. TDA Fecha Simple con Positivo . . . 31
2.4. Un clase para cilindros . . . 31
2.5. TDA Fracción . . . 31
2.6. Adivina el número . . . 32

Teoría

2.A Conceptos Básicos

La **abstracción** es el proceso mental por el que captamos las características principales de un concepto o proceso descartando los detalles. El proceso aísla conceptualmente las distintas partes, propiedades o cualidades de un objeto para estudiarlas por separado. En definitiva, categorizamos elementos en grupos y cada grupo es una abstracción que ignora determinadas características de los elementos del grupo. Cada grupo resalta los aspectos relevantes del objeto de estudio.

La abstracción permite estudiar un sistema complejo a diferentes niveles de detalle; es decir, refleja un modelo jerárquico. Si se prefiere, la abstracción permite hacer una descomposición en la que varía el nivel de detalles. Es por ello que la abstracción se usa en muchas áreas de la ciencia de la computación para reducir la complejidad de los problemas/tareas a niveles que sean manejables. De hecho la programación modular es una metodología que aborda la descomposición de problemas basándose en la abstracción.

Las propiedades de una descomposición útil son:

- Cada uno de los niveles de abstracción deben de estar al mismo nivel.
  Por ejemplo, consideremos como objetos las casas que constan de distintos elementos. Una abstracción de esos elementos es 'Cocina" y otro nivel de abstracción es "tornillo de la puerta del dormitorio". Hemos dividido conceptualmente dos grupos de elementos de una vivienda pero una cocina tiene un nivel de abstracción mayor que el tornillo de la puerta. El primero obvia más detalles que el segundo pues éste es mucho más concreto. En una cocina siguen quedando muchos más elementos que podrían seguir estudiándose por separado y si se sigue el proceso de "descomposición" llegaríamos a elementos tan irreducibles como el tornillo de la puerta del dormitorio. En términos de programación significa que cuando se divida un programa en módulos convienen que todos los módulos se encuentren al mismo nivel de abstracción, y no que unos traten problemas muy simples y otros problemas muy complejos.
- Cada parte debe poder ser abordada por separado, sin la necesidad de depender del resto de los grupos.
  En término de programación modular significa que cada módulo deber ser cohesivo: debe llevar a cabo una tarea bien definida en el nivel de detalle que le corresponda.
  y no debe de existir (demasiado) acoplamiento.
- La solución de cada parte debe poder unirse al resto para obtener la solución final.
  En término de programación modular significa, por un lado, que los módulos no deben de estar acoplados. Obviamente los módulos no pueden ser independientes entre sí porque entonces serían partes independientes del problema, así que debe de estar conectados. Pero las conexiones de cada módulo con el resto del programa deben ser las imprescindibles, las mínimas. Cada módulo debe ofrecer a los demás los procedimientos/funciones de lo que hace pero no aquellas partes de cómo lo hacen.
  Por otro lado significa que las soluciones que ofrece un módulo se aportan a otros módulos y éstos a unos terceros y así sucesivamente, y todo a través de esas conexiones mínimas. La composición de las soluciones de los subproblemas de cada módulos, una vez combinadas, permiten resolver el problema original.

2.B Mecanismos de Abstracción

Existen dos mecanismos para realizar la abstracción

- **Abstracción por parametrización.** Está relacionada con la abstracción procedimental.
  Es cuando se introducen parámetros para abstraer valores concretos. Un parámetro representa a un conjunto de elementos específicos. Por ejemplo, todos los números naturales pueden quedar identificados con esta declaración `int num`. Esto nos permite identificar `num` con el 1, 2, ...
  Ya que los elementos específicos se pueden usar en procedimientos concretos, esta abstracción también nos permite usar parámetros en los procedimientos. Así, al introducir parámetros abstraemos un número infinito de cálculos. Por ejemplo, en vez de usar `trasladarAlPto(persona, (1, 3))` se puede usar `trasladarAlPto(Objeto obj, Punto p)` lo que nos permite trasladar cualquier objeto a un punto del plano. Notar también que también nos abstraemos de la ejecución concreta. Nos da igual cómo se haga el traslado, solo sé que si le doy un objeto y un punto lo hará correctamente.
- **Abstracción por especificación.** En esta abstracción incluimos más detalles que en la abstracción por parametrización. Es cuando asociamos a la abstracción un documento en el que indicamos
  - Un nombre. Por ejemplo el nombre de un procedimiento.
  - Una descripción precisa. Describe en qué consiste la abstracción pero solo menciona lo imprescindible. Por ejemplo, la descripción del comportamiento de un procedimiento.
  - Las condiciones. Los estados que deben cumplirse.
    Si la abstracción es procedimental, podemos conseguir una especificación introduciendo detalles como precondiciones/requisitos, postcondiciones/efecto.
    - Precondiciones o requisitos. Las condiciones exigidas para que el procedimiento se comporte como se prevée.
    - Postcondiciones. Las afirmaciones que serán ciertas si se cumplieron las precondiciones. Podemos dividirlas en varias partes según los detalles de la abstracción a realizar.

  Además, la especificación debe ser legible lo que quiere decir que debe ser entendible.

  Por ejemplo, el siguiente código no funcionará siempre

```
def inverso (valor: double) -> double:
    el_inverso = 1/valor
    return el_inverso
```

por lo que requiere de una especificación En este caso, una especificación para esta función puede ser

```
def inverso (valor: double) -> double:
    """
    Función inverso: calcula el inverso de un número real.
    Descripción: devuelve una aproximación del inverso calculando 1/valor
    Requisitos: el valor de entrada deber ser un valor real no nulo
    Modificaciones: no realiza ninguna modificación de ningún parámetro
    Retorno: un valor real
    """
    el_inverso = 1.0/valor
    return el_inverso
```

Notar que aquí hemos dividido las postcondiciones en: descripción, requisitos, modificaciones y retorno.

2.C Tipos de Abstracción

Podemos distinguir tres tipos de abstracción:

- **Abstracción procedimental.** Es la que consiste en definir un conjunto de procedimientos como abstracción de operaciones.
  Una suma no es solo la suma de dos números se puede abstraer para definir suma como una operación de agregación de cantidades u objetos, con lo que se puede hablar de suma de cadena de caracteres, matrices, listas, ...
- **Abstracción de iteración.** Es la que permite trabajar sobre colecciones de objetos y permite pasar de un elemento al siguiente sin saber cómo se organizan de forma interna.
  Por ejemplo, el recorrido de un array usando un bucle `while(i<len(vector))` se puede generalizar a **while(vector.hasNext())** donde ya no depende del índice.
- **Abstracción de datos.** Es la que consiste en considerar una conjunto de objetos y un conjunto de operaciones (procedimientos) sobre los objetos, lo que los dota de un comportamiento.
  Por ejemplo, en matemáticas un grupo es una estructura algebraica formada por un conjunto no vacío y una operación interna que combina cualquier par de elementos para obtener un tercero dentro del mismo conjunto y que satisface las propiedades asociativa, existencia de elemento neutro y simétrico. No importa cómo son los objetos de la estructura.

2.C.1 Abstracción Procedimental

La abstracción precedimental consiste en crear **procedimientos y funciones** como abstracciones de las operaciones. Es decir, a partir de un conjunto preciso de operaciones abstraemos una única operación en forma de función o procedimiento para expresar el ***qué*** hacen. Las técnicas de Programación Estructurada y Refinamiento por Pasos Sucesivos usan mucho este tipo de abstracción para romper el problema en problemas más pequeños.

Toda operación tiene asociada una sintaxis y una semántica:

- Sintaxis: cómo se escribe. Es una signatura que consta del nombre de la operación, los parámetros (número, orden y tipo) y lo que devuelve.
- Semántica: qué significa. Establece las condiciones que deben cumplir los argumentos y el efecto de la operación.

Los operadores permiten claramente una abstracción por especificación, pues se necesita de un documento que permita conocer los aspectos sintácticos y semánticos para poder usar la función/procedimiento y conocer lo que va a realizar, respectivamente. No nos interesa saber cómo funciona.

Hemos visto que, en general, una especificación consta de unas precondiciones y unas postcondiciones. Para estructurar mejor la información se pueden usar estas cláusulas:

- **Nombre.** Hace referencia al nombre de la operación y que debe ser un nombre representativo de la operación que realiza.
- **Parámetros**. Explica el significado de cada uno de los objetos de entrada indicando su tipo.
- **Retorno.** Describe el valor que retorna indicando su tipo, solo para funciones.
- **Precondiciones.** Establecen las restricciones de uso. Son prerrequisitos para su correcto funcionamiento por lo que, de no cumplirse, no se asegura un comportamiento correcto del operador.
- **Descripción** o efecto. Descripción textual del comportamiento de la operación (el efecto), si se cumplen las precondiciones. En su caso, una descripción de la combinación de las operaciones empleadas y la creación de nuevos objetos a partir de los argumentos.
- **Modifica.** Indica los parámetros de entrada que se modificarán. Esta cláusula puede ir en la descripción.

- **Excepciones.** Describe el comportamiento del método/función cuando las precondiciones no se cumplen. No es una cláusula obligatoria.

Existen herramientas que ayudan a la especificación de los procedimientos como por ejemplo, **doc++**[1], **doxygen**[2], etc ... Todos funcionan de forma similar. Se trata de poner comentarios en el programa usando marcas que ayuden a la herramienta a detectar que se tiene una especificación del procedimiento. Entonces exporta el contenido del contenido comentado junto con la signatura del método a un formato legible como HTML.

En lo que a nosotros respecta, la especificación informal de una procedimiento/función se puede establecer de esta manera:

```
operacion nombre (ent id:tipo, ....) return tipo
    Precondiciones: Indica los datos, id, de entrada que se necesitan
    Retorna: Indica el tipo de dato que retorna
    Descripción: Descripción textual del comportamiento de la operación (el efecto)
    Excepciones: Indica las excepciones (opcional)
```

2.C.2 Abstracción de Datos

El tercer tipo de abstracción es la abstracción de datos es la que permite trabajar con datos con una serie de operaciones (es la que más nos interesa. Podemos distinguir 3 niveles de abstracción:

- **Tipos de datos integrados.** Son los tipos de datos que ofrecen los lenguajes de programación, que también reciben el nombre de tipos fundamentales.

  En Python se distinguen:

  - Los tipos de datos simples: numéricos (enteros, reales y complejos) y booleanos.
  - Los tipos de datos compuestos: cadena de caracteres (strings), secuencias (rangos, listas y tuplas), mapas (diccionarios), conjuntos.

  Otros lenguajes de programación solo contemplan como tales a los tipos de datos simples.
- **Tipos de datos definidos por el usuario** o programador. Son los que pueden construir los programadores agrupando datos de diferentes tipos.

  Es usual usar lo que se llama una estructura (struct), en cuyo caso los datos recibe el nombre de campos. Unos ejemplos son los siguientes.

```
struct Fecha
{
    int dia;
    char mes[10];
    int anno;
}
```

```
struct Persona
{
    Fecha fecha;
    char nombre[45];
}
```

- **Tipos de datos abstractos.** Este tipo de abstracción no se limitan como las estructuras a considerar una agrupación de los datos sino que los datos se contemplan como modelos a los que se les agregan propiedades operacionales mediante una especificación. Un TDA es una entidad abstracta formada por datos y operaciones.

  Si consideramos un conjuntos de datos como un array 2D esto sería una estructura. Si disponemos de dos array 2D de la misma dimensión podríamos hacer una suma de datos término a término y esto no sería más que una operación sobre dos datos estructurados. Pero si abstraemos estos datos a nivel de modelo podríamos hablar de la entidad matriz y de la operación suma de matrices. Desde este nuevo nivel de abstracción el cómo están almacenados los datos es irrelevante (p.e. podría ser un array 2D u otra estructuras basadas en tuplas o listas)

2.D Tipos de Datos Abstractos

Los Tipos de Datos Abstractos son modelos matemáticos que constan de un nombre publico para identificar a un conjunto de datos (valores) junto con un conjunto de operaciones bien definidas sobre los datos (como una estructura algebraica). Como modelos, es irrelevante cómo se almacenan los datos y cómo se implementan las operaciones.

Para todo TDA hemos de abordar tres tareas:

- Especificación: Descripción del TDA.
- Representación: Forma concreta en que se representarán los datos.
- Implementación: La forma específica en que se codifica la representación en un lenguaje de programación.

[1] http://docpp.sourceforge.net
[2] https://www.doxygen.nl

2.D.1 Especificación de Datos Abstractos

La especificación de un TDA no es diferente a las condiciones generales de cómo debe ser una especificación, solo que ahora hemos de tener en cuenta lo siguiente.

- La especificación debe ser general; es decir, no se puede pensar en tipos de datos concretos o estructuras de datos concretas. No piense en **int** sino en $\mathbb{N}$ o **Numeric**.
- Un TDA consta de dos partes: datos y operaciones. Por tanto aparecerán dos partes en la especificación.
  - Definición. En esta parte se describe o define el TDA, los términos que sean necesarios para comprender el resto de la especificación así como el listado de otros TDAs que se puedan necesitar para entender su definición y operaciones.
    En concreto para su definición se tendrá que indicar el dominio de los nuevos objetos que modela. Se puede recurrir a notaciones matemáticas conocidas como, por ejemplo, usando la notación de conjuntos $\{s_1, s_2, \ldots\}$, notación de intervalos $[a, b]$, expresiones regulares, ... o bien indicando el conjunto de reglas que permiten construir los objetos como, por ejemplo, una lista o es un secuencia de 0-elementos o es una lista seguida de 1-elemento.
  - Operaciones. En esta parte se indica tanto la sintaxis como la semántica de las operaciones del modelo.
    Como esta parte hace referencia a una abstracción operacional sobre datos abstractos, se emplearan las especificaciones operacioneales en los términos que hemos indicado anteriormente como pre/post-condiciones o un esquema más detallado.

El conjunto de operaciones se pueden dividir en los siguientes grupos:

- Operaciones de creación (o constructoras). Son las que determinan cómo se puede crear e inicializar el tipo de dato. Permiten crear un ente del modelo sobre el que se puede empezar a trabajar.

```
Matriz A = Matriz()
Matriz B = Matriz.identidad()
```

- Operaciones de modificación (o combinación). Son las que permiten alterar el estado del TAD o de los datos que contiene.

```
lista.add(elemento)
lista.cambia(posicion, nuevo_valor)
```

- Operaciones de consulta (u observación). Son las que informan sobre el estado del TAD o de los datos, sin crear modificación alguna.

```
lista.size()
lista.contiene(valor)
```

- Operaciones de persistencia. Son las que permiten preservar el estado del TAD de forma permanente (guardarlo) así como los que permiten recuperar el estado del mismo (leerlo).

```
matriz.save("salida.dat")
lista.read("entrada.dat")
```

Una tarea importante en la definición de un TDA es establecer las operaciones que se usarán para manejar los conjuntos de datos. Aunque es una tarea que dependerá de los problemas que se quieran resolver con el TDA, puede considerar estas reglas generales:

- Determinar las operaciones fundamentales.
  Debe existir un número mínimo de operaciones que garanticen la abstracción. A estas operaciones las llamaremos **primitivas** y cumplen estas dos condiciones:
  - La supresión de una de ellas conlleva que encontramos problemas que no se pueden resolver porque nos faltan operaciones.
  - Ese conjunto de operaciones permite construir cualquier otro tipo de operación.
- Determinar algunas operaciones no fundamentales.
  No se deben de incluir un exceso de operaciones. No tiene sentido incluir operaciones que nunca se usarán o son poco útiles, además de que complica el mantenimiento a nivel de implementación. Incluya, al menos, aquellas que se hagan con más frecuencia.

2.D.1.1 Especificación Formal

Una especificación formal es una especificación axiomática y algebraica de un TDA, llámese T, junto con las funciones que define sus operadores, en concreto:

- Constructores. El conjunto de operaciones del TDA que permite construir un valor de tipo T.

$$c_1 :\longrightarrow T, \qquad c_2 : V_1 \longrightarrow T, \qquad c_3 : V_1 \times V_2 \longrightarrow T, \qquad \ldots \qquad c_n : \prod_{i=1}^{n-1} V_i \longrightarrow T$$

- Modificación. El conjunto de operaciones del TDA que permite construir un nuevo valor de tipo T a partir de un valor de tipo T dado, pero que no son constructores.

$$m_1 : T \longrightarrow T, \;\; m_2 : T \times V_1 \longrightarrow T, \;\; m_3 : T \times V_1 \times V_2 \longrightarrow T, \ldots m_k : T \times \prod_{i=1}^{k-1} V_i \longrightarrow T$$

- Consulta. El conjunto de operaciones que a partir de un valor de tipo T retornan un valor, que no es de tipo T.

$$q_1 : T \longrightarrow V^1, \;\; q_2 : T \times V_1 \longrightarrow V^2, \;\; q_3 : T \times V_1 \times V_2 \longrightarrow V^3, \ldots q_r : T \times \prod_{i=1}^{r-1} \longrightarrow V^r$$

  donde el superíndice debe entenderse como un índice y no como una potencia.

En ocasiones se indica la semántica para cada operación.

Ejemplo 2.1

a especificación formal de los números naturales puede establecerse de esta forma.

TDA Número Natural

- Representa cantidades enteras no nulas.
- Usa: booleanos ($\mathbb{B}$).
- Sintaxis:
  - $cero \to \mathbb{N}$ /*El cero es natural*/
  - $sucesor : \mathbb{N} \to \mathbb{N}$ /*El siguiente es natural*/
  - $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ /*La suma*/
  - $escero : \mathbb{N} \to \mathbb{B}$ /*comprueba si es el cero*/
  - $igual : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ /*comprueba si son iguales*/
- Semántica:
  - Variables
    - n, m: de tipo natural
    - $true, false$: los valores booleanos
  - Ecuaciones
    - $n + m = \overbrace{sucesor(..(sucesor(n))...)}^{m}$
    - $sucesor(n) = n + 1$
    - $escero(n) = \begin{cases} true & \text{si } n \text{ es } cero \\ false & \text{si } n \text{ no es } cero \end{cases}$

Ejemplo 2.2

a especificación formal del TDA fecha podría ser la siguiente:

TDA Fecha

- Representa fechas válidas de acuerdo al calendario Gregoriano.
- Usa: naturales ($\mathbb{N}$), booleanos ($\mathbb{B}$)
- Sintaxis:
  - $crear : [1 \ldots 31] \times [1 \ldots 12] \times \mathbb{Z} - \{0\} \longrightarrow Fecha$
  - $anterior : Fecha \longrightarrow Fecha$

- $siguiente: Fecha \longrightarrow Fecha$
- $suma: Fecha \times \mathbb{Z} \longrightarrow Fecha$
- $iguales: Fecha \times Fecha \longrightarrow booleano$
- $da: Fecha \longrightarrow \mathbb{N}$
- $mes: Fecha \longrightarrow \mathbb{N}$
- $año: Fecha \longrightarrow \mathbb{Z} - \{0\}$
- ...etc...

▪ Semántica:

- Variables
  - dd, mm, aa son de tipo natural, representan día, mes y año.
  - $fecha$: $dd/mm/aa$
  - $true, false$: los valores booleanos
- Ecuaciones
  - $crear(dd,\ mm,\ aa) = dd/mm/aa$
  - $dia(fecha) = fecha.dd$
  - $mes(fecha) = fecha.mm$
  - $año(fecha) = fecha.aa$
  - $anterior(fecha) = \begin{cases} (dd-1,\ mm,\ aa) & \text{si } fecha.dd > 1 \\ (lastdia(mm-1),\ mm-1,\ aa) & \text{si } fecha.dd = 1 \text{ y } fecha.mm > 1 \\ (31,\ 12,\ aa-1) & \text{si } fecha.dd = 1 \text{ y } fecha.mm = 1 \end{cases}$
  - ...etc...

2.D.1.2 Especificación Informal

Una especificación informal consisten en una especificación donde se describe de la forma más completa y escueta el TDA junto con la especificación de su operaciones. Se puede realizar según este esquema:

```
TDA nombre
    Descripción: Descripción textual del tipo
    Usa: Lista de otros TDA que pueda necesitar para su representación u
         operaciones
    Operaciones: Lista de operaciones junto una descripción de su
                 comportamiento (el efecto).
```

Ejemplo 2.3

a especificación informal del TDA fecha podría ser la siguiente:
TDA Fecha

- Representa fechas válidas de acuerdo al calendario Gregoriano.
- Usa: enteros, booleanos
- Operaciones:
  - Crear (entero d, entero m, entero a) : Fecha
    - Precondiciones: a$\neq$ 0, 1 $\leq$m$\leq$ 12 y 1 $\leq$d$\leq$ 31 de acuerdo al calendario gregoriano.
      Dados 3 números enteros que representan el día, mes y año, respectivamente, se obtiene la fecha compuesta por estos 3 valores
  - anterior (Fecha f) : Fecha
    Retorna la fecha anterior a la fecha f dada
  - siguiente (Fecha f) : Fecha
    Retorna la fecha siguiente a la fecha f dada
  - suma (Fecha f, entero d) : Fecha
    Retorna la fecha que corresponda de sumar d días a la fecha f dada
  - iguales (Fecha f1, Fecha f2) : booleano

Indica con $true$ si las dos fechas son iguales, y $false$ sin son distintas

- `día (Fecha f) : entero`
  Retorna el día de una fecha
- `mes (Fecha f) : entero`
  Retorna el mes de una fecha
- `año (Fecha f) : entero`
  Retorna el año de una fecha. No puede ser nulo.

2.D.2 Representación de TDAs

Definido un TDA se ha de seleccionar una representación de los objetos para implementar los datos y operaciones en términos de esa representación. La representación elegida debe permitir representar todos los valores de su dominio y la implementación de todas las operaciones.

La estructura de datos elegida para su representación se le denomina **tipo rep**. Se debe documentar cómo se almacenan los datos en el nuevo tipo de datos **rep**; es decir, debe documentarse la relación que existe entre el tipo **rep** y el tipo abstracto que se construya. Esta relación se llama **función de abstracción**:

$$Abst : \textbf{rep} \longrightarrow \mathcal{A}$$

es una función sobreyectiva del conjunto de objetos que pueden representarse al conjunto de objetos abstractos. Es decir:

- Todos los objetos abstractos tendrán una representación en el conjunto origen.
- Una o varias representaciones se pueden asignar a los objetos abstractos. P.e. las representaciones $[1, 2]$ y $[2, 1]$ pueden representar al conjunto $\{1, 2\}$.
- Determina que valores en la representación se pueden asignar a objetos abstractos. P.e. en una representación por ternas para una fecha, los valores $[100, 200, 2021]$ no son válidos porque ni hay 200 meses en un año ni ninguno tiene 100 días.

Ejemplo 2.4

$c_0 + c_1x + c_2x^2 + \ldots$. Se opta para su representación una estructura de datos formado por un entero para representar el grado del polinomio y un array de reales para representar los coeficientes:

```
struct rep {
   int grado;
   double[] coef;
}
```

La función de abstracción puede definirse como:

$Abst(\texttt{r}) =$ `r.coef[0] + r.coef[1]`x `+ r.coef[2]`x^2 `+ ... + r.coef[r.grado]`$x^{\texttt{r.grado}}$

siempre y cuando `r.grado` sea no negativo y `r.coef` sea un array que contenga (`r.grado`+1) elementos.

Notar que la función de abstracción indica para cada representación válida cómo se obtiene el tipo abstracto correspondiente. Para indicar cuál es el conjunto de valores que son válidos para representar a un tipo abstracto se usa el **invariante de la representación** que es un predicado I : **rep** $\longrightarrow \mathbb{B}$ que es cierto para los objetos de **rep** que sean legítimos.

Ejemplo 2.5

i se considerar representar a todos los polinomios $c_0 + c_1x + c_2x^2 + \ldots$. mediante la estructura

```
struct rep {
   int grado;
   double[] coef;
}
```

El invariante de la representación puede definirse como:

$I(\texttt{r})$=`(r.grado`$\not<$`0) AND r.grado = len(r.coef)`

Ejemplo 2.6

Para representar una fecha se usa la estructura

```
struct rep {
   int dia;
   int mes;
   int anio;
}
```

- Invariante de la representación. $I(r)$ es la conjunción de los siguientes predicados.
  - 1 $\leq$ r.dia $\leq$ 31
  - 1 $\leq$ r.mes $\leq$ 12
  - r.mes$\in${4, 6, 9, 11} $\rightarrow$ r.dia$\leq$30
  - r.mes=2 AND bisiesto(r.anio) $\rightarrow$ r.dia$\leq$29
  - r.mes=2 AND NOT bisiesto(r.anio) $\rightarrow$ r.dia$\leq$28
- Función de abstracción: $Abst$(r)=r.dia/r.mes/r.anio

2.D.3 Implementación de TDAs

Un TDA es el resultado de un proceso de abstracción que puede representarse mediante un **tipo rep**. Para ser útil debe ser implementado en un lenguaje de programación para que los usuarios del TDA puedan usarlo en sus programas.

Lo que necesita conocer el usuario del TDA es su nombre, su dominio (tipos de objetos con los que trabaja) y su interface (las operaciones asociadas). Estos 3 elementos componen precisamente la especificación del TDA, que recibe el nombre de parte pública del TDA. Por contra, a los usuarios no les interesa conocer tanto cómo se define y cómo deben implementarse (estructura de datos + algoritmos), componentes que reciben el nombre de parte privada del TDA. La ventaja de tener una parte pública y otra privada es que el usuario puede usar la interface pública independientemente de cómo se haya implementado internamente, de hecho se pueden tener diferentes implementaciones para un mismo TDA, cada una con sus ventajas e inconvenientes.

Así pues, observamos que para implementar un TDA se tienen que cumplir varios requisitos:

- Encapsulamiento. Es un mecanismo que facilita la agrupación de los datos y sus métodos. El TDA debe constituir un módulo o componente que pueda implementarse del resto del sistema. Esto permite que pueda rediseñarse (cambiar su implementación) y reutilizarse sin afectar al resto del programa.
- Ocultación de la información. Es el mecanismo por el que se restringe el acceso a algunas componentes del TDA. Esto permite:
  - La privacidad de la representación. El usuario no puede saber cómo se ha implementado el TDA pues la implementación permanece oculta. El usuarios solo debe conocer la interface pública que es la que viene dada por la especificación.
  - La protección de los datos. El usuario no puede tener permiso para acceder a los campos internos o métodos de la implementación pues podría modificar los valores del TDA sin pasar por la interface pública.

La Programación Orientada a Objetos permite implementar de una forma natural los TDA. En POO una clase es la implementación de una abstracción de un conjunto de objetos. Toda clase define una plantilla que consta de atributos (variables) y métodos (funciones). Todo objeto de una clase consta de un estado (valores concretos de los atributos) y de un comportamiento (los métodos de la clase). Los objetos son, por tanto, instancias o casos concretos de la plantilla que define la clase y las variables de un programa POO hacen referencia a objetos.

Ejemplo 2.1. Se podría definir la clase Persona como una abstracción de los objetos que tienen un nombre y una fecha de nacimiento junto con algunos comportamientos como, por ejemplo, ser capaz de calcular los días entre su fecha de nacimiento y otra dada. **En pseudo-código:**

```
class Persona {
   String nombre;
   Date fechaNacimiento;

   int edad(Date fecha) {
      return abs(fecha.toInt() - fechaNacimiento.toInt());
```

```
    ...
}
```

El siguiente código crea el objeto particular llamado Daniel y que nació el 1 de enero de 2001. Posteriormente muestra el número de días transcurridos para la fecha dada:

```
daniel = Persona("Daniel", Date(1,1,2001))
print(f"Han transcurrido {daniel.edad(Date(21, 12, 2222))} días")
```

□

En POO aparece de forma natural:

- la abstracción, que queda representada por las clases. En POO una clase se construyen considerando lo esencial de los objetos que se quieren representar.
- el encapsulamiento, pues en POO las clases representan conjuntos de objetos donde todos ellos se ajustan a una plantilla (todos tienen los mismos atributos y los mismos métodos o acciones).
- la ocultación de información, pues en POO se puede establecer qué atributos y métodos pueden permanecer ocultas (o visibles) a otros objetos.

Además, en POO se tienen varias características que son muy adecuadas para trabajar con TDAs:

- Un tipo de operador de un TDA son los constructores, que son los que crean una instancia del TDA del tipo de dato TDA con un operador de inicialización. Además en TDA la instancia se puede modificar y consultar. En POO un objeto es un caso particular de la plantilla que define a la clase a la que pertenece ese objeto. En este paradigma de programación lo objetos son creados, modificados y destruidos ... igual que las instancias de un TDA.
- En POO se trabaja con herencia. Una clase hereda de otra clase si la primera es una especialización de la segunda. P.e. la clase rectángulo es una especialización (particularización) de la clase figura geométrica. En TDAs no encontramos con situaciones similares. P.e. una cola es una especilización de una lista.
- En POO se tiene polimorfismo (hacer lo mismo de forma diferente). En TDAs esto también ocurre con frecuencia. P.e. la suma de números es una operación polimórfica: la suma se hace de forma diferente dependiendo de la naturaleza de los números. No es lo mismo sumar naturales, que sumar fracciones o sumar números complejos pero aún haciéndose de forma diferente todo es sumar números.

Python

2.E Python

Para la implementación de los TDAs se usará Python. En esta sección vamos a dar un repaso breve, pero lo suficientemente completo de este lenguaje como para abordar los ejercicios de esta sesión.

2.E.1 Breve Historia de Python

Python es un lenguaje creado por Guido van Rossum (Haarlem, Países Bajos, 31/1/1956). En diciembre de **1989** buscó un proyecto de programación como hobby para pasar las vacaciones de Navidad basado en el lenguaje **ABC** (script). Eligió el nombre de Python para el proyecto (como fan de Monty Python's Flying Circus)

En **1991** van Rossum publicó el código 0.9.0 por 1ª vez en `alt.sources`. Tiene un ruerte influencia de **Modula** y presenta clases con `herencia`, manejo de `excepciones`, funciones y tipos modulares. Se basa en sistema de `módulos`. La primera versión de Python se lanzó con licencia de Código Abierto. `https://docs.python.org/3/license.html`

En **1994** se formó `comp.lang.python`, el foro de discusión principal de Python. En **1999** se lanza la primera versión de la serie 1.0 donde se introduce programación funcional así como las funciones `reduce()`, `filter()` y `map()` inspiradas en **Lisp**. En **1999** van Rossum realizó una propuesta a DARPA (Agencia de Proyectos de Investigación Avanzados de Defensa) llamada `Computer Programming for Everybody`, consiguiendo que el lenguaje fuera patrocinado.

En mayo de **2000**, Guido y el equipo central de desarrollo de Python formaron `BeOpen PythonLabs` de BeOpen.com. Sacan la versión 2.0. En octubre de 2000, el equipo de `PythonLabs` se trasladó a Digital Creations (ahora Zope Corporation; consulte https://www.zope.org/). En **2001**, se formó Python Software Foundation, una organización sin fines de lucro creada específicamente para poseer propiedad intelectual relacionada con Python.

Desde la versión 2.1. todas tienen la *Python Software Foundation License*.

- **Python 2.x:** generación de listas basada en la programación funcional **Haskell**, recolección de basura (inventado por John McCarthy como parte de **Lisp**), generadores (basado en Icon), la unificación de tipo en Python (escritos en **C**) y clases (escritos en Python) dentro de una jerarquía (lo convierte en un lenguaje basado en un modelo POO puro). La v.2.7 dejó de tener soporte el 1/1/2020.

- **Python 3.x** (desde 2008): Hace limpieza de código y deja de ser complatible con 2.x.. Incluye una lista interminable de mejoras en cada versión. La última en la ver. 2.10.2 del 21/2/2022 que incluye la **`coincidencia de patrones estructurales`**.

2.E.2 Características y Uso

El índice TIOBE calcula la popularidad de los lenguajes de programación usando 25 motores de búsqueda de distintos países (google, wikipedia, amazon). La popularidad se asocia al uso: Mayor uso →más dudas →más búsquedas

En octubre de 2021 Python ocupó el primer puesto en el índice Tiobe, convirtiéndose en el tercer lenguaje que lidera el índice en sus más de 20 años de existencia. Python, C y Java acaparan el 42 % de "la popularidad"de todos los lenguajes de programación.

Inicialmente los **objetivos** marcados por Guido cuando lo presentó a DARPA:

- Python debería ser **fácil**, intuitivo y tan potente como sus principales competidores.
- El proyecto sería de **código abierto** para que cualquiera pudiera colaborar.
- El código escrito en Python sería tan comprensible como cualquier texto en **inglés**.
- Python debería ser apto **para las actividades diarias** permitiendo la construcción de prototipos en **poco tiempo**.

Lo que es hoy en día:

- **Multiplataforma** (Windows, Linux/UNIX, macOS, iOS, iPadOS y otros).
- Lenguaje de programación **interpretado**.
- **Multiparadigma:** imperativo (modular y orientado a objetos) y funcional.
- **Dinámico:** 1) No necesita declaración explícita del tipo, 2) Se determina en tiempo de ejecución.
- **Fuertemente tipado** (no se puede cambiar el tipo de dato). No se puede hacer `'1'+2`.
- Permite la **inclusión** de módulos en C, C++. De hecho está implementado sobre C.

