Trabajos de Programación con Python

Trabajos de análisis de datos con Python en el ámbito de los negocios y retos de programación, con soluciones

Alfredo Sánchez Alberca
asalber@ceu.es
https://aprendeconalf.es

Tabla de contenidos

Prefacio

¡Bienvenido a los Trabajos de Programación con Python!

Este libro contiene una colección de trabajos resueltos de análisis de datos con Python en el ámbito de la inteligencia de los negocios. Cada trabajo plantea un problema real a partir de un conjunto de datos abiertos y una lista de requisitos que hay que resolver, primero con las estructuras de datos básicas de Python y después con la librería Pandas y los gráficos de Matplotlib.

Incluye además una colección de retos de programación: problemas cortos pero con un grado de complejidad mayor que los de la sección de ejercicios, pensados para seguir avanzando en el dominio de Python. Es recomendable haber intentado todos los problemas de los Ejercicios de Programación con Python antes de intentar los retos.

Cada trabajo y cada reto incluye su solución en un desplegable, junto con un enlace para abrirla y ejecutarla en Google Colab.

Este libro es un complemento ideal del Manual de Python y de los Ejercicios de Programación con Python, donde se explican los conceptos necesarios para resolver los trabajos y los retos.

Trabajos de análisis de datos

1. Contratos menores de Madrid
2. Emisiones de Madrid
3. Deuda pública
4. Airbnb Madrid

Retos de programación

5. Ajedrez
6. Laberinto
7. Estrellas regulares

Datos

Los conjuntos de datos utilizados en los trabajos están en la carpeta `datos` del repositorio.

Licencia

Parte I.

Trabajos de análisis de datos

1. Contratos menores de Madrid

El objetivo de este trabajo es hacer un análisis de los contratos menores del Ayuntamiento de Madrid desde 2015 para hacer una comparativa por años y ver qué empresas han resultado más beneficiadas.

1.1. Conjunto de datos

Datos abiertos del Ayuntamiento de Madrid: Contratos menores

Deben usarse los datos desde 2015.

1.2. Requisitos obligatorios

1. Crear una función que reciba una empresa y una lista de años y devuelva un diccionario con el número de contratos y el total facturado por la empresa esos años.
2. Crear una función que reciba una sección y una lista de años y devuelva un diccionario con el número de contratos y el total facturado a la sección esos años.
3. Crear una función que reciba una empresa, una sección y una lista de años y devuelva un diccionario con el número de contratos y el total facturado por la empresa a la sección esos años.
4. Crear una función que reciba una rango de años y un número entero n e imprima la lista de las n empresas que más han facturado durante esos años ordenadas de mayor a menor facturación y genere un gráfico con esa información.
5. Crear una función reciba una rango de años y un número entero n y genere un gráfico con la evolución anual del total facturado por las n empresas que más han facturado.
6. Crear una función reciba una rango de años y genere un gráfico con la evolución anual del total facturado a las secciones.

Solución

1.2.1. Preprocesamiento de datos

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import datetime as dt
from urllib.error import HTTPError

try:
    df = pd.read_csv(
 ↪  'http://aprendeconalf.es/python/trabajos/datos/contratos-menores-madrid.csv'
 ↪  , sep=';')
except HTTPError:
    print('La url no existe')
else:
    # Preprocesamiento de datos
    # # Ordenar el dataframe por años
    df = df.sort_values(['AÑO'])
    print(df)
```

1.2.2. Requisito 1

Crear una función que reciba una empresa y una lista de años y devuelva un diccionario con el número de contratos y el total facturado por la empresa esos años.

```
def empresa(df, nif):
    """
    Función que devuelve el nombre de la empresa con el nif dado.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de los contratos menores.
        - nif: Es una cadena con el NIF del contratista.
    Devuelve:
        El nombre del contratista.
    """
    return df[df['NIF']==nif]['CONTRATISTA'].iloc[0]
```

```python
def facturacion_empresa_años(df, nif, años):
    """
    Función que devuelve un diccionario con el número de contratos y
 ↪  el total facturado por la empresa durante una lista de años.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de los contratos menores.
        - nif: Es una cadena con el NIF del contratista.
        - años: Es una lista con los años.
    Devuelve:
        Un diccionario con el número de contratos y el total
 ↪  facturado por la empresa con el nif dado durante los años
 ↪  indicados.
    """
    # Filtro de la empresa y los años
    df1 = df[(df['NIF'] == nif) & (df['AÑO'].isin(años))]
    return {'Número de contratos': df1['IMPORTE'].count(), 'Total
     ↪  facturado':df1['IMPORTE'].sum()}

# Ejemplo
print(facturacion_empresa_años(df, 'B28380582', [2018, 2019]))
```

1.2.3. Requisito 2

Crear una función que reciba una sección y una lista de años y devuelva un diccionario con el número de contratos y el total facturado a la sección esos años.

```python
def gasto_seccion_años(df, seccion, años):
    """
    Función que devuelve un diccionario con el número de contratos y el total gastado por una sección del ayuntamiento durante una lista de años.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos de los contratos menores.
        - seccion: Es una cadena con la sección del ayuntamiento que ordena el gasto.
        - años: Es una lista con los años.
    Devuelve:
        Un diccionario con el número de contratos y el total facturado por la empresa con el nif dado durante los años indicados.
    """
    # Filtro de la sección y los años
    df1 = df[(df['SECCION'] == seccion) & (df['AÑO'].isin(años))]
    return {'Número de contratos': df1['IMPORTE'].count(), 'Total gastado':df1['IMPORTE'].sum()}

# Ejemplo
print(gasto_seccion_años(df, 'EMPRESA MUNICIPAL DE TRANSPORTES S.A.', [2018, 2019]))
```

1.2.4. Requisito 3

Crear una función que reciba una empresa, una sección y una lista de años y devuelva un diccionario con el número de contratos y el total facturado por la empresa a la sección esos años.

```python
def facturacion_empresa_seccion_años(df, nif, seccion, años):
    """
    Función que devuelve un diccionario con el número de contratos y
 ↪  el total facturado por la empresa a una seccion durante una
 ↪  lista de años.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de los contratos menores.
        - nif: Es una cadena con el NIF del contratista.
        - seccion: Es una cadena con la sección del ayuntamiento que
 ↪  ordena el gasto.
        - años: Es una lista con los años.
    Devuelve:
        Un diccionario con el número de contratos y el total
 ↪  facturado por la empresa con el nif dado a la sección dada
 ↪  durante los años indicados.
    """
    # Filtro de la empresa, la sección y los años
    df1 = df[(df['NIF'] == nif) & (df['SECCION'] == seccion) &
 ↪  (df['AÑO'].isin(años))]
    return {'Número de contratos': df1['IMPORTE'].count(), 'Total
     ↪  facturado':df1['IMPORTE'].sum()}

# Ejemplo
print(facturacion_empresa_seccion_años(df, 'B80176936', 'EMPRESA
 ↪  MUNICIPAL DE TRANSPORTES S.A.', [2018, 2019]))
```

1.2.5. Requisito 4

Crear una función que reciba una rango de años y un número entero n e imprima la lista de las n empresas que más han facturado durante esos años ordenadas de mayor a menor facturación y genere un gráfico con esa información.