Su gran popularidad se debe, en parte, a que hoy en día es **EL** lenguaje de programación de referencia para

- El **desarrollo web.**
  - Pyramid, Django y Flask.
  - Frameworks que integran protocolos y reducen el tiempo de desarrollo.
- **Ciencias de los datos**
  - NumPy (matrices), Pandas (manipulación y limpieza), Plotly y Seaborn (visualización estadística de datos), Shap (explicación de modelos), etc.
  - Bibliotecas de desarrollo ayudan a extraer información de los datos
- **Inteligencia Artificial y Aprendizaje Automático**
  - Keras / Tensorflow / Scikit-learn / PyTorch (machine learning), OpenCV (visión artificial), PyKE (programación lógica), ...
- **Aplicaciones empresariales**
  - Es un lenguaje robusto que puede manejar múltiples **solicitudes** de **bases de datos** a la vez.
  - Legibilidad, funcionalidad y escalabilidad permanece igual.

También se usa en otros ámbitos como los siguientes:

- **Sector educativo**
  - Fácil de aprender para principiantes ya que su sintaxis “coincide” con la del inglés.
  - Potenciado por el desarrollo de cursos y programas educativos on-line. *Python* tiene 123 resultados en EdX y 878 resultados en Coursera.
- **Web scraping**
  - Recopila información de las páginas web.
  - Ejemplos son PythonRequest, Selenium, MechanicalSoup.
- **Desarrollo de Juegos**
  - Pygame, PyKyra, Pyglet, PyOpenGL, Kivy, Panda3D, Cocos2D, ...
  - No es lo más usado, pero a veces surgen juegos destacados. P.e. Campo de batalla 2 (2000)
- **Desarrollo de software**
  - Simplica el proceso de desarrollo de software para aplicaciones complejas.
  - Se usa para la gestión del proyecto, como lenguaje de apoyo.

- **GUI de escritorio**
  - Tkinter, PyQt, PyGUI y WxPython
  - Crear GUI de alta calidad de manera eficiente.
- **Sistemas operativos**
  - Scripts para mantenimiento del sistema.
  - macOS no puede vivir sin Python 2.7 ... hasta la ver. 12.3 (Monterey), pero incluye la versión 3 con XCode.

Algunas aplicaciones conocidas que usan Python son:

- Windows (la instalación asistida)
- Bit Torrent (descargas p2p)
- Google App Engine (servicio de alojamiento web)
- Ubuntu Software Center (gestor de aplicaciones)
- Dropbox (servicio de almacenamiento)
- Uber (lo usa con Big Data Analytics de IBM)
- Blender (Modelado 3D)
- Instagram (red social)
- Pinterest (red social)
- Reddit (red social)
- Youtube (red social)
- Spotify (servicio de streaming de música)
- Netflix (servicio de streaming de video)

2.E.3 Variables, Expresiones y Conversión de Tipo

Tipos de Datos. En programación imperativa se distinguen dos tipos de datos: elementales y compuestos.

- **Tipos de datos primitivos (o elementales)**
  - Formado por un único elemento.
  - Tipos: carácter, númerico, booleano, enumerado.
- **Tipos de datos compuestos**
  - Formado por una agrupación de elementos.
  - Tipos: string, array, registro, conjunto, lista, diccionario.

Variable y expresiones. Las **variables** son las ubicaciones de almacenamiento de los datos. Cada variable se caracteriza por su **nombre** y el **valor** almacenado. **Declarar** una variable es dar la orden para establecer el tipo de dato que se almacenará e identificar la zona de memoria con el identificador establecido. **Asignar** valor a una variable es almacenar una información en la zona de memoria del identificador. La asignación se realiza con `=`. La primera asignación se llama **inicialización**. Una **constante** es una variable para la que solo se puede realizar la inicialización (no caben más asignaciones).

Python es dinámicamente tipado:

- No existe la declaración de forma explícita.
- Al realizar una asignación, de forma implícita realizará la declaración.
- **Si se asigna posteriormente un valor de un tipo diferente, cambiará su declaración.**
- En Python no existen las constantes.

Las variables usan la convención de nombres snake_case y cuando se quiere considerar que una variable es una constante su identificador tiene todos sus caracteres en mayúsculas[3]. Recuerde, en Python puede cambiar el nombre de "una constante".

```
numero_entero: int = 2
numero_real: float = 10.01
caracter: str = '2'              # Se interpreta como un string
booleano: bool = True
string: str = " una cadena "     # Se interpreta como un string
```

[3] PEP 8 – Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/

```
# Asignación múltiple
numero_entero, numero_real, booleano= 2, 10.01, False

# Destrucción de variables
del(numero_entero)   # No es usual esta instrucción

# Las "constantes" se escriben en mayúsculas
PI: real = 3.1415      // La "constante" PI en Python. Ojo, puede cambiar su valor.
```

Se distinguen varios tipos de Expresiones:

- **Expresiones numéricas**
  - Aritméticas: suma (`+`), resta y negación (`-`), multiplicación (`*`), división (`/`), división entera (`//`). módulo (`%`), exponente (`**`).
- **Expresiones de strings**
  - `+`, Concatena dos cadenas.
  - `*`, Producto de un número natural por una cadena. Concatena la cadena tantas veces como indique el número natural.
  - `cadena[i]`, retorna el carácter i-ésimo.
  - `cadena[-i]`, retorna el carácter i-ésimo.
  - `cadena[i:j:k]`, retorna el string formado por los elementos entre i y j, cada k índices.
- **Expresiones booleanas**
  - Comparaciones: `x == y`, `x != y`, `x > y`, `x >= y`, `x < y`, `x <= y`
  - Operaciones: `e1 and e2`, `e1 or e2`, `not e`

Los resultados de las expresiones se guardan mediante la **asignación**. Las asignaciones no se consideran formalmente como expresiones, pero sí las construyen. P.e. `x = x/10` ≡ `x /= 10`

Las asignaciones que usa Python son: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `//=`, `**=` (algunas son solo para expresiones numéricas).

Precedencia. El orden de precedencia de los operadores en Python los puede ven en la Figura 2.1.

Mutabilidad y Casting. Una variables es **mutable** si se puede cambiar el valor de la variables sin cambiar su referencia en memoria.

La instrucción `id(var)` muestra la referencia de la variable `var`. Esta instrucción nos permite comprobar la mutabilidad de una variable. Si se hacen dos asignaciones a una variable e `id()` cambia, la variables es inmutable. Números, booleanos y strings son inmutables. Compruébalo !!

El **casting** es el proceso por el que el valor de una variable se interpreta como otro tipo de dato. Python no admite casting, pero sí tiene funciones para el **cambio de tipo**: `int()`, `float()`, `str()`.

2.E.4 Programación Estructurada

La programación estructurada es un paradigma de programación que se basa en el **Teorema del Programa Estructurado**, propuesto por Böhm-Jacopini, que demuestra que todo programa puede escribirse utilizando tres estructuras básicas, llamadas **estructuras de control**:

- **Estructura secuencia**, consta de una secuencia de instrucciones directas.
- **Estructura condicional**, que consta de una sentencias condicional y sus correspondiente cuerpos.
- **Estructura iterativa**, que consta de bucles (o sentencias iterativas).

Estructura Secuencial. Consta de una secuencia de **órdenes directas**. El conjunto de instrucciones **vienen dadas por el lenguaje** de programación.

- **Sentencias de Asignación**, consistentes en el paso de valores de una expresión o literal a una zona de la memoria.
- **Lectura**, `input()`, consistente en recibir desde un dispositivo de entrada algún dato.
- **Escritura**, `print()`, consiste en mandar a un dispositivo de salida algún valor.
- **Tamaño**, `len()`, calcula el número de datos que tiene una secuencia (p.e. un string).
- **Identificación**, `id()`, retorna la referencia de una variable.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	**not**	Unario	—	6
Conjunción	**and**	Binario	Por la izquierda	7
Disyunción	**or**	Binario	Por la izquierda	8

Figura 2.1: Orden de Precedencia Fuente: `Introducción a la Programación con Python` (página 37)

- **Tipo**, `type()`, indica el tipo de dato de un literal o de una variable.
- ...

En **Python** tenemos:

- **Sentencias de asignación:** `https://docs.python.org/3/reference/simple_stmts.html`
- **Built-in Functions:** `https://docs.python.org/3/library/functions.html`

Estructura condicional. Es aquella que ejecuta ciertas órdenes si se cumple una condición booleana.
En **Python** se usa la sentencia compuesta con cláusulas **`if`**, **`elif`**, **`else`**.

- Condicional simple.

```
if condicion:
    estructura
```

```
if condicion: sentencia  # Inline
```

- Condicional doble.

```
if condicion:
    estructura_if
else:
    estructura_else
```

```
var = exp_si_true  if  condicion else exp_si_false # Inline
```

- Condicional anidado.

```
if condicion1 [op condicion2 [op condicion3] ... ]:
    estructura_if
elif condicion:
    estructura_else_if
else  # Casi obligado si se usa elif.
    estructura_else
```

Estructura Iterativa. La versión imperativa de esta estructura consta de los siguientes pasos:

1. Se parte de una variable, que se llamará **variable de control** y que se inicializará a cierto valor.
2. Entonces se comprueba una **condición** booleana donde interviene la variable de control.
3. Si la condición es cierta, entonces se ejecutarán nuevas **estructuras**.
4. Entres las **estructuras** habrá alguna secuencial que **modifique la variable de control**.

5. Se vuelve paso 2.

El proceso **se repite** hasta que la variable de control tome un valor que hace que la condición booleana es falsa.
En **Python** se usa la sentencia **while**:

```
control = valor_inicial
while expresion_booleana_con_la_var_de_control:
    estructuras
    modificar la variable de control
else: # opcional
   estructuras
```

El bloque **else no se realizará** si se ejecutara la sentencia **break** en el bloque **while**.
Existen dos sentencias, **break** y **continue**, que te pueden ser útiles:

- **break**
  - Solo puede ocurrir sintácticamente en un bucle **for**[4] o **while**.
  - Terminará el bucle adjunto más cercano y omitirá la cláusula opcional **else**.
- **continue**
  - Solo puede ocurrir sintácticamente en un bucle **for** o **while**.
  - Continúa con la siguiente iteración del bucle más cercano.
  - No ejecutará lo que aparezca después de **continue**

Ejercicio 2.1

Aquí hay un error muy común ¿cuál?

```
control = 1
while control < 10:   # Se puede cambiar la condición
    if control % 2 == 0:
        continue
    elif control % 5 == 0:
        break
    print(control, end=" ")
    control = control + 1  # Se debe modificar la var. de control
else:
    print("No debería ejecutarse")
```

2.E.5 Programación Procedimental

Una **función** es una **secuencia** de instrucciones identificada con un **nombre** que retorna un valor. En **Python** una función puede retornar varios valores y, en este caso, **"empaqueta"** todos los datos de retorno en un tipo de datos llamado **tupla**.

```
def nombre_funcion ( lista de parámetros ) -> tipo de dato de retorno:
   estructuras de la función
   return valores
```

Un **procedimiento** es una función que no tiene la instrucción de retorno.

```
def nombre_procedimiento ( lista de parámetros ) [-> None]:
   estructuras del procedimiento
```

Si una función/método tiene n-parámetros podemos invocar a la función con n-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetros, etc ... Son parámetros **posicionales**.

```
def fun(a, b, c, d):  # Tiene 4 parámetros
    print(a, b, c, d)

fun(1, 2, 3, 4) # Invocamos con 4 parámetros posicionales.
```

[4] El bucle **for** se estudiará en POO.

Los **k-últimos parámetros** de un función pueden ser **opcionales**.

- Los opcionales determinan un valor **literal** **por defecto**.
- **Primero** los obligatorios **y después** los opcionales (o por defecto).

```
def fun(a, b, c=3, d=4): #  2 posicionales + 2 opcionales.
    pass
```

Python permite invocar por **palabras claves** (keywords).

- Usar **keyword =** especifica el nombre del parámetro en la invocación.
- El orden de los parámetros pueden cambiarse.
- En la declaración de la función, los keywords siempre se pondrán al final.

```
# Para la función anterior
fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
fun(1, 2, d=4, c=3) # Los keywords al final
fun(d=4, 1, 2, c=3) # Incorrecto
```

Cuando no se conoce el número de argumentos que se usarán, se usa el **Packing Arguments** (empaquetamiento de argumentos) con el operador *.

```
>>> def fun(*args):   # Empaquetará todos los argumentos
...     print(args)
...
>>> fun(1, 2, 3, 4)   # Invocación desempaquetada
(1, 2, 3, 4)
```

En el ejemplo, todos los argumentos se agrupan en una **tupla**.

También existe el **Unpacking Arguments.** Dada una función/método con varios parámetros podemos empaquetar los argumentos con *.

```
>>> def fun(a, b, c, d):   # Desempaquetará si le envían \
...     print(a, b, c, d)  # los argumentos empaquetados
...
>>> lista = [1, 2, 3, 4] # Las listas las veremos más tarde.
>>> fun(*lista)          # Invocación empaquetada
1 2 3 4
```

Para acceder a uno de lo argumentos empaquetados se usan **índices**:

```
>>> def fun(*args):
...     print(len(args), args[1]) # Muestra el cardinal y el 2o argumento
...
>>> fun(1, 2, 3, 4)
4 2
```

Estaría mejor acceder a ellos con un nombre (keyword). De hecho, podemos **empaquetar y usar keywords** usando **diccionarios**, con **. **Un diccionario es** como un array pero los índices se sustituyen por claves.

```
>>> def fun(**kwargs):
...     print(f"{len(kwargs)} elementos", kwargs) # Muestra el diccionario
...
>>> fun(a=1, b=2, c=3, d=4)    # Invocación con keywords
4 elementos {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> diccionario = {'p1': 1, 'p2': 2, 'p3':3}  # Los veremos
>>> fun(**diccionario)         # Invocación con empaquetado
3 elementos {'p1': 1, 'p2': 2, 'p3': 3}
```

Si se quieren usar los 3 modos de pasar argumentos, el orden debe ser:

1. posicionales
2. empaquetados sin keyword
3. empaquetados con keyword

```
>>> def fun(a, b, *args, **kwargs):
...     print(a, b)  # Muestra los posicionales
...     print(args)  # Muestra los empaquetados sin keyword
...     print(kwargs)# Muestra los empaquetados con keyword
...
>>> fun(1, 2, 3, 4, 5, p1=6, p2=7, p3=8)
1 2
(3, 4, 5)
{'p1': 6, 'p2': 7, 'p3': 8}
```

EJERCICIO 2.2

Se define `def func(x, y, op = "+")`. ¿Cuánto valdrá `x`, `y` y `op` en los siguientes casos?

- `func(2, 3, "*")` ▪ `func( (2, 3), "*")` ▪ `func( *(2, 3), "*")`
- `func( *(2, 3, "*"))` ▪ `func( (2, 3, "*"), 4)` ▪ `func( *(2, 3, "*"), 4)`

Docstrings. Toda función debe contener su **especificación informal**. La documentación se hace en Python mediante **docstrings**: Cadenas que empiezan y terminan con triple comillas (simples o dobles) Existen varios **formatos** (lenguajes de marcas):

- reStructuredText. El estandar de Python.
- Google. La mejor alternativa "no-nativa".
- Numpy. La propuesta de Numpy.
- etc ...

```
def func (x: int) -> str:
    """
    Función general que no hace nada
    reStructuredText

    :param x: La entrada, es entero
    :return: El valor de retorno
    """
    pass
```

```
def func (x: int) -> str:
    """
    Función general que no hace nada
    Google

    Args:
        x: La entrada, es entero

    Returns:
        El valor de retorno
    """
    pass
```

Funciones Lambda o Anónimas. El λ-cálculo es un sistema formal matemático desarrollado en los años 1930s diseñado para trabajar con la noción de función, aplicación de funciones y recursión. Proporciona una semántica simple para la computación y la primera simplificación es que el λ-cálculo **trata las funciones de forma anónima**. La escritura anónima de $\text{square\_sum}(x, y) = x^2 + y^2$ es $(x, y) \mapsto x^2 + y^2$.

En **Python** las expresiones anónimas se llaman **Funciones Lambda** y tiene la siguiente expresión:

```
lambda argumentos: expresión
```

En la definición deberá tener en cuenta:

- Puede usar cualquier número de argumentos.
- Solo habrá una única expresión.

```
>>> 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 clausura como `map()`, `filter()`, `reduce()`, ... para trabajar con colecciones. Se verán más adelante.

2.E.6 Programación Modular

Es un paradigma de programación que consiste en **dividir un gran programa** (posiblemente de miles de líneas) **en módulos** (ficheros). Los módulos constan a su vez de **funciones**, **variables**, **clases**. Algunos módulos usarán otros módulos pero habrá uno (y solo uno), llamado **módulo principal**, que no puede ser usado por los demás módulos. El módulo principal es el encargado de ir resolviendo cada una de las subtareas.

En Python la modularidad se consigue con **import**. Ejemplo de uso de módulos:

```
>>> import math
>>> print("PI=", math.pi)
PI= 3.141592653589793
```

```
>>> import math as m
>>> print("PI=", m.pi)
PI= 3.141592653589793
```

```
>>> from math import pi, cos
>>> print("cos(PI)=", cos(pi))
cos(PI)= -1.0
```

2.E.7 Programación Orientada a Objetos

La estructura básica para definir una clase en Python es la siguiente:

```
class NombreDeLaClase: # Notación Camel
    def __init__(self, p1, p2, ...):
        self._var1 = p1
        self._var2 = p2
        ...  # más atributos

    def metodo1 (self, parametros):
        acciones

    def metodo2 (self, parametros):
        acciones

     .... # más métodos

objeto = NombreDeLaClase(argumentos para __init__)
```

Todos los métodos que tienen la forma: **__metodo__(...)** se denominan **métodos mágicos**. Destacamos:

- __new__(cls, ...). Es el método creador. **No debe escribirse nunca**.
- __init__(self, ...). **Es un método inicializador** de instancia que se invoca siempre, una vez construido el objeto con __new__(cls, ...).
- etc ...

Trabajar con **TDAs** como modelos **ayuda** muchísimo a la resolución de problemas y diseño de algoritmos; pero al final hay que implementarlos en un lenguaje de programación usando sus **tipos de datos integrados**. Todos los tipos de datos integrados en Python son objetos y, por tanto, si queremos construir nuevos tipos de datos debemos saber cómo construir y manipular nuevos objetos.

Deberá tener presente que Python es un lenguaje de POO puro. En consecuencia, cualquier operación que realice conlleva manipular los atributos del objeto y, por tanto, invocará de forma directa o indirecta a los métodos del objeto.

Por ejemplo, en todos los **Tipos de Datos Numéricos** tenemos estos miembros/métodos y operaciones:

- Miembros: **.real**, **.imag**
- Métodos: **.conjugate()**,
- Operaciones: +, -, *, /, //.
- Los métodos mágicos sobre operaciones matemáticas, asignaciones extendidas, operaciones unarias y operaciones de comparación.

Esto quiere decir que cuando realice una operación numérica invocará al método mágico correspondiente. Por ejemplo,

- Cuando escriba **3+5**, internamente se ejecutará 3.__add__(5). Siendo el **3** y el **5** dos objetos.
- Si escribe **3-5**, internamente se ejecutará 3.__sub__(5).
- etc ...

Lo mismo ocurre cuando invoque a un función sobre un número. Por ejemplo, invocando a **abs(3)** estará invocando a 3.__abs__(). También ocurre lo mismo al usar operadores de comparación: usar < es invocar al método .__lt__(), usar == invocará al método .__eq__(), etc ...

De entre todos los operadores el más esencial es sin duda la asignación =. Cada vez que realice una asignación de la forma estará construyendo un objeto nuevo y cada vez que se crea un objeto nuevo se invocará al método .__init__().

Esto es muy importante a tener en cuenta porque:

- A ser un lenguaje no tipado, la asignación determina su tipo. En consecuencia una variable puede referenciar incialmente a un entero para terminar referenciando a una lista.

```
a = 2  % a es un objeto entero
a = 2 == 3  % a es un objeto booleano
a = NombreDeLaClase(p1, p2) % a es un objeto de NombreDeLaClase
```

- Todas las variables tienen un ámbito local salvo que se diga lo contrario (se explicará posteriormente).

Para completar los **Tipos de Datos Numéricos** en Python, indicar que se tienen los siguientes.

- **Enteros:**
  - Una variable se declara entera si se le asigna un número sin punto decimal.
  - Función constructora: `int()`.
  - Métodos: `.bit_length()`: N^o de bits necesarios.
  - Funciones: `bin`(n). Convierte a binario.
- **Reales:**
  - Una variable se declara real si se le asigna un número con punto decimal.
  - Función constructora: `float()`.
  - Métodos: `.is_integer()`: True si el valor es igual al de su parte entera.
- **Complejos:**
  - Constructor: `complex([real[, imag]])`

Podemos aumentar el conjunto de datos numéricos usando librerías. Destacamos aquí el módulo **Fracciones:**

- Requiere `from fractions import Fraction`
- Constructores: `Fraction(numerator=0, denominator=1)`, `Fraction(float)`, `Fraction(string)`, ...
- Miembros: `.numerator`, `.denominator`
- Métodos: `.as_integer_ratio()`: simplifica la fracción.
- https://docs.python.org/3/library/fractions.html

Otros tipos de datos integrados son los **Tipos de Datos Booleanos.** Usa las constantes True y False. Una variable a la que se le asigne uno de estos valores será una variable booleana. Las expresiones booleanas se construyen con:

- Comparaciones: `x == y`, `x != y`, `x > y`, `x >= y`, `x < y`, `x <= y`
- Operaciones: `e1 and e2`, `e1 or e2`, `not e`

Recuerde que todas estas operaciones invocará a sus correspondiente método mágicos.

Los TDAs más relevantes requieren el uso de colecciones. Python proporciona un conjunto de tipos de datos llamados **secuencias** (colecciones) o tipos de datos secuenciales. Se explicarán con más detenimiento en sesiones posteriores (de teoría/prácticas).

Recuerde seguir PEP 8 para las normas de estilo.

PEP 8 – Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/

Ejercicios

2.1 Un clase para números positivos

★☆☆ ✍

`assert`, `isinstance` y algún método mágico

Con la clase **`Positivo`** se quiere implementar los números naturales (enteros positivos). Para su implementación, sabemos que no existe este tipo de dato pero sí un tipo de dato para los naturales (`int`). Por ello, se opta por esta implementación:

```python
class Positivo:
    __slots__ = '_num'

    def __init__(self, num: int):
        pass

    def __str__(self) -> str:
        pass

def una_prueba():
    pass

if __name__ == '__main__':
    una_prueba()
```

Debes completar el código teniendo en cuenta los siguientes puntos, consultando la documentación de **Python** si fuera necesario:

- El atributo **_num** es el que almacenará el número entero positivo. De acuerdo a esto ?cuál es la función de abstracción?
- El método de inicialización **__init__()** debe generar un error si se intenta crear un objeto con un entero no positivo. Usa las instrucciones **assert** y **isinstance** para generar un error si el valor dado no es un entero positivo. De acuerdo a esto ¿cuál es el invariante de la representación?
- El método **__str__() : str** retornará en forma de string (**str**) el valor de **_num**.

Como ejemplo de uso define la función **def** una_prueba() con el siguiente código.

```python
el_tres = Positivo(3)     # Crea un objeto que representa al 3
print(el_tres)            # Muestra 3
el_cero = Positivo(0)     # Debe mostrar un error
el_pi = Positivo(3.14)    # Debe mostrar un error
```

2.2 TDA Fecha Simple

★☆☆

Implementación de Fecha con clases

Implementa el TDA fecha usando la representación indicada en el Ejemplo 2.6. Para ello debes completar el siguiente código:

```python
class Fecha:
    __slots__ = '_dia', '_mes', '_agno'

    def __init__(self, dia: int, mes: int, agno: int):
        pass

    def _set_agno(self, agno: int) -> None:
        pass

    def _set_mes(self, mes: int) -> None:
        pass

    def _set_dia(self, dia: int, mes: int) -> None:
        pass

    def get_dia(self) -> int:
        pass

    def get_mes(self) -> int:
        pass
```

```
    def get_agno(self) -> int:
        pass

    def __str__(self) -> str:
        pass
```

Ten en cuenta que:

- El método `def __init__()` debe invocar a los métodos `def _set_xxx`. Por ejemplo, se invocará a la asignación del año como `self._set_agno(agno)`.
- Los métodos `def _set_xxx` controlarán si los argumentos son correctos, en especial para los días (que dependen de los meses). Usa las instrucciones `assert` y `isinstance` para generar un error si los valores dados no se corresponden con la especificación. No es necesario comprobar que los años sean bisiestos.
  Observa que empiezan con "guión bajo". Esto quiere decir que los métodos `_set_xxx` no deberían ser invocados desde otras clases (es decir, no se puede modificar una fecha una vez creada).
- Los métodos `def get_xxx` permiten consultar los datos de un fecha. Estos métodos no empiezan con "guión bajo". Esto quiere decir que pueden ser invocados desde otras clases.
- El método `def __str__()` retornará la cadena "dd/mm/aa". Se invocará cuando uses la instrucción `print()`.
- Construye un método de prueba para imprimir fechas válidas y crear fechas inválidas (contemplando todas las posibilidades del invariante de la representación.

2.3 TDA Fecha Simple con Positivo

★☆☆

Otra Implementación de Fecha

Implementa el TDA Fecha del ejercicio anterior cambiando el tipo `int` del día y del mes por el tipo `Positivo` del primer ejercicio.

Crea un fichero .py con un nombre nuevo y reescribe `class Fecha`. No le cambies el nombre a la clase, solo al fichero.

2.4 Un clase para cilindros

★☆☆

función de abstracción, invariante de la representación

Se quiere construir el TDA Cilindro Recto con las operaciones básicas del cálculo de volumen y superficie.

- Defina el TDA.
- Defina la función de abstracción.
- Defina el invariante de la representación.
- Implemente en Python el TDA.
- Añada a la implementación el método mágico `__str__()` para imprimir toda la información disponible de un cilindro.

2.5 TDA Fracción

★★☆

Implementación de Fracción

Escribe al definición informal del TDA, su función de abstracción y el invariante de la representación.

Responde a lo indicado e implementa el TDA teniendo en cuenta que una fracción `fr` puede usar los siguientes métodos:

- `fr.suma(f: Fraccion)` realiza la asignación: `fr = fr+f`.
- `fr.resta(f: Fraccion)` realiza la asignación: `fr = fr-f`.
- `fr.multiplica(f: Fraccion)` realiza la asignación: `fr = fr * f`.
- `fr.divide(f: Fraccion)` realiza la asignación: `fr = fr / f`.
- `fr.simplifica()` para obtener su fracción más simplificada posible.

¿Cuáles son los métodos mágicos que debes de implementar para realizar estas operaciones `f1==f2` y `f1<f2`?

Te puede interesar el algoritmo de Euclides https://es.wikipedia.org/wiki/Algoritmo_de_Euclides#Descripci%C3%B3n_formal

2.6 Adivina el número

★☆☆

Lenguaje Imperativo en Python

Construir el siguiente juego de adivinación de números.

- El humano introduce un número.
- El ordenador debe de adivinar el número introducido por el usuario.
- El humano indicará si el número dado por el ordenador es mayor o menor al que ha introducido.
- La partida no termina hasta que el ordenador lo adivine.
- Los números estarán en el intervalo $[0, 100]$.
- Cuando lo adivine se mostrará un mensaje indicando el número de intentos realizados.
- Descubierto el número, se preguntará si se quiere otra partida.
- Si no se quiere otra partida se preguntará si se desea salir del juego.
- Si se deseara salir, se debe confirmar la respuesta. Si la respuesta es "no confirmo", volverá a preguntar si quiere otra partida.
- Realizar un programa que haga todo lo anterior sin la participación de humano alguno. Por comodidad se puede suponer siempre que el número dado por el humano es siempre el mismo.

Resuelve este ejercicio usando solo programación estructurada (sin clases). **Si tienes problemas para resolver este ejercicio, probablemente te falte base para poder afrontar la POO.**

Sesión 3

Colecciones

Índice Parcial **Página**

3.A. Estructuras de Datos . . . 33
3.B. Secuencias de Python . . . 34
3.C. Colecciones no Secuenciales de Python . . . 39
3.D. Iteración en Colecciones . . . 42
3.E. Construcción de Contenedores por comprehensión . . . 42
3.F. Métodos Mágicos . . . 43
3.G. Arrays en Python . . . 45
3.G.1. La Estructura Array . . . 45
3.G.2. Listas como arrays . . . 46
3.G.3. Array compacto . . . 47
3.G.4. ¿Por qué necesitamos arrays? Implementación en Python . . . 48
3.1. Construcción por comprehensión . . . 49
3.2. Más Ejercicios con Secuencias . . . 49
3.3. Un poco de estadística . . . 50
3.4. TDA Bag . . . 50
3.5. Anagramas . . . 51
3.6. TDA Map . . . 52
3.7. TDA Set . . . 53
3.8. Cursos de un Título . . . 53
3.9. Iterador de listas . . . 54
3.10. TDA Array1D . . . 55
3.11. TDA Array 2D . . . 56
3.12. Acceder a datos indexados . . . 57
3.13. Matrices . . . 58
3.14. Juego de la Vida . . . 58
3.15. Reversi . . . 59

Teoría

3.A Estructuras de Datos

Trabajar con TDAs, como modelos matemáticos, nos ayuda muchísimo a la hora de construir algoritmos. Pero en algún momento deben de implementarse y la implementación depende del lenguaje de programación. Es por ello que esta sesión vamos a aprender cuáles son las estructuras de datos que nos ofrece el lenguaje **Python** para, posteriormente, implementar los primeros TDAs en este lenguaje.

Una variable almacena un valor que se encuentra en una dirección de memoria. De una forma simplista, la variable puede referenciar a un único valor o al inicio de un conjunto de valores. En el primer caso decimos que la variable responde a un tipo simple y en el segundo caso decimos que la variable responde a un tipo compuesto para indicar a una forma de organizar conjuntos de datos elementales.

Existen muchas formas de organizar un conjunto de datos. Las distintas formas en las que son combinados reciben el nombre de **estructuras de datos**. A diferencia del tipo de dato elemental o simple, que solo puede almacenar un valor, los

datos estructurados o estructuras de datos pueden contener varios datos simultáneamente. A la acción de construir un tipo de dato compuesto se le llama **composición**.

Una **estructura de datos** es una agrupación de datos que se trata como una unidad. Puede contener solo datos elementales o bien datos estructurados o bien datos elementales y estructurado.

Existen muchas formas de clasificar las estructuras de datos. Una de esas formas es la siguiente:

- **Por su naturaleza** se distinguen las estructuras **homogéneas y heterogéneas**.
  - **Estructura homogénea**. Es la estructura en la que todos los datos elementales son del mismo tipo de dato (p.e. vectores, tablas, matrices n-dimensionales.)
  - **Estructura heterogénea**. Es cuando alguno de los datos elementales es de distinto tipo al resto (p.e. registros)
- **Por el uso de la memoria** se distinguen entre **contiguo y enlazadas**
  - Las **estructuras contiguas** son aquellas en las que los datos que contienen se almacenan de forma "contigua" a partir de una posición de memoria: los datos se almacenas en posiciones consecutivas de memoria. La estructura no puede cambiar de tamaño: se deben ocupar todas las posiciones para no desperdiciar espacio y se requiere de otra estructura mayor si faltaran posiciones. La ventaja es que un dato de la estructura se puede localizar directamente a partir de su posición relativa respecto de la posición inicial del primer dato.
  - Las **estructuras dinámicas** son aquellas cuyos datos se almacenan en posiciones no-consecutivas de memoria. No está limitada a una cantidad fija de posiciones por lo que pueden aumentar o disminuir en cantidad y, por tanto, presentan un tamaño variable sin que el programador pueda determinarlo previamente. La única restricción de estas estructuras son la memoria disponible.
- **Por la posibilidad de modificarlos** se distinguen entre **mutable e inmutables**
  - Las **estructuras mutables** son aquellas cuyo estado puede cambiar una vez creadas.
  - Las **estructuras inmutables** son aquellas cuyo estado no puede cambiar una vez creadas.

Python integra las siguientes estructuras de datos.

- **Estructuras mutables.** Se pueden cambiar sus términos una vez creada.
  - **Listas:** **secuencia** de elementos arbitrarios. Se escriben entre corchetes y se separan con comas.
    Ej: `[1, "hola", 3]`.
  - **Conjuntos:** colección de elementos arbitrarios **únicos**. Se escriben entre llaves y se separan con comas.
    Ej: `{1, "hola", 3}`.
  - **Diccionario:** conjuntos con objetos indexados. Cada elemento consta de un par `clave: valor`. La clave puede ser cualquier valor inmutable.
    Ej: `["a": 1, "b": "hola", "c": 3]`.
- **Estructuras inmutables.** No puede cambiar sus términos una vez creada.
  - **Strings:** **secuencia** de valores que representan códigos Unicode. Se escriben entre comillas.
    Ej: `"hola"`, `'adiós'`
  - **Tuplas:** **secuencia** de elementos arbitrarios. Se escriben entre paréntesis y se separan con comas.
    Ej: `(1, "hola", 3)`.
  - **Rangos:** **secuencias** que se construyen con `range([start,] stop [[, step]])`.
    Ej: `range(1:10:2)`.
  - **Conjuntos congelados** (Frozensets). La versión inmutable de los conjuntos.
    `frozenset({1, "hola", 3})`

Otra clasificación, distingue los siguientes grupos (`https://docs.python.org/es/3/library/stdtypes.html`): Secuencias, cadenas de caracteres (string), conjuntos (set, frozenset), mapas (dict).

Desarrollemos cada uno de estos tipos de datos integrado; pero, por comodidad, juntaremos las secuencias con los strings en un mismo apartado.

3.B Secuencias de Python

Los TDAs más relevantes requieren el uso de colecciones. Python proporciona un conjunto de tipos de datos llamados **secuencias** o tipos de datos secuenciales entre los que se encuentran los siguientes tipos de datos lineales integrados de este lenguaje: tuplas, rangos, listas. Aunque Python trata a los strings como otro tipo de dato, en este documento lo trataremos como secuencias porque comparten muchos operadores.

Las secuencias o tipos de datos secuenciales (o en secuencia) representan a colecciones finitas de datos referenciados por un índice. Alternativamente, una secuencia esta formada por una colección de objetos, o términos, indexados con

índices 0, 1, 2 ...

Se puede **acceder** a sus elementos de distintas formas:

- `s[i]`, el elemento i-esimo de `s`,
  Cuenta +1 desde el 0 si $i \geq 0$, o cuenta -1 desde la `len(s)-i` si $i < 0$.
- `s[i:j]`, la rebanada (slice) de `s` desde `i` hasta `j`. Selecciona todos los elementos con índice t tale que $i \leq t < j$.
- Algunas secuencias también admiten la **"división ampliada"** con un tercer parámetro de "paso": `[i:j:k]` selecciona todos los elementos con índice t donde $t = i + n \times k$, $0 \leq n$ e $i \leq t < j$.

Funciones comunes

- `len(s)`, que devuelve el número de elementos de una secuencia.
  Si `len(s)==n`, el conjunto de **índices** es $\{0, 1, \ldots, n-1\}$.
- `min(s)` / `max(s)`: el item más pequeño/grande de `s`.
- `sum(s)`: la suma de los elementos de `s`.
- `sorted(s)`: una lista con los elementos ordenados de `s`.
- `enumerate(s)`: un enumerado de la lista `s`.
- `any(s)`: devuelve True si bool(x) es True para cualquiera de los elementos de la secuencia.
- `all(s)`: devuelve True si bool(x) es True para todos los elementos de la secuencia.

Métodos comunes

- `s.count(x)`: el número total de ocurrencias de `x` en `s`.
- `s.index(x[, i[, j]])`: el índice de `x` en `s` [después de `i` [y antes de `j`]]

Operaciones comunes en estos elementos son:

- `x in s`: Es True si `x` es un item de `s`.
- `x not in s`: Es True si `x` no es un item de `s`.
- `s + t`: concatena `s` y `t`
- `n * s`: añade `s` un total de `n`-veces (en algunas secuencias).

```
>>> secuencia = (list(range(1, 3)) + list(range(3, 6))) * 2
>>> print(len(secuencia))
10
>>> 5 in secuencia
True
>>> count = 0
>>> for i in secuencia:
...   if i == 5:
...       count += 1
...       print(f'Encontré el 5: {count} vez/veces')
...
Encontré el 5: 1 vez/veces
Encontré el 5: 2 vez/veces
>>> secuencia.index(5)
4
>>> secuencia.count(5)
2
```

Strings. Construcción: `""`, `''`, `"cadena"`, `'cadena'`, `str()`, `str("cad")`.

Un **string** es una cadena o secuencia de valores que representan códigos Unicode y se escriben entre comillas dobles. Python no tiene un tipo char y en su lugar cada carácter se representa como un objeto string con longitud 1.

Funciones asociadas a los caracteres son:

- `ord(char)`: retorna el código decimal de un char
- `chr(codigo)`: retorna el char dado el código (número natural) con la función .

Son métodos de las cadenas:

- `.lower()`/`.upper()`: Retorna una copia de la cadena de caracteres con todas las letras en **minúsculas / mayúsculas**.
- `.capitalize()`: Retorna una copia de la cadena con el **primer carácter** en mayúsculas y el resto en minúsculas.
- `.title()`: Igual pero el primer carácter **de cada palabra** de la cadena.

- `.casefold()`: Retorna el texto de la cadena, normalizado a minúsculas. Los textos **normalizados** pueden usarse para realizar búsquedas textuales independientes de mayúsculas, minúsculas y caracteres idiomáticos.
- `.count(sub[, start[, end]])`: Retorna el número de **ocurrencias** no solapadas de la cadena sub en el rango [start, end].
- `.find(sub[, start[, end]])`: Retorna el menor índice de la cadena s donde se puede **encontrar** la cadena sub, considerando solo el intervalo s[start:end]
- `.split(sep=None, maxsplit=-1)`: Retorna los distintos substring comprendidos entre dos separadores. El separador es el valor de `sep`. El parámetro `maxsplit` indica el máximo número de divisiones. Si no se indica ningún separador elimina todos los espacios y devuelve sólo las palabras que conforman la cadena.
- `.join(lista)`: retorna un string formado por los elementos de la lista.

Hay muchas más funciones en https://docs.python.org/es/3/library/stdtypes.html#text-sequence-type-str.

Ejemplo 3.1

```
>>> str = ' tengo espacios '
>>> print(str, "len=", len(str))
 tengo espacios  len= 16
>>> str.split()
['tengo', 'espacios']
>>> str.join(['ahora', 'extras'])
'ahora tengo espacios extras'

>>> i = 2; j = 11; k = 4
>>> print(str[i])  # Mostrar el elemento i=2
e
>>> print(str[i:j]) # Mostrar los elmentos entre i=2 hasta 10<j=11
engo espa
>>> print(str[i:j:k]) # Mostrar los elementos 2=i+0*k, 6=i+1*k, 10=i+2*k
e a
>>> print(str[-i])  # Mostrar el elemento i=-2, se corresponde con el i=15
s
>>> print(str[-j:-i]) # Mostrar los elementos entre j=-11 (es el 1) hasta el i=-2 (sin incluir)
o espacio
>>> print(str[j:i:-k]) # Adivina
cen
```

Nota *La documentación oficial de **Python** distingue 3 tipos de secuencias básicas: listas, tuplas y rangos. Y considera tipos de secuencias adicionales las específicas para datos binarios (bytes, bytearray, memoryview) y texto (strings).*

Listas. Construcción: `[]`, `[x, y, ...]`, `list()`, `list(iterable)`, por comprehensión.

Una lista es una secuencia ordenada de elementos arbitrarios que es mutable.

El constructor `list()` construye una lista vacía. También se construye con un par de corchetes. Los demás constructores crean listas con elementos.

Ejemplo 3.2

```
>>> lista = ['uno', 'dos', 'tres'];
>>> print(lista, "len=", len(lista))
['uno', 'dos', 'tres'] len= 3
```

También se puede usar el operador * si desea repetir una lista varias veces.

```
>>> l1 = ['uno', 'dos'];
```

```
>>> l2 = ['tres', 'cuatro'];
>>> l1 + l2
['uno', 'dos', 'tres', 'cuatro']
>>> 3*l1
['uno', 'dos', 'uno', 'dos', 'uno', 'dos']
```

Dispone de las operaciones/funciones:

- **`s[i] = x`**. El elemento **`i`** de s es reemplazado por **`x`**.
- **`s[i:j] = t`**. La rebanada de valores de s que van de **`i`** a **`j`** es reemplazada por el contenido del iterador **`t`**.
- **`del s[i:j]`**. Equivalente a **`s[i:j] = []`**.
- **`s[i:j:k] = t`**. Los elementos de **`s[i:j:k]`** son reemplazados por los elementos de **`t`**.
- **`del s[i:j:k]`**. Borra los elementos de **`s[i:j:k]`** de la lista.

Y los siguientes métodos:

- **`.append(x)`** **Añade** un valor. Alternativametne **`lista = l1 + l2`** concatena listas.
- **`.insert(i, x)`**, **inserta** un ítem en una posición dada. [0, 1, 2].insert(1, 10) == [0, 10, 1, 2]
- **`.extend(iterable)`**, **añade** cualquier objeto iterable a la lista.
- **`.pop([x])`**, **extrae y quita** el último elemento o el elemento indicado.
- **`.remove(x)`**, **elimina** el primer elemento que sea igual a **`x`**.
- **`.index(x[,start[,end]])`**, **indica** el índice del elemento **`x`** en el rango dado.
- **`.sort(*, reverse=False)`**, **ordena** los elementos de la lista in situ

```
>>> # Crea
>>> lista = ['uno', 'dos', 'tres']; lista
['uno', 'dos', 'tres']
>>> # Añade
>>> lista = lista + [4]; lista
['uno', 'dos', 'tres', 4]
>>> lista.append(5); lista
['uno', 'dos', 'tres', 4, 5]
>>> lista.insert(0, 'cero'); lista
['cero', 'uno', 'dos', 'tres', 4, 5]
>>> # Elimina
>>> lista.remove ('dos'); lista # Por elemento
['cero', 'uno', 'tres', 4, 5]
>>> del lista[0]; lista  # Por índice
['uno', 'tres', 4, 5]
>>> lista.pop(); lista  # Extrae y quita el último elemento
5
['uno', 'tres', 4]
>>> lista.pop(1); lista # Extrae y quita el elemento indicado
'tres'
['uno', 4]
>>> # Encontrar
>>> lista.index ('dos');
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'dos' is not in list
>>> 'dos' in lista
False
>>> 4 in lista
True
```

Tenga en cuenta que los elementos de una lista son arbitrarios, por lo que puede contener elementos de distintos tipos:

```
>>> lista = ['uno', 2, ['tres', 4]]; lista
['uno', 2, ['tres', 4]]
```

```
>>> lista[1]
2
>>> lista[2]
['tres', 4]
>>> lista[2][0]
'tres'
```

Tuplas. Construcción: `(x, y, ...)`, `tuple(iterable)`, por comprehensión.

La versión inmutable de las listas son las tuplas. Una tupla es una secuencia ordenada de elementos arbitrarios. Se declara con apertura de paréntesis, seguido de los elementos separados por comas y cierre de paréntesis. Una tupla vacía está formada por un par de paréntesis. Una vez creada no se puede modificar pero sí se puede consultar su contenido usando índices o el operador **`in`**.

Las tuplas son más rápidas que las listas y como solo se puede iterar entre ellos, es mejor usar una tupla que una lista si se va a trabajar con un conjunto estático de valores. Se pueden convertir tuplas en listas y viceversa.

- **`tuple()`** recibe como parámetro una lista y devuelve una tupla.
- **`list()`** recibe como parámetro una tupla y devuelve una lista.

Ejemplo 3.3

```
>>> tupla = ('uno', 'dos', 'tres'); type(tupla)
<class 'tuple'>
>>> lista = ['uno', 'dos', 'tres']; type(lista)
<class 'list'>
>>> t = tuple(lista); type(t)
<class 'tuple'>
>>> l = list(tupla); type(l)
<class 'list'>
>>> print(tupla, "len=", len(tupla))
('uno', 'dos', 'tres') len= 3
>>> 2 in tupla
False
>>> tupla[1]
'dos'
```

Rangos. Construcción: `range([start,] stop[, step])`

El tipo **`range`** representa una secuencia inmutable de números y se usa usualmente para bucles que se deben de ejecutar un número de veces (bucles **`for`**)

Un rango se construye con uno, dos o 3 naturales:

- **`range(stop)`** construye un rango que empieza en 0 y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- **`range(start, stop)`** construye un rango que empieza en `start` y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- **`range(start, stop, step)`** construye un rango que empieza en `start`, finaliza en `stop` y cada elemento de la secuencia se diferencia del anterior en una `step`-unidades.

Ejemplo 3.4

```
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple(range(10, 100, 20))
(10, 30, 50, 70, 90)
>>> tuple(range(100, 10, -20))
```

```
(100, 80, 60, 40, 20)
```

Nota *"Los rangos implementan todas las operaciones comunes de las secuencias, excepto la concatenación y la repetición. La ventaja de usar un objeto de tipo range en vez de uno de tipo list o tuple es que con range siempre se usa una cantidad fija (y pequeña) de memoria, independientemente del rango que represente (Ya que solamente necesita almacenar los valores para start, stop y step, y calcula los valores intermedios a medida que los va necesitando)".*

3.C. Colecciones no Secuencias de Python

En este grupo tenemos a los conjuntos (set, frozenset) y los mapas (dict).

Conjuntos. Construcción: `{}`, `{x, y, ...}`, `set()`, `set(iterable)`, por comprehensión.

Estos tipos de datos representan a conjuntos no ordenado de elementos (objetos únicos). Sus términos no pueden ser indexados por ningún subíndice. La función incorporada `len()` devuelve el número de elementos en un conjunto.

Se utilizan para pruebas rápidas de pertenencia, la eliminación de elementos duplicados de una secuencia y el cálculo de operaciones matemáticas como la intersección, unión, diferencia y diferencia simétrica.

Se tienen dos tipos de conjuntos:

- Conjuntos (Set)
  Representan un conjunto mutable. Se declara con apertura de llaves, los elementos separados por comas y cierre de llaves. Usar solo la apertura y cierre de llaves sin elementos define un diccionario, no un conjunto. Para definir un conjunto vacío usa el constructor `set()`.
- Conjuntos congelados (Frozensets)
  Representan un conjunto inmutable. Se crean con el constructor `frozenset()`.

Ejemplo 3.5

```
>>> conjuntoMutable = {"Perl", "Python", "Java"}; print(conjuntoMutable)
{'Java', 'Python', 'Perl'}
>>> set(("Perl", "Python", "Java"))
{'Java', 'Python', 'Perl'}
>>> frozenset(conjuntoMutable)
frozenset({'Java', 'Python', 'Perl'})
```

Nota *Los conjuntos no pueden estar formado por elementos mutables.*

```
>>> set( {1: 2, 3: 4} )
{1, 3}
>>> set( [1   , "h", (0, (3, 4)) ] ); set((0, 1)) # ok
{(0, (3, 4)), 1, 'h'}
{0, 1}
>>> set(([0], [1])) # no ok. Los elementos son listas y éstas son mutables.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Algunos operadores que se tienen con conjuntos son;

- `conjA - conjB` realiza la diferencia del `conjA` con el conjunto `conjB`.
- `conjA -= conjB` calcula la diferencia del `conjA` con el conjunto `conjB` y el resultado se almacena en `conjA`.
- `conjA | conjB` calcula la unión del `conjA` con el conjunto `conjB`.
- `conjA |= conjB` calcula la unión del `conjA` con el conjunto `conjB` y el resultado se almacena en `conjA`.
- `conjA & conjB` calcula la intersección de los `conjA` y `conjB`.

- `conjA &= conjB` calcula la intersección del `conjA` con el conjunto `conjB` y el resultado se almacena en `conjA`.
- `conjA <= conjB` indica si el `conjA` es un subconjunto del conjunto `conjB`.
- `conjA < conjB` indica si el `conjA` es un subconjunto propio del conjunto `conjB`.
- `conjA >= conjB` indica si el `conjA` es un superconjunto del conjunto `conjB`.
- `conjA > conjB` indica si el `conjA` es un superconjunto propio del conjunto `conjB`.

Los métodos equivalentes a estos operadores son:

- `.difference()`: retorna la diferencia de dos conjuntos. `conjA.difference(conjB)` Alternativamente se puede usar el operador `-`.
- `.difference_update()`: retorna la diferencia de dos conjuntos. `conjA.difference(conjB)`. Alternativamente se puede usar el operador `-=`.
- `.union()`: la unión de conjuntos. `conjA.union(conjB)`. Alternativamente se puede usar el operador `|`.
- `.update()`: modifica el conjunto con la unión de conjuntos y el resultado lo almacena en el primer conjunto. `conjA.update(conjB)`. Alternativamente se puede usar el operador `|=`.
- `.intersection()`: la intersección de conjuntos. `conjA.intersection(conjB)`. Alternativamente se puede usar el operador `&`.
- `.intersection_update()`: modifica el conjunto con la intersección de conjuntos. Alternativamente se puede usar el operador `&=`.
- `.issubset()`: indica si un conjunto es subconjunto de otro. `conjA.issubset(conjB)`. Alternativamente se puede usar el operador `<=`.
- `.issuperset()`: indica si un conjunto es un superconjunto de otro. `conjA.issuperset(conjB)`. Alternativamente se puede usar el operador `>=`.

Otros métodos que se pueden realizar con los conjuntos son:

- `.add(x)`: **añadir** un elemento `x`.
- `.discard(x)`: **elimina** el elemento `x`.
- `.remove(x)`: igual que `.discard()` **pero** si el elemento no existe aparecerá un mensaje de error.
- `.clear()`: **borrar** todos los elementos.
- `.copy()`: realizar una **copia** del conjunto.
- `.isdisjoint(conj)`: indica si `conj` **es disjunto** con el conjunto actual.
- `.pop()`: **borra y retorna** un elemento arbitrario del conjunto.

Diccionarios Construcción: `{}`, `{k1: v1, k2: v2, ...}`, `dict()`,`dict(iterable)`, por comprehensión.

Los mapeos representan a conjuntos finitos de objetos pero indexados. La notación de subíndice `m[k]` selecciona el elemento indexado por `k` en el mapeo `m`. Los índices no tienen que ser necesariamente números naturales. Para ello se utilizan tablas hash y políticas de resolución de colisiones.

De forma nativa Python implementa los **diccionarios**, que son su forma de llamar a los mapeos. Representan a una colección mutable y no ordenada de pares clave-valor. La clave puede ser cualquier valor inmutable (ese es el valor que indexa en el mapeo). La razón es que la implementación eficiente de los diccionario requiere una clave que permanezca constante.

Se dispone de la función `len()` que devuelve el número de elementos en un mapeo. Se pueden construir diccionarios vacíos con `{}` y `dict()`. Mediante asignación se define un diccionario usando llaves e indicando los pares (clave, valor):

```
dic = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Ejemplo:

```
dic = {
    'Madrid': 'España',
    'París': 'Francia',
    'Londres': 'Inglaterra'
}
```

Mediante el constructor se pasa como argumento una lista con tuplas (clave, valor):

```
dic = dict([
    (<key>, <value>),
    (<key>, <value>),
      .
      .
    (<key>, <value>)
])
```

Ejemplo:

```
dic = dict([
    ('Madrid': 'España'),
    ('París': 'Francia'),
    ('Londres': 'Inglaterra')
])
}
```

En el caso de que las claves sean string se puede simplificar esta segunda opción quitando la notación de tuplas y realizando la asignación `clave = valor`.

```
dic = dict(
    <key>=<value>,
    <key>=<value>,
      .
      .
    <key>=<value>
)
```

Ejemplo:

```
dic = dict(
    Madrid='España',
    París='Francia',
    Londres='Inglaterra'
)
}
```

Operadores con diccionarios son:

- `d[key] = value`: **Asigna** el valor `value` a `d[key]`.
- `len(d)`: retorna el número de entradas almacenadas; es decir, el número de pares (clave, valor).
- `del d[k]`: borra la clave k junto con su valor.
- `key [not] in` d: indica si la clave [no] está en d.

Métodos con diccionarios son:

- `.setdefault(key,[v])`: **Retorna** el valor de `key`, pero si `key` no existe le asigna el valor `v`. **Muy útil** para para incializaciones.
- `copy()`: realiza una copia del diccionario.
- `.keys()` y `.values()` retornan la lista de **claves** y **valores**, respectivamente.
- `.get(k)`: retorna **el valor de la clave** dada.
- `.items()`: retorna una **lista** cuyos elementos son tuplas **(clave, valor)**.
- `.popitem()`: **Borra el último** para clave-valor añadido al diccionario y lo retorna como tupla.
- `.pop(k)`: **Retorna** el valor de la clave dada **y elimina** del diccionario el par (key,pop(key)).
- `.update(otroDict)`: **fusiona** las claves y valores de los diccionarios. Sobreescribe los valores que tengan la misma clave.
- `.clear()`: **Limpia** el diccionario de pared clave-valor.
- `.items()`: retorna una lista cuyos elementos son tuplas (clave, valor).

```
>>> d = {1:'value','key':2};
>>> print(d)
{1: 'value', 'key': 2}
>>> print(d.keys())
dict_keys([1, 'key'])
>>> print(d.items())
dict_items([(1, 'value'), ('key', 2)])
>>> print("d[1] = ", d[1]);
d[1] =  value
>>> print("d['key'] = ", d['key']);
d['key'] =  2
>>> d1 = dict([[1,2],[3,4]])   ; print(d1)
{1: 2, 3: 4}
>>> d2 = dict([(3,26),(4,44)]) ; print(d2)
{3: 26, 4: 44}
>>> d1.update(d2); print(d1)
{1: 2, 3: 26, 4: 44}
```

Puede interesar `defaultdict` de `collections`. Ej: `d = defaultdict(lambda: "No existe")`. Si `d[clave]` no existe invocará a la función dada pero no lanzará un error.

3.D. Iteración en Colecciones

La Abstracción de Iteración permite recorrer elementos de un contenedor **sin tener en cuenta** su representación interna. Se recurre a un **iterador** que sabe cómo recorrer al contenedor. La abstracción tiene la forma:

```
Para cada elemento P de Contenedor
    acción sobre P
```

En Python algunos contenedores se llaman **iterables**.

- Todas las **secuencias** estudiadas previamente son objetos iterables.
- Los iterables responden a la abstracción de iteración.
- La abstracción se particulariza como:

```
for x in contenedor:
    acción con x
[else : acciones]
```

Si bien para el caso de diccionarios, se tienen estas opciones:

```
for k[,v] in diccionario:
    acción con k [,v]
[else : acciones]
```

```
for x in diccionario.values():
    acción con x
[else : acciones]
```

- En todos los casos se crea un **iterador** interno que recorre los elementos hasta recibir `StopIteration`.
- Se puede obtener el iterador de una secuencia con `iter(contenedor)`.
- **Los iteradores son iterables** que se recorren con `next(iterador)`.
- Un iterador lanza un error si `next()` recibe el valor `StopIteration`.

3.E. Construcción de Contenedores por comprehensión

Un contenedor por comprensión es el que se construye basado en la notación matemática de creación de conjuntos (**builder notation**[1]). Por ejemplo, podemos construir: $S = \{ \underbrace{2 \cdot x}_{\text{Expresión salida}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{Conj. entrada}} , \underbrace{x^2 > 3}_{\text{Predicado}} \}$

Definir conjuntos basándonos en propiedades también se conoce como comprensión de conjuntos, abstracción de conjuntos o definición por intención de un conjunto.
En Python la construcción por comprehensión[2] se expresa como

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

donde:

- `< >` representa a `[ ]` si quiere construir **listas**, representa a `( )` si quiere construir **tuplas** y representa a `{ }` para construir **conjuntos**.
  En el siguiente ejemplo se construye un conjunto porque se usan llaves:

```
>>> {2*x for x in range(5) if x*x > 3}
{8, 4, 6}
```

- `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]
```

  Notar que el condicional sirve para filtrar los valores de `x`. Solo a aquellos valores de la secuencia que cumplan la condición se les aplicará la expresión.
- `expresion(x)` es cualquier expresión sobre `x`. Algunos ejemplos son:
  - No alterar el valor de `x`; es decir, `expresion(x)=x`.
  - Aplicar alguna operación o función. P.e. `expresion(x)=x**2`.
  - Usar alguna método. P.e. `expresion(x)=x.upper()`.
  - Usar expresiones ternarias. P.e. `expresion(x)=x if x%2==0 else 1000`.

Para **diccionarios** usa la expresión:

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

[1] https://en.wikipedia.org/wiki/Set-builder_notation
[2] https://en.wikipedia.org/wiki/List_comprehension

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.

3.F Métodos Mágicos

Cuando se realiza una función u operación sobre un objeto es posible que se delegue la acción en algún método del propio objeto. En estos casos se pueden utilizar los métodos mágicos. Estos métodos se pueden sobrescribir para que la función integrada tenga el comportamiento esperado.

A modo de ejemplo, se indican algunos de estos métodos mágicos junto con la función u operación asociada considerando una situación particular.

Nota *Los métodos mágicos ya fueron introducidos en la pág. 28. Si no los recuerda, lea antes los contenidos de esa sección.*

Suponga que se construye un TDA para el que se considera que la representación será un valor entero (el peso) y un conjunto de datos (un contenedor).

```
struct UnTDA
{
    entero peso;
    secuencia contenedor;
}
```

Para su implementación en Python se opta por usar un `int` para el peso y una lista, `list`, para el contenedor.

- La construcción de objetos para esta clase se realizará con la sentencia `un_objeto = UnTDA()`; es decir, se considera que solo se deben asignar valores por defecto al estado del objeto. La construcción de objetos invoca a dos métodos mágicos, siendo `__init__(self)` el que sirve para establecer el estado inicial del objeto recién construido. ¿Recuerda cuál es el otro método mágico y que no se debe de reescribir nunca?
  En consecuencia, optamos por definir el método de inicialización de esta forma:

```
>>> class UnTDA:
...     #....
...     def __init__(self):
...       self.peso = 0
...       self.contenedor = list()
...
```

- Imagine ahora que quiere usar alguna sentencia para conocer el número de elementos del contenedor del objeto. Una forma de hacerlo es declarando un método, al que llamaremos `def cardinal(self)`, cuya definición se limita a retornar el tamaño de la lista asociada al contenedor. En este caso la clase se definiría como sigue:

```
>>> class UnTDA:
...     #...
...     def cardinal(self):
...       return len(self.contenedor)
...
```

```
# Ejemplo de uso
un_objeto = UnTDA()
# Por aquí lo trasteamos
# Imprimimos el cardinal del contenedor
print( un_objeto.cardinal() )
```

  De acuerdo al ejemplo: para conocer el `.cardinal()` del objeto, delegamos su cálculo a la función `len()` sobre el atributo `self.contenedor`.
  Otra opción es no construir dicho método, pero a cambio poder usar usar la función integrada `len()` cuando nos convenga; es decir, en vez de escribir `print(un_objeto.cardinal())` poder escribir `print(len(un_objeto))`. En Python, la función integrada `len`)) invoca al método mágico `def __len__(self)` del objeto que se pasa como argumento en dicha función. Así, si se opta por esta segunda solución, la clase deberá definirse de esta otra forma:

```
>>> class UnTDA:
...     #...
...     def __len__(self):
...       return len(self.contenedor)
...
```

```
# Ejemplo de uso
un_objeto = UnTDA()
# Por aquí lo trasteamos
# Imprimimos el cardinal del contenedor
print( len(un_objeto) )
```

- Suponga ahora que quiere mostrar la información del objeto con la instrucción `print(un_objeto)`. En este caso se imprimirá lo que retorne la instrucción `str(un_objeto)` y como esta, a su vez, invoca al método mágico `def __str__(self)` se deberá reescribir este método mágico de la siguiente forma:

```
>>> class UnTDA:
...     #...
...     def __str__(self):
...         return '(' + str(self.peso) + ',' + \
...                str(len(self)) + ')'
...
```

En este caso, la instrucción `print(un_objeto)` imprimirá un par de valores que se mostrarán entre paréntesis y separados por una coma. El primer valor se corresponde al valor del atributo peso y el segundo al tamaño del contenedor. Para este segundo valor se supone que se ha redefinido el método mágico **`def`** `__len__(self)` para que retorne el cardinal del contenedor (punto anterior).

Nota *El funcionamiento e implementación del método mágico `__str__()` ya lo ha practicado en la sesión anterior.*

- Suponga que considera que un objeto de esta clase precede a otra si su peso es menor y que quiere hacer la comprobación de si cierto objeto precede a otro.
  Bueno, una forma de hacerlo es construyendo un método, por ejemplo con el nombre `is_lower(self, el_otro)` que retorne **`True`** si `self.peso <= el_otro.peso` y **`False`** en otro caso. Para su uso tendrá que usar la notación punto: `objeto.is_lower(el_otro)`.
  Otra forma puede ser usar directamente esta expresión `un_objeto <= el_otro`. Si usa esta expresión de operación (que no es un método) entonces Python invoca al método mágico **`def`** `__le__(self, el_otro)` y será este el que tenga que reescribir para que retorne **`True`** si `self.peso <= el_otro.peso` y **`False`** en otro caso.

  Dispone de todos estos métodos mágicos para definir los operadores $<$, $\leq$, $==$, $>$ y $\geq$ entre objetos: `__lt__(self, other)`, `__le__(self, other)`, `__eq__(self, other)`, `__ne__(self, other)`, `__gt__(self, other)`, `__ge__(self, other)`.

  Destacar aquí el funcionamiento de la comparación de igualdad:
  por defecto $x == y$ se define como **True if x is y else NotImplemented**
  - **`is` comprueba la identidad**: es **`True`** sii a y b son el mismo objeto.
  - **`==` comprueba la igualdad**: es **`True`** sii `__eq__` es **`True`**.

  No confunda **`is`** con **`==`**.

  El primero comprueba una igualdad en la referencia de los objetos. El segundo invoca a un método mágico.
- Existen métodos mágicos que se asocian a clases que contengan contenedores (p.e. secuencias). Como nuestro ejemplo tienen un contenedor cabe plantearse definir métodos mágicos relacionados con contenedores. Indicaremos solo dos:`__iter__(self)` y `__contains__(self, item)`.
  Antes de explicarlos debe entender qué es un iterador (llamado **`e`**) y un interable: dado un contenedor, un iterador del contenedor es un objeto asociado a dicho contenedor que es capaz de pasar al siguiente elemento del contenedor. Un iterador se crea cada vez que se necesita recorrer el contenedor. Si sobre un contenedor se puede construir un iterador, se dice que el contenedor es un iterable.
  Un ejemplo sencillo con listas de Python es el siguiente:

```
>>> mi_lista = ['h', 'o', 'l', 'a']
>>> string = ''
>>> for e in mi_lista:
...   string = string + e
...
>>> print(string)
hola
```

  Aquí el bucle **`for`** crea un iterador que apunta al primer elemento del iterable `mi_lista` y en cada iteración el iterador se actualiza al siguiente elemento del iterable hasta que no queden elementos.
  Con esta aclaración, continuamos con nuestro problema. Imagine que quiere recorrer los elementos del contenedor de su TDA mediante un **`for`**, para ello el **`for`** necesita tener un iterador que apunte al primer elemento del contenedor. **El método que construye iteradores** se llama **`def`** `__iter__(self)` y deber retornar un objeto de tipo iterador. Como nuestro contenedor en el TDA es una lista y ésta ya sabe crear iteradores, podemos retornar como iterador del cotenedor lo que nos devuelva el creador de iteradores de las listas. El código quedaría así:

```
>>> class UnTDA:
...     #...
...     def __iter__(self):
...         return self.contenedor.__iter__()
...
```

Es resumen, para construir un iterador sobre iterables de cierta clase `class UnTDA` puede delegar en el método de construcción de iteradores sobre dichos iterables. Otro modo de construir iterables lo desarrollará como ejercicio.

El otro método mágico que se usa con secuencias es `__contains__(self, item)`. Está asociado con la comprobación de si un elemento (item) se encuentra en la secuencia. Se invoca cuando se utiliza **`in`** y se usa así: `item in secuencia` o `item not in secuencia`.

Es decir, cuando el intérprete se encuentra `item in secuencia`, entonces se invoca al método mágico `__contains__(self, item)`. Si no lo defines entonces Python iterará sobre la secuencia usando un iterador propio y retornará `True` si lo encuentra y `False` en otro caso.

En principio, cuando trabaje con colecciones de datos, no tendrá que definir `__contains__(self, item)` si los iteradores creados con `__iter__(self)` recorren toda la colección. El cómo se construyen estos iteradores es uno de los ejercicios de esta sesión.

Aquí se ha mostrado unos ejemplos sencillos del vínculo que hay entre construccción de objetos y `__init__(self)`, la función `len()` y el método `__len__(self)`, la función `print()` y `__str__(self)`, los operador de comparación con los métodos mágicos de comparación, la creación de iteradores con la sentencia `for` e `in`. Pero la lista es más amplia:

- A Guide to Python's Magic Methods: https://rszalski.github.io/magicmethods/
- Lista de métodos mágicos asociados a cada operación: https://docs.python.org/3/library/operator.html?highlight=operations

§ 3.G. Arrays en Python

3.G.1. La Estructura Array

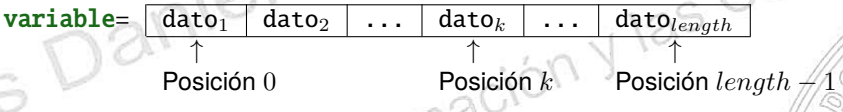
La estructura más básica para almacenar y recuperar un conjunto de datos es el array. La mayoría de los lenguajes de programación proporcionan este tipo de dato estructurado como un tipo de dato integrado/primitivo.

Los arrays son **conjuntos** de datos donde todos son del mismo tipo de dato y para referirse a un dato individual, se utiliza un **índice** que indica el desplazamiento desde el primer elemento del array. Es similar a la notación matemática de variables con subíndices x_i, solo que en los lenguajes de programación se suelen usar corchetes `x[i]`. Así, un array puede contemplarse como una colección de datos que expresado de forma matemática es de la forma $\{x_0, x_1, \ldots, x_n\}$ mientras que expresado en un lenguaje de programación es de la forma { `x[0]`, `x[1]`, ..., `x[n]` } .

Debes saber distinguir muy bien entre posición y elemento de la posición:

- La **posición** (o índice) es el lugar que ocupa cada "caja" (o celda) en el conjunto de array.
- El **elemento** es el dato que hay almacenado en dicha posición (o caja).