```python
def empresas_mayor_facturacion(df, años, n = 10):
    """
    Función que imprime una tabla con las n empresas que más han facturado durante los años indicados y genera un gráfico con esa información.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos de los contratos menores.
        - años: Es una lista con los años.
        - n: Es el número de empresas en la tabla (10 por defecto)
    """
    # Filtro de los años
    df1 = df[df['AÑO'].isin(años)]
    # Agrupar por empresas
    df1 = df1.groupby('NIF')['IMPORTE'].sum()
    # Ordenar descendentemente por importe
    df1 = df1.sort_values(ascending=False)
    # Obtener los n primeros y establecer el nombre de la empresa como índice.
    df1 = df1[:n].rename(lambda x: empresa(df, x))
    # Imprimirlos
    print(df1)
    # Inicializamos el gráfico
    fig, ax = plt.subplots(figsize=(6, 8))
    # Dibujar el diagrama de barras
    df1.plot(kind = 'bar', ax = ax)
    # Título del gráfico
    ax.set_title(str(n) + ' empresas con mayor facturación (años ' + str(años) + ')')
    # Título del eje x
    ax.set_ylabel('Importe total en €')
    # Ajustar los márgenes del gráfico
    plt.tight_layout()
    plt.gcf().subplots_adjust(bottom=0.6)
    # Guardamos el gráfico
    plt.show()
    return

# Ejemplo
empresas_mayor_facturacion(df, [2018,2019], n = 10)
```

1.2.6. Requisito 5

Crear una función reciba una rango de años y un número entero n y genere un gráfico con la evolución anual del total facturado por las n empresas que más han facturado.

```
def evolucion_empresas_mayor_facturacion(df, inicio, fin, n = 10):
    """
    Función que crea un gráfico con la evolución anual del total
 ↪  facturado por las n empresas con mayor facturación en un rango
 ↪  de años dado.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - inicio: Es un entero con el año inicial.
        - fin: Es un entero con el año final.
        - n: Es el número de empresas en el gráfico (10 por defecto)
    """
     # Filtro de los años
    df1 = df[(df['AÑO'] >= inicio) & (df['AÑO'] <= fin)]
    # Agrupar por empresas
    df2 = df1.groupby('NIF')['IMPORTE'].sum()
    # Ordenar descendentemente por importe
    df2 = df2.sort_values(ascending = False)
    # Obtener la lista de los nifs de las n empresas con mayor
     ↪  facturación
    nifs = df2[:n].index
    # Filtrar las n empresas con mayor facturación en el rango de
     ↪  años
    df1 = df1[df1['NIF'].isin(list(nifs))]
    # Agrupar por años y empresas
    df1 = df1.groupby(['AÑO', 'NIF'])['IMPORTE'].sum()
    # Establecer como índice el nombre de la empresa
    df1 = df1.rename(lambda x: empresa(df, x), level = 1)
    # Inicializamos el gráfico
    fig, ax = plt.subplots(figsize=(10, 4))
    # Desagrupamos por empresas y generamos el gráfico de líneas
    df1.unstack().plot(legend = True, ax = ax)
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Título del gráfico
    ax.set_title('Evolución facturacion ' + str(n) + ' empresas con
 ↪  mayor facturación')
    # Título del eje x
    ax.set_ylabel('Importe anual en €')
    # Establecer las marcas del eje x
    ax.set_xticks(range(inicio, fin+1))
    # Ajustar los márgenes del gráfico
    # plt.tight_layout()
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
evolucion_empresas_mayor_facturacion(df, 2017, 2019)
```

1.2.7. Requisito 6

Crear una función reciba una rango de años y genere un gráfico con la evolución anual del total facturado a las secciones.

```
def evolucion_facturacion_secciones(df, inicio, fin):
    """
    Función que crea un gráfico con la evolución de la facturacion
 ↪  anual por secciones en un rango de años dado.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - inicio: Es un entero con el año inicial.
        - fin: Es un entero con el año final.
    """
     # Filtro de los años
    df1 = df[(df['AÑO'] >= inicio) & (df['AÑO'] <= fin)]
    # Agrupar por años y secciones
    df1 = df1.groupby(['AÑO', 'SECCION'])['IMPORTE'].sum()
    # Inicializamos el gráfico
    fig, ax = plt.subplots(figsize=(10, 4))
    # Desagrupamos por secciones y generamos el gráfico de líneas
    df1.unstack().plot(legend = True, ax = ax)
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Título del gráfico
    ax.set_title('Evolución facturacion por secciones')
    # Título del eje x
    ax.set_ylabel('Importe anual en €')
    # Establecer las marcas del eje x
    ax.set_xticks(range(inicio, fin+1))
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
evolucion_facturacion_secciones(df, 2017, 2019)
```

Abrir la solución en Google Colab

2. Emisiones de Madrid

El objetivo de este trabajo es comprobar si las restricciones de tráfico establecidas en Madrid Central han servido para reducir significativamente las emisiones de gases contaminantes.

2.1. Conjunto de datos

Datos abiertos del Ayuntamiento de Madrid: Calidad del aire. Datos diarios años 2001 a 2019.

2.2. Requisitos

1. Crear una función que reciba una estación de medición y una magnitud y devuelva una lista con todas las mediciones de la magnitud en la estación.
2. Crear una función que reciba un mes y una estación de medición y devuelva un diccionario con las medias de las magnitudes medidas por la estación durante ese mes.
3. Crear una función que reciba un mes y una magnitud y devuelva un diccionario con las medias de las estaciones de medición de la magnitud durante ese mes.
4. Crear una función que reciba un rango de fechas y una estación de medición y genere un gráfico con la evolución diaria de las magnitudes de esa estación en las fechas indicadas.
5. Crear una función que reciba un rango de fechas y una magnitud y genere un gráfico con la evolución diaria de la magnitud para cada estación de medición en las fechas indicadas.
6. Crear una función que reciba una magnitud y genere un gráfico con las medias mensuales para cada estación de medición.
7. Crear una función que reciba un mes y una magnitud y devuelva un diccionario con las medias de la magnitud dentro de Madrid Central y fuera de ella.
8. Crear una función que reciba una magnitud y genere un gráfico con las medias mensuales dentro de Madrid Central y fuera de ella.

Solución

2.2.1. Preprocesamiento de datos

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import datetime as dt

# Códigos de las magnitudes contaminantes medidas
magnitudes = {
    '01':'Dióxido de Azufre',
    '06':'Monóxido de Carbono',
    '07':'Monóxido de Nitrógeno',
    '08':'Dióxido de Nitrógeno',
    '09':'Partículas < 2.5  m',
    '10':'Partículas < 10  m',
    '12':'Óxidos de Nitrógeno',
    '14':'Ozono',
    '20':'Tolueno',
    '30':'Benceno',
    '35':'Etilbenceno',
    '37':'Metaxileno',
    '38':'Paraxileno',
    '39':'Ortoxileno',
    '42':'Hidrocarburos totales(hexano)',
    '43':'Metano',
    '44':'Hidrocarburosno metánicos (hexano)'
}

# Códigos de las estaciones de medición.
estaciones = {
    '001':'Pº. Recoletos',
    '002':'Glta. de Carlos V',
    '035':'Pza. del Carmen',
    '004':'Pza. de España',
    '039':'Barrio del Pilar',
    '006':'Pza. Dr. Marañón',
    '007':'Pza. M. de Salamanca',
    '008':'Escuelas Aguirre',
    '009':'Pza. Luca de Tena',
    '038':'Cuatro Caminos',
    '011':'Av. Ramón y Cajal',
    '012':'Pza. Manuel Becerra',
    '040':'Vallecas',
    '014':'Pza. Fdez. Ladreda',
    '015':'Pza. Castilla',
    '016':'Arturo Soria',
    '017':'Villaverde Alto',
    '018':'Calle Farolillo',
    '019':'Huerta Castañeda',
    '036':'Moratalaz',
    '021':'Pza. Cristo Rey',
    '022':'Pº. Pontones',
    '023':'Final C/ Alcalá',
    '024':'Casa de Campo',
    '025':'Santa Eugenia',
    '026':'Urb. Embajada (Barajas)',
    '027':'Barajas',
    '047':'Méndez Álvaro',
```