Los valores posibles de tales índices se llama **rango** y, para un array de tamaño `n`, toma valores entre `0` y `n-1` (o si se prefiere, entre `0` y `variable.length-1`). En cada posición se almacena un dato.



Ejemplo 3.6

El conjunto de datos $\{2, 4, 6\}$ se puede almacenar utilizando un estructura de array. Una posibilidad es realizar esta asignación: `x[0]=2`, `x[1]=4`, `x[2]=6`.

Los arrays son estructuras mutables en el sentido de que se puede modificar cada uno de sus valores, pero también es inmutable en el sentido de que no se puede cambiar su tamaño una vez establecido. Si un array es de tamaño 10 (es capaz de almacenar 10 datos) podemos modificar cada uno de los datos pero no podemos cambiar la cantidad de datos que puede almacenar.

Hay varias formas de contemplar los arrays:

- Matemáticamente, un array es una **función** definida como la secuencia de valores:

$$\begin{array}{rccl} array: & \{0,1,2,\ldots,n-1\} & \longrightarrow & \{x|\ x \text{ es de tipo } \texttt{tipo\_de\_dato}\} \\ & i & \longmapsto & array(i) \end{array}$$

  como tal función tiene un conjunto imagen $\{array(0),\ldots,array(n-1)\}$ que se corresponde con **una lista ordenada** de valores según su posición o índice.
  Entonces, usando un índice podemos acceder a los elementos de esa lista para conocer el valor del dato o modificarlo.
- Desde una perspectiva más computacional, un array es un **conjunto de datos** del mismo tipo almacenados en la memoria del ordenador en posiciones adyacentes que se agrupan bajo un nombre común y se pueden tratar como una unidad. Es decir, es una estructura de datos homogénea y contigua.
- Pero también se puede contemplar como **un conjunto de variables** (cada posición tiene asociada una variable), siendo `x[k]` la k-ésima variable y como tal variable podemos asignar valores (p.e. `x[k] = 4.6;`) o utilizar los valores que almacena para realizar operaciones (p.e `x[k]+7.8`)

En cualquier caso, ambas perspectivas contemplan lo mismo: un conjunto de valores a los que se accede por un índice. Utiliza la perspectiva que tu resulte más intuitiva.

3.G.2. Listas como arrays

Python por defecto **no proporciona** la estructura de datos array pero sí proporciona la estructura `list` como el tipo de dato para codificar una secuencia mutable. Podríamos usar un objeto de tipo `list` para simular la existencia de un array.
En Python no es difícil simular la construcción de un array por literales.

```
>>> array = [1, 2, 3]  #  Construcción de un "array" con valores literales.
```

Tampoco si se desea inicializar un array 1D con todos su valores iguales, el proceso más sencillo es este:

```
>>> array = [1]*3  # Una lista que concatena 3 veces la lista [1].
>>> print(array)
[1, 1, 1]
```

Incluso tampoco es difícil simular un array 2D (matrices) pues basta construir tantas listas anidadas como se requieran y a cada una asignarle el valor literal.

```
>>> matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> print(matriz[1][2])  # Acceder al elemento 2 de matriz[1]
6
```

Se podría pensar que el siguiente código para simular un array 2D es correcto, pero no funcionará. Descubra por qué.

```
>>> n = 2
>>> m = 4
>>> matriz = [[0] * m] * n  # Una matriz nula de n x m valores
>>> print(matriz)
[[0, 0, 0, 0], [0, 0, 0, 0]]
>>> matriz[0][1] = 10 # Modifica solo el valor (0,1) ??
>>> print(matriz)     # Pero también ha modificado el (1, 1) !!!
[[0, 10, 0, 0], [0, 10, 0, 0]]
```

En su lugar puede usar este código correcto:

```
>>> for i in range(0, n):
...     matriz[i] = [0] * m  # Una matriz nula de n x m valores
...
>>> print(matriz)
[[0, 0, 0, 0], [0, 0, 0, 0]]
>>> matriz[0][1] = 10 # Modifica solo el valor (0,1)
>>> print(matriz)     # Sin modificar el (1, 1)
[[0, 10, 0, 0], [0, 0, 0, 0]]
```

Una forma de simular la asignación de valores diferentes en una matriz es usar un bucle anidado:

```
>>> array = []
>>> for i in range(n):
...     fila = []
...     for j in range(m):
...         fila.append(i*10 + j) # o bien introducir un valor o invocar a función
...     array.append(fila)  # Añadir la fila al array `array`
...
>>> print(array)   # Matriz de n x m valores.
[[0, 1, 2, 3], [10, 11, 12, 13]]
```

3.G.3. Array compacto

Una forma de trabajar con arrays en Python es usar el módulo **array**, que permite representar un array de **valores básicos**: caracteres, enteros y reales. Veamos cómo realizar las 3 operaciones básicas.

- Creación. En teoría, para la creación de un array hay que indicar el número de valores que almacenará, pero en Python con **array** no se necesita. Un **array** se crea indicando el código de tipo.

  ```
  >>> import array
  >>> array.typecodes  # Mostrar todos los códigos de tipo
  'bBuhHiIlLqQfd'
  >>> arr=array.array('i')
  >>> print(arr)
  array('i')
  ```

  El código de tipo puede ser cualquier de los siguientes caracteres `'bBuhHiIlLqQfd'`. Así, por ejemplo, 'i' indica que se trabajará con enteros con signo con 2 bytes, 'f' que serán reales float de 4 bytes, etc ... El significado de cada carácter lo puede encontrar en la documentación de Python.
- Inicialización. Se disponen de los métodos
  - **append(x)**, añade un valor al final del array
  - **insert(i, x)**, inserta el valor x en la posición i

  ```
  >>> arr.append(10); arr.append(5); arr.append(1);
  >>> arr.insert(0,0)
  >>> arr.insert(10, 3) # Si la posición no existe actúa como append
  >>> print(arr)
  array('i', [0, 10, 5, 1, 3])
  ```

- Acceso. Se usa la notación corchete usual de los arrays

  ```
  >>> for i in range(0, len(arr)):
  ...     print(arr[i])
  ...
  0
  10
  5
  1
  3
  ```

- Modificación. Se accede a la posición deseada y se hace la asignación usual con **=**

  ```
  >>> arr[0]=arr[1]
  >>> print(arr)
  array('i', [10, 10, 5, 1, 3])
  ```

Algunos métodos útiles de estos objetos con:

- `count(x)`. Devuelve el número de apariciones de x en el array.
- `index(x)`. Devuelve el índice más pequeño de la primera aparición de x en la matriz.
- `pop([i])`. Elimina el elemento con el índice i de la matriz y lo devuelve.
- `remove(x)`. Elimina la primera aparición de x de la matriz.
- `reverse()`. Invierta el orden de los elementos de la matriz.

Esta forma de trabajar con arrays le puede ser útil si solo quiere trabajar con caracteres, enteros y reales. No es útil para trabajar con otros tipos de datos. Por tanto, estamos obligados a buscar un modo alternativo.

3.G.4 ¿Por qué necesitamos implementar los arrays? Implementación en Python

En los dos apartados anteriores ha visto cómo se usan las listas como si fueran arrays y el tipo de dato array que le ofrece Python. Entonces ¿necesitamos la implementar un tipo de datos abstracto para trabajar con arrays en Python? La respuesta es sí y podemos dar, al menos, tres motivos para ello:

- Por un lado, porque si se trabaja con los arrays proporcionados por las librerías de Python no **es posible trabajar con estructuras de datos** lo que convierte a la librería array en algo inútil salvo para casos particulares.
- Por otro lado, porque **una lista contiene más espacio de almacenamiento del necesario para almacenar los elementos que se encuentran actualmente en la lista**. Este espacio adicional, puede ser hasta el doble de la capacidad necesaria. Esto permite expandir una lista de forma rápida a medida que se agregan nuevos elementos, pero no siempre se requiere hacer una expansión de una lista. Así que un array es el tipo de datos más adecuado para secuencias en la que se conoce el número máximo de elementos desde el principio, mientras que la lista es más adecuada cuando el tamaño de la secuencia necesita cambiar después de que la lista haya sido creada.
- Un tercer motivo para decidir si se usa un array o una lista está en cómo se vaya a utilizar la secuencia. La estructura array consta de un conjunto limitado de operaciones **para acceder** a los elementos individuales que componen la secuencia; **pero si se necesitan otras operaciones**, como por ejemplo eliminar elementos u ordenarlos, entonces una lista es una opción mejor.

Aunque Python no dispone del tipo de dato array, si dispone de la librería `ctypes` que permite usar estructuras del lenguaje C que sí dispone de arrays. Para construir los arrays necesita basarse en el siguiente código:

```
>>> import ctypes   # Mandatorio

>>> size = 3
>>> ArrayType = ctypes.py_object * size # Mandatorio esta línea con la siguiente
>>> miArray = ArrayType()               # Mandatorio esta línea con la anterior

>>> miArray[0] = 100
>>> for i in range(size):      # Recorrer el array
...   print(miArray[i])
...
100
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: PyObject is NULL
```

Notar las tres líneas que son obligadas. Si no se hace así, no se podrán construir Arrays en Python. Notar también que al mostrar el contenido del array se muestra que el objeto es `NULL` (no hay referencia a los objetos) porque no se ha inicializado todo el array, solo su primera posición. El acceso al valor de un elemento no se puede realizar si **antes** no se ha realizado una asignación. Notar también que `NULL` es una palabra de C pero no es de Python - en Python es `None`.

Podemos entonces definir una clase que construya arrays 1D (una secuencia de datos) de esta forma:

```
from ctypes import py_object, Array

class Array1D:
    __slots__ = '_size', '_array'

    def __init__(self, size: int):
        assert size > 0, "Array size must be > 0 Exception"
        self._size: int = size
```

```
        self._array: Array[py_object] = (py_object * size)()
        self.clear(None) # Initialize each element
```

Y que podemos extender a arrays 2D de forma sencilla (un array de arrays):

```
class Array2D:
    __slots__ = '_the_rows'

    def __init__(self,  *args) -> None:
        rows = len(args)  # args is a tuple
        assert rows > 0, "Array rows must be > 0 Exception"
        # Create a 1D array to store an array reference for each row
        self._the_rows: Array1D = Array1D(rows)
        # Create the 1D arrays for each row of the 2-D array.
        for i in range(rows):  # type: int
            assert args[i] > 0, "Array cols must be > 0 Exception"
            self._the_rows[i] = Array1D(args[i])
            i += 1
```

Ejercicios

En estos ejercicios vamos a repetir parcialmente algunas funcionalidades que ya tiene **Python** con sus colecciones (contenedores, estructuras de datos). Queremos replicar esta funcionalidad para que adquieras destreza con los TDA y las estructuras de datos de Python. Por ello, implementarás TDAs usando algunos de las estructura de datos integradas en el lenguaje.

Lo primero que debe de hacer es recordar/memorizar qué métodos existen para las colecciones. Para ello puede volver a leer las secciones previas y https://docs.python.org/3.9/tutorial/datastructures.html.

3.1 Construcción por comprehensión

★★☆ ✍

Construcción de Listas

- Construir una lista de n-ceros. Ej: [0, 0, 0]
- Construir una matriz nula (listas de una lista). Ej: [[0, 0], [0, 0]]
- Convertir una matriz en una lista. Ej: [[1, 2], [3, 4]] → [1, 2, 3, 4]
- Almacenar todos los valores pares de una matriz en una lista
- Convertir todos los valores nulos de una lista en -1. Ej: [2, 0] → [2, -1]
- Indicar si todos los valores de una lista son positivos. Usa `all()`. Ej: [1, 2, 3, -1] es False.

3.2 Más Ejercicios con Secuencias

★★★

Listas de tuplas

- `s.isdigit()` y `s.isalpha()` indica si `s` es un dígito o una letra, respectivamente. ¿Cómo saber si una cadena de caracteres está formado solo por números usando `all()`? Consulte la documentación de Python para saber más de estos métodos/función.
- Extraer todas las combinaciones posibles de las tuplas de 2 argumentos dadas por dos listas. Si l1 = (4, 5), l2=(7, 8), sol=(4, 7), (4, 8), (5,7), (5, 8), (7, 4), (7, 5), ...
- Dada una lista de tuplas quedarse con aquellas cuya longitud esté en un rango [m, M].
- Dadas dos listas de parejas no ordenadas obtener el conjunto intersección de dichas parejas. P.e. para l1 = [(3, 4),(5, 6)], l2 = [(5, 4),(4, 3)], la salida : (3, 4)
- Dada una lista de tuplas, extraer las tuplas que tienen todos su elementos divisibles por K.
- Dada una lista de Tuplas, elimine todas las tuplas con todos los valores None.

3.3 Un poco de estadística

★★☆

Listas y diccionarios

Este ejercicio no es difícil pero requiere de ciertos conocimientos del lenguaje. Estas son algunas pistas:

- `sum(lista)` calcula la suma de los elementos de una lista.
- `max(lista)` calcula el valor máximo de los elementos de una lista.
- `len(lista)` calcula el número de elemento de una lista.
- `lista.sort()` ordena los elementos de la lista.
- `lista[i]` accede el elemento i-ésimo de la lista.
- `lista.append(val)` añade el valor **`val`** a la lista.
- `dict[n]` accede al valor almacenado en el diccionario con clave **`n`**.
- `dict.setdefault(n, 0)`: si en el diccionario no existe la clave **`n`** añade una clave/valor con valores **`0`**/**`n`**. Si existe la clave **`n`** no hace nada.
- `dict.values()` retorna una lista con todos los valores del diccionario.

Usando todo lo anterior:

- Construir una función que retorne la media de una lista.
- Definir una función que calcule la mediana de una lista. Previa ordenación, es el elemento central si es impar o el valor medio si es par.
- Definir una función que calcule las modas de una lista. Guarda en un diccionario la frecuencia de cada elemento: la clave/valor es elemento/suFrecuencia. Uslo para determinar las mayores frecuencias.

Este ejercicio no requiere clases.

3.4 TDA Bag

★☆☆

Implementación de Bag con `list`

Un bag (bolso) es un contenedor que almacena una colección de items donde no importa la posición dónde se almacenó cada uno ni el número de veces que aparecen. Imagínalo como una bolsa de la compra.

Un BAG se podría implementar usando las estructuras de datos de **Python**. Dos opciones son listas y diccionarios. Con listas podemos almacenar cualquier tipo de objeto, incluidos duplicados, lo que quiere decir que cada item tiene su propia referencia. Con diccionarios cada item único puede almacenarse en la parte de la *clave* del par *clave/valor* y un contador que almacene el número de veces que aparece el item en la parte del *valor*. La primera estructura es más adecuada si sabemos que tendremos pocos elementos repetidos, mientras que la segunda es más adecuada si aparecen muchos items repetidos. En este ejercicio se pide implementar el **`TDA Bag`** usando la estructura de datos **`list`** de **Python**.

La definición del TDA Bag la estableceremos así:

Definición 3.1: 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*.

Nota

- *Se asume que para aplicar **todos** estos operadores del TDA (salvo el constructor) una variable de tipo Bag debe ser dada. Así por ejemplo **`len() : int`** realmente debería ser **`len(mybag) : int`**, el operador **`contains(item) : bool`** realmente debería ser **`contains(bag, item) : bool`**, y así sucesivamente.*
  *El **`bag`** NO se especifica para simplificar la notación y resulte más cómoda la lectura.*

- *Por otro lado, debe tener en cuenta que si se implementara el TDA como POO, entonces estos operadores realmente sería métodos de una clase. La implementación de* **`len() : int`** *realmente debería ser* **`bag.len() : int`**, *la implementación de* **`contains(item) : bool`** *será como* **`bag.contains(item) : bool`**, ...
- *Pero si además usa* **Python** *deberá de tener en cuenta que algunos de estos métodos deberán de ser mágicos: La implementación de* **`len() : int`** *realmente sería la implementación de* **`__len__() -> int`**, *etc ...*

  Recuerda existen funciones y operadores en Python que cuando son usados con ciertos objetos, entonces se invocan a sus correspondientes métodos mágicos de la clase del objeto.

Esta observación, deberá tenerla presente para el resto de los TDA que se estudien en el curso.

Para ayudar en la implementación correcta de TDA Bag usando el tipo de dato **`list`** de **Python**, se describe cómo se implementará cada operación de Bag:

- Se usara como estructura una clase con un único atributo de tipo **`list`**.
- Una bolsa está vacía si su atributo es una lista vacía.
- El tamaño de la bolsa se determina por el tamaño de la lista.
  En **Python**, para usar **`len(myBag)`** debes definir el método **`__len__()`**.
- Determinar si la bolsa contiene un elemento específico consiste en comprobar si dicho elemento se encuentra en la lista referenciada por el atributo.
  En **Python**, la forma usual para comprobar si un item se encuentra en una secuencia se usar el operador **`in`**: `item in secuencia`. Para definir este operador hay que definir el método **`__contains__()`**.
- Cuando se agrega un nuevo artículo a la bolsa, se puede agregar al final de la lista, ya que no hay un orden específico de los artículos en una bolsa.
  Usa una de las formas de añadir elementos a la lista referenciada por el atributo.
- La extracción de un artículo de la bolsa se realizará delegando en la operación equivalente para el tipo **`list`** de **Python** sobre el atributo de la clase.
- Los elementos de un bolso se recorrerán usando un iterador de listas.
  Los iteradores en una clase se construyen en **Python** definiendo el método **`__iter__()`**. Define este método en la clase Bag invocando al método **`__iter__()`** de la clase **`list`** y retornando su iterador.
- Opcionalmente, si quieres imprimir los items del bolso puedes hacerlo así:

```
def __str__(self):
    return self._list.__str__()
```

  Toma como referencia este código para resolver algunos de los apartados anteriores.

Construye un programa que ponga de manifiesto el uso correcto de todos los métodos.

Por supuesto, no olvides esta prueba:

```
for item in myBag:
    print(item)
```

3.5 Anagramas

★★☆✍

Anagramas en una lista de palabras

Dada una lista de palabras, se te pide que agrupes las palabras que sean anagramas entre sí. Dos palabras son anagramas si contienen las mismas letras, pero en un orden diferente. Por ejemplo, .ªmorz roma"son anagramas. Para ello se considera la siguiente solución:

- Como estructura de datos se usará la clase **`AnagramFinder`** que consta de un campo que es un diccionario. También tienen los métodos **`.add_word()`** que añade una palabra al diccionario y **`.return_anagrams()`** que retorna los anagramas.
- Como solución, se considera el siguiente programa principal.
  1. Considerara una lista de palabras. Por ejemplo:

```
words = ["ramo", "amor", "roma", "perro", "pera", "arco", "carro", "mar", "ram", "casa", "saca
         "tarde", "dater", "paso", "sopa", "roca", "caro", "rima", "mira"]
```

  2. Construye un objeto de la clase **`AnagramFinder`**.
  3. Para cada palabra construir una clave/valor y añadirla al diccionario del objeto construido en el paso anterior:
     - Ordenar las letras de la palabra alfabéticamente.

- Usar la palabra ordenada como una clave y agregar la palabra original al valor correspondiente a esa clave. El valor es un bag.

4. Imprime los anagramas invocando al método del objeto que retorna los anagramas.

Llama al fichero solución

3.6 TDA Map

★☆☆

Implementación de Map con `list`

La definición del TDA Map la estableceremos así:

Definición 3.2: 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.

Implementa el TDA Map con la estructura de datos que aquí se define.

```
class Map:
    __slots__ = '_dict'  # Solo tendrá un atributo y de tipo list

    class _MapEntry:     # Clase interna. Registros que se guardarán en _dict
        __slots__ = 'key', 'value'
        def __init__(self, key: object, value: object) -> None:
            self.key = key
            self.value = value

# A partir de aquí se implementan los métodos de Map
```

Es decir, un objeto de tipo `Map` tiene el atributo **`self._dict`** que es de tipo lista y se quiere que los elementos de esta lista sean del tipo **`Map._MapEntry`** (que indica que **`_MapEntry`** es una clase miembro de la clase **`Map`**). Estos elemento almacenan parejas key/value.

Observa que **`self._dict`** es un atributo de tipo lista, por lo que **`self._dict[pos]`** accederá al elemento de la posición **`pos`** de dicha lista (que es de tipo **`Map._MapEntry(key, value)`**. Así, **`self._dict[pos]._key`** accederá al atributo **`_key`** del elemento de la posición **`pos`** de la lista **`self._dict`**.

Observa también que para construir un mapa debes usar el constructor **`Map()`**, y que endrás que usar el constructor **`Map._MapEntry(key, value)`** para invocar al inicializador de objetos de la clase interna **`_MapEntry`** de **`Map`**.

Para resolver este ejercicio en **Python** tendrás que definir los métodos mágicos: **`__len__()`** para **`len()`**, **`__contains__()`** para **`contains()`** que en **Python** es **`in`**, e **`__iter__()`** para **`iterator()`**.

Usa el siguiente método auxiliar (privado) para implementar los métodos **`add()`**, **`remove()`**, **`valueOf()`** y **`contains()`** (este último es realmente **`__contains__()`**). ¿Qué es lo que hace este método auxiliar?

```
    def _find_position(self, key: object) -> Optional[int]:
        for i in range(len(self)):
            if self._dict[i].key == key:
```

```
            return i
        return None
```

3.7 TDA Set

★★☆

Implementación de Set con `list`

Un conjunto se puede contemplar como un Bag donde **no se permite** la repetición de elementos. Construye un módulo con el TDA Set usando la estructura de datos `list` de Python.

La definición del TDA Set la estableceremos así:

Definición 3.3: 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.

Para resolver este ejercicio en Python tendrás que definir los métodos mágicos: `__len__()` para **`len()`**, `__contains__()` para **`contains()`** que en Python es `in`, `__eq__()` para **`equal()`** que en Python es `==`, e `__iter__()` para **`iterator()`**.

3.8 Cursos de un Título

★★☆ ✍

Ejemplo de uso de Set y Map

Para este ejercicio usa tu implementación del TDA Map y del TDA Set limitándote al uso de los operadores del TDA. Si alguno no lo has implementado usa entonces la estructura de datos `dict` y `set` que te ofrece Python.

Implementa las siguientes clases con las especificaciones que se indican:

- **`Asignatura`**. Una asignatura se caracterizará por un nombre, código numérico de la asignatura y curso en el que se imparte (un número entre 1 y 4). Estos 3 atributos se pueden consultar. Dos asignaturas son iguales si tienen el mismo código (implementa el método `__eq__()`).

- **Asignaturas**. Representa a un conjunto de asignaturas. Se pueden añadir asignaturas al conjunto y puede calcular un diccionario cuya clave es el curso y el valor es el conjunto de asignaturas de dicho curso.
- **Alumno**. Cada alumno se caracteriza por tener un nombre y apellido, un número de identificación y una fecha de nacimiento (es un TDA). Cada alumno se matricula de un conjunto de asignaturas (de tipo **Asignaturas**) que pueden ser de distintos cursos. La lista de asignaturas de un alumno se puede consultar.
- **Alumnos**. Es un conjunto de alumnos al que se le pueden añadir alumnos. Puede retornar un diccionario con clave igual al curso y valor el conjunto de alumnos que están matriculados en ese curso. También puede calcular un diccionario con clave igual al código de una asignatura y cuyo valor es el conjunto de alumnos matriculados en dicha asignatura. Observa que para este segundo diccionario tendrás que suministrar por parámetro la lista de asignaturas. Construye una función auxiliar que calcule el conjunto de alumnos que están matriculados en una asignatura (dada por su código) y úsala para el segundo diccionario.

Para comprobar el funcionamiento correcto de tus clases construye 6 asignaturas y las almacenas en una variable de tipo **Asignaturas**. Comprueba que se pueden calcular las asignaturas de cada curso.

Construye también 4 alumnos y asígnales algunas asignaturas de las 6 construidas anteriormente. Almacénalos en una variable de tipo **Alumnos** y construye los dos diccionarios que pueden calcularse en esta clase.

3.9 Iterador de listas

★★☆ ✍

Crear un iterador

Ya sabe que existen 3 tipos de abstracciones. Una de ellas es la abstracción por iteración. En la implementación de sus TDA diccionario, conjunto y bolsa tuvo construir iteradores delegando en el constructor de iteradores de la clase `listo` `dict`. En este ejercicio estudiará como implementar sus propios iteradores para cualquier contenedor de datos que construya y para ello tomaremos como referencia el tipos de dato integrado `list` (https://docs.python.org/3/library/stdtypes.html).

Consideremos el siguiente código en **Python** :

```
>>> lista = [1, 2, 3, 4] # Crea una lista literal
>>> for elem in lista:  # Itera sobre los elementos de la lista
...   print(f'Elemento {elem}')
...
Elemento 1
Elemento 2
Elemento 3
Elemento 4
```

El bucle `for` muestra que una lista es de tipo iterable. Esto significa lo siguiente:

- La lista, que es un contenedor, tienen el método `container.__iter__()` que retorna un iterador, `iterator`, que es creado por este método.
- El iterador es un objeto que recorre la lista y tiene dos métodos:
  - `iterator.__iter__()`, que retorna la referencia del propio iterador.
  - `iterator.__next__()`, que retorna el siguiente elemento del contenedor `container`. Cuando no hay más elementos lanza un error con la sentencia `raise StopIteration`.

Puede consultar https://docs.python.org/3/library/stdtypes.html#iterator-types si quiere consultar la documentación oficial.

En este ejercicio, vamos a crear nuestra propia lista con su propio iterador. Para ello sigue los siguientes pasos escribiendo todo el código en un único fichero.

1. Construye la clase `MiLista` con el atributo `_lista` que es de tipo `list`.
2. Añade el método de inicialización `__init__()` que recibe como argumento una lista ya creada previamente.
3. Añade el método `__iter__()` en la clase `MiLista` para que construya y retorne un iterador que sea una instancia de la clase `_IteradorDeMiLista`.
4. Para ello necesitará construir la clase `_IteradorDeMiLista` que tendrá estos dos atributos: (1) `_listRef` que almacena la referencia de la instancia de `MiLista`, y (2) `_pos` que es un entero que se inicializa a 0. `_pos` tomará valores en el conjunto de índices de la lista `_listRef`.
5. Añade al iterador el método `__iter__()`, en la clase `_IteradorDeMiLista`, para que retorne una referencia a sí mismo.

6. Y por último añade al iterador el método **__next__()**, en la clase **_IteradorDeMiLista**, para que retorne el elemento de la lista dado por **_pos** e incrementa este valor en dos unidades. Ten en cuenta que si **_pos** se corresponde a un índice que está fuera de rango entonces debe ejecutar la sentencia **raise StopIteration**.
   Notar que este iterador retornará solo los valores de las posiciones pares.

Usa este ejemplo de uso para comprobar que lo has implementado correctamente:

```
>>> miLista = MiLista([0,1,2,3,4,5,6,7,8])
>>> for elem in miLista:
...   print(elem)
...
0
2
4
6
8
```

¿Cómo se podría modificar el código del iterador para que recorra todas las posiciones pero primero que muestre los valores de las posiciones pares y después las de las impares?

.....

3.10 TDA Array1D

★★☆✍

Implementar TDA Array1D

Resueltos todos los ejercicios anteriores, debería poder implementar los array 1D en **Python** con las siguientes especificaciones.

Definición 3.4: 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 3.12.
- **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 3.12.
- **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 3.9

Se podría alegar que un Array no es un TDA pues realmente se implementan a nivel de hardware pero a este nivel solo podemos hacer unas pocas operaciones. Sin embargo, desde la perspectiva de un TDA le hemos dotados de un iterador y de más operaciones (p.e. **length()**). Es decir, se proporciona un alto nivel de abstracción a los array implementados a nivel de hardware.

Como ejemplo de uso, el siguiente código debe funcionar:

```
from TDAArray import Array1D
import random

def main():
    # Construcción de un array de 3 elementos
    mi_array = Array1D(3)
    # Asignación de  datos  aleatorios
```

```
    for i in range(len(mi_array)):
        mi_array[i] = random.random()

    print("Todos los datos son aleatorios")
    for value in mi_array:   # Check iterator
        print(value)

if __name__ == '__main__':
    main()
```

Se espera que construya el TDA Array1D de acuerdo a lo indicado en la página 48 (altamente recomendado), claro que, alternativamente puede construir la clase Array1D usando listas como arrays (desaconsejado). Recuerde que una cosa es el TDA y otra bien distinta es la estructura de datos que se usa para su implementación. En este caso para el TDA Array1D puede usar dos estructuras de datos pero, por definición de array, es más correcta la estructura que usa **ctypes**.

3.11 TDA Array 2D

★★★✍

Implementación del TDA Array 2D

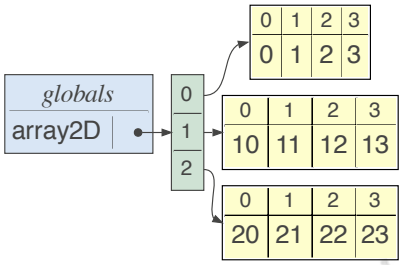
Si el valor de cada posición de un Array1D es una referencia a otro Array1D, entonces obtenemos un Array 2-dimensional.

Una tabla formada por $3 \times 4 = 12$ datos.
Las filas de la tabla se enumeran en **0..2** y las columnas en **0..3**.

Ejemplo 3.1.

	0	**1**	**2**	**3**	←**j**
0	0	1	2	3	
1	10	11	12	13	
2	20	21	22	0	
↑					
i					

Implementación de una tabla de 3 filas y 4 columnas con un array 2D.



□

Pero un array 2D no está obligado a que cada fila tenga el mismo número de columnas. Por ejemplo, la primera fila podría tener un valor, la segunda 10 valores, la tercera 5 valores, etc.

En este ejercicio deberá de implementar el TDA Array 2D usando el TDA Array 1D y con las siguientes especificaciones.

Definición 3.5: 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.

Yo te construyo el método de inicialización y lo demás corre de tu cuenta:

```
class Array2D:
  _the_rows: Array1D

  def __init__(self, *args: int):
    cols = tuple(args)
    rows = len(cols)
    assert rows > 0, "Array rows must be > 0 Excepcion"
    self._the_rows = Array1D(rows)
    for i in range(rows):
      assert cols[i] > 0, "Array cols must be > 0 Excepcion"
      self._the_rows[i] = Array1D(cols[i])
```

Cuando en un método o función se usa ***args** se está indicando que se considera una cantidad indeterminada de parámetros y que todos ellos serán empaquetados. Con el constructor indicado, se puede crear una instancia de Array2D indicando el número de columnas que tendrá cada fila. Por ejemplo, si se invocara como `array2D = Array2D(4, 7)` se construirá un array 2D cuya primera fila tendrá 4 elementos y la segunda fila tendrá 7 elementos.

3.12 Acceder a datos indexados

★☆☆

Métodos de indexación **[ ]**.

Supón que construyes la clase Matriz con los atributos:

`._rows: int`, `._cols: int` y `._data: List[List[float]]`.

Para acceder/modificar cierto elemento del array (usando POO) será necesario construir algunos métodos, que llamaremos **getItem(r: int, c: int) : float**, **setItem(r: int, c: int, v: float)** donde: **r** es la fila, **c** es la columna y **v** es el nuevo dato. Por ejemplo, para obtener el elemento (2, 0) de la matriz se usará la notación punto de la forma `matriz.getItem(2, 0)`. Pero al trabajar con matrices ¿no sería deseable acceder con la notación indexada `matriz[2, 0]` en vez de usar la notación punto? Sin duda para manipular datos indexados es más cómodo usar índices que usar métodos.

Python permite acceder a elementos mediante índices si la clase implementa los métodos mágicos **__getitem__(self, index)** (para retornar el valor del índice dado) y **__setitem__(self, index, value)** (para asignar un nuevo valor a la posición del índice dado).

En ambos métodos **index** es una **tupla** que contendrá uno o varios números de tipo **int** que representará a los índices. En el caso de que **index** tenga un solo índice se interpretará como un entero.

La invocación a los métodos mágicos se hará con la escritura **objeto[i1, i2, ...]**, poniendo entre corchetes los índices. No confundir con la notación lista **iterable[i1][i2]...** de listas o tuplas.

Como ejemplos simples, suponiendo que las clases de los objetos que se indican ya tienen implementados los métodos mágicos adecuadamente:

- datos[4], invocará a **__getitem__(4)** e **index** consta de un solo valor, el 4.
- datos[4] = 20, invocará a **__setitem__(4, 20)** e **index** consta de un solo valor.
- matriz[2, 4], invocará a **__getitem__((2,4))** e **index** consta de una tupla con dos valores.
- matriz[2, 4] = 20, invocará a **__setitem__((2, 4), 20)** e **index** será la tupla (2, 4).

Lo que se propone en este ejercicio es que en vez de implementar los métodos **getitem()** y **setitem()** implementes los métodos **__getitem__(self, index)** y **__setitem__(self, index, value)** donde **index** es una tupla. Una tupla es un tipo de dato similar a una lista pero es inumutable (ver https://docs.python.org/3/library/stdtypes.html para detalles). Implementados estos métodos podrá acceder al atributo que almacena todos los datos usando dos índices, donde el primero representa a la fila y el segundo a la columna.