2.2.2. Requisito 1

Crear una función que reciba una estación de medición y una magnitud y devuelva una lista con todas las mediciones de la magnitud en la estación.

```
def estacion_magnitud(df, estacion, magnitud):
    """
    Función que devuelve la lista de valores de una estación y
 ↪  magnitud.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - estacion: Es una cadena con el código de la estación de
 ↪  medición.
        - magnitud: Es una cadena el código de la magnitud medida.
    Devuelve:
        Una lista con los valores de la magnitud medidos en la
 ↪  estación indicada.
    """
    # Filtro de la estación y la magnitud
    df1 = df[(df['ESTACION'] == estacion) & (df['MAGNITUD'] ==
 ↪  magnitud)]
    return list(df1['VALOR'])

# Ejemplo
print(estacion_magnitud(df, '050', '12'))
```

2.2.3. Requisito 2

Crear una función que reciba un mes y una estación de medición y devuelva un diccionario con las medias de las magnitudes medidas por la estación durante ese mes.

```python
def medias_mes_estacion(df, mes, estacion):
    """
    Función que devuelve la media de las magnitudes medidas en una
 ↪  estación en un mes concreto.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - mes: Es una cadena de dos caractares con el número de mes
 ↪  en formato mm.
        - estacion: Es una cadena con el código de la estación de
 ↪  medición.
    Devuelve:
        Un diccionario cuyos pares tienen como clave las magnitudes
 ↪  y como valores las medias del mes en la estación indicada.
    """
    # Filtro de la estación y el mes
    df1 = df[(df['ESTACION'] == estacion) & (df['MES'] == mes)]
    # Agrupación por magnitud y cálculo de la media
    return {magnitudes[k]:np.mean(v) for k, v in
     ↪  df1.groupby('MAGNITUD')['VALOR']}

# Ejemplo
print(medias_mes_estacion(df, '03', '050'))
```

2.2.4. Requisito 3

Crear una función que reciba un mes y una magnitud y devuelva un diccionario con las medias de las estaciones de medición de la magnitud durante ese mes.

```python
def medias_mes_magnitud(df, mes, magnitud):
    """
    Función que devuelve la media de una magnitud en mes concreto en
 ↪  cada estación de medición.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - mes: Es una cadena de dos caractares con el número de mes
 ↪  en formato mm.
        - magnitud: Es una cadena con el código de la magnitud
 ↪  medida.
    Devuelve:
        Un diccionario cuyos pares tienen como clave las estaciones
 ↪  y como valores las medias del mes de la magnitud indicada.
    """
    # Filtro de la magnitud y el mes
    df1 = df[(df['MAGNITUD'] == magnitud) & (df['MES'] == mes)]
    # Agrupación por estación y cálculo de la media
    return {estaciones[k]:np.mean(v) for k, v in
     ↪  df1.groupby('ESTACION')['VALOR']}

# Ejemplo
print(medias_mes_magnitud(df, '12', '12'))
```

2.2.5. Requisito 4

Crear una función que reciba un rango de fechas y una estación de medición y genere un gráfico con la evolución diaria de las magnitudes de esa estación en las fechas indicadas.

```python
def evolucion_estacion(df, estacion, inicio, fin):
    """
    Función que crea un gráfico con la evolución de las magnitudes
 ↪  de una estación de medición en un rango de fechas.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - estacion: Es una cadena con el código de la estacion de
 ↪  medición.
        - inicio: Es una cadena con la fecha inicial en formato
 ↪  dd-mm-aaaa.
        - fin: Es una cadena con la fecha final en formato
 ↪  dd-mm-aaaa.
    """
    # Añadir columna con el nombre de las magnitudes
    df['NOMBRE MAGNITUD'] = df['MAGNITUD'].apply(lambda x:
 ↪  magnitudes[x])
    # Filtro de la estación y rango de fechas
    df1 = df[(df['ESTACION'] == estacion) & (df['FECHA'] >= inicio)
 ↪  & (df['FECHA'] <= fin)]
    # Establecemos la columna fecha como índice del DataFrame
    df1.set_index('FECHA', inplace = True)
    # Inicializamos el gráfico
    fig, ax = plt.subplots()
    # Agrupamos los datos por magnitud y generamos el gráfico de
     ↪  líneas
    df1.groupby('NOMBRE MAGNITUD')['VALOR'].plot(legend = True)
    # Reducimos el eje x un 30% para que quepa la leyenda
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.7, box.height])
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
evolucion_estacion(df, '017', '2018-03-01', '2018-06-30')
```

2.2.6. Requisito 5

Crear una función que reciba un rango de fechas y una magnitud y genere un gráfico con la evolución diaria de la magnitud para cada estación de medición en las fechas indicadas.

```python
def evolucion_magnitud(df, magnitud, inicio, fin):
    """
    Función que crea un gráfico con la evolución de las mediciones
 ↪  de una magnitud en cada estación en un rango de fechas.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - magnitud: Es una cadena con el código de la magnitud
 ↪  medida.
        - inicio: Es una cadena con la fecha inicial en formato
 ↪  dd-mm-aaaa.
        - fin: Es una cadena con la fecha final en formato
 ↪  dd-mm-aaaa.
    """
    # Añadir columna con el nombre de las estaciones
    df['NOMBRE ESTACION'] = df['ESTACION'].apply(lambda x:
 ↪  estaciones[x])
    # Filtro de la magnitud y el rango de fechas
    df1 = df[(df['MAGNITUD'] == magnitud) & (df['FECHA'] >= inicio)
 ↪  & (df['FECHA'] <= fin)]
    # Establecemos la columna fecha como índice del DataFrame
    df1.set_index('FECHA', inplace = True)
    # Inicializamos el gráfico
    fig, ax = plt.subplots()
    # Agrupamos los datos por estación y generamos el gráfico de
     ↪  líneas.
    df1.groupby('NOMBRE ESTACION')['VALOR'].plot(legend = True)
    # Reducimos el eje x un 30% para que quepa la leyenda
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.7, box.height])
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Guardamos el gráfico
    plt.show()
    return

# Ejemplo
evolucion_magnitud(df, '12', '2018-03-01', '2018-06-30')
```

2.2.7. Requisito 6

Crear una función que reciba una magnitud y genere un gráfico con las medias mensuales para cada estación de medición.

```
def evolucion_medias_magnitud(df, magnitud):
    """
    Función que crea un gráfico con la evolución de las medias de
 ↪  una magnitud en cada estación.
    Parámetros:
        - df: Es un DataFrame con la información de la base de datos
 ↪  de emisiones.
        - magnitud: Es una cadena con el código de la magnitud
 ↪  medida.
    """
    # Añadir columna con el nombre de las estaciones
    df['NOMBRE ESTACION'] = df['ESTACION'].apply(lambda x:
 ↪  estaciones[x])
    # Filtro de la magnitud
    df1 = df[df['MAGNITUD'] == magnitud]
    # Establecemos la columna fecha como índice del DataFrame
    df1.set_index('FECHA', inplace = True)
    # Inicializamos el gráfico
    fig, ax = plt.subplots()
    # Agrupamos los datos por estación y generamos el gráfico de
     ↪  líneas.
    df1.groupby(['ANO','MES','NOMBRE
 ↪  ESTACION']).mean().unstack()['VALOR'].plot(ax = ax, legend =
 ↪  True)
    # Reducimos el eje x un 30% para que quepa la leyenda
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.7, box.height])
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Guardamos el gráfico
    plt.show()
    return

# Ejemplo
evolucion_medias_magnitud(df, '12')
```

Abrir la solución en Google Colab

3. Deuda Pública

El objetivo de este trabajo es hacer un análisis del endeudamiento público por países.

3.1. Conjunto de datos

Datos del banco mundial: Deuda pública países y PIB países.

3.2. Requisitos

Sin usar la librería Pandas:

1. Crear una función que reciba un país y un tipo de deuda y devuelva un diccionario con todos los periodos y la cantidad de deuda en esos periodos de ese país y tipo de deuda.
2. Crear una función que reciba un país y un tipo de deuda y devuelva un diccionario con el mínimo y el máximo de deuda de ese tipo para ese país.
3. Crear una función que reciba un país y un año, y devuelva un diccionario con la deuda interna y la deuda externa de ese país en ese año.
4. Crear una función que reciba un país y un año, y devuelva un diccionario con la deuda en moneda local y la deuda en moneda extranjera de ese país en ese año.

Usando Pandas:

5. Preprocesar el fichero de deuda pública para obtener un data frame con el país, el tipo de deuda, la fecha y la cantidad de deuda.
6. Crear una función que reciba un país y una fecha y devuelva un diccionario con la deuda total interna, externa, en moneda local, en moneda extranjera, a corto plazo y a largo plazo, de ese país en esa fecha.
7. Crear una función que reciba un tipo de deuda y una fecha, y devuelva un diccionario con la deuda de ese tipo de todos los países en esa fecha.
8. Crear una función que reciba un país y una fecha y dibuje un diagrama de sectores con la deuda interna y la deuda externa de ese país en esa fecha.
9. Crear una función que reciba un país y una fecha, y dibuje un diagrama de barras con las cantidades de los distintos tipos de deudas de ese país en esa fecha.

10. Crear una función que reciba una lista de países y un tipo de deuda y dibuje un diagrama de líneas con la evolución de ese tipo de deuda de esos países (una línea por país).
11. Crear una función que reciba un país y una lista de tipos de deuda y dibuje un diagrama de líneas con la evolución de esos tipos de deuda de ese país (una línea por tipo de deuda).
12. Crear una función que reciba una lista de países y una lista de tipos de deuda, y dibuje un diagrama de cajas con las deudas de esos tipos de esos países (una caja por país y tipo de deuda).
13. Preprocesar el fichero del PIB crear un data frame con el país, la fecha y el PIB.
14. Crear una función que reciba un país y dibuje la evolución de la deuda pública total como porcentaje del PIB.
15. Crear una función que reciba un país devuelva un diccionario con los años y si el endeudamiento en esa fecha era insostenible. Se considera un endeudamiento insostenible si durante los tres años anteriores el porcentaje de deuda pública con respecto al PIB es superior al 20 %.

Solución

3.2.1. Sin usar la librería Pandas

3.2.2. Requisito 1

Crear una función que reciba un país y un tipo de deuda y devuelva un diccionario con todos los periodos y la cantidad de deuda en esos periodos de ese país y tipo de deuda.

```
# Preprocesamos primero el fichero para obtener un diccionario de
 ↪  diccionarios con las series de deudas de cada país.
from urllib import request
from urllib.error import URLError
# Leemos el fichero desde la url.
try:
    f = request.urlopen(␣
 ↪  'http://aprendeconalf.es/python/trabajos/datos/deuda.csv')
    # El fichero no existe
except URLError:
    print('¡La url no existe!')
else:
    # Creamos una lista con las líneas del fichero.
    lineas = f.read().decode('utf-8').splitlines()
    # Dividimos la primera línea del fichero que contiene los
     ↪  nombres de las columnas y creamos una lista con los nombres
     ↪  de las columnas.
    columnas = lineas[0].split(',')
    # Creamos un diccionario vacío para ir añadiendo las series de
     ↪  deuda de cada país.
    deudas = {}
    # Recorremos las lineas del fichero desde la 1 hasta el final
    for linea in lineas[1:]:
        # Creamos el diccionario que contendrá la información de una
         ↪  serie país-tipo deuda.
        serie = {}
        # Creamos una lista con los campos partiendo la línea por el
         ↪  carácter ,.
        campos = linea.split(',')
        # Recorremos los campos de la línea
        for i in range(4, len(columnas)):
            # Para cada campo añadimos al diccionario el par con
             ↪  clave el nombre de la columna y valor el campo de la
             ↪  posición i.
            serie[columnas[i][:6]] = campos[i]
        # Añadimos el diccionario a la lista de alojamientos.
        deudas[(campos[1], campos[3])] = serie

    print(deudas)
```

```python
def deuda_pais(deudas, pais, tipo):
    ''' Función que recibe un diccionario con las deudas de cada
     ↪  país y devuelve la serie de deuda de un tipo y un país dado.

    Parámetros:
        - deudas: Es un diccionario de diccionarios donde las claves
 ↪  del diccionario principal son tuplas (país, tipo de deuda) y los
 ↪  valores son diccionarios con las deudas de cada trimestre.
        - pais: Es una cadena con el nombre del país.
        - tipo: Es una cadena con el tipo de deuda.

    Devuelve: Un diccionario con las deudas trimestrales del tipo y
 ↪  el país dados.
    '''
    return deudas[(pais, tipo)]

# Ejemplo
print(deuda_pais(deudas, 'AUS', 'DP.DOD.DLTC.CR.M1.PS.CD'))
```