Como ejemplo de uso, el siguiente código debería funcionar:

```
from TDAArray import Array2D

def main():
    # Building an array2D (4-rows and columns 2, 4 ,6, 2)
    mi_array = Array2D(2, 4, 6, 2)
    # Asignación de  datos  aleatorios
    for i in range(mi_array.num_rows()):
        for j in range(mi_array.num_cols(i)):
            mi_array[i, j] = (i+1)*10+(j+1)
    # Print data
    for value in mi_array:   # Check iterator
        print(value)

if __name__ == '__main__':
    main()
```

Aplica esta técnica de indexación de datos siempre que puedas en tus propios tipos de datos (Array1D, Array2D, Matriz, ...). Es más **¿a que esperas para implementar estos métodos en los dos ejercicios anteriores?**

3.13 Matrices

★★☆

Matriz con indexación **[r,c]**.

Construye la **clase Matriz** con los 3 atributos que se han indicado de tal forma que el siguiente código funcione. Las datos de la matriz se generan con el criterio que se desee.

```
m = Matriz(3, 6)
print(m[2, 0])  # __getitem__() retorna el dato (2, 0)
print(m)        # usar self[i, j] en __str__() para que imprima la matriz por filas
m[2, 0] = 100   # __setitem__() modifica el dato (2, 0)
print(m[2])     # __getitem__() retorna la fila 2
```

Observa que para este ejercicio, **__getitem__(self, index)** tendrá que analizar el caso en el que reciba uno (para retornar una fila) o dos enteros (para retornar un valor) .

3.14 Juego de la Vida

★★☆

Aplicación de Array 2D

Un autómata celular bidimensional consiste en un array $n \times m$ donde cada celdilla (llamada célula) solo toma dos posibles valores, que indicaremos por **False** (está muerta) y **True** (está viva). En cada iteración las célula cambia su valor en función de los valores de las 9 células que la rodea en la iteración anterior. Se utilizan los siguientes criterios:

1. Una célula muere si dicha célula no tiene más de 1 vecino vivo o si tiene más de 3 vecinos vivo.
2. Se reemplaza una célula muerta por una viva si dicha célula tiene exactamente 3 vecinos vivos.
3. Una célula viva permanecerá en ese estado si tiene 2 o 3 vecinos vivos.
4. Se considera que la frontera es periódica: las células de los bordes "se unen" con las células del borde opuesto (el array tiene forma de toro - "donut").

Los autómatas celulares son modelos matemáticos y existen varios tipos. El que aquí se indica también se le llama Juego de la Vida y se puede estudiar su comportamiento usando el siguiente TDA.

Definición 3.6: TDA Juego de la Vida

El Juego de la Vida es un autómata celular que se representa en un grid o matriz $n \times m$. Cada celda representa a un célula que puede estar viva o muerta de acuerdo a ciertas reglas de reproducción.

- **`LifeGrid (nrows, ncols)`**. Crea un array 2-dimensional que constará de **`nrows`**-índices fila y cada una tendrá y **`ncols`** columnas. Los argumentos tienen que ser positivos.
- **`numRows() : int`**. Retorna el número de filas del grid.
- **`numCols(row) : int`**. Retorna el número de columnas del grid.
- **`deadCell(i, j) : None`**. Mata a la célula de la posición **`(i, j)`**.
- **`liveCell(i, j) : None`**. Resucita a la célula de la posición **`(i, j)`**.
- **`isLiveCell(i, j) : boole`**. Indica si la célula de la posición **`(i, j)`** está viva o no.
- **`numLiveNeighbors(i, j) : int`**. Indica cuántas células vecinas de la posición **`(i, j)`** están vivas.
- **`evolve() : None`**. Modifica todas las células del grid de acuerdo a las reglas de evolución.

Obviamente, para su implementación tendremos un atributo que será un **`Array2D`** con lo que **tienen la obligación de utilizar** los métodos de esta clase. Así, por ejemplo, la implementación de **`deadCell(i, j)`** es simplemente invocar al método **`setItem(i, j)`** de **`Array2D`**. Tenga en cuenta en la implementación las particularidades de Python.

Como ejemplo de uso haga un programa que cree un juego de la vida y lo haga evolucionar 10 veces. Para ello implemente las funciones **`inicializar(automata: LifeGrid)`** donde todas las células estén muertas menos las que considere y **`mostrar(automata: LifeGrid)`** que muestre el estado del autómata en cada paso evolutivo.

3.15 Reversi

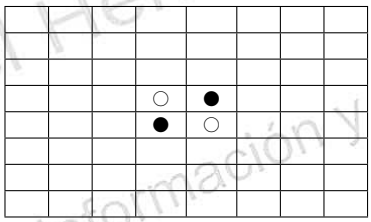
★★★

Aplicación de Array 2D

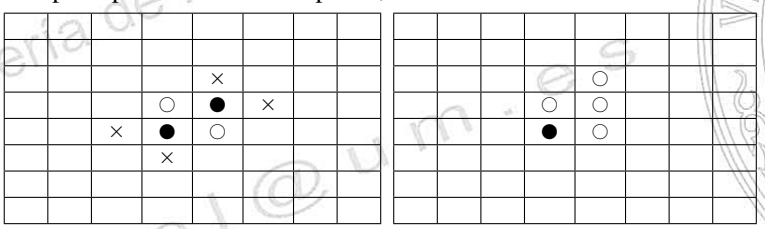
Reversi es un juego para dos jugadores sobre un tablero de 8×8 celdas. Cada jugador tiene un número de fichas ilimitadas. Las fichas circulares tienen un color diferente en cada una de sus caras: por un lado es negra y por el otro lado es blanca. Un jugador juega a blancas - JugadorB - y otro jugador juega a negras - JuegadorN -. En cada turno se añade un ficha nueva y algunas fichas blancas del tablero pasarán a negras (si juega el jugadorN) o algunas fichas blancas del tablero pasarán a blancas (si juega el jugadorB). Cuando ya no se puedan poner fichas nuevas, ganará el jugador que tenga sobre el tablero más fichas de su color.

En cada turno, un jugador deberá de poner una ficha de su color en un celda. La celda tiene que cumplir que desde esa posición (1) deben existir en el tablero al menos una celda de su color en horizontal, vertical o diagonal, y (2) entre la celda donde se colocará la ficha y la otra celda del tablero deberán existir fichas del jugador contrario. Una vez colocada la ficha en la celda, se dará la vuelta a las fichas del jugador contrario que se encuentren entre la celda y la otra celda del tablero. En el caso de que un jugador no pueda poner ficha porque no es capaz de encontrar una celda válida el turno pasa al otro jugador. El juego finaliza cuando ninguno de los jugadores encuentran celdas válidas.

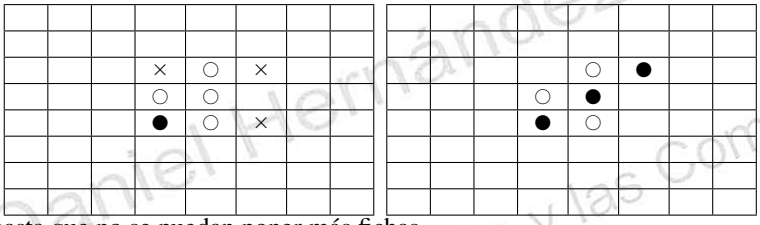
La configuración inicial del juego siempre empieza con 4 fichas ocupando las posiciones centrales y de la siguiente forma:



Siempre empiezan las blancas, por lo que de acuerdo a las reglas de movimiento las celdas candidatas $\times$ se muestran a la izquierda. Si el JugadorB optara por la celda más superior, el tablero se modifica como se muestra a la derecha:



A continuación se muestran las opciones del JugadorN y su movimiento final.



El juego continúa así hasta que no se puedan poner más fichas.

Se pide lo siguiente:

- Realiza un proceso de abstracción para definir el TDA Reversi que contenga al menos los siguientes métodos: el constructor de juegos para tableros de tamaño $n \times n$ ($n \geq 4$ y par), el que inicializa un tablero, el que indica qué jugador gana, el que indica si una celda es un movimiento legal para un jugador, el que indica si una celda está ocupada, el que realiza un movimiento legal para un jugador.

  Recuerda que los métodos son públicos, es decir, son los que se podrán usar desde otro módulo (p.e. un programa principal). Puedes construir métodos auxiliares (privados) para ciertos métodos públicos pero esos no se indican en el TDA.
- Como ejemplo de uso, desarrolla un programa que permita al ordenador jugar contra sí mismo y tenga un función que permita mostrar en la consola la situación del tablero en cada turno.

  Para que sea visible, hazlo para un tablero de 4×4.

Sesión 4

Listas

Índice Parcial **Página**

4.A. TDAs Lineales . . . 61
4.B. Implementación de TDAs lineales . . . 62
4.C. TDA Lista . . . 63
4.C.1. Implementación con Arrays . . . 63
4.C.2. Implementación con Estructuras Enlazadas . . . 64
4.D. Búsqueda y Ordenación . . . 67
4.D.1. Búsqueda . . . 68
4.D.1. Ordenación . . . 69
4.1. Listas con Array1D . . . 72
4.2. Lista Simplemente Enlazada . . . 72
4.3. Lista Simplemente Enlazada con Índices . . . 72
4.4. Lista Doblemente Enlazada . . . 73
4.5. Módulo de Búsqueda . . . 73
4.6. Módulo de Ordenación . . . 73
4.7. Ordenar a los alumnos . . . 74

Teoría

4.A TDAs Lineales

Son aquellos que se pueden expresar como una sucesión finita de elementos $a_1, a_2, \ldots, a_n$. Aunque cada TDA tiene sus propias operaciones, las más importantes son:

- Agregar un elemento a la sucesión.
- Eliminar un elemento de la sucesión.
- Consultar un elemento de la sucesión.

Destacan los TDAs con ordenación lineal que son aquellos donde cada objeto tiene exactamente un predecesor inmediato o un sucesor inmediato. Destacan el primer objeto (que no tiene predecesor) y el último objeto (que no tiene sucesor). Matemáticamente,

Una relación binaria $a \preceq b$ (a precede o es igual a b) entre dos objetos es una ordenación lineal sii cumple las siguientes propiedades:

Completa o total. $\forall\ a, b$, o bien $a \preceq b$ o bien $b \preceq a$,

Transitiva. si $a \preceq b$ y $b \preceq c$, esto implica que $a \preceq c$, y

Antisimétrica. Si se cumple $a \preceq b$ y $b \preceq a$, se sigue que $a = b$

Diremos que $a \prec b$ (a precede a b) sii $a \preceq b$ y $a \neq b$.

Las operaciones adicionales en este tipo de contenedor son:

- Acceder al k-ésimo objeto del orden. En particular, al primer o último objeto en el orden lineal,
- Recorrer todos los objetos del contenedor en dicho orden.
- Dado un objeto a que puede, o no, estar en el contenedor,
  - buscar el objeto predecesor o sucesor en el contenedor.

- contar todos los objetos que le preceden o todos los que le suceden.

- Inserta un nuevo objeto o reemplazar el objeto en la k-ésima posición,
- Dada una referencia al k-ésimo objeto, inserte un nuevo objeto antes o después de ese objeto; o eliminar el objeto antes o después del k-ésimo objeto.

Algunos ejemplos de este tipo abstracto de contenedores son: string, listas, listas ordenadas, pilas, colas, colas de prioridad.

También se incluyen en este tipo de TDA a aquellos donde la relación entre ellos no es relevante. P.e. conjuntos abstractos o conjuntos asociativos abstractos.

Las operaciones en este tipo de contenedor son las operaciones que tiene cualquier contenedor general:

- Acceder al número de objetos del contenedor,
- Determinar si el contenedor está vacío,
- Insertar un nuevo objeto en el contenedor,
- Determinar si un objeto está en el contenedor (membresía),
- Sacar un objeto del contenedor y
- Quitar todos los datos (limpiar) del contenedor.

En la Sesión 3 ya estudió los tipos de datos integrados en Python relacionados con las secuencias. También implementó los diccionarios, conjunto y bags usando el tipo de dato `list`. En esta sesión vamos a estudiar la forma generar de implementar los TDA lineales e implamentaremos algunos en particular.

4.B Implementación de TDAs lineales

Existen 3 grandes grupos de estructuras de datos para implementar los TDAs lineales:

- Arrays. La colección de objetos se sitúan en cada una de las posiciones del array.
- Otras estructuras integradas del lenguaje. En Python, por ejemplo, se pueden usar la secuencia más adecuada.
- Estructuras enlazadas. Una estructura enlazada es aquella que se construye utilizando como estructura básica una que tiene el siguiente patrón:

```
struct node {
   TD valor;
   node reference1;
   node reference2;
   ...
}
```

Esta estructura básica, llamada también como **nodo, consta de un campo donde se almacena un valor y uno o varios campos que almacenan referencias a la misma estructura básica** (hace referencia a otros nodos del mismo tipo).

Nota *Un nodo nos sirve para representar muchos conceptos como los elementos de una lista o los vértices de un árbol o grafo. Además esta representación puede ser distinta incluso para representar el mismo concepto. Por ejemplo, se puede representar una lista como una lista simplemente enlazada, una lista doblemente enlazada o una lista circular. Como verá más adelante, cada una de las 3 representaciones indicadas usan una estructura de nodo, solo que cada una de las representaciones usa las referencias del nodo de forma diferente. Los nodos también se usa en técnicas de búsqueda (búsqueda en anchura, en profundidad, Dijkstra, etc). En la práctica, nos podemos encontrar con la necesidad de mezclar nodos de distintos tipos, naturaleza y propósitos. Por ejemplo, usar nodos para representar una lista y usar otros nodos para hacer una búsqueda a partir de una lista que está representada mediante nodos.*

El concepto de nodo siempre es el mismo: es una representación que consta de valor+referencias. Cómo y para qué se va a usar condiciona al valor que almacena (o valores) y la cantidad de referencias.

Uno podría argumentar que si un lenguaje de programación ya tiene implementado los TDAs para qué dedicar esfuerzos en implementarlos de nuevo. Hay dos motivos obvios por los que conviene aprender a implementarlos:

1. En una asignatura de programación debe conocer cómo se implementan los TDAs. Es un conocimiento básico de programación.
   Es como decir que para qué aprender derivadas si ya hay programas informáticos que las calculan.
2. El TDA implementado por el lenguaje puede contener primitivas insuficientes o poco adecuadas para nuestros propósitos.

Puede verse obligado a tener que extender la clase que los implementa o simplemente adaptarla a su necesidades. Pero esto no será posible si no conoce el punto anterior.

4.C TDA Lista

Definición 4.1: 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.
- **`add(value) : None`**. Añade un nuevo valor a la lista.
- **`pop(value) : Value`**. Retorna el primer valor **`value`** que aparezca en la lista, eliminando dicho valor de la lista.
- **`peek(value) : int`**. Retorna la posición donde aparezca el primer valor **`value`** que aparezca en la lista. El elemento no se elimina de la lista.
- **`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.

4.C.1 Implementación con Arrays

La forma más sencilla para representar una lista $< a_0, a_1, \ldots, a_{n-1} >$ de n-elementos con arrays es usar como representación un simple array de objetos, `object[] list` de tamaño n, donde cada posición almacena un elemento de la lista sin quedar huecos libres en el array. Pero no es difícil ver que esta representación es muy ineficiente pues cada vez que se añada o elimine un elemento de la lista será necesario reconstruir el array de nuevo. Por ello vamos a usar la siguiente representación:

```
struct rep {
   int pos;
   int len;
   object[] list;
}
```

La **función de abstracción**: $Abst$: **rep** $\longrightarrow \mathcal{A}$ la definiremos como

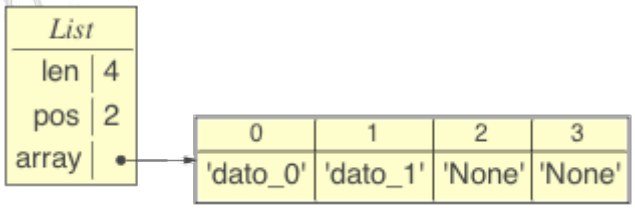
$$Abst(\texttt{r}) = <\texttt{r.list[0], r.list[1], ..., r.list[len-1]}>$$

Las representaciones legítimas vienen dadas por un predicado, el **invariante de la representación** I : **rep** $\longrightarrow \mathbb{B}$, que lo definimos de esta manera:

$$I(\text{r}) = \begin{cases} \forall p, p < \text{r.pos} \Rightarrow \text{r.list[r.p]} \neq NULL \;\&\& \\ \forall p, p \geq \text{r.pos} \Rightarrow \text{r.list[r.p]} = NULL \;\&\& \\ \text{r.list} \neq NULL \;(\text{r.len} > 0) \end{cases}$$

Es decir, en nuestra estructura, `list` almacenará los elementos de la lista, `len` almacena el tamaño de la lista y `pos` almacena la siguiente posición no vacía. En el array todos los valores deberán de estar de forma consecutiva; es decir, no se permitirá que existan entre dos valores posiciones sin valor asignado. Esto obliga a que el primer elemento que se añada a la lista ocupará la posición `0`.

Ejemplo 4.1. Si se considera la estructura indicada para una capacidad de 4 elementos, se puede representar cualquier lista que conste como mucho de 4 términos. En particular, la lista $< dato_0, dato_1 >$ se puede representar de esta forma:



□

Notar que en todas las primitivas de la definición de una lista, el valor de **`pos`**, será un entero entre 0 y **`length()-1`**. Indicar también que un array se dimensiona a cierto tamaño y, por tanto, la lista tendrá una cantidad finita de elementos (la del tamaño del array). Un exceso de elementos generaría un error de capacidad. Además, un iterador para una lista no es más que un iterador del array de la estructura de representación.

4.C.2 Implementación con Estructuras Enlazadas

Las Estructuras Enlazadas son las que cumplen estas condiciones:

- Representan a colecciones.
- Constan de nodos (pág 62).
- Cada nodo contiene información de interés.
- Cada nodo contiene referencia a otros nodos.
- Los nodos se crean cuando son necesario por lo que no tienen por qué almacenarse de forma consecutiva en la memoria.

La versión más simple de un nodo es la que consta de un valor y una referencia a otro nodo. La estructura construida con estos nodos se llama **Estructura Simplemente Enlazada** y es la que usaremos para representar a una lista.

Una **Lista Simplemente Enlazada** es la representación de una lista que usa una Estructura Simplemente Enlazada:

```
struct node {
   TD valor;
   node next;
}
```

La **función de abstracción**: $Abst$: **node** $\longrightarrow \mathcal{A}$ la definiremos como

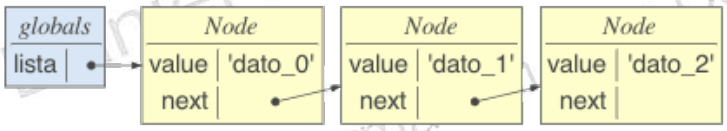
$$Abst(\texttt{r}) = <\texttt{r.valor, r.next.valor, ..., r.next.....next.valor}>$$

Las representaciones legítimas vienen dadas por un predicado, el **invariante de la representación** I : **node** $\longrightarrow \mathbb{B}$, que se define con las siguientes condiciones:

- Consta de una colección de nodos donde cada uno tiene una referencia a otro nodo (cuyo campo llamaremos **`siguiente`**/next).
- Dos nodos diferentes no tienen como referencia **`siguiente`** el mismo nodo.

- Se tiene una variable externa que hace referencia a aquel nodo tal que a partir de él se puede recorrer todos los elementos de la lista pasando por la referencia del **`siguiente`**.

Ejemplo 4.2. De esta forma se representará la lista $< dato_0, dato_1, dato_2 >$ según una lista simplemente enlazada:



□

Observaciones:

- Las estructuras enlazadas, a diferencia del array, no necesita hacer reservas de posiciones consecutivas de memoria, con lo que la lista puede aumentar mientras que exista memoria suficiente.
- Los elementos `Node.value` de la lista se localizan a través de la posición (referencia) del nodo `Node` que lo contiene. Así pues, **`pos`** en todas las primitivas debería ser una referencia a un nodo, mientras que en un array la posición es un índice del array.

Operaciones básicas en estructuras simplemente enlazadas

Las operaciones que mejor debes entender son

- Recorrer los nodos
- Buscar un nodo
- Añadir un nodo como siguiente de otro.
- Eliminar el nodo siguiente de otro.

Recorrer los nodos se puede expresar de la siguiente forma:

```
actual = primer nodo de la lista
mientras actual is not None:
  trabajar con actual
  actual = actual.next
```

Trabajar con actual suele consistir en invocar a alguna función pasando como parámetro `actual` (p.e. `print(actual.value)`). No es necesario recorrer toda la lista. En ocasiones el número de actualizaciones de `actual` viene dado por el valor **`pos`** de las primitivas.

Buscar un nodo se puede expresar de la siguiente forma:

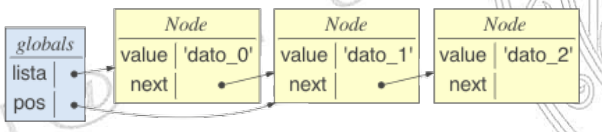
```
actual = primer nodo de la lista
mientras actual is not None y actual.value != target:
  actual = actual.next
trabajar con actual
```

Este código es el típico código para la función que indica si existe algún valor en la lista (retornando un booleano) y para la función que retorna la posición (o referencia) donde se encuentra el valor encontrado.

Añadir un nodo como siguiente de otro consta de los siguientes pasos:

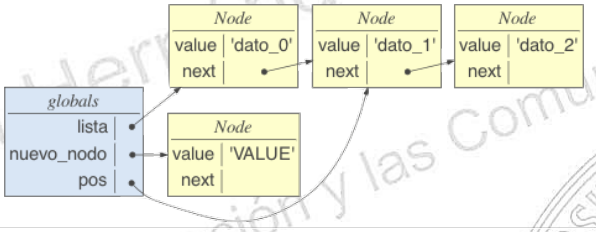
1. Tener la referencia del nodo referente.

```
pos = referencia
```



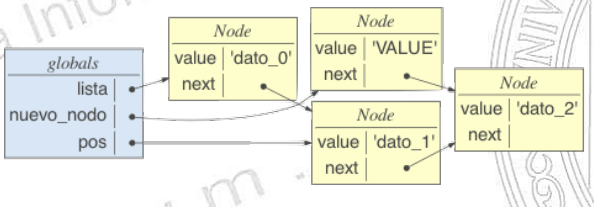
2. Crear un nodo que contenga el valor a añadir en la lista.

```
nuevo_nodo = Node(VALUE)
```



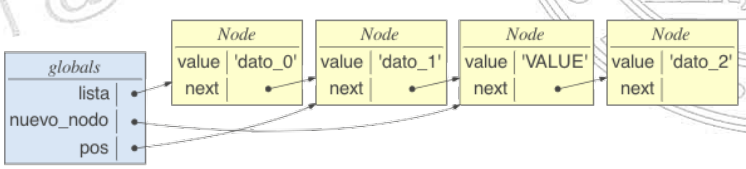
3. Enlazar el nuevo nodo con el siguiente del nodo referente.

```
nuevo_nodo = pos.next
```



4. Enlazar el nodo referente con el nuevo nodo.

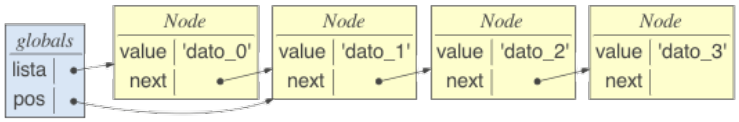
```
pos.next = nuevo_nodo
```



Eliminar el nodo siguiente de otro consta de los siguientes pasos:

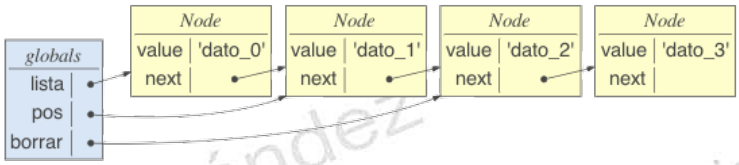
1. Tener la referencia del nodo referente.

```
pos = referencia
```



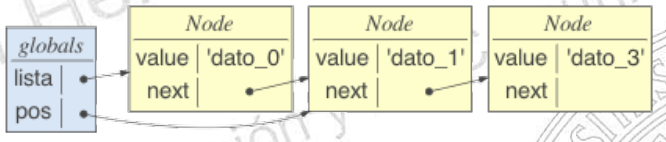
2. Enlazar el nodo referente con el nodo siguiente de su nodo siguiente.

```
borrar = pos.next
pos.next = borrar.next
```



3. Liberar la memoria del nodo siguiente del nodo referente.

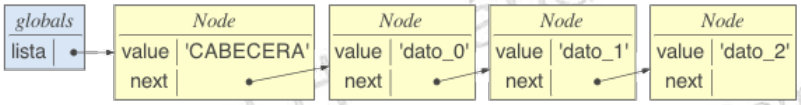
```
gc.collect()
```
En la práctica, esta invocación no se realiza.



Un punto a tener en cuenta al trabajar con estructuras de datos enlazadas es que con el fin de simplificar la escritura de las funciones, en el sentido de tener que evitar casos especiales en el proceso de inserción y eliminación, es evitar cambiar la referencia de la variable externa al inicio de la estructura.

Por ejemplo, en un lista simplemente enlazada, si se eliminara el primer elemento de la lista, entonces la referencia de la variable externa tendrá que cambiarse para que apunte al segundo elemento de la lista. De la misma forma, si se quiere añadir un elemento antes del primer elemento de la lista, entonces la referencia de la variable externa tendrá que cambiar al nuevo elemento insertado.

Una forma de evitar estos casos (aquellos en los que la variable externa cambia su valor de referencia) es considerar la existencia de un nodo intermedio entre la variable externa y el primer término de la lista. Es un nodo que recibe el nombre de **nodo cabecera** (head) que no contendrá ningún valor y como elemento siguiente hará referencia al primer elemento de la lista, así mismo la variable externa siempre hará referencia al nodo cabecera (y nunca cambiará).



Un problema que presenta las listas simplemente enlazadas es que obtener ciertas posiciones para ciertas operaciones puede ser un proceso muy costoso. Una de esas situaciones es **añadir un elemento al final** de la lista. Por ejemplo, si en una lista de 10^6 elementos se quiere añadir un elemento al final de la misma, será necesario recorrer todos los elementos para poder modificar la referencia del campo siguiente del último nodo. Una forma de resolver este problema es considerar en la estructura una referencia al último nodo:

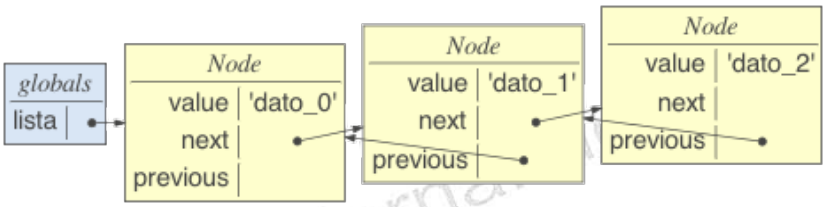
```
struct List {
  struct node {
    TD valor;
    node next;
  }

  node first
  node last
}
```

Otra situación complicada es cuando se necesita recuperar la posición anterior a una posición dada. Por ejemplo, si en una lista de 10^6 elementos se quiere añadir un elemento en la penúltima posición de la misma, será necesario recorrer los primeros $10^6 - 1$ elementos de la lista. La situación se agrava si se quisiera recorrer la lista en orden inverso y mostrar sus valores: primero sería necesario recorre 10^6 elementos, después $10^6 - 1$, después $10^6 - 2$, etc. Una forma de solucionar este problema es considerar **Estructuras Enlazadas con dos enlaces**.

Una **Lista Doblemente Enlazada** es una representación que usa una Estructura Enlazada con dos Enlaces. El primer enlace hace referencia al elemento siguiente en la lista y el segundo enlace hace referencia al elemento anterior en la lista.

```
struct node {
   TD valor;
   node next;
   node previous;
}
```

Ejemplo 4.3. La lista doblemente enlazada (sin cabecera) para la lista $< dato_0, dato_1, dato_2 >$ es:



□

4.D Búsqueda y Ordenación

En todas las estructuras de datos que conlleven una colección de datos (p.e. arrays, conjuntos, ficheros, mapas, ...) hay dos tipos de operadores/funciones imprescindibles para el recorrido en la colección de datos y su manipulación: la búsqueda y la ordenación.

- El objetivo de la búsqueda es encontrar un dato concreto. Por ejemplo, en una lista de números naturales localizar el mayor número primo.
- La ordenación está relacionada con modificar/crear una estructura para que sus elementos se almacenen ordenados según cierto criterio. Es usual que esta estructura sea una lista. Por ejemplo, dada una lista de números naturales crear una nueva lista donde estén ordenarlos de forma creciente.

Nota *En lo que sigue, supondremos que la búsqueda y ordenación se hará sobre listas. También se asume que de la asignatura de Fundamentos de Programación se conocen perfectamente los algoritmos de búsqueda secuencial y binaria, así como los algoritmos de ordenación de la burbuja, selección e inserción.*

4.D.1 Búsqueda

Un **algoritmo de búsqueda** es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos. Por ejemplo, encontrar el registro correspondiente a cierta canción en un array de canciones.

Dependiendo de cómo se encuentren ordenados los elementos de la colección, podemos aplicar uno de los dos siguientes algoritmos:

- o búsqueda secuencial, cuando los elementos de la colección no están ordenados
- o búsqueda binaria, para cuando los elementos están ordenados.

Pero ¿cuándo una lista de objetos esta ordenada?

- Una lista `vec` se dice que tiene un orden ascendente si
  $$i \leq j \text{ implica que } vec[i] \leq vec[j]$$
- Una lista `vec` se dice que tiene un orden descendente si
  $$i \leq j \text{ implica que } vec[i] \geq vec[j]$$
- Una lista `vec` se dice que está ordenada si presenta un orden ascendente o un orden descendente

El orden puede venir dado (impuesto) o puede que tengas que definirlo tú. Todo depende del tipo de dato que contenga la lista.

- Si el tipo de dato de la lista son números, el orden viene dado por el orden de los números tal y como lo conoces.
- Si la colección es de caracteres, el orden viene dada por la tabla de códigos.
- Si el conjunto es de valores booleanos, se suele considerar que **`true`** es mayor que **`false`**; pero este orden no siempre viene implementado en los lenguajes de programación. Es posible que el orden lo tengas que implementar tú. **Processing** no tienen implementado ningún orden en los booleanos pero para **Python** son números naturales.
- Si el array es de String, el orden será el lexicográfico.
- Si el array es de arrays, tú tendras que definir el orden.
- Si la colección es de registros, tú tendrás que establecer qué campo o conjuntos de campos son los que establecen que un registro sea posterior que otro.

Búsqueda secuencial. La búsqueda secuencial se usa cuando los datos no están ordenados. Se puede expresar de la siguiente manera:

```
def busqueda_secuencial(lista, dato) -> Optional[int]
  pos = 0
  for pos in range(len(lista)):
    Si lista[pos].value == dato:
      return pos
  return None
```

Búsqueda Binaria. La búsqueda binaria se usa cuando los datos están ordenados. Se puede expresar de la siguiente manera, supuesto que la lista ya ha sido ordenada previamente:

```
def busqueda_binaria(lista, dato) -> Optional[int]
  size = len(lista)
  inf = 0
  sup = size-1

  while inf <= sup:
    pos = ((sup - inf) // 2) + inf // División entera: se trunca la fracción
    if lista[pos].value == dato:
      return pos
    else:
      if dato < lista[pos]:
        sup = pos - 1
      else:
        inf = pos + 1

  return None
```

4.D.1 Ordenación

Un **algoritmo de ordenación**[1] u ordenamiento es aquel que está diseñado para ordenar los elementos de una lista. Para esto será necesario aclarar qué se entiende por la expresión **vec[pos1]<vec[pos2]**, aspecto que ya se ha comentado. Dicha expresión de comparación deberá interpretarse como una función u operación construida por el programador.

Ordenación de Burbuja. La Ordenación de burbuja (Bubble Sort en inglés) funciona revisando cada elemento de la lista que va a ser ordenado con el siguiente, intercambiándolos de posición si están en el orden equivocado y si están en posiciones correctas continúa el proceso con el siguiente elemento. Más concretamente, en cada iteración recorre toda la lista, desde el inicio de la lista y hasta el final. En el recorrido compara cada par de elementos adyacentes y si no están en el orden esperado los intercambia de lugar. El proceso continúa hasta que no haya más elementos que necesiten ser comparados. Se puede expresar de la siguiente manera:

```
def sort_bubble(vec):
  tam = length(vec)
  do:
    nuevoTam = 0
    for pos in range(1, tam):
      if vec[pos-1] > vec[pos]:
        intercambiar(vec[pos-1], vec[pos])
        nuevoTam = pos
    tam = nuevoTam
  mientras tam != 0
```

Ordenación por selección. Imagina que dispones de n-valores (todos) distintos en un vector y sabes que están desordenados. ¿Dónde colocarías el elemento más pequeño? ¿y el segundo más pequeño? ¿y el tercero? .... ¿encuentras un patrón? Es fácil, hay que colocar el k-ésimo valor más grande en la k-ésima posición del array. A partir de esta idea nos planteamos el siguiente algoritmo de ordenación:

1. Buscar el mínimo elemento de la lista e intercambiarlo con el primero,
2. buscar el siguiente mínimo en el resto de la lista e intercambiarlo con el segundo

   ...

n. y así sucesivamente hasta llegar al último.