3.2.3. Requisito 2

Crear una función que reciba un país y un tipo de deuda y devuelva un diccionario con el mínimo y el máximo de deuda de ese tipo para ese país.

```python
def rango_deuda(deudas, pais, tipo):
    ''' Función que devuelve el mínino y el máximo de deuda de un
     ↪  tipo y un país dado.

    Parámetros:
        - deudas: Es un diccionario de diccionarios donde las claves
 ↪  del diccionario principal son tuplas (país, tipo de deuda) y los
 ↪  valores son diccionarios con las deudas de cada trimestre.
        - pais: Es una cadena con el nombre del país.
        - tipo: Es una cadena con el tipo de deuda.

    Devuelve: Un diccionario con el mínimo y el máximo de la deuda
 ↪  trimestral del tipo y el país dados.
    '''

    deuda = deudas[(pais, tipo)]
    return {'Mínimo': min(deuda.values()), 'Máximo':
     ↪  max(deuda.values())}

# Ejemplo
print(rango_deuda(deudas, 'AUS', 'DP.DOD.DLTC.CR.M1.PS.CD'))
```

3.2.4. Requisito 3

Crear una función que reciba un país y una fecha, y devuelva un diccionario con la deuda interna y la deuda externa de ese país en ese año.

```
def deuda_interna_externa(deudas, pais, fecha):
    ''' Función que devuelve la deuda interna y externa de un
     ↪  trimestre y un país dado.

    Parámetros:
        - deudas: Es un diccionario de diccionarios donde las claves
 ↪  del diccionario principal son tuplas (país, tipo de deuda) y los
 ↪  valores son diccionarios con las deudas de cada trimestre.
        - pais: Es una cadena con el nombre del país.
        - fecha: Es una cadena con el año y el trimestre.

    Devuelve: Un diccionario con la deuda interna y externa del
 ↪  trimestre y el país dados.
    '''

    deuda_interna = deudas[(pais, 'DP.DOD.DECD.CR.PS.CD')]
    deuda_externa = deudas[(pais, 'DP.DOD.DECX.CR.PS.CD')]
    return {'Deuda interna':deuda_interna[fecha], 'Deuda
     ↪  externa':deuda_externa[fecha]}

# Ejemplo
print(deuda_interna_externa(deudas, 'AUS', '2015Q1'))
```

3.2.5. Requisito 4

Crear una función que reciba un país y un año, y devuelva un diccionario con la deuda en moneda local y la deuda en moneda extranjera de ese país en ese año.

```python
def deuda_moneda_local_extranjera(deudas, pais, fecha):
    ''' Función que devuelve la deuda en moneda local y extranjera
     ↪  de un trimestre y un país dado.

    Parámetros:
        - deudas: Es un diccionario de diccionarios donde las claves
 ↪  del diccionario principal son tuplas (país, tipo de deuda) y los
 ↪  valores son diccionarios con las deudas de cada trimestre.
        - pais: Es una cadena con el nombre del país.
        - fecha: Es una cadena con el año y el trimestre.

    Devuelve: Un diccionario con la deuda en moneda local y
 ↪  extranjera del trimestre y el país dados.
    '''

    deuda_moneda_local = deudas[(pais, 'DP.DOD.DECN.CR.PS.CD')]
    deuda_moneda_extranjera = deudas[(pais, 'DP.DOD.DECF.CR.PS.CD')]
    return {'Deuda en moneda local':deuda_moneda_local[fecha],
     ↪  'Deuda en moneda extranjera':deuda_moneda_extranjera[fecha]}

# Ejemplo
print(deuda_moneda_local_extranjera(deudas, 'AUS', '2015Q1'))
```

3.2.6. Usando la librería Pandas

3.2.7. Requisito 5

Preprocesar el fichero de deuda pública para obtener un data frame con el país, el tipo de deuda, la fecha y la cantidad de deuda.

```
import pandas as pd

from urllib.error import HTTPError

try:
    deuda = pd.read_csv(
 ↪  'http://aprendeconalf.es/python/trabajos/datos/deuda.csv')
except HTTPError:
    print('La url no existe')
else:
    deuda = deuda.melt(id_vars=['Country Name', 'Country Code',
 ↪  'Series Name',
                               'Series Code'], var_name='Fecha',
                                ↪  value_name='Cantidad')
    # Renombramos los nombres de las columnas que queremos
    deuda.rename(columns={'Country Name': 'Pais', 'Country Code':
 ↪  'PaisId',
                          'Series Name': 'Tipo', 'Series Code':
                           ↪  'TipoId'}, inplace=True)
    # Extraemos los 6 primeros caracteres de la columna Fecha
    deuda['Fecha'] = deuda.Fecha.str[0:6]
    # Renombramos los tipos de deuda
    tipos = {'DP.DOD.DECD.CR.PS.CD': 'Deuda interna',
 ↪  'DP.DOD.DECN.CR.PS.CD': 'Deuda en moneda local',
 ↪  'DP.DOD.DECX.CR.PS.CD': 'Deuda externa',
            'DP.DOD.DECF.CR.PS.CD': 'Deuda en moneda extranjera',
             ↪  'DP.DOD.DLTC.CR.M1.PS.CD': 'Deuda a lago plazo',
             ↪  'DP.DOD.DSTC.CR.PS.CD': 'Deuda a corto plazo'}
    deuda['TipoId'] = deuda.TipoId.apply(
        lambda x: tipos[x] if x in tipos.keys() else x)

    deuda
```

3.2.8. Requisito 6

Crear una función que reciba un país y una fecha y devuelva una serie con la deuda total interna, externa, en moneda local, en moneda extranjera, a corto plazo y a largo plazo, de ese país en esa fecha.

```python
def resumen_deuda(deudas, pais, fecha):
    ''' Función que devuelve la deuda total interna, externa, en
     ↪  moneda local, en moneda extranjera, a corto plazo y a largo
     ↪  plazo, de un país y una fecha dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - pais: Es una cadena con el nombre del país.
        - fecha: Es una cadena con el año y el trimestre.

    Devuelve: Una serie con la deuda total interna, externa, en
 ↪  moneda local, en moneda extranjera, a corto plazo y a largo
 ↪  plazo, del país y la fecha dados.
    '''

    # Filtramos el país, la fecha y los tipos de deuda
    deuda_filtro = deuda[(deuda.PaisId == pais) & (deuda.Fecha ==
 ↪  fecha) & deuda.TipoId.isin(
        ['Deuda interna', 'Deuda en moneda local', 'Deuda externa',
 ↪  'Deuda en moneda extranjera', 'Deuda a lago plazo', 'Deuda a
 ↪  corto plazo'])]
    # Devolvemos la serie de la columna Cantidad tomando como índice
     ↪  la columna del tipo de deuda.
    return pd.Series(list(deuda_filtro.Cantidad),
     ↪  index=deuda_filtro.TipoId)


# Ejemplo
print(resumen_deuda(deuda, 'AUS', '2015Q1'))
```

3.2.9. Requisito 7

Crear una función que reciba un tipo de deuda y una fecha, y devuelva una serie con la deuda de ese tipo de todos los países en esa fecha.

```
def resumen_deuda(deuda, tipo, fecha):
    ''' Función que devuelve la deuda de todos los países de un tipo
     ↪  y en una fecha dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - tipo: Es una cadena con el tipo de deuda.
        - fecha: Es una cadena con el año y el trimestre.

    Devuelve: Una serie con la deuda de todos los países del tipo y
 ↪  la fecha dados.
    '''

    # Filtramos el tipo de deuda y la fecha
    deuda_filtro = deuda[(deuda.TipoId == tipo) & (deuda.Fecha ==
 ↪  fecha)]
    # Devolvemos la serie de la columna Cantidad tomando como índice
     ↪  la columna del país.
    return pd.Series(list(deuda_filtro.Cantidad),
     ↪  index=deuda_filtro.Pais)


# Ejemplo
print(resumen_deuda(deuda, 'Deuda externa', '2015Q1'))
```

3.2.10. Requisito 8

Crear una función que reciba un país y una fecha y dibuje un diagrama de sectores con la deuda interna y la deuda externa de ese país en esa fecha.

```
import matplotlib.pyplot as plt

def sectores_deuda_externa_interna(deudas, pais, fecha):
    ''' Función que dibuja un diagrama de sectores con la deuda
     ↪  interna y externa de un país y una fecha dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - tipo: Es una cadena con el tipo de deuda.
        - fecha: Es una cadena con el año y el trimestre.
    '''

    # Filtramos el país, la fecha y los tipos de deuda
    deuda_filtro = deuda[(deuda.PaisId == pais) & (
        deuda.Fecha == fecha) & deuda.TipoId.isin(['Deuda interna',
 ↪  'Deuda externa'])]
    # Creamos una serie de la columna Cantidad tomando como índice
     ↪  la columna del tipo de deuda.
    serie = pd.Series(list(deuda_filtro.Cantidad),
 ↪  index=deuda_filtro.TipoId)
    # Creamos la figura y los ejes
    fig, ax = plt.subplots()
    # Dibujamos el diagrama de sectores
    serie.plot(kind='pie', autopct='%1.0f%%', ax=ax)
    # Añadimos el título
    ax.set_title('Deuda externa vs interna de ' + pais + ' en ' +
 ↪  fecha, loc="center",
                 fontdict={'fontsize': 14, 'fontweight': 'bold',
 ↪  'color': 'tab:blue'})
    # Eliminamos la etiqueta del eje y
    ax.set_ylabel('')
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
print(sectores_deuda_externa_interna(deuda, 'AUS', '2015Q1'))
```

3.2.11. Requisito 9

Crear una función que reciba un país y una fecha, y dibuje un diagrama de barras con las cantidades de los distintos tipos de deudas de ese país en esa fecha.

```
def barras_tipos_deuda(deuda, pais, fecha):
    ''' Función que dibuja un diagrama de barras con las cantidades
     ↪  de los distintos tipos de deudas de un país y una fecha
     ↪  dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - pais: Es una cadena con el nombre del país.
        - fecha: Es una cadena con el año y el trimestre.
    '''

    # Filtramos el país, la fecha y los tipos de deuda
    deuda_filtro = deuda[(deuda.PaisId == pais) & (deuda.Fecha ==
 ↪  fecha)]
    # Creamos la figura y los ejes
    fig, ax = plt.subplots()
    # Dibujamos el diagrama de barras
    deuda_filtro.plot(kind='bar', x='Tipo', y='Cantidad', ax=ax)
    # Añadimos el título
    ax.set_title('Deuda por tipología de ' + pais + ' en ' + fecha,
 ↪  loc="center",
                 fontdict={'fontsize': 14, 'fontweight': 'bold',
 ↪  'color': 'tab:blue'})
    # Eliminamos la leyenda
    ax.legend().remove()
    # Guardamos el gráfico.
    plt.show()
    return


# Ejemplo
print(barras_tipos_deuda(deuda, 'AUS', '2015Q1'))
```

3.2.12. Requisito 10

Crear una función que reciba una lista de países y un tipo de deuda y dibuje un diagrama de líneas con la evolución de ese tipo de deuda de esos países (una línea por país).

```python
def evolucion_tipo_deuda(deuda, paises, tipo):
    ''' Función que dibuja un diagrama de líneas con la evolución de
    ↪ un tipo de deuda y una lista de países dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - paises: Es una lista de cadenas con los nombres de los
    ↪ países.
        - tipo: Es una cadena con el tipo de deuda.
    '''

    # Filtramos los países y el tipo de deuda
    deuda_filtro = deuda[(deuda.PaisId.isin(paises)) & (deuda.TipoId
    ↪ == tipo)]
    # Convertimos la fecha en el índice
    deuda_filtro.set_index('Fecha', inplace=True)
    # Creamos la figura y los ejes
    fig, ax = plt.subplots()
    # Dibujamos el diagrama de barras
    deuda_filtro.groupby('PaisId').Cantidad.plot(legend=True, ax=ax)
    # Añadimos el título
    ax.set_title('Evolución de ' + tipo, loc="center",
                 fontdict={'fontsize': 14, 'fontweight': 'bold',
    ↪ 'color': 'tab:blue'})
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
evolucion_tipo_deuda(deuda, ['GEO', 'SLV', 'MDA'], 'Deuda interna')
```

3.2.13. Requisito 11

Crear una función que reciba un país y una lista de tipos de deuda y dibuje un diagrama de líneas con la evolución de esos tipos de deuda de ese país (una línea por tipo de deuda).

```
def evolucion_deuda_pais(deuda, pais, tipos):
    ''' Función que dibuja un diagrama de líneas con la evolución de
     ↪  unos tipos de deuda y un país dado.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - pais: Es un cadena con el nombre del país.
        - tipos: Es una lista de cadenas con los tipo de deuda.
    '''

    # Filtramos el país y los tipos de deuda
    deuda_filtro = deuda[(deuda.PaisId == pais) &
 ↪  (deuda.TipoId.isin(tipos))]
    # Convertimos la fecha en el índice
    deuda_filtro.set_index('Fecha', inplace=True)
    # Creamos la figura y los ejes
    fig, ax = plt.subplots()
    # Dibujamos el diagrama de barras
    deuda_filtro.groupby('TipoId').Cantidad.plot(legend=True, ax=ax)
    # Añadimos el título
    ax.set_title('Evolución de los tipos de deuda de ' + pais,
 ↪  loc="center",
                 fontdict={'fontsize': 14, 'fontweight': 'bold',
 ↪  'color': 'tab:blue'})
    # Guardamos el gráfico.
    plt.show()
    return


# Ejemplo
evolucion_deuda_pais(deuda, 'SLV', ['Deuda interna', 'Deuda
 ↪  externa'])
```