Esta secuencia de acciones determinan el siguiente patrón repetitivo:

1. Buscar el mínimo elemento de la lista que está entre la posición **pos** y el final de la lista (inicialmente **pos=0**)
2. Intercambiarlo con el que está entre en la posición **pos**
3. Hacer **pos++**

El algoritmo se puede expresar así:

```
def sort_selection(vec):
  for pos in range(len(vec)):
    pos_minimo = pos
    for siguiente in range(pos+1, tam):     # Busca el valor más pequeño
      if vec[pos_minimo] > vec[siguiente]:  # de entre los siguientes
        pos_minimo = siguiente;
    intercambiar(vec[pos], vec[pos_minimo])
```

Ordenación por inserción. El planteamiento de este algoritmo es el siguiente. Si tuvieses una lista de elementos, para los que sabes que todos los elementos que hay antes de cierta posición **pos-1** sus elementos ya están ordenados ¿cuál es la posición que le corresponde al elemento de la posición **pos** en esa lista? Pues se trataría simplemente de recorrer la lista desde el principio y encontrar **el primer elemento que sea mayor que él** en la lista (si lo hubiera). Si se encontrase en cierta posición **k**, entonces desplazaríamos toda la sublista de las posiciones **[k, pos-1]** a las posiciones **[k+1,pos]**, y el elemento que se encontraba en **pos** se pasaría a la posición **k**. Es decir, insertamos el elemento de la posición **pos** en la posición **k**.

[1] Puede visualizar el funcionamiento de los algoritmos que se muestran en esta sección en https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

El proceso de inserción también se puede hacer desde el "último" elemento de la sublista: Considera una variable **anterior** que represente a las posiciones que se encuentran antes de **pos**, se puede repetir el ciclo de pasar el elemento que está en la posición **anterior** a la posición **anterior+1** (y decrementando en una unidad el valor de **anterior**) mientras que el elemento original que estaba en **pos** **sea más pequeño** que los elementos determinados por **anterior**. Cuando **anterior** corresponda a una posición cuyo valor sea mayor que el valor que originalmente estaba en **pos** deja de hacer los intercambios, en cuyo caso dejará un "hueco" que es aprovechado para insertar el valor que se encontraba en **pos**. El algoritmo se puede expresar así:

```
def sort_insertion(vec)
  for pos in range(1, len(vec)):
    valor = vec[pos]
    anterior = pos - 1
    while anterior >= 0 and vec[anterior] > valor:
      vec[anterior+1] = vec[anterior]
      anterior = anterior - 1
    vec[anterior + 1] = valor
```

Ordenación por casilleros. Todos los algoritmos anteriores son algoritmos de orden $O(n^2)$. Esto quiere decir que el número de operaciones que tiene que hacer, en el peor de los casos, es proporcional a n^2 para n igual al número de datos a ordenar. En otras palabras, en el peor de los casos, el ordenador que quiera ordenar mil objetos tendrá que realizar un número de operaciones proporcional a $(10^3)^2 = 10^6$. Teniendo en cuenta que hay algoritmos en el mundo de la informática muchísimo más complicados y que prácticamente nunca acaban, se podría pensar que estos algoritmo "no están nada mal"; pero si te digo que hay muchos algoritmos de ordenación cuya complejidad computacional es mucho menor (son del orden $O(n\ log(n))$ y $O(n)$) y que los peores algoritmos de ordenación son los de orden $O(n^2)$ ... tengo que decirte que se te han explicado los algoritmos más sencillos pero también los más ineficientes si tienes que ordenar muchos, muchos datos.

Para mejorar la eficiencia de los algoritmos anteriores, te mostramos un algoritmo muy simple de orden $O(n)$, que es la ordenación por casilleros (o cubos - bucket sort o bin sort, en inglés).

El algoritmo se divide en dos partes.

1. En la primera, se considerar una serie de casilleros o cubos en los que solo debes introducir los elementos de la lista a ordenar que cumpla las condiciones del casillero. Las condiciones entre los casilleros deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos cubos distintos.

   Por ejemplo, si vas a ordenar un lista de números naturales $\{29, 25, 3, 49, 9, 37, 21, 43\}$ puedes definir el cubo k como el que contiene los números comprendidos entre $k \times 10$ y $(k + 1) \times 10 - 1$. Con lo que los cubos quedan como $C_0 = \{3, 9\}$, $C_2 = \{29, 25, 21\}$, $C_3 = \{37\}$ y $C_4 = \{49, 43\}$
2. En la segunda parte se ordenan los elementos de cada uno de esos casilleros con otro algoritmo de ordenación (que podría incluso ser distinto según el casillero) o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos.

   Por ejemplo, tras ordenar los cubos anteriores se quedan como $C_0 = \{3, 9\}$, $C_2 = \{21, 25, 29\}$, $C_3 = \{37\}$ y $C_4 = \{43, 49\}$. Y el orden de la lista final queda como "la concatenación" de los elementos de cada uno de los cubos: $\{3, 9, 21, 25, 29, 37, 43, 49\}$.

El algoritmo puede expresarse en 4 pasos:

1. Crear una colección de casilleros vacíos, que ya está "ordenados" entre sí.
2. Colocar cada elemento a ordenar de la lista en un único casillero.
3. Ordenar los elementos de cada casillero según el algoritmo que se considere más adecuado.
4. Devolver los elementos de cada casillero "concatenados".

El algoritmo se puede expresar así:

```
def sort_basket(vec)
  # Primer paso: crea los casilleros de forma adecuada
  cubos = new Coleccion (nCubos)

  # Segundo paso: introduce cada elemento en un casillero
  for pos in range(len(vec)):
    insertar(vec[pos], cubos)
```

```
  # Tercer paso: ordena los casilleros
  for pos in range(nCubos):
    cubos[pos].sort()

  # Cuarto paso: devuelve la solución
  return concatena(cubos)
```

Merge Sort (Ordenación por mezcla). Este algoritmo recursivo se basa en una idea muy sencilla. Para ordenar una lista basta ordenar la primera mitad lo que nos da una lista1 y ordenar la segunda mitad lo que nos da una lista2. Entonces basta recorrer uno a uno los elementos de lista1 y compararlos con los elementos de lista2. Si el elemento actual de lista1 es más pequeño se añade al resultado y en otro caso se añade el elemento actual de lista2.

Existen dos versiones de este algoritmo. Se muestra una de ellas copiada de Wikipedia. Si quiere conocer la segunda versión consulte el enlace anterior.

```
function merge_sort(list m)
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m <= 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)
```

```
function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) <= first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Ejercicios

4.1 Listas con Array1D

★★☆

Implementar TDA List como Array

Usa la definición del TDA List de esta sesión para implementarlo en **Python** usando para su implementación el TDA Array1D de la sesión anterior. Llama a tu fichero **`ListArray.py`** y a la clase **`List`**.

Para su implementación usa la siguiente estructura[2]:

```
class Lista:
    _lista : Array1D # Array de valores
    _pos: int        # Posición
```

donde `_lista` será de tipo **`Array1D`** de cierto tamaño n fijo y `pos` indica la posición donde se almacenará el siguiente elemento a añadir en el array `_lista`. Inicialmente `pos` valdrá 0 (es decir, el primer elemento que se añada en el array será en la posición de índice 0). El array irá cambiando en tiempo de ejecución de acuerdo a los siguientes criterios:

- Si se añade un nuevo elemento sin especificar su posición y quedan posiciones libres, el dato se almacenará en la posición `pos`. Pero si se especificara su posición, entonces todos los elementos desde esa posición hasta el final incrementará su posición en una unidad. En cualquiera de los dos casos se incrementará `pos` en una unidad. Así, `pos` se incrementa conforme se vayan añadiendo elementos.
- Si se añade un nuevo elemento pero todas las posiciones están ocupadas, entonces crea un nuevo array con el doble de capacidad, copia los elementos existentes y añade el nuevo elemento teniendo en cuenta el punto anterior. Después descarta el array usado hasta el momento para usar el que tiene doble capacidad.
- Si se elimina un elemento de una posición, desplaza todos los elementos que se encuentran en las posiciones siguientes a una posición anterior para que no queden huecos libres "al final". Decrementa `pos` en una unidad.
- Si queda un porcentaje A de posiciones libres (p.e. $A = 50\,\%$), entonces se creará un nuevo array con un porcentaje B de posiciones (p.e. $B = 75\,\%$) y se copiará en él los elementos existentes. El nuevo array nunca tendrá una capacidad inferior al valor n fijo. Después descarta el array usado hasta el momento para usar el que tiene dos tercios de capacidad.

Observa que si se trabajara con Array1D de forma estricta (el número de posiciones es siempre igual al número de elementos de la lista), entonces sería necesario crear un array con una dimensión más (si se va a añadir un elemento) o una dimensión menos (si se va a borrar un elemento), copiar los elementos restantes al nuevo array y descartar el array inicial. Desde una perspectiva computacional el coste para usar esta estructura es excesivo (imagina que trabajaras con millones de datos). Por ello, como alternativa, se trabaja con la estructura indicada y que viene de forma nativa en muchos lenguajes de programación.

Ya que se implementará esta estructura en **Python**, los datos se pueden acceder por índice (notación de array) y no por método (notación punto de POO). Para ello implementa los métodos mágicos **`__getitem__()`** y **`__setitem__()`**.

4.2 Lista Simplemente Enlazada

★★☆

Implementar TDA List con Nodos

Usa la definición del TDA List de esta sesión e implementala en **Python** usando una estructura simplemente enlazada. Llama a tu fichero **`ListSimpleRefLink.py`** y a la clase **`List`**.

Recuerda que en esta implementación, el valor del parámetro **`pos`** en todas las primitivas debería ser una referencia a un nodo.

4.3 Lista Simplemente Enlazada con Índices

★★☆ ✍

Implementar TDA List con índices y nodos

En los dos ejercicios anteriores se han implementado dos formas de trabajar con listas: una con índices (arrays) y otra con referencias (nodos). También sabemos que si se usa un TDA Lista nuestro programa debe diseñarse e implementarse independientemente del tipo de implementación del TDA. Por tanto, necesitamos que los valores **`pos`** sean del mismo tipo en ambas implementaciones de la lista.

[2] Aquí usaremos el identificador **`Lista`** para evitar problemas con **`List`** de `typing` de Python

Queremos que **pos** sea un entero en cualquier implementación. En un array se corresponderá con uno de los índices del array y en una estructura enlazada nos indicará el número de veces que tendremos que actualizar las referencias de los nodos.

Utiliza el siguiente esquema para implementar una lista simplemente enlazada usando un entero para acceder a los nodos. Recuerda implementar los métodos mágicos **__getitem__()** y **__setitem__()**.

```
class Lista:
    class __Node:
        __slots__ = '_value', '_next'
        _value: object
        _next: Optional['Node']

        ....

    __slots__ = '_head', '_len'
    _head: Node
    _len: int

    def __get_node(self, pos: int) -> __Node:
        if pos == -1 or self.empty():
            return self._head

        assert 0 <= pos < len(self), "Posición fuera de límites"
        node = self._head.next
        for i in range(pos):
            node = node.next
        return node
```

Observa que se usa una clase interna que es invisible fuera de la clase (se usa dos líneas bajas antes del nombre de la clase). Observa también que esta implementación del TDA Lista usa un nodo cabecera.

4.4 Lista Doblemente Enlazada

★★☆

Implementar TDA List con Nodos

Usa la definición del TDA List de esta sesión para implementarlo en **Python** usando para su implementación una estructura doblemente enlazada. Llama a tu fichero **ListDoubleLink.py** y a la clase **List**.

4.5 Módulo de Búsqueda

★☆☆ ✍

Implementar algoritmos de búsqueda

Crea el módulo 🖿 `Searching > SearchingList.py` e implementa en él los dos algoritmos de búsqueda (secuencial y binaria) trabajando con las listas de Python.

Cuando veas que funciona correctamente, adapta las funciones para que sean métodos de las implementaciones que hayas hecho del TDA Lista.

Es decir, si tienes la función `def search_secuencial()` modifícala para que sea el método `def search_secuencial(self)`, que deberás añadir al tipo Lista que has implementado en la sesión anterior.

4.6 Módulo de Ordenación

★★☆ ✍

Implementar algoritmos de ordenación

Implementa un módulo con distintas técnicas de ordenación de listas. Deberá contener, al menos, la ordenación por casilleros y la ordenación por mezcla. Este módulo lo aplicaremos en la siguiente sesión para aplicar técnicas de búsqueda en espacio de estados que es la forma más general de resolver problemas en Inteligencia Artificial.

- Llama a tu fichero **Sort** y a las funciones las llamas **sort_busket()** y **sort_merge()**. Ambas funciones tendrán como parámetro de entrada una lista de tipo `list` de Python - ver página 71.
  Opcionalmente añade las siguientes funciones: **sort_bubble()**, **sort_selection()**, **sort_insertion()**.

- Implementa el método **`def sort(self, method)`** como una primitiva más del tipo lista. El parámetro **`method`** indicará la función a aplicar para la ordenación.

4.7 Ordenar a los alumnos

★☆☆ ✍

Aplicación de los algoritmos de ordenación. Diccionarios

En el contexto del Ejercicio 3.8 crea la clase **`Acta`**. El acta de una asignatura es un documento que consta de una lista donde, en cada fila, figura un alumno junto con la calificación obtenida. Para este ejercicio se supondrá que todos los alumnos tienen una calificación numérica real en el intervalo $[0, 10]$ y con un valor decimal. Se pide:

- Crea un conjunto de no menos de 10 alumnos. No es importante que los datos de los alumnos tengan sentido, lo importante es que se distingan. Puedes usar un bucle **`for`** para acelerar el proceso.
- Construye entonces un acta con esos alumnos y asigna a cada uno una nota aleatoria. Recurre al módulo `https://docs.python.org/es/3/library/random.html`
- Ordena a los alumnos del acta de acuerdo a sus calificaciones.
  Mira los métodos mágicos de comparación `__cmp__()`, `__lt__()`, `__qt__()`, ... A efectos de depuración, necesitarás que todos los alumnos tengan siempre la misma calificación "aleatoria" en cada ejecución. Para conseguir que en cada ejecución siempre se generan los mismos números aleatorios usa siempre la misma semilla, p.e. **`seed(0)`**.
- Construye un diccionario que indique cuántos alumnos han suspendido ($nota < 5$), han aprobado ($5 \leq nota < 7$), tienen notable ($7 \leq nota < 9$) o son de sobresaliente ($nota \geq 9$).
  El diccionario lo puedes construir usando **`dict()`** de **Python** o **`Map()`** del Ejercicio 3.6.

Sesión 5

Pilas y Colas

Índice Parcial **Página**

5.A. TDA Pila . . . 75
5.B. TDA Cola . . . 76
5.C. TDA Cola de Prioridad . . . 76
5.D. Problemas de Búsqueda . . . 77
5.1. Pilas . . . 79
5.2. Colas . . . 79
5.3. Resolución de Laberintos . . . 79
5.4. Laberintos con Heurística . . . 81
5.5. Resolución de n-Puzzle . . . 81

Teoría

5.A TDA Pila

Las listas estudiadas en el sesión anterior son secuencias de datos donde no hay restricciones para consultar, insertar o borrar sus valores. Las Pilas se pueden contemplar como una lista donde existen restricciones en la consulta, inserción y borrado de sus valores. En concreto, en una pila (stack) sigue el criterio LIFO: "Last input, first output'. Esto quiere decir que solo se puede consultar o eliminar el último elemento introducido. Se llama tope al extremo de la lista donde tienen lugar las inserciones y supresiones en la lista.

Definición 5.1: 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.

Las pilas se pueden representar con las mismas estructuras estudiadas para las listas. Si se utilizara una estructura con capacidad finita y la pila se llenara entonces no cabe añadir más elementos. De añadirse entonces se producirá un error conocido como **desbordamiento** - overflow.

Los métodos de la estructura de datos integrado **list** de Python hacen fácil usar las listas como pilas pues esta estructura de datos dispone de los métodos **append()** - que añade un elemento en la última posición de la lista - y **pop()** - que sin argumento recupera el elemento de la última posición de la lista -.

```
>>> stack = [3, 4, 5]
>>> stack.append(6);
>>> print(stack)
[3, 4, 5, 6]
>>> stack.pop()
6
>>> print(stack)
[3, 4, 5]
```

5.B TDA Cola

Las Colas se pueden contemplar como otro tipo de lista donde existen restricciones en la consulta, inserción y borrado de sus valores. En concreto, en una cola (queue) sigue el criterio FIFO: "First input, first output'. Esto quiere decir que los elementos se introducen al final de la lista pero solo se puede consultar o eliminar el primer elemento que se introdujo.

Definición 5.2: 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.

Se puede implementar una cola en Python usando **collections.deque**.

```
>>> from collections import deque
>>> cola = deque([3, 4, 5])
>>> cola.append(6); cola.append(7)
>>> print(cola)
deque([3, 4, 5, 6, 7])
>>> cola.popleft(); cola.popleft()
3
4
>>> print(cola)
deque([5, 6, 7])
```

Por descontado que las colas también se pueden representar con las mismas estructuras estudiadas para las listas.

5.C TDA Cola de Prioridad

Es una colección de elementos ordenados de acuerdo a una clave que el usuario asigna cuando añade un elemento a la cola. El elemento con clave mínima será el siguiente en ser eliminado de la cola (p.e. un elemento con clave 1 tendrá prioridad sobre un elemento con clave 2). La clave suele ser numérica pero puede usarse cualquier objeto siempre que el objeto admita una ordenación $a < b$, para cualquier instancia a y b.

Si una cola de prioridad tuviera varias entradas con claves equivalente, los métodos **min()** y **pop()** seleccionarán uno arbitrario si hay varios mínimos.

Una forma de implementar una cola de prioridad es usar una lista ordenada. Los métodos **`min()`** y **`pop()`** retornarían el primer elemento de la lista y el método **`add(k ,v)`** insertaría la tupla/pareja en aquella posición de la lista que garantice que la lista sigue ordenada.

Otra forma de implementar una cola de prioridad es usar una lista no ordenada. Como una cola de prioridad ordena sus elementos y la lista no lo está, se necesitará un método para encontrar la pareja con clave mínima:

```
class UnsortedPriorityQueue():
  def _find_min(self):
    assert not self.is empty(), "Cola vacía"
    small = self._data.first()
    actual = self._data.after(small)
    while actual is not None:
      if data.element( ) < small.element():
        small = actual
      actural = self. data.after(actual)
    return small

  def __init__(self):
    self._data = List()

  # Termina con los métodos del TDA
```

Una tercera forma de implementar este TDA es usando Heap (un tipo especial de árbol binario) y que se comentará en la página 94.

5.D Problemas de Búsqueda

Existen muchas estrategias para resolver problemas. Una de ellas es la basada en la búsqueda en espacio de estados. Un problema se puede contemplar como una situación en la que se parte de una situación inicial (no deseada) y se quiere llegar a una situación final (una de las deseadas). Resolver el problema consiste en partir de la situación inicial y realizar una acción para llegar a otra situación, a partir de esta situación se toma otra acción para llegar a otra situación, ... y así sucesivamente se van realizando acciones hasta llegar a una situación que se corresponda con la solución del problema.

Una de esas situaciones recibe el nombre de estado. Un estado caracteriza una situación concreta del problema en un instante de tiempo y no presenta ningún tipo de ambigüedad. Algunos ejemplos son:

- Una configuración válida de un juego de tablero (ajedrez, damas, parchís, ...) es un estado del problema. Aquí el problema es conseguir ganar al contrario en el juego de tablero elegido.
- Una configuración válida de un juego tipo puzzle (un puzzle de piezas, sudoku, ...) es un estado del problema. Aquí el problema es resolver un solitario.
- Estar en una posición GPS de un trayecto entre dos puntos es un estado del problema de encontrar una ruta entre dos puntos seleccionados previamente.
- Tener un nivel de conocimientos y destrezas en una asignatura es un estado del problema "hay que aprobarla".

Una solución en los problemas representados mediante estados es una secuencia de acciones que permiten pasar de un estado a otro empezando por el estado inicial hasta llegar a un estado final. Gráficamente se puede representar mediante un árbol de búsqueda.

A nivel de implementación se necesita una estructura de **Node** que consta de los siguientes campos:

- **`_state`**: Almacena un **estado**
- **`_parent`**: Almacena una referencia al **nodo** del estado que lo generó.

También se necesita una función con los parámetros:

- **initial**: El estado inicial.
- **`goal_test`**: la función (callable) que recibe como entrada un estado y retorna un booleano indicando si el estado es un estado solución o no.
- **`successors`**: la función (callable) que recibe como entrada un estado y retorna una lista de estados sucesores.

Su cuerpo es el siguiente:

```
def search(initial, goal_test, successors):
  frontera = {Nodo(initial, None)}  # LISTA de nodos,
                                    # empezando por el estado inicial
  explorados = {initial}            # CONJUNTO de estados detectados
                                    # también empieza con el estado inicial
  mientras que la frontera no esté vacía:
     nodo_actual = extraer un nodo de frontera (y borrarlo)
     estado_actual = estado del nodo_actual
     si goal_test(estado_actual):
         retornar camino solución para el nodo_actual
     para cada estado de successors(estado_actual):
         si estado está en explorados:
             pasar al siguiente estado
         añadir estado a explorados
         añadir el nodo(estado, nodo_actual) a frontera
  retornar None
```

El retorno del camino solución para el nodo actual es un proceso que consta de los siguientes pasos:

1. Construir una lista (guardará la secuencia de estados que llevan a la solución)
2. Recorrer la lista de nodos que empieza en nodo_actual mientras que exista un nodo siguiente e ir añadiendo sus estados a la lista construida[1].
3. Invertir el orden de la lista para que empiece desde el estado inicial y finalice en el estado solución encontrado.

Un aspecto que debe quedar claro es ¿qué nodo de la frontera debe extraerse para continuar la búsqueda? Esta es una cuestión clave, pues es lo que determinará la estrategia de la búsqueda. Podemos distinguir 4 estrategias a partir del cuerpo de la función:

- Búsqueda en anchura. Se selecciona un nodo de entre los nodos del nivel menos profundo del grafo de búsqueda que aún no haya sido estudiado.
- Búsqueda en profundidad. Se selecciona un nodo de entre los nodos del ultimo nivel generado.
- Búsqueda uniforme. Se selecciona uno de los nodos que tenga menos coste acumulado. El coste acumulado de un nodo n se define como $g(n) = g(n_{padre}) + c(n_{padre}, n)$ donde $c(n_{padre}, n)$ es un valor no negativo que indica cuánto cuesta realizar la acción para pasar de n_{padre} a n.

  Esto supone que en la clase **Node** habrá que añadir un campo más que represente el valor de $g()$ y por tanto su constructor tendrá 3 parámetros.

  Por ejemplo, podemos medir el coste de ir de un punto a otro del mapa de carreteras por el número de kilómetros. Vemos que ir del punto A al punto B son 10Km e ir del punto B al punto C son 30Km. Si realizamos una búsqueda uniforme empezando por A: el coste acumulado para A es 0, el coste acumulado para B es 10 (si se llegó a él a través de A) y el coste acumulado para C es 40 (si se llegó a él a través de B).
- Búsqueda A^*.

  Que define el algoritmo clásico de la IA. Se selecciona uno de los nodos que tenga menos coste $f(n) = g(n)+h(n)$, donde:

  - $g(n) = g(n_{padre}) + c(n_{padre}, n)$, es el coste acumulado.
  - $h(n)$ es una función heurística que asigna valores no negativos.

    El valor heurístico asociado a un estado es un valor que no sobreestima el coste real para llegar desde ese estado al estado final.

    Continuando con el ejemplo anterior de la ruta A-B-C. Si nuestro objetivo es llegar a C, un valor heurístico para A es cualquier valor entre 0 y 40. Cualquier valore subestima el valor real que es 40. Una subestimación (no sobreestimación) es un valor optimista sobre el coste real. Cuanto más se aproxime la subestimación al valor real, el algoritmo de búsqueda funcionará mejor.

    Si se sobrestima los valores heurísticos el algoritmo no convergerá a la solución. En el ejemplo, no sería correcto dar una valor heurístico a A por encima de 40.

  Notar que para implementar este algoritmo será necesario modificar la clase **Node** para que tenga dos campos: uno para almacenar el valor de $g()$ y otro para almacenar el valor de $h()$ lo que supone que el constructor tendrá 4 parámetros.

  Además, nuestra función de búsqueda tendrá esta signatura:

[1] Si no recuerdas como se recorre una lista repasa las operaciones básicas de estructuras enlazadas en la página 64 y el Ejercicio 4.5.

```python
def astar(initial: T,
          goal_test: Callable[[T], bool],
          successors: Callable[[T], List[T]],
          heuristic: Callable[[T], float]
          )
          -> Optional[List[T]]
```

Ejercicios

5.1 Pilas

★☆☆ ✍

Implementar TDA Stack

Usa la definición del TDA Stack de la página 75. Implementa este TDA usando Arra1D con capacidad limitada, listas de Python y listas simplemente enlazadas.

5.2 Colas

★☆☆ ✍

Implementar TDA Queue

Usa la definición del TDA Queue de la página 76 Implementa este TDA usando Arra1D con capacidad limitada, listas de Python y listas simplemente enlazadas.

5.3 Resolución de Laberintos

★★☆ ✍

Búsqueda en anchura y profundidad

Implementa dos versiones del algoritmo de búsqueda en espacio de estados de la página 77:

- **`dfs()`**: es la versión que implementa el algoritmo de búsqueda en profundidad (depth-first search). En este caso la frontera es una lista de nodos con restricciones. En concreto será una pila (LIFO). Con ello, en cada iteración aumentamos el nivel de la búsqueda.
- **`bfs()`**: es la versión que implementa el algoritmo de búsqueda en anchura (breadth-first search). Es la que considera que la lista de nodos frontera es una cola (FIFO). Así, en cada iteración siempre analizará los nodos de los niveles previos.

Para comprobar que la implementación es correcta resolveremos el problema de encontrar un camino solución en un laberinto formados por un grid de celdas cuadrados. Cada celda puede estar libre o bloquedada y de entre las libres hay dos celdas destacadas: la celda de inicio que es el punto de partida y la celda objetivo que es el punto de salida. Si representamos cada celda mediante una pareja $(fila, columna)$ tenemos un problema de búsqueda en espacio de estados:

- Estado inicial: $(inicio.fila, inicio.columna)$
- Estado objetivo: $(objetivo.fila, objetivo.columna)$
- Solución: una secuencia de estados que empieza en el estado inicial y finaliza en el estado objetivo.
  El elemento siguiente a $(fila, columna)$ en la secuencia será uno de estos cuatro estados $(fila \pm 1, columna \pm 1)$ y siempre y cuando no se salga de los límites del grid.

Para resolver el problema creará el fichero `Maze.py` siguiendo los siguientes pasos:

1. Copiar el siguiente código:

```python
from typing import NamedTuple
from enum import Enum

class Cell(str, Enum):
    EMPTY = " "
    BLOCKED = "#"
    START = "S"
    GOAL = "G"
    PATH = "."

class MazeLocation(NamedTuple):
```

```
    row: int
    column: int
```

Son dos clases hijas, una de **NamedTuple** y otra de **(str, Enum)** .

- **NamedTuple** es la clase que permite trabajar con tuplas cuyos campos tienen nombre y por tanto podemos acceder a su valores usando la notación punto.
  La clase **MazeLocation** es la que condificará los estados del problema.
- **Cell** define un tipo enumerado. Sus valores se acceden como **Cell.EMPTY**, **Cell.BLOCKED**, etc y se interpretan como constantes (realmente son instancias únicas y no podrán ser modificadas). Los enumerados se usan cuando se quieren usar tipos de datos con una cantidad limitada de valores y con valores concretos identificativos. Entiéndelo como una extensión de los booleanos.
  Para acceder a los miembros en enumeraciones mediante sentencias de la forma Cell.BLOCKED no funcionará porque realmente son objetos. Como tales constan de atributos. Si tiene un miembro enum y puede acceder a su `name` o `value`:

```
>>> Cell.BLOCKED
<Cell.BLOCKED: '#'>
>>> Cell.BLOCKED.name
'BLOCKED'
>>> Cell.BLOCKED.value
'#'
```

  Para saber más sobre enumeraciones consulte https://docs.python.org/es/3/library/enum.html.

2. Crea el fichero **Maze.py** e implementa la clase **Maze** para trabajar con laberintos formados por un grid de cuadrados compuesto de **rows**-filas y **columns**-columnas. Aparte de los dos atributos que almacenan las dimensiones, dispondrá de un atributo para representar al grid y que será una lista de listas de elementos de tipo **Cell**. Para las listas se usará el tipo de dato List de Python, por lo que a cada celda se accederá mediante la notación típica de listas: `_grid[fila][columna]`.
   También dispone dos atributos de tipo **MazeLocation** que almacenará el estado inicial y final.
   Para construir un laberinto se necesita conocer sus dimensiones, el estado inicial y el estado final. Entonces se construye un grid formado inicialmente por celdas vacías. Posteriormente cambia algunas celdas vacías a celdas bloqueadas.
   El cambio de celdas se hace recorriendo todas las celdas y para cada una se lanza un número aleatorio según una uniforme (0,1) y si el número está por debajo de un cierto umbral, entonces se realiza el cambio. Usa **random.uniform** y un umbral con valor 0.2 si quieres que (aproximadamente) el 20 % de las celdas cambien su valor.
3. Añade a **Maze** los métodos que se pasarán como argumentos en los algoritmos de búsqueda **dfs()** y **bfs()**:
   - `def goal_test(self, maze_location: MazeLocation) -> bool`.
     Dada una localización (o estado) de tipo `MazeLocation` indica si se corresponde con el estado final.
   - `def succesors(self, ml: MazeLocation) -> List[MazeLocation]`.
     Dado un estado **(ml.row, ml.column)** retorna los 4 estados **(ml.row ± 1, ml.column ± 1)** descartando aquellos que se salgan de las dimensiones del grid y aquellos que se correspondan a celdas bloqueadas.

   Añade cuantos métodos consideres a **Maze** para que te permita resolver este problema. Entre ellos deberás añadir:
   - `def mark(self, path: List[MazeLocation])`.
     Dado un camino solución (lista de estados) cambiar las celdas correspondientes del grid por **Cell.PATH**.
   - `def __str__(self) -> str`
     Para mostrar el laberinto con **print()**.
4. Crea un fichero para comprobar que todo funciona. Tendrás que importar las clases **Maze**, **dfs** y **bfs** y ejecutar los siguientes pasos:
   - Crear un laberinto **maze**.
   - Invocar a `solution = busqueda(maze.start, maze.goal_test, maze.succesors)`
   - Mostrar el laberinto indicando si se ha encontrado solución o no.

5.4 Laberintos con Heurística

★★★

Aplicar A^*

Implementa el algoritmo A^* y aplícalo al problema del laberinto. Usa como función heurística la distancia de Manhattan que hay entre el estado actual y el estado objetivo. La definición de esta distancia la tienes en el ejercicio anterior.

5.5 Resolución de n-Puzzle

★★★

Implementar A^*

El n-Puzzle es un problema clásico de búsqueda con heurísticas. Consta de un grid cuadrado formado por $k \times k = n+1$ celdas. Por ejemplo, el 8-puzzle es un grid de 3×3, el 15-puzzle es un grid de 4×4 celdas, etc ... El grid consta de n-piezas colocadas "al azar" en el grid y cada una tiene un número del 1 al n. El objetivo final es conseguir una configuración (estado) que ordena de forma creciente las piezas de izquierda a derecha y de arriba a abajo.
Un par de ejemplos de estados solución son:

Solución del 8-puzzle:

1	2	3
4	5	6
7	8	

Solución del 15-puzzle:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Dado un estado, una situación de n-Puzzle, existen 4 estados que se pueden llegar a construir a partir de él. Cada uno de ellos se consigue intercambiando la celda vacía por una celda enumerada adyacente siempre que esté en la misma fila o columna. Es decir, partiendo de un estado general

a	b	c
d		e
f	g	h

se pueden llegar a uno de los siguientes estados:

a		c
d	b	e
f	g	h

a	b	c
	d	e
f	g	h

a	b	c
d	e	
f	g	h

a	b	c
d	g	e
f		h

El algoritmo de búsqueda adecuado para este tipo de problemas es el A^* pero hemos de definir una función de costes para los costes acumulados y una heurística.

- Como función de costes, se puede considerar la unidad. Es decir $g(n) = g(n_{padre}) + 1$.
- Como función heurística se puede considerar la suma de todas distancias de cada pieza a su posición. Dicha distancia se define de la siguiente forma:
  Si una pieza p se encuentra en la posición $A = (x, y)$ y la posición final de la pieza es la posición $B = (p.x, p.y)$, la distancia se define como: $d(A, B) = |x - p.x| + |y - p.y|$. Esta distancia se llama distancia de Manhattan.

El objetivo de este ejercicio es implementar el algoritmo A^* para resolver este problema.

Luis Daniel Hernández
Dpto. Ingeniería de la Información y las Comunicaciones
ldaniel@um.es

Sesión 6

Árboles y Grafos

Índice Parcial **Página**

6.A. Recursividad . . . 83
6.B. TDA Árbol . . . 85
6.B.1. Árbol Binario . . . 89
6.B.3. Árboles binarios de búsqueda . . . 92
6.B.3. Heap . . . 94
6.B.3. Representación de los árboles . . . 95
6.C. TDA Grafo . . . 97
6.C.1. Recorrido en Grafos . . . 99
6.C.2. Representaciones de un Grafo . . . 99
6.1. Árboles . . . 100
6.2. TDA Árbol Binario . . . 100
6.3. Adivinador . . . 100
6.4. Árboles Binarios de Búsqueda . . . 101
6.5. Heap Min . . . 101
6.6. Árbol de partición . . . 101
6.7. Árboles de Codificación . . . 102
6.8. Caminos de coste mínimo . . . 102
6.9. Árboles Generadores . . . 104

Teoría

6.A Recursividad

Existen 3 formas de definir un concepto:

- **Extensiva o extensional:** La definición de un término consta del listado de todos los entes que pertenecen a la clase indicada por el término.
  Por ejemplo, “Alumnos de la UMU” consistiría en un listado de todos los alumnos de la UMU.
- **Comprensiva o intensional:** el término queda definido proporcionando todas las propiedades requeridas sobre “las cosas”. Las propiedades demarcan la palabra definida.
  Por ejemplo, “Alumnos de la UMU” constaría de aquellas personas del mundo que están matriculadas en la UMU.
- **Recursiva:** el término queda definido mediante un proceso basado en su propia definición. Se parte de una definición basada en ejemplos o casos particulares.
  Por ejemplo, “Alumnos de la UMU” se puede definir a partir de un alumno de cada titulación y definir el proceso SerCompañeroDeTitulación.

Una definición recursiva es una definición que consta de varias partes:

- **Regla Base:** una definición para unos casos concretos. Es el conjunto inicial sobre el que haremos la primera definición. Es imprescindible para hacer una definición recursiva.
- **Regla Recursiva:** Es un conjunto de reglas que aplicándolas obtiene las definiciones para todos los demás casos. Es una parte imprescindible. También recibe el nombre de pasos recursivos.

- **Regla de Exclusión:** Es un conjunto de reglas que indica cuándo un objeto no se ajusta al concepto en términos de la regla base y la regla recursiva. Normalmente esta parte va implícita y es opcional.

Como caso particular se tiene la definición inductiva, que es una definición recursiva que viene en términos de números naturales.

Ejemplo 6.1. Algunos ejemplos de definiciones recursivas son:

- Definición del conjunto: los descendientes de Adán.
  Una persona es **descendiente de** Adán si
  - **Regla Base:** la persona es un hijo de Adán.
  - **Regla Recursiva:** la persona es un hijo de un **descendiente de** Adán.
  - **Regla de Exclusión:** No hay nadie más que sea descendiente de Adán (salvo los que cumplan la regla base y la regla recursiva).
- Definición de un conjunto: los números pares desde el 0. Un número n **es par** sii
  - **Regla Base:** El número n es el 0.
  - **Regla Recursiva:** Si $n-2$ es **es par**.
- Definición de objetos basados en estructuras: Una lista.
  L **es una lista** de elementos del conjunto A sii
  - **Regla Base:** L es la lista vacía, $[]$.
  - **Regla Recursiva:** Si $L=[a|L']$ se cumple que $a \in A$, y L' es una **es una lista**.

Observa que en todos los casos se tiene

- **en la regla base:** se dice qué objetos concretos cumplen la definición.
- **en la regla recursiva:** vuelve a aparecer la definición (está en rojo).

□

De entre los ejemplos de las definiciones nos centramos en la definición de lista. Observa que no solo define qué es una lista sino que además nos indica cómo deben de construirse: dada una lista, hay que añadir un elemento más al inicio de la misma. Esto da lugar a un construcción recurrente de listas. Todo concepto recursivo se construye a partir de unas reglas para las que **se requiere de una clara ordenación** que permita el crecimiento gradual de objetos en el dominio que estamos definiendo. Para dicho orden y crecimiento es fundamental disponer de algún procedimiento que garantice cómo se construye un objeto nuevo a partir de uno o varios de los ya construido. Si el procedimiento construye siempre objetos nuevos y diferentes de los ya construidos tendrás una buena definición recurrente; si no es así tendrás un conflicto en la definición.

Los procedimientos de construcción de objetos es lo que formalmente se conoce como **Recursividad Estructural**, que son aquellos en los que partiendo de una serie de objetos $\{y_1, y_2, \ldots, y_n\}$ se construye un nuevo objeto x.

Ejemplo 6.2 (Construcción de paredes). Construir una pared de ladrillos responde a una recursividad estructural pues empiezas por una pared que está formada solo por un ladrillo, el y_1. A continuación coges otro, el y_2, y lo unes al primero para obtener un nuevo objeto: la pared x_1, formada por la unión de y_1 e y_2. A continuación coges otro, el y_3, y lo unes al x_1 para obtener un nuevo objeto: la pared x_2 ... y así sucesivamente. El procedimiento de construcción de paredes aquí expuesto responde a una definición recursiva:
y es una pared sii

- o bien y es un ladrillo,
- o bien y está compuesta por una pared y' y un ladrillo x.

□

Muchos procesos matemáticos responden a procesos recursivos similares a la "construcción de paredes".

Ejemplo 6.3 (Factorial de un natural). Un ejemplo típico de proceso recurrente es el producto de los n-primeros números naturales conocido como el factorial de n, $n!$:

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

El factorial de n se puede calcular así:

```
int producto = 1;
int numeroActual = 1;
mientras (numeroActual <= n) {
  producto = producto * numeroActual;
  numeroActual = numeroActual + 1;
}
```

En cada paso de la iteración los valores `producto` y `numeroActual` se van actualizando y nos acercan a la solución buscada.

Pero observa detenidamente la expresión: `producto = producto * numeroActual`. El proceso iterativo expuesto responde por tanto a una definición recursiva:

Un producto es el resultado de multiplicar **un producto** por un número,

pues lo definido aparece en la definición y donde el producto que se construye se obtiene a partir de un producto previo (**¡le precede en el orden!**) y un nuevo objeto.

En definitiva, la expresión responde a una definición/proceso recurrente; pero ¿cómo definirlo? La respuesta está en el proceso iterativo. El paso inicial de la iteración nos determina la regla base, el punto de inicio: `producto=1`. El último paso de la iteración nos determina la regla recursiva (los demás casos): `n!= (n-1)! * n`. Esto es:

$$n! = \begin{cases} 1 & \text{si } n = 1 \\ n \times (n-1)! & \text{si } n > 1 \end{cases}$$

□

La recursion es una alternativa a la iteración en la solución de problemas, especialmente si estos tienen naturaleza recursiva. Normalmente, una solución recursiva es menos eficiente que la solución iterativa debido al tiempo adicional de llamada a procedimientos. Sin embargo, en muchos casos, la recursión permite especificar una solución mas simple y natural para resolver un problema que en otro caso seria mas difícil. Por esta razón la recursividad es una herramienta muy potente para la resolución de problemas de programación.

La programación como técnica para la resolución de problemas también contempla la recursividad. Una primera definición al concepto de recursividad desde la perspectiva de la programación es: "una función es recursiva si en su cuerpo se invoca a sí misma". Al igual que en el planteamiento teórico, no se puede entrar en un ciclo sin fin. Solo se puede construir una función recursiva si sus acciones responden a una recursividad estructural (porque hay una definición recursiva que las justifica) y, por tanto, toda función recursiva deberá contemplar una regla base, una regla recursiva y, si fuese necesario, una regla de exclusión.

Ejemplo 6.4. En el ejemplo anterior hemos justificado cómo el procedimiento del cálculo del factorial de un número es un procedimiento recursivo. Además dicho procedimiento da como resultado un valor numérico lo que nos permite expresar dicho proceso con el siguiente código:

```
def  factorial (n: int):
    if n < 1:
        return -1  # Regla de exclusión. Un -1 es un error. No existe el factorial.
    if n == 1:
        return 1   # Regla base. El producto del primer número natural.
    return n *  factorial(n-1) #  Regla recursiva. El producto de n-números naturales.
}
```

La recursividad que aquí se muestra consiste en que la función **`factorial()`** se invoca a sí misma dentro del cuerpo de la función. Pero esta no es la única forma de recursividad.

□

Las estructuras que vamos a estudiar en este capítulo son estructuras recursivas y, por tanto, muchos de sus métodos también serán recursivos. Recuerde que todo procedimiento recursivo tiene un método alternativo iterativo pero, aunque el procedimiento recursivo es computacionalmente más costoso, la expresión de los procesos es mucho más simple por lo que merece la pena su uso.

6.B TDA Árbol

Los árboles son tipos de datos no lineales, dinámicos y jerárquicos:

- No lineales porque cada elemento del árbol puede tener de 0 a varios sucesores.
- Dinámicas porque pueden cambiar de forma y de tamaño durante la ejecución del programa.

- Es un TDA con ordenación jerárquica. Una relación binaria $a \preceq b$ se dice que es un orden jerárquico sii hay un solo objeto raíz r y se cumple:
  - Para todo a, $a \preceq a$ (Reflexiva),
  - Para todo a, $r \preceq a$ (existe un mínimo en el orden).
  - Si $a \preceq c$ y $b \preceq c$, esto implica que $a \preceq b$ o $b \preceq a$.
  - La relación es antisimétrica: $a \preceq b$ y $b \preceq a$, esto implica que $a = b$
  - Es transitiva: $a \preceq b$ y $b \preceq c$, esto implica que $a \preceq c$

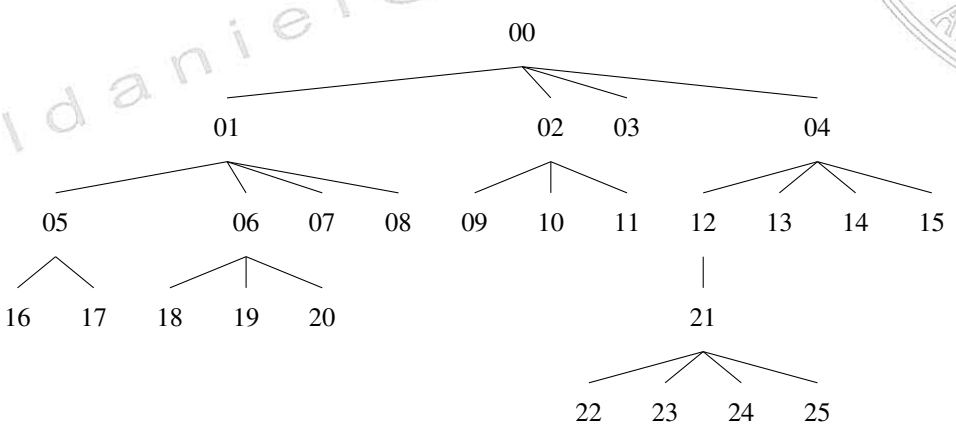
  En este orden pueden existir elementos no comparables. Se dice que dos objetos a y b son comparables sii $a \preceq b$ o $b \preceq a$; de lo contrario, se dice que no son comparables.

Definición 6.1: 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.

Ejemplo 6.5.



En la figura tenemos el árbol formado por el 00 y una lista de 4 árboles.

- El primero está formado por el 01 y 4 árboles
- El segunda consta del 02 y 3 árboles.
- El tercero lo forma el 03 y cero-árboles.
  Éste también se puede definir como el que está definido por el 03 y un número (el que queramos) de árboles vacíos.
- Y el cuarto está formado por el 04 y 4 árboles.

□

Asociado a todo árbol tenemos las siguientes definiciones:

- Se llama vértice o **nodo**[1] a cada uno de los datos de un árbol.
  Ejemplo: El árbol del Ejemplo 6.5 consta de 26 nodos.
- La lista de árboles de la colección se llaman **subárboles** del árbol.
  Ejemplo: En el árbol del Ejemplo 6.5 el nodo 00 tiene 4 subárboles y el nodo 06 tienen 3 subárboles.
- El dato distinguido en la definición recibe el nombre de **nodo raíz**.
  Ejemplo: Si en el Ejemplo 6.5 consideramos el árbol completo, el 00 es el nodo raíz.
- Los nodos raíz de los subárboles se llaman **nodos hijos** del nodo raíz del árbol.
  Ejemplo: En el árbol del Ejemplo 6.5 el nodo 00 tiene como nodos hijos a los nodos 01, 02, 03 y 04. Y el nodo 02 tiene como hijos al 09, 10 y 11.
- Todos los nodos (de los subárboles) de un árbol distinto de la raíz se llaman **nodos descendientes** del nodo raíz del árbol.
  Ejemplo: En el árbol del Ejemplo 6.5 el 00 tiene como descendientes a todos los demás nodos y el 20 es un descendiente de los árboles que tienen como raíz al 00, al 01 y al 06.

[1] No debe confundir este concepto de nodo como vértice con el concepto de nodo como representación de la página 62. Recuerda distinguir siempre el concepto (un vértice o valor) de la representación (una estructura valor+referencias).

- El nodo raíz también se llama **nodo padre** de los nodos raíz de los subárboles. Los **nodos antecedentes** de un nodo son todos aquellos que tienen como descendiente al nodo.
  Ejemplo: En el árbol del Ejemplo 6.5 el 04 es el nodo padre de los nodos 12, 13, 14 y 15. El nodo 00 es antecedente del resto de los nodos y en particular padre del 01, 02, 03 y 04. Los antecedentes del 23 son 21, 12, 04 y 00.
- El dato distinguido en la definición recibe el nombre de **nodo hoja** si el árbol no tienen ningún subárbol. O si se prefiere es el que no tiene descendientes.
  Ejemplo: En el Ejemplo 6.5 algunos nodos hojas son 03, 07, 09, 15, 20, ...
- Un nodo interno es aquel que no es ni nodo padre ni es nodo hoja. Es decir, tiene un padre y al menos un hijo.
  Ejemplo: En el Ejemplo 6.5 algunos nodos hojas son 01, 06, 12, 21, ... son nodos internos.
  Ejemplo: En el Ejemplo 6.5 algunos nodos hojas son 03, 07, 09, 15, 20, ...
- Se llama **camino** entre dos nodos n y m a una secuencia de nodos $[a_0 = n, a_1, a_2, \ldots, a_k = m]$ donde cada nodo es padre del siguiente.
  Ejemplo: En el Ejemplo 6.5 algunos caminos son (04, 12, 21, 22), (00, 01, 06, 19), ...
- A la relación que existe entre un nodo raíz y un nodo hijo se le llama **arista** (se representa mediante una línea o una flecha).
  Ejemplo: Si se tienen el camino (04, 12, 21, 22) es porque se tienen las aristas 04→12, 12→21 y 21→22.
- Se llama **longitud** del camino al número de aristas que hay en un camino.
  Así, una arista es un camino de longitud 1. Por ello, se representan de forma indistinta (04, 12) que 04→12 o (12, 21) que 12→21.
- La **altura** de un nodo es la longitud del camino más largo que le une con cualquiera de sus descendientes. Por definición, la altura de un nodo hoja es 0.
  Ejemplo: En el Ejemplo 6.5 la altura del nodo 17 es 0, la del 01 es 2 y la del nodo 00 es 4.
- La **profundidad** de un nodo es la longitud del camino que une a la raíz con dicho nodo. Por definición, la profundidad del nodo raíz es 0.
  Ejemplo: En el Ejemplo 6.5 la profundidad del nodo 00 es 0, la del nodo 10 es 2, la del 22 es 4.
- Se llama **nivel** al conjunto de nodos que están a la misma profundidad. El nivel 0 está formado por el nodo raíz, el nivel 1 está formado por los hijos del nodo raíz, etc
  Ejemplo: En el Ejemplo 6.5 los nodos del nivel 1 son {01, 02, 03, 04} y los del nivel 4 son {22, 23, 24, 25}.
- Un **árbol n-ário** es aquel en el cada nodo no tienen más de n-hijos.
  Ejemplo: En el Ejemplo 6.5 los árboles con raíz el 00 y el 21 son árboles 4-ários. Son 3-ários los que tienen como raíz al 02 y al 06. Por contra es 2-ário el que tiene por raíz al 05.
  Notar que por definición, un árbol 2-ários también es 3-ário, y estos son 4-ários, etc ..

Definición 6.2: 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.

¿Qué pasaría si al trabajar con un contenedor iterable de nodos del árbol, el árbol fuera modificado cambiando valores, añadiendo nodos o borrando nodos?

6.B.1 Árbol Binario

De entre los árboles 2-ários distinguimos los árboles binarios.

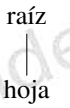
Definición 6.3: Árbol Binario

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.

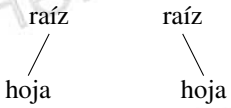
La diferencia entre un árbol binario y otros es que en los binarios damos nombre a los subárboles y estos siempre existen. Es bien diferente a un árbol 2-ários donde sus susbárboles no se distinguen.

Ejemplo 6.6. Imagina que tienes un árbol formado por 2 nodos.

Si se considera que el árbol es 2-ário, solo puede construir un árbol:



Peros si se considera que el árbol es binario, se pueden construir dos árboles:



En uno el hijo izquierdo tiene un nodo y el hijo derecho es vacío, y en el otro el hijo izquierdo es vacío y el hijo derecho tiene un nodo. □

La razón por la que los árboles binarios se utilizan con más frecuencia que los árboles n-arios es que los binarios proporcionan una ventaja de velocidad real (cuando el árbol está bien equilibrado). De hecho tiene muchas aplicaciones. En cada aplicación el árbol binario se adapta para el propósito específico. Se mencionan algunos.

- Árbol de búsqueda binario. Se utiliza en muchas aplicaciones de búsqueda donde los datos entran/salen constantemente, como los objetos map y set en las bibliotecas de muchos idiomas.
- Partición de espacio binario. Determina que objetos de un escenario deben renderizarse antes. Se utiliza en casi todos los videojuegos 3D.
- Montones. Se usan para implementar colas de prioridad. Estas colas se usan en sistemas operativos (prioridad de procesos) y en algoritmo de búsqueda de caminos (A* se usa en aplicaciones de AI , incluyendo robótica y videojuegos).

- Huffman Coding Tree. Se utiliza como algoritmo de compresión, como los utilizados por los formatos de archivo .jpeg y .mp3.

Sin entrar en aplicaciones concretas y pensando en árboles binarios en general, nos encontramos que para realizar alguna acción sobre todos los nodos de un árbol binario será necesario establecer algún tipo de recorrido. Se sigue uno de estos dos recorridos: recorrido en anchura o recorrido en profundidad. Veamos los pseudocódigos usando una estructura de nodo con dos referencias: `izquierdo` y `derecho`.

- El recorrido en anchura consiste en actuar sobre el nodo raíz o nivel cero, después sobre los nodos del nivel 1 después sobre los nodos del nivel 2 y así sucesivamente. Para realizar este recorrido se necesita recurrir a una cola:

```
def recorrido_anchura (nodo):
    cola = Cola()
    cola.encola(nodo)
    while not cola.isEmpty():
        node = cola.desencola()
        accion(node) # P.e. print(node)
        if  nodo.izquierdo is not None:
            cola.encola(nodo.izquierdo)
        if nodo.derecho is not None:
            cola.encola(nodo.derecho)
```

- El recorrido en profundidad sigue alguna de las siguientes 3 estrategias recursivas:

1. Recorrido prefijo.

```
def recorrido_prefijo (nodo):
    accion(nodo)
    recorrido_prefijo(nodo.izquierdo)
    recorrido_prefijo(nodol.derecho)
```

2. Recorrido infijo.

```
def recorrido_infijo (nodo):
    recorrido_infijo(nodo.izquierdo)
    accion(nodo)
    recorrido_infijo(nodol.derecho)
```

3. Recorrido postfijo.

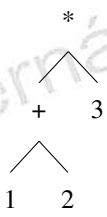
```
def recorrido_postfijo (nodo):
    recorrido_postfijo(nodo.izquierdo)
    recorrido_postfijo(nodol.derecho)
    accion(nodo)
```

Se pueden hacer recorridos en profundidad sin recursividad utilizando pilas:

```
def recorrido_profundidad (nodo):
    pila = Pila()
    pila.push(nodo)
    while not pila.isEmpty():
        node = pila.pop()
        accion(node) # P.e. print(node)
        if  nodo.izquierdo is not None:
            pila.push(nodo.izquierdo)
        if nodo.derecho is not None:
            pila.push(nodo.derecho)
```

- En todas las estrategias se supone que nodo es una referencia al nodo raíz del árbol.

Ejemplo 6.7. Supongamos que se tiene el siguiente árbol que representa a una operación aritmética:



El objetivo de este ejemplo es realizar un recorrido completo para imprimir todos los nodos. Para ello se cambia `def recorrido` por `def imprimir` para que tenga la función un nombre representativo sobre lo que se quiere hacer. También se cambiará `accion(nodo)` por `print(nodo)` entendido que tal impresión muestra correctamente el contenido del nodo.

El resultado de la impresión en cada caso es:

- Recorrido prefijo: $*+123$
- Recorrido infijo: $1+2*3$
- Recorrido postfijo: $12+3*$

□

Como se puede ver del ejemplo, algunos recorridos son mejores que otros. Así, por ejemplo, imprimir los nodos según el recorrido infijo mostrará los mismos símbolos que los usados al escribir expresiones aritméticas. Las otras dos no nos resultan familiares como expresiones aritméticas. Sin embargo, si realizamos el recorrido prefijo el resultado es la expresión aritmética en notación Polaca. En esta notación, si se realiza la última operación más a la derecha después de leer un segundo número (no entramos en más detalles) resulta que es la forma más sencilla para un ordenador de calcular el resultado de una expresión aritmética. En resumen, si una expresión aritmética está expresada en forma de árbol entonces el recorrido infijo para imprimir es el mejor recorrido pero el recorrido prefijo es el mejor recorrido para calcular el resultado de la expresión. La conclusión es que **se deberá de usar el recorrido más adecuado en función del tipo de acción que queramos realizar sobre un árbol**.

Se indican algunos ejemplos de acciones que llevan recorridos[2][3]:

- **mostrar(arbol)**: Mostrar los elementos de un árbol como si fuera una expresión aritmética:
  - Caso base: Mostrar los elementos de un árbol vacío es hacer nada.
  - Caso general: Realizar un recorrido infijo (mostrando los nodos)
- **mostrar(arbol)**: Mostrar los elementos de un árbol como si fuera una estructura de directorio:
  - Caso base: Mostrar los elementos de un árbol vacío es hacer nada.
  - Caso general: Realizar un recorrido prefijo (mostrando los nodos y realizando mayor tabulación a mayor profundiad).
- **borrar(arbol)**: Borrar los nodos empezando por el nivel más profundo.
  - Caso base: Borrar los elementos de un árbol vacío es hacer nada.
  - Caso general: Realizar un recorrido postfijo (realizando como acción la eliminación del nodo raíz).
- **copiar(arbol): arbol**: Copiar un árbol empezando por la raíz
  - Caso base: Si el árbol está vacío no hay nada que copiar.
  - Caso general:
    - Crear un nuevo árbol binario con valor igual al nodo dado.
    - Añadir como hijo izquierdo la copia del hijo izquierdo del nodo dado.
    - Añadir como hijo derecho la copia del hijo derecho del nodo dado.
    - Retornar el nuevo árbol

    Se sigue un orden prefijo.
- **contar(arbol): int**: Contar el número de elementos que cumplan cierto criterio.
  - Caso base: Si el árbol está vacío no existen elementos que comparar.
  - Caso general: Retornar num = numR + numIzdo + numDcho

[2] Se asume que se trabaja con la estructura de la alternativa 1 (un nodo y un árbol es lo mimos). Las acciones se muestran como funciones.

[3] En Canal Youtube Juan A. Sánchez: Árboles puede encontrar la implementación de todos estos métodos en lenguaje C, implementado con la estructura de la alternativa 1 y con la estructura Primer Hijo - Hermano derecho. Los tipos de estructuras se explican más adelante.

- numR=1 si el nodo raíz cumple el criterio. Será 0 si no lo cumple.
- numIzdo es el número de elementos encontrados en la hijo izquierdo.
- numDcho es el número de elementos encontrados en la hijo derecho.

Da igual la estrategia usada.

- **buscar(arbol, valor): bool**: Buscar un elemento en el árbol.
  - Caso base:
    - Si el árbol está vacío no existe el elemento.
    - Si el elemento es el nodo raíz indicar que se encontró.
  - Caso general: Si el elemento no está en la raíz
    - Buscar el elemento en el hijo izquierdo. Si lo encontró, indicar que se encontró.
    - Buscar el elemento en el hijo derecho. Si lo encontró, indicar que se encontró.
    - Indicar que no se encontró.

  Indicar que lo encontró es hacer `return True`, en caso contrario `return False`.
- **comparar(arbol1, arbol2): bool**: Comparar dos árboles.
  - Caso base:
    - Si los dos son vacíos entonces indicar que son iguales.
    - En otro caso, si uno de ellos es vacío indicar que son distintos (porque el otro sí tendrá elementos)
  - Caso general:
    - Si los valores de los dos nodos son distintos indicar que son distintos.
    - Calcular resultadoIzdo como el resultado de la comparación de los hijos izquierdos de los dos nodos.
    - Calcular resultadoDcho como el resultado de la comparación de los hijos izquierdos de los dos nodos.
    - Retornar resultadoIzdo&&resultadoDecho
- **altura(arbol): int**: Calcular la altura de un árbol
  - Caso base:
    - La altura de un árbol formado por un nodo hoja es 0
    - La altura de un árbol vacío no está definido.
  - Caso general: El nodo tiene algún hijo.
    - Calcular alturaIzda como la altura del hijo izquierdo del nodo.
    - Calcular alturaDcha como la altura del hijo derecho del nodo.
    - Retornar 1+ max(alturaIzda, alturaDcha)
- **hojas(arbol): int**: Calcular el número de hojas.
  - Caso base:
    - Si el árbol está formado por un nodo hoja retornar +1.
    - Si es un árbol vacío retornar 0.
  - Caso general: El nodo tiene algún hijo.
    - Calcular hojasIzda como el número de hojas del hijo izquierdo del nodo.
    - Calcular hojasDcha como el número de hojas del hijo derecho del nodo.
    - Retornar hojasIzda+hojasDcha.

6.B.2 Árboles binarios de búsqueda

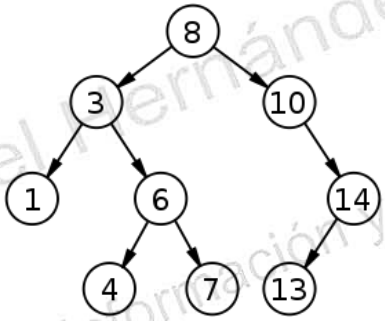
Estos árboles son árboles binarios que cumplen la siguientes 2 propiedades:

- 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.

Ejemplo 6.8. Un ejemplo de árbol binario de búsqueda:



□

Ya conoce la técnica de búsqueda binaria en secuencias ordenadas y sabe también que es un técnica eficiente. Esta estructura busca lo mismo pero con estructuras enlazadas. De hecho si se realiza un recorrido **inorden** de sus elementos obtenemos una lista ordenada de todos sus elementos. También hay que tener en cuenta que se necesita que el árbol esté bien balanceado para que sus algoritmos resulten eficientes.

Los algoritmos más relevantes en estos árboles son los siguientes.

- **buscar(arbol, valor): bool**: Algoritmo de búsqueda.
  1. Si el árbol está vacío indicar fracaso.
  2. Si el elemento está en la raíz indicar éxito.
  3. Si el elemento a buscar es menor que el nodo raíz buscar en el subárbol izquierdo.
  4. Si el elemento a buscar es mayor que el nodo raíz buscar en el subárbol derecho.
- **añadir(arbol, valor)**: Añadir un elemento en el árbol. Los nuevos elemento se añaden siempre como nodos hojas.
  1. Si el árbol es nulo, crear un árbol (nodo) con el elemento y terminar.
  2. Si el dato es menor que la raíz añadir el elemento en la rama izquierda
  3. Si el dato es mayor que la raíz añadir el elemento en la rama derecha.
- **minimo(arbol): valor**: Buscar el valor mínimo de un árbol.
  1. Si el árbol es nulo, no existe
  2. Si el árbol no tiene hijo izquierdo, el mínimo es la raíz.
  3. Si el árbol tiene hijo izquierdo, el mínimo es el mínimo del hijo izquierdo.
- **maximo(arbol): valor**: Buscar el valor máximo de un árbol.
  Es el proceso anterior pero hay que cambiar hijo izquierdo por hijo derecho.
- **borrar(arbol, valor): arbol**: Eliminar el nodo que tiene un valor. El árbol resultante tiene que seguir siendo un árbol binario de búsqueda por lo que distinguiremos varios casos:
  1. Si el árbol (o nodo) es nulo, nada que borrar.
  2. Si el nodo contiene el valor distinguir dos casos
     a) El nodo solo tiene un hijo, se reemplaza por su hijo.
     Dependiendo del lenguaje, tendrás que borrar el nodo de forma explícita antes del retorno. En Python no es necesario.
     b) El nodo/árbol tiene los dos hijos. Se reemplaza con el menor de sus hijos mayores.
        1) Buscar el valor mínimo del árbol derecho: v_{min}.
        2) Sustituir el valor del nodo por ese valor mínimo.
        3) El nodo que contienen v_{min}, por definición no puede contener hijo izquierdo; pero sí hijo derecho. De ser así, asignar como árbol derecho el resultado de eliminar el nodo que tenga el valor mínimo empezando por el árbol derecho. Es decir, proceder a eliminar este nodo (casos 1 o 2.a)) para realizar al eliminación completa.:
        `arbol.dcho = borrar(arbol.dcho, vmin)`
        4) Retornar el nodo.

     Alternativamente se puede buscar el valor máximo por la rama izquierda (de sus hijos menores).
  3. Si el valor es más pequeño que el valor del nodo, suprimir el nodo que tenga el valor empezando por el hijo izquierdo. Asignar al hijo izquierdo el resultado del borrado.
  4. Si el valor es mayor que el valor del nodo, suprimir el nodo que tenga el valor empezando por el hijo derecho. Asignar al hijo derecho el resultado del borrado.

5. Retornar el nodo.

6.B.3 Heap

El término heap se puede traducir por montículo, cúmulo, montón o pila. Mantendremos el término en inglés.

Un heap es un árbol binario donde cada nodo consta de una pareja $(clave, valor)$. Existe una ordenación entre los nodos:

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

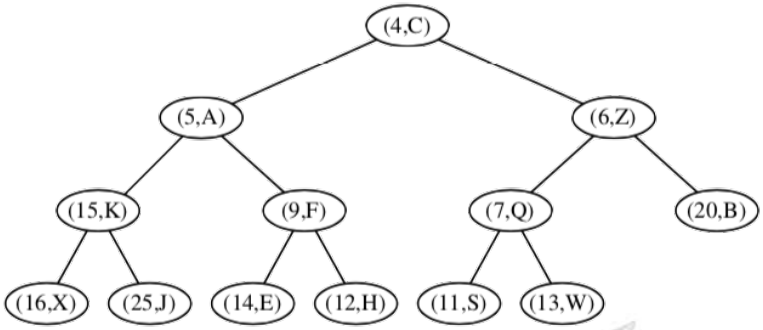
En muchos casos la clave se obtiene a partir del valor. Por ejemplo, si los valores son numéricos entonces se puede usar como clave dicho valor numérico. Si los valores son objetos de tipo Persona, se puede usar como clave el nombre de la persona o la edad de la persona o una combinación de ambos. En estos casos, la ordenación puede reescribirse como:

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

Todo heap satisface estas dos propiedades:

- Ordenación del Heap: El valor de cada nodo es mayor o igual que la valor almacenado en el padre. Como consecuencia de esta propiedad, el camino desde la raíz a una hoja están en orden no decreciente. Además, siempre se almacena el valor mínimo en la raíz del árbol. En este caso se llama Min-Heap.
  Se puede cambiar el criterio de ordenación: El valor de cada nodo es menor o igual que la valor almacenado en el padre. En este caso el camino desde la raíz a una hoja están en orden no creciente y la raíz almacenará el valor máximo. En este caso se llama Max-Heap.
- Árbol binario completo: Un árbol binario es completo si al tener el árbol profundidad h, se cumple que los niveles 0, 1, 2, ..., $h-1$ tienen el número máximo de nodos posible (es decir, dos nodos) y los nodos restantes en el nivel h residen en las posiciones más a la izquierda de ese nivel.

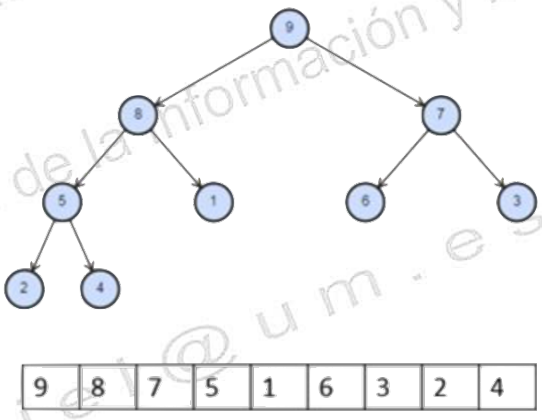
Ejemplo 6.9. Un ejemplo de Heap-Min:



Fuente: *Data Structures and Algorithms in Python* □

Se puede implementar una Cola de Prioridad (sección) con un Heap. Además un heap se peede/debb implementar como un lista (p.e. un array1D). En dicha lista, si un nodo padre del árbol se encuentra en la posición p de la lista, entonces sus nodos hijos se encuentran en las posiciones $2p+1$ y $2p+2$. Así mismo, si un nodo hijo del árbol se encuentra en la posición p de la lista, entonces sus nodo padre se encuentran en la posición $(p-1)/2$.

Ejemplo 6.10. Un ejemplo de Heap-Max:



Fuente: *Problem Solving in Data Structures Algorithms Using Python Programming Interview Guide by Hemant Jain* □

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 Heap-Min
  El nuevo elemento se añade como último elemento del árbol. Pero puede que incumpla la propiedad de la heap: que el nuevo elemento sea menor que su padre. Si fuera el caso se aplica filtrado hacia arriba para restaurar la propiedad de orden:
  - Si el nuevo elemento es más pequeño que su padre, intercambiar ambos elementos.
  - Repetir el paso anterior tantas veces como sea necesario.

  El filtrado termina cuando la clave alcanza la raíz o un nodo tal que su padre tienen una clave menor..
- 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-Min.
  Los pasos a realizar son:
  - Se guarda el dato de la raíz.
  - Se elimina el último elemento del heap y se almacena en la raíz.
  - Se hace filtrado hacia abajo para restaura la propiedad de orden:
    - El nuevo elemento colocado en la raíz se intercambia por el más pequeño de sus dos hijos.
    - Repetir el paso anterior tantas veces como sea necesario.

  El filtrado termina cuando la clave alcanza un lugar correcto donde puede estar insertado.

Los heap se usan en multitud de situaciones. Comentamos brevemente algunas:

- Heapsort. Es una de los mejores métodos de ordenación en cualquier escenario. Consiste en crear un heap-min o un heap-max, según el caso, a partir de una secuencia (p.e. un lista) y retornar la lista resultante de aplicar de forma reiterada la extracción del nodo raíz. Se puede hacer de dos formas:
  - Se insertan los elementos uno a uno en un heap.
  - BuildHead: Se añaden los elementos desordenados en un árbol binario completo. Posteriormente se hace filtrado hacia abajo de cada uno de sus elementos recorriendo los elementos desde la posición del último elemento del array hasta el primero.
- Algoritmos de selección. Encontrar ciertos valores destacados de una colección. P.e. el mínimo, el máximo, la mediana o los k-elementos más grandes.
- Colas de prioridad. Las colas de prioridad se usan en algoritmos sobre grafos como el algoritmo de Prim, de Dijkstra, uniforme o A*.

6.B.3 Representación de los árboles

Representación general. Tenemos varias alternativas siguiendo las definiciones:

Alternativa 1. Hemos indicado que un árbol puede ser nulo o puede constar de un dato y una lista de varios subárboles. Esto responde a la estructura:

```
struct Arbol {
   TDato dato
   Secuencia_Arbol subarboles
}
```

A nivel de implementación `Secuencia_Arbol` tendrá que especificarse: puede ser una lista, un array, ...

Alternativa 2. En las definiciones de los elementos de un árbol también hemos hablado de nodos (valores destacados) y nodos hijos (valores destacados de los subárboles). Esto responde a esta estructura:

```
struct Arbol {
   Nodo raiz
}
```

```
struct Nodo {
   TDato dato
   Secuencia_Arbol subarbolesHijos
}
```

Notar que la estructura `Arbol` de la primera alternativa es la estructura `Nodo` de la segunda alternativa (solo cambia el nombre). Si optas por la primera estructura puedes entonces usar el nombre que te resulte más conveniente: `Arbol`, `Nodo`,

`TreeNode`, ... teniendo en cuenta que en esa primera representación el concepto de nodo y árbol es indistinto.

Observa que nos encontramos en la misma situación de las listas con estructuras enlazadas. ¿ Recuerdas que en listas enlazadas se podía tener un primer nodo que no representaba a ningún valor de la lista pero que estaba ahí porque facilitaba algunas operaciones? Por supuesto también se podía trabajar con una estructura de lista enlazada sin nodo cabecera. Aquí está pasando de alguna forma mismo. Vale, ¿pero cuál es la mejor? Pues depende. En la práctica podemos encontrar dos formas básicas de representar un árbol y ambas responden a Estructuras Enlazadas.

- Alternativa 1.
  En esta representación, árbol y nodo se consideran lo mismo y se representa de la misma forma. Ya es cuestión de gustos si la estructura se llama Arbol o Nodo.
  En esta representación un árbol es vacío si la variable de tipo árbol es None. La ventaja de esta representación es que se usa una sola estructura lo que pueda facilitar algunos métodos de programación. Sin embargo, los métodos que se definan pueden requerir cierta especificación adicional. Por ejemplo, si se considera el método de comparación **equ()** ¿qué se está comparando dos árboles o dos nodos?
- Alternativa 2.
  En esta representación un árbol consta de un campo que hace referencia a una estructura de tipo nodo que contiene la lista de nodos hijos.
  En esta representación un árbol es vacío si el campo `nodo` de la estructura `Arbol` es None.
  Con esta representación se pueden añadir más métodos propios de un árbol que no requieran recursividad en el recorrido de los nodos. También se pueden usar métodos con nombres iguales en ambas estructuras. Por ejemplo, se puede implementar el método **equ()** en ambas estructuras. Una sería comprobar si dos árboles son iguales (ver definición anterior) y para la otra comprobar si los valores de los dos nodos son iguales.

Representación de árboles binarios: padre-hijo. En el caso de que el árbol sea un **árbol binario**, se utiliza una estructura donde se distinguen cada uno de los hijos.

```
struct Nodo {
   TDato dato
   Nodo izquierdo
   Nodo derecho
}
```

Recuerda que en listas podía interesar usar listas doblemente enlazada para poder acceder no solo al nodo siguiente sino también al nodo anterior. En las estructuras mencionadas tenemos enlaces a los nodos hijos (los nodos siguientes) y se puede, si interesa, tener otro campo **para referenciar al nodo padre**. Añadir este campo en la estructura puede ayudar a resolver problemas como el de encontrar el camino entre un nodo y el nodo raíz o el problema de calcular la profundidad de un nodo. Para algunos problemas concretos se requiere esta referencia al padre.

Representación de árboles binarios: hijo-hermano_derecho. Otra forma de representar a un árbol cualquiera es mediante una estructura doblemente enlazada es con la siguiente estructura como Primer Hijo - Hermano Derecho:

```
struct Nodo {
   TDato dato
   Nodo primerHijo
   Nodo hermanoDerecho
}
```

Con esta estructura se pone de manifiesto que todo Nodo consta de una lista de hijos siendo los dos primeros los que se indican en la estructura. Además indica que todo Nodo forma parte de un nivel que viene dada por la lista de hermanos que define el campo hermanoDerecho.

Observar que es la misma estructura que la del árbol binario solo que ahora las referencias de los campos tienen otro significado. La ventaja de esta estructura es que se puede extender muchos algoritmos de árboles binarios a árboles n-ários usando las estrategias de recorrido indicadas anteriormente pero con pequeñas modificaciones (hay que tener en cuenta que pueden existir varios hermanos derechos).

Por ejemplo, los algoritmos de impresión en preorden y postoden son:

```python
def preorden (nodo: Nodo):
    return if nodo is None
    print(nodo)
    nodo = nodo.primerHijo
    while nodo not is None:
        preorden(nodo)
        nodo = nodo.hermanoDerecho
```

```python
def postorden (nodo: Nodo):
    return if nodo is None
    nodo = nodo.primerHijo
    while nodo not is None:
        postorden(nodo)
        nodo = nodo.hermanoDerecho
    print(nodo)
```

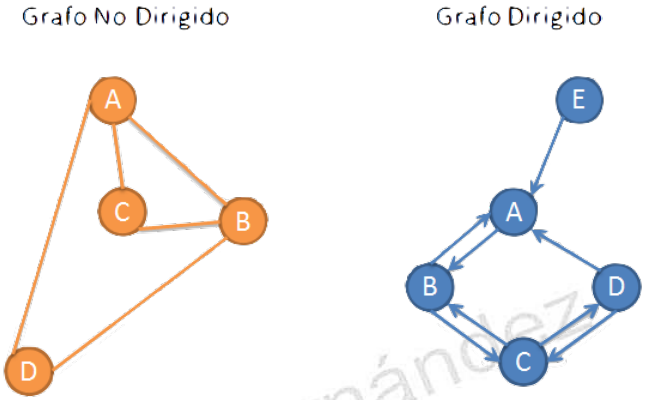
6.C TDA Grafo

Los grafos son tipos de datos no lineales que sirve para representar relaciones binarias entre elementos. Un grafo se define como un par de conjuntos $G = (V, E)$, donde V es un conjunto finito de elementos llamados vértices y E es un conjunto finito de arcos. Un arco es una tupla (u, v) donde $u, v \in V$. La tupla puede contener una tercera componente formada por un número real que recibe el nombre de peso y represente un costo para ir del vértice u al vértice v.

Los arcos pueden ser dirigido o no dirigidos. Serán no-dirigidos si la relación E es una relación simétrica. Esto define lo Grafos Dirigidos (o Digraph) cuando sus arcos son dirigidos y los Grafos no Dirigidos si la relación es simétrica.

Gráficamente se puede representar como una serie de puntos en el plano, uno por cada vértice y flechas que conectan los puntos para representar los arcos (u, v). En el caso de que exista las tupla (u, v) y (v, u) entonces se tienen una arista (una sola línea sin puntas de flecha que conecta a los dos vértices).

Ejemplo 6.11. Ejemplos de grafos son los siguientes

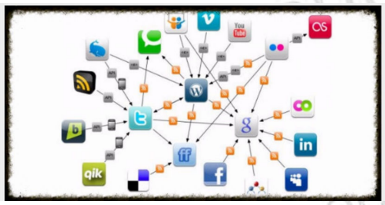


Fuente: https://blogs.ua.es/jabibics/2011/01/05/capitulo-7-tipos-de-grafos/ □

Los grafos se usan en muchos problemas reales. Mencionamos algunos:

- Hemos de colocar un recurso (p.e. una fábrica) en un lugar que optimice la recogida de recursos y distribución de productos.
- Secuencia de estados. Autómatas finitos (p.e. juegos)
- Buscar el camino mínimos entre un punto geográfico y otro.
- Determinar el flujo de pasajeros que se pueden transportar en avión desde una ciudad a otra (los aviones pueden ir directamente o haciendo escala en otros lugares).
- Diseñar una red de comunicación sin que se pierda calidad.
- Reconstruir la red social de un individuo para fines comerciales.

Ejemplo 6.12. Ejemplos de grafos en situaciones reales



Fuente:
https://jcrd0730.wixsite.com/estr/single-post/2016/05/25/aplicaciones-de-grafos-1 □

Algunas definiciones relacionadas con los grafos son:

- Un **bucle** o loop es una arista que conecta un vértice consigo mismo.
- **Grado de entrada** (in-degree) de un vértice v es el número de arcos de entrada en el vértice v. Notar que un loop en un vértice aporta +1 en el cálculo.
- **Grado de salida** (out-degree) de un vértice v es el número de arcos de salida en el vértice v. Notar que un loop en un vértice aporta +1 en el cálculo.
- **Grado** de un vértice v es la suma de in-degree y out-degree del vértice v. Notar que un loop en un vértice aporta +2 en el cálculo.
- Un **camino** entre los vértices u y v es una secuencia de vértices $[a_1 = u, a_2, \ldots, a_n = v]$ tales que (a_i, a_{i+1}) es un arco.
- Un **ciclo** es un camino que empieza y termina en el mismo vértice e incluye al menos un vértice.
- Un **grafo acíclico** es aquel que no tiene ciclos.
- Un nodo v es **alcanzable** desde el vértice u si hay un camino entre los vértices u y v.
  En un grafo no dirigido si v es alcanzable desde u entonces u es alcanzable desde v. Pero en un grafo dirigido si v es alcanzable desde u entonces no se garantiza que u sea alcanzable desde v.
- Un **grafo conexo** cumple que todo nodo v es alcanzable desde todos los nodos restantes del grafo.
- Un **bosque** (forest) es un grafo sin ciclos.
- Un **subgrafo** de un grafo G es un grafo que contiene algunos vértices de G y algunos de los arcos que los conectan.
- Un **árbol** se define como un grafo acíclico conexo.
  La definición que se da aquí de árbol no es la definición de árbol que se vio al definir el TDA Árbol (era una definición recursiva). Es decir, aquí no hay un nodo raíz, hijos, ascendentes, etc. Sin embargo se puede comprobar que se puede considerar un nodo en el árbol-grafo que puede actuar como raíz y a partir de él construir niveles hasta llegar a una estructura que responden a la definición recursiva de árbol.
  Algunos resultados interesantes que después se usan en los algoritmos son:
  Dado un grafo G, son equivalentes:
  - G es un árbol
  - G es un grafo sin ciclos pero si se agrega cualquier arista e a G resulta un grafo con exactamente un ciclo simple y ese ciclo contiene a e.
  - Existe exactamente un camino simple entre todo par de vértices.
  - G es conexo, pero si se quita cualquier arista a G queda un grafo no conexo.

Operaciones que pueden realizarse sobre un grafo son:

- Agregar o quitar vértices.
- Agregar o quitar arcos/aristas.
- Buscar un dato (vértice) en el grafo.
- Comprobar si dos vértices están unidos por un camino.
- Obtener los nodos adyacentes a uno dado.
- Obtener todos los vértices.
- Iterar sobre los vértices.
- Obtener un árbol generador del grafo.
  Un árbol generador es un subgrafo del grafo que es un árbol y tiene el mismo conjunto de vértices que el grafo.

6.C.1 Recorrido en Grafos

Un recorrido en grafos es similar al recorrido en árboles. Podemos distinguir las siguientes técnicas:

- Recorrido en anchura. Se parte de un nodo inicial y se introduce en una cola. Entonces, mientras que la pila no esté vacía extraemos el elemento de la pila, lo marcamos como visitado y añadimos a la cola todos los adyacentes al nodo que no hayan sido visitados.
- Recorrido en profundidad. Es igual que en anchura pero se usa una pila.
  Alternativamente se puede usar recursividad. Cada vez que se considera un adyacente de un nodo, se vuelve a invocar a la función de recorrido.

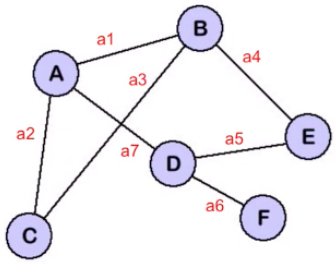
6.C.2 Representaciones de un Grafo

Las representaciones principales de grafos son las siguientes:

- **Matriz de incidencia (MI):** se utiliza una matriz de tamaño $k \times n$ donde las filas representan a las aristas y las columnas a los vértices. La posición $MI[i, j]$ indica qué relación tiene la arista i-ésima con el vértice j-esimo.
  Si la arista i-ésima se conecta el vértice j con el vértice k, $a_i = (v_j, v_k)$, entonces $MI[i, j] = -1$ y $M[i, k] = +1$.
  Es decir, -1 indica que el vértice es de "salida" del arco y $+1$ indica que el vértice es la "llegada" del arco.
  En el caso de que el arco tenga algún peso, se cambia el valor 1 por el valor del peso.

  Ejemplo 6.13. Matriz de incidencia

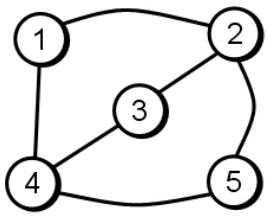
	A	B	C	D	E	F
a1	1	1	0	0	0	0
a2	1	0	1	0	0	0
a3	0	1	1	0	0	0
a4	0	1	0	0	1	0
a5	0	0	0	1	1	0
a6	0	0	0	1	0	1
a7	1	0	0	1	0	0



  Fuente: https://youtu.be/0PjgvCYHbJA □

- **Matriz de adyacencia (MA):** Se utiliza una matriz de tamaño $n \times n$ donde las filas y las columnas hacen referencia a los vértices. Cada combinación de fila y columna representa un vértice y el valor almacenado en esa localización indica la existencia o no de un arco. Si la celda $MA[i, j] = 1$ indica la existencia de un arco, pero si $MA[i, i] = 0$ indica que no hay un arco que una el vértice i-ésimo con el j-ésimo.
  En el caso de que el arco tenga algún peso, se cambia el valor 1 por el valor del peso.

  Ejemplo 6.14. Matriz de adyacencia

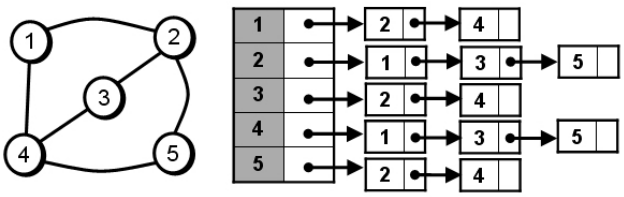


M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

  Fuente: https://es.wikipedia.org/wiki/Grafo □

- **Lista de adyacencia (LA):** Se utiliza un vector de tamaño n (un elemento por cada vértice) donde $LA[i]$ almacena la referencia a **una lista** de los vértices adyacentes a i. También se puede usar un diccionario para los vértices adyacentes (p.e. `map<int, list<int>>`) si los vértices a almacenar son enteros.
  En una red esta lista almacenará también la longitud de la arista que va desde i al vértice adyacente.
  En el caso de que el arco tenga algún peso, se añadirá el valor del peso a cada elemento de la lista.

  Ejemplo 6.15. Listas de adyacencias para un grafo que utiliza una lista simplemente enlazada.



Fuente: https://es.wikipedia.org/wiki/Grafo □

Ejercicios

6.1 Árboles

★☆☆

Implementar TDA Árbol

Usa la definición del TDA Árbol de la página 88.
Implementa este TDA usando el tipo **list** de **Python** usando la Alternativa 2 y con referencia al padre.

6.2 TDA Árbol Binario

★☆☆ ✍

Implementar TDA Árbol Binario

Define el TDA Binary Tree según lo indicado en la sección 6. Implementa este TDA usando la Alternativa 2.
Implementa no menos de 5 métodos basados en recorridos. Se recomiendan para el aprendizaje: mostrar (dependiendo de cierto parámetro de tipo enumerado hacer recorrido prefijo, infijo y postfijo), copiar (implementando los métodos mágicos correspondientes), buscar un valor, calcular la altura de un árbol, calcular el número de hojas.

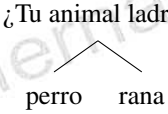
6.3 Adivinador

★★☆

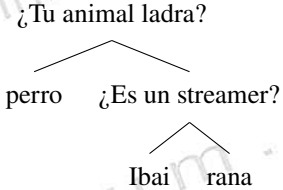
Implementar TDA Binary Tree

Usa el TDA implementado en el apartado anterior para desarrollar una versión reducida de la aplicación https://es.akinator.com/ (puedes jugar online, ¡hazlo!).

El programa aprende el personaje imaginado por el usuario de la siguiente manera. En la primera iteración al programa afirma que piensas en un personaje (p.e. una rana), el que tú consideres lo suficientemente raro como para que no acierte. Obviamente el usuario afirmará que se equivocó con lo que el programa pedirá al usuario que le indique cuál es el personaje en le que estaba pensando (p.e. un perro) y qué pregunta permite discernir entre los personajes (p.e. ¿tu animal ladra?). Con estos 3 datos construye el siguiente árbol binario.



En las siguiente iteraciones, se limita a recorrer cada una de las ramas en función de la respuesta que le de el usuario a cada una de las preguntas hasta llegar a un nodo hoja. Si acierta, el programa ganó; pero si no acierta pedirá una nueva pregunta que le permita discernir entre el personaje de ese nodo hoja con el personaje que esté pensando el usuario.
Imagina ahora que el usuario piensa en el streamer Ibai. Partiendo del árbol anterior el programa preguntará si el personaje ladra, diremos que no y pedirá una pregunta. Podría generar el siguiente árbol:



Observa que todos los nodos internos son preguntas y todos los nodos hojas son los personajes: NO es necesario ningún atributo adicional para saber si es un nodo pregunta o un nodo personaje.

Haz un programa que construya un árbol de adivinación según se ha indicado anteriormente. Puedes partir de un árbol ya creado internamente que conste de varios personajes y preguntas. ¡¡Reta a tus conocidos demostrando que construyes programas "inteligentes"!!

6.4 Árboles Binarios de Búsqueda

★★☆

Implementar TDA Binary Search Tree

Define el TDA Binary Search Tree implementando todo lo indicado en el apartado 6. .

Responde e implementa la solución a las siguientes cuestiones:

- ¿Cómo se puede transformar una lista en un árbol binario de búsqueda?
- ¿Qué recorrido debería de hacerse en un árbol binario de búsqueda para obtener todos los valores ordenados en orden creciente?
- La respuesta a ambas preguntas nos da un algoritmo de ordenación de listas:
  1. Dada una lista (desordenada) se construye un árbol binario de búsqueda con sus elementos (de acuerdo a la solución a la primera pregunta).
  2. Retornar en otra lista el resultado del recorrido indicado (de acuerdo a la solución de la segunda pregunta).
  3. Si se requiere una ordenación decreciente basta invertir la lista obtenida en el punto anterior.

¿Serías capaz de añadir tu paquete de métodos de ordenación con esta nueva técnica?

6.5 Heap Min

★☆☆

Implementar TDA Heap Min

Define el TDA Heap implementando lo operadores de creación, inclusión y extracción explicados en la página 94.

Como representación usa una estructura con tres atributos: un **booleano** para determinar si el heap es min o max, un **entero** para el tamaño del Heap (número de nodos que hay en el heap) y **un contenedor lineal** con los elementos del heap. Alternativamente si el contenedor es una lista, no se necesita el entero pues vendrá dado por `len(lista)`.

Observa que los valores de los nodos deberán de tener definido algún tipo de orden, lo que se traduce en sobreescribir correctamente los métodos mágicos asociados a las operaciones de comparación.

6.6 Árbol de partición

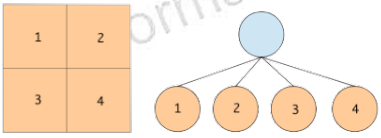
★☆☆

Implementación de un árbol de partición

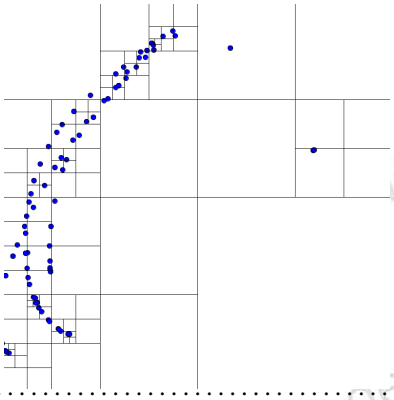
Un problema que se plantea en Geometría Computacional es buscar representaciones eficientes de la partición del espacio. Algunas de ellas se basan en usar árboles.

Se puede usar una Binary space partitioning o Partición Binaria del Espacio (BSP), que consiste en subdividir recursivamente el espacio en elementos convexos usando hiperplanos. Esta subdivisión da lugar a una representación jerárquica de la escena usando un árbol binario. Esta representación permite hacer dibujos de una escena de forma correcta y rápida en programas de diseño o animación: primero se dibuja lo más cercano y después lo más lejano.

En este ejercicio vas a hacer una partición cuaternaria del plano. Para ello vas a usar una árbol 4-ario. Se considera que un nodo representa a la región rectangular del plano con vértices opuestos. Cada nodo consta de 4 hijos. El valor de cada hijo es una referencia a otro nodo que representa uno de los cuadrantes del nodo padre.



Se parte de un escenario o plano en el que se sitúan varios objetos, cada uno situado en una posición del plano. Al algoritmo que calcula el árbol de partición se le suministra los vértices opuestos que definen una región y la lista de objetos que están en dicha región. Inicialmente, la región es el plano completo del escenario y la lista completa de objetos. En el caso de que un cuadrante contenga al menos un número K de objetos (que también se pasa por parámetro), se invoca de forma recursiva para generar el árbol de partición para ese cuadrante. Al final, cada nodo hoja contendrá entre 0 y K-1 objetos.



En la figura se ve que en el cuadrante 4 de la escena no hay objetos, por lo que no se hace partición. Sin embargo en el cuadrante 2 de la escena se detectan más de 3 objetos por lo que se hace una partición en 4 cuadrantes. De estos cuatro: los 3 primeros cuadrantes no se particionan porque ninguno de ellos tiene más de 3 objetos, pero su cuarto cuadrante sí porque tiene más de 3 objetos (están superpuestos). Lectura análoga hay que hacer de los cuadrantes 1 y 3 de la escena.

6.7 Árboles de Codificación

★★☆

Codifica El Quijote

Dado un texto podemos crear un árbol que permite codificar el texto para que ocupe menos espacio. Para ello se necesita conocer la frecuencia de aparición de cada carácter del texto (incluidos los espacios).

Entonces se construye un árbol de codificación partiendo de n-arboles con un solo nodo, uno por cada carácter del texto junto con su frecuencia, para terminar construyendo un árbol binario cuyas hojas son los nodos de partida. Creados los n-árboles iniciales, se toman los dos árboles con **menor** frecuencia y se unen creando un nuevo árbol. La etiqueta del nodo raíz del nuevo árbol contendrá la suma de las frecuencias de los dos nodos árboles que se unen y tendrá dos hijos, formados por los dos árboles que se han unido. También se etiquetan las dos ramas del nuevo árbol. Por comodidad se suele considerar 0 el de la izquierda y 1 el de la derecha. Los dos árboles que se han fusionado ya no participarán en el proceso. Entonces, se vuelve a seleccionar otros dos árboles con cuyos nodos raíces tengan **menor** frecuencia y se repite el proceso una y otra vez hasta que solo quede un árbol binario.

Para obtener el código de un carácter, se comienza un código vacío y se iniciará un recorrido del árbol desde la hoja donde está el símbolo hasta llegar al nodo raíz del árbol. Durante el recorrido se irá añadiendo al código un 0 o un 1, dependiendo de la rama. Tras llegar a la raíz, se invertirá el código. P.e. si en el recorrido ascendente para la letra 'A' el código fue '0111000', el código que se asignará a 'A' será '0001110'.

Para obtener el carácter de un código (descodificación) se comienza el recorrido desde la raíz del árbol y se visita la rama de la izquierda si el primer elemento código era un 0, o se visitará la rama de la derecha si el primer elemento del código era un 1, previa eliminación del primer elemento del código. Entonces, se repite el proceso para el nodo visitado el código restante. Por ejemplo, si el código es '010' se visitarán las ramas: izquierda, derecha e izquierda. El carácter del nodo hoja será el carácter asociado al código.

Lo normal es que tanto la codificación como la descodificación use una tabla y descarte el árbol. Al usar tablas, se suelen ordenar los códigos de acuerdo a su orden lexicográfico.

Codifica y descodifica el libro completo de `El ingenioso hidalgo don Quijote de la Mancha`. https://fegalaz.usc.es/~gamallo/aulas/lingcomputacional/corpus/quijote-es.txt

¿Cuánto ocupa el fichero original ? ¿cuánto ocupa cuando lo codificas mediante árboles binarios?

6.8 Caminos de coste mínimo

★★☆

Algoritmo de Dijkstra, Coste Uniforme y A*

Implementa el **algoritmo de Dijkstra** que permite obtener los caminos de coste mínimo de un nodo a todos los demás.

Este algoritmo considera que se usa una cola de prioridad (ver apartado 6) o un Heap-Min (ejercicio 6.5). En definitiva, se tiene una lista que ordena los elementos que contiene de forma creciente usando como criterio un peso positivo asociado a cada elemento. El peso se puede interpretar como un coste o una distancia.

1. Seleccionar un nodo origen s.
2. Inicialización:
   - Para cada nodo v del grafo hacer
     - $distancia[v] = \infty$. En principio es imposible llegar a cualquier nodo.
     - $padre[v] = None$. Indica desde dónde se generó.
     - $visitado[v] = False$. Indica si un nodo fue visitado.
   - $distancia[s] = 0$

- $cola.add(s, distancia[s])$

3. Mientras que la cola no esté vacía hacer:
   a) $u = cola.pop()$. Extrae el elemento de menor coste.
   b) $visitado[u] = True$
   c) Para todo v en $Adyacentes(u)$ hacer:
      Si no $visitado[v]$ y $distancia[v] > distancia[u] + peso(u, v)$ hacer
      1) $distancia[v] = distancia[u] + peso(u, v)$
      2) $padre[v] = u$
      3) $cola.add(v, distancia[v])$

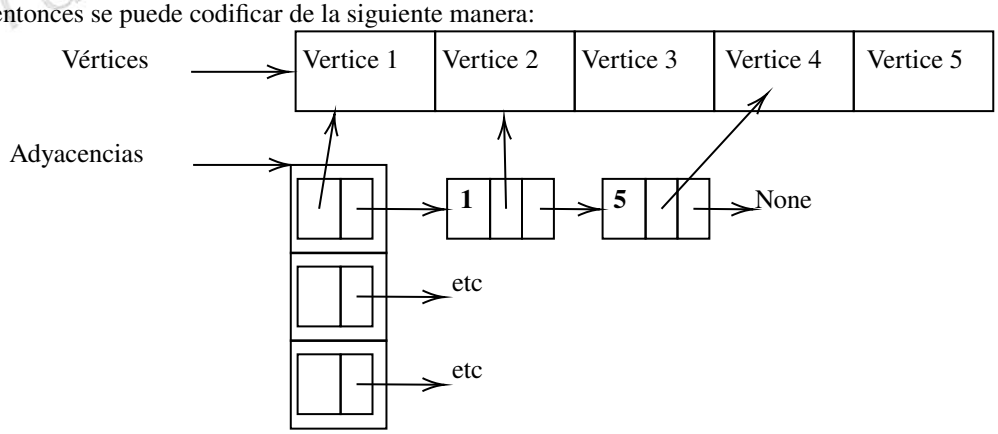
Si se recorre la lista $padre[v]$ en orden inverso se obtendrá el camino de menor coste que conecta el nodo inicial s con el nodo v.

Para la implementación usa como representación del grafo una **lista de adyacencia**. Observa que se usa una lista simplemente enlazada y que ya la implementaste en el Ejercicio 4.3. Solo tendrás que modificar la estructura del nodo para que tenga ahora 3 campos: la referencia al objeto, el coste y el siguiente.

- La referencia almacena el lugar donde esté almacenado el vértice del grafo (que puede ser cualquier tipo de objeto).
- El coste indica el coste del arco que hay desde el vértice del nodo cabecera hasta el vértice del nodo.
- El siguiente almacena una referencia al nodo siguiente.

Además, conviene tener una lista de los objetos que forman los vértices del grafo a efectos de que la lista de adyacencia no guarde copias de los objetos, sino una referencia directa a ellos.

Por ejemplo, para el grafo de la página 99, si se considera que el coste del vértice 1 al vértice 2 es 1 y del vértice 1 al vértice 4 es 5, entonces se puede codificar de la siguiente manera:



Como alternativa a las listas se pueden usar diccionarios:

```
vertices: Dict = {id1: valor_vertice_1,
                  id2: valor_vertice_2,
                  id3: valor_vertice_3,
                  ...
                  }

adyacencias: Dict = {id1: lista_adyacentes_del_vertice_id1, # Lista simplemente enlazada
                     id2: lista_adyacentes_del_vertice_id2,
                     id3: lista_adyacentes_del_vertice_id3,
                     ...
                     }
```

Una modificación del algoritmo es el **algoritmo de Coste Uniforme** (sección 5) que busca el camino de menor coste entre el nodo inicial y uno nodo destino. En este caso, en el paso 3.b) se comprobará si el nodo u es un nodo destino. Si lo fuera, entonces finaliza el algoritmo. Es muy fácil obtener el algoritmo de Coste Uniforme usando el código del Ejercicio 5.3, que ya has implementado.

El **algoritmo A*** (sección 5, ejercicio 5.4), puede usar este algoritmo para calcular el valor heurístico, $h(n)$, de cada nodo al nodo origen. Notar que para el A*, su punto destino es el origen de Dijkstra.

Un ejemplo de uso lo encontramos en problemas de transporte para encontrar el camino más corto entre un origen y un destino (p.e. cuando Google Maps te indica la cantidad de Kms que vas a recorre entre dos puntos geográficos). En videojuegos se usa para que los personajes no controlados por el jugador encuentren el camino más corto desde donde se encuentran hasta donde tú los mandes o para buscar el camino más corto para ¡matarte!.

6.9 Árboles Generadores

★★☆

Algoritmo de Prim

Implementa el algoritmo de Prim que permite obtener el grafo generador de menor costo de un grafo.

1. Elegir un nodo origen.
2. Agregar el nodo a la lista de visitados.
3. Agregar sus adyacentes a la lista L, que es una lista ordenada de los nodos (por costes).
4. Mientras que la lista L no esté vacía:
   a) Extraer el mejor nodo de L
   b) Si el nodo no está en la lista de visitados
      1) Agregar a la lista ordenada L los adyacentes del nodo que no hayan sido visitados.
      2) Agregar el nodo al árbol solución.
5. Retornar el árbol solución.

Un ejemplo de uso es el siguiente. Imagina que una empresa eléctrica tiene que hacer un nuevo tendido de cables que debe de pasar por varias localidades empezando desde una localidad que es la que recibe el suministro. En el modelo (grafo), toda localidad se conecta con otra localidad siempre que sea viable hacer un tendido entre las dos (lo que conllevará un coste positivo). El resultado final es un grafo no dirigido pesado. El algoritmo de Prim obtiene un árbol que contiene a todas las localidades, cada una se conectará con otra siempre que sea la conexión más económica de entre todas las posibles conexiones.