3.2.14. Requisito 12

Crear una función que reciba una lista de países y una lista de tipos de deuda, y dibuje un diagrama de cajas con las deudas de esos tipos de esos países (una caja por país y tipo de deuda).

```
def cajas_deuda(deuda, paises, tipos):
    ''' Función que dibuja un diagrama de cajas con las deudas de
     ↪  unos tipos y unos países dados.

    Parámetros:
        - deuda: Es un DataFrame con las deudas de los países.
        - paises: Es una lista de cadenas con los nombres de los
 ↪  países.
        - tipos: Es una lista de cadenas con los tipo de deuda.
    '''

    # Filtramos el país y los tipos de deuda
    deuda_filtro = deuda[(deuda.PaisId.isin(paises)) &
                         (deuda.TipoId.isin(tipos))]
    # Creamos la figura y los ejes
    fig, ax = plt.subplots()
    # Dibujamos el diagrama de cajas
    deuda_filtro.boxplot(column='Cantidad', by=['PaisId', 'TipoId'],
 ↪  ax=ax)
    # Añadimos el título
    ax.set_title('Deuda de ' + ', '.join(paises) + '\n(' + ',
 ↪  '.join(tipos) + ')',
                 loc="center", fontdict={'fontsize': 14,
 ↪  'fontweight': 'bold', 'color': 'tab:blue'})
    plt.suptitle('')
    # Rotamos las etiquetas del eje x
    plt.xticks(rotation=90)
    # Guardamos el gráfico.
    plt.show()
    return


# Ejemplo
cajas_deuda(deuda, ['BOL', 'MDA', 'SLV'], ['Deuda interna', 'Deuda
 ↪  externa'])
```

3.2.15. Requisito 13

Preprocesar el fichero del PIB crear un data frame con el país, la fecha y el PIB.

3.2.16. Requisito 14

Crear una función que reciba un país y dibuje la evolución de la deuda pública total como porcentaje del PIB.

3.2.17. Requisito 15

Crear una función que reciba un país devuelva un diccionario con los años y si el endeudamiento en esa fecha era insostenible. Se considera un endeudamiento insostenible si durante los tres años anteriores el porcentaje de deuda pública con respecto al PIB es superior al 20 %.
Abrir la solución en Google Colab

4. Airbnb Madrid

El objetivo de este trabajo es comprobar si se está utilizando la plataforma Airbnb por parte de empresas, en lugar de particulares, para alquiler turístico en el centro de Madrid.

4.1. Conjunto de datos

Datos abiertos de Inside Airbnb: Fichero alojamientos Madrid detallado.

4.2. Requisitos obligatorios

Sin usar Pandas:

1. Extraer del fichero de alojamientos una lista con todos los alojamientos, donde cada alojamiento sea un diccionario que contenga el identificador del alojamiento, el identificador del anfitrión, el distrito, el precio y las plazas.
2. Crear una función que reciba la lista de alojamientos y devuelva el número de alojamientos en cada distrito.
3. Crear una función que reciba la lista de alojamientos y un número de ocupantes y devuelva la lista de alojamientos con un número de plazas mayor o igual que el número de ocupantes.
4. Crear una función que reciba la lista de alojamientos un distrito, y devuelva los 10 alojamientos más baratos del distrito.
5. Crear una función que reciba la lista de alojamientos y devuelva un diccionario con los anfitriones y el número de alojamientos que posee cada uno.

Usando Pandas:

6. Preprocesar el fichero de alojamientos para crear un data frame con las variables id, host_id, listing_url, room_type, neighbourhood_group_cleansed, price, cleaning_fee, accommodates, minimum_nights, minimum_cost, review_scores_rating, latitude, longitude, is_location_exact. Eliminar del data frame cualquier fila incompleta. Añadir al data frame nuevas variables con el coste mínimo por noche y por persona (que incluya los gastos de limpieza).

7. Crear una función que reciba una lista de distritos y devuelva un diccionario con los tipos de alojamiento en ese distrito y el porcentaje de alojamientos de ese tipo.
8. Crear una función que reciba una lista de distritos y devuelva un diccionario con el número de alojamientos que cada anfitrión ofrece en esos distrito, ordenado de más a menos alojamientos.
9. Crear una función que reciba devuelva un diccionario con el número medio de alojamientos por anfitrión de cada distrito.
10. Crear una función que reciba una lista de distritos y dibuje un diagrama de sectores con los porcentajes de tipos de alojamientos en esos distritos.
11. Crear una función que dibuje un diagrama de barras con el número de alojamientos por distritos.
12. Crear una función que dibuje un diagrama de barras con los porcentajes acumulados de tipos de alojamientos por distritos.
13. Crear una función reciba una lista de distritos y una lista de tipos de alojamientos, y dibuje un diagrama de sectores con la distribución del número de alojamientos de ese tipo por anfitrión.
14. Crear una función que dibuje un diagrama de barras con los precios medios por persona y día de cada distrito.
15. Crear una función que reciba una lista de distritos y dibuje un gráfico de dispersión con el coste mínimo por noche y persona y la puntuación en esos distritos.
16. Crear una función que reciba una lista de distritos y dibuje dos histogramas con la distribución de precios por persona y día, uno para los alojamientos con título en inglés y otro para los alojamientos con títulos en español. Si la distribución es muy asimétrica, aplicar una transformación logarítmica. ¿Hay diferencias entre los precios de los alojamientos en inglés y el español? Nota: Para identificar el idioma puede usare el módulo langdetect.

Solución

4.2.1. Sin usar la librería Pandas

4.2.2. Requisito 1

Extraer del fichero de alojamientos una lista con todos los alojamientos, donde cada alojamiento sea un diccionario que contenga el identificador del alojamiento, el identificador del anfitrión, el distrito, el precio y las plazas.

```
from urllib import request
from urllib.error import URLError
# Leemos el fichero desde la url.
try:
    f = request.urlopen(
 ↪  'http://aprendeconalf.es/python/trabajos/datos/madrid-airbnb-listings-small.csv'
 ↪  )
    # El fichero no existe
except URLError:
    print('¡La url no existe!')
else:
    # Creamos una lista con las líneas del fichero.
    lineas = f.read().decode('utf-8').splitlines()
    # Extraemos los nombres de las columnas de la primera fila
     ↪  partiendo la cadena por el carácter de tabulación.
    columnas = lineas[0].split('\t')
    seleccion = ['id', 'host_id', 'neighbourhood_group_cleansed',
 ↪  'accommodates', 'price']
    # Creamos un diccionario para traducir el nombre de las columnas
    traduccion = {'id':'id', 'host_id':'anfitrion',
 ↪  'neighbourhood_group_cleansed':'distrito',
 ↪  'accommodates':'plazas', 'price':'precio'}
    # Creamos la lista de alojamientos
    alojamientos = []
    # Recorremos las lineas del fichero desde la 1 hasta el final
    for linea in lineas[1:]:
        # Creamos el diccionario que contendrá la información del
         ↪  alojamiento.
        alojamiento = {}
        # Creamos una lista con los campo partiendo la línea por el
         ↪  carácter de tabulación.
        campos = linea.split('\t')
        # Recorremos los campos de la línea
        for i in range(len(columnas)):
            # Para cada campo añadimos al diccionario el par con
             ↪  clave el nombre de la columna y valor el campo de la
             ↪  posición i.
            if columnas[i] in seleccion:
                alojamiento[traduccion[columnas[i]]] = campos[i]
        # Añadimos el diccionario a la lista de alojamientos.
        alojamientos.append(alojamiento)

    print(alojamientos)
```

4.2.3. Requisito 2

Crear una función que reciba la lista de alojamientos y devuelva el número de alojamientos en cada distrito.

```
def alojamientos_distritos(alojamientos):
    '''
    Función que devuelve un diccionario con el número de
 ↪  alojamientos en cada distrito.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.

    Devuelve: Un diccionario con el número de alojamientos por
 ↪  distrito.
    '''

    # Creamos el diccionario
    alojamiento_distritos = {}
    # Recorremos la lista de alojamientos
    for alojamiento in alojamientos:
        # Si el distrito ya aparece como clave del diccionario,
         ↪  incrementamos su valor en uno
        if alojamiento['distrito'] in alojamiento_distritos.keys():
            alojamiento_distritos[alojamiento['distrito']] += 1
        # Si el distrito no aparece como clave del diccionario, lo
         ↪  añadimos con valor 1.
        else:
            alojamiento_distritos[alojamiento['distrito']] = 1
    return alojamiento_distritos

# Ejemplo
print(alojamientos_distritos(alojamientos))
```

4.2.4. Requisito 3

Crear una función que reciba la lista de alojamientos y un número de ocupantes y devuelva la lista de alojamientos con un número de plazas mayor o igual que el número de ocupantes.

```python
def filtrar_plazas(alojamientos, plazas):
    '''
    Función que devuelve una lista con los alojamientos que tienen
    un número de plazas mayor o igual que uno dado.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
    diccionario contiene los datos de un alojamiento.
    Devuelve: Un diccionario con el número de alojamientos por
    distrito.
    - plazas: Es un entero con el número mínimo de plazas.

    Devuelve: Una lista con los alojamientos que tienen un número de
    plazas mayor o igual que plazas.
    '''

    return [alojamiento for alojamiento in alojamientos if
     int(alojamiento['plazas']) >= plazas]

# Ejemplo
filtro = filtrar_plazas(alojamientos, 10)
print(filtro)
```

4.2.5. Requisito 4

Crear una función que reciba la lista de alojamientos un distrito, y devuelva los 10 alojamientos más baratos del distrito.

```python
def alojamientos_baratos(alojamientos, distrito, n):
    '''
    Función que devuelve una lista con los n alojamientos más
 ↪  baratos en un distrito dado.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    Devuelve: Un diccionario con el número de alojamientos por
 ↪  distrito.
    - distrito: Es una cadena con el nombre del distrito.
    - n: Es un entero con el número de alojamientos a devolver.

    Devuelve: Una lista con los n alojamientos más baratos del
 ↪  distrito dado.
    '''

    # Filtramos los alojamientos del distrito
    alojamientos_distrito = [alojamiento for alojamiento in
 ↪  alojamientos if alojamiento['distrito'] == distrito]
    # Definimos una función de ordenación con la clave para la
     ↪  ordenación
    def orden(dict): return float(dict['precio'][1:])
    # Ordenamos la lista de alojamientos con la función de
     ↪  ordenación
    ranking_alojamientos = sorted(alojamientos_distrito, key =
 ↪  orden)
    return ranking_alojamientos[:n]

# Ejemplo
top_arganzuela = alojamientos_baratos(alojamientos, 'Arganzuela',
 ↪  10)
print(top_arganzuela)
```

4.2.6. Requisito 5

Crear una función que reciba la lista de alojamientos y devuelva un diccionario con los anfitriones y el número de alojamientos que posee cada uno.

```python
def alojamientos_anfitriones(alojamientos):
    '''
    Función que devuelve un diccionario con el número de
    alojamientos de cada anfitrión.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
    diccionario contiene los datos de un alojamiento.

    Devuelve: Un diccionario con el número de alojamientos por
    anfitrión.
    '''

    # Creamos el diccionario
    alojamiento_anfitriones = {}
    # Recorremos la lista de alojamientos
    for alojamiento in alojamientos:
        # Si el anfitrión ya aparece como clave del diccionario,
        incrementamos su valor en uno
        if alojamiento['anfitrion'] in
        alojamiento_anfitriones.keys():
            alojamiento_anfitriones[alojamiento['anfitrion']] += 1
        # Si el anfitrión no aparece como clave del diccionario, lo
        añadimos con valor 1.
        else:
            alojamiento_anfitriones[alojamiento['anfitrion']] = 1
    return alojamiento_anfitriones

# Ejemplo
anfitriones = alojamientos_anfitriones(alojamientos)
print(anfitriones)
```

4.2.7. Usando la librería Pandas

4.2.8. Requisito 6

Preprocesar el fichero de alojamientos para crear un data frame con las variables id, host_id, listing_url, room_type, neighbourhood_group_cleansed, price, cleaning_fee, accommodates, minimum_nights, minimum_cost, review_scores_rating, latitude, longitude, is_location_exact. Eliminar del data frame cualquier fila incompleta. Añadir al data frame nuevas variables con el coste mínimo por noche y por persona (que incluya los gastos de limpieza).

```
import pandas as pd

# Creamos un DataFrame desde la url del fichero csv
try:
    alojamientos =
 ↪  pd.read_csv('../datos/madrid-airbnb-listings-small.csv', sep =
 ↪  '\t')
except URLError:
    print('La url no existe')
else:
    # Renombramos los nombres de las columnas que queremos
    alojamientos.rename(columns = {'host_id': 'anfitrion',
 ↪  'listing_url': 'url', 'room_type':'tipo_alojamiento',
 ↪  'neighbourhood_group_cleansed':'distrito', 'price':'precio',
 ↪  'cleaning_fee':'gastos_limpieza', 'accommodates':'plazas',
 ↪  'minimum_nights':'noches_minimas',
 ↪  'review_scores_rating':'puntuacion'}, inplace = True)
    # Filtramos las columnas que quermos
    alojamientos = alojamientos[['id', 'anfitrion', 'url',
 ↪  'tipo_alojamiento', 'distrito', 'precio', 'gastos_limpieza',
 ↪  'plazas', 'noches_minimas', 'puntuacion']]
    # Eliminamos las filas con valores desconocidos
    alojamientos = alojamientos.dropna()
    # Eliminamos el carácter $ de las columnas del precio y
     ↪  gastos_limpieza y las convertimos a float
    alojamientos['precio'] =
 ↪  alojamientos.precio.str.replace(',','').str[1:].astype('float')
    alojamientos['gastos_limpieza'] =
 ↪  alojamientos.gastos_limpieza.str[1:].astype('float')
    # Creamos una nueva columna con el precio por persona
     ↪  multiplicando el precio diario por el número mínimo de
     ↪  noches, sumando los gastos de limpieza y finalmente
     ↪  dividiendo por el número mínimo de noches y el número de
     ↪  plazas.
    alojamientos['precio_persona'] = (alojamientos.precio *
 ↪  alojamientos.noches_minimas + alojamientos.gastos_limpieza) /
 ↪  (alojamientos.noches_minimas + alojamientos.plazas)

alojamientos
```

4.2.9. Requisito 7

Crear una función que reciba una lista de distritos y devuelva un diccionario con los tipos de alojamiento en esos distritos y el porcentaje de alojamientos de ese tipo.

```
def tipos_alojamientos_distritos(alojamientos, distritos):
    '''
    Función que devuelve una serie con el porcentaje de tipos de
 ↪  alojamientos en una lista de distritos dada.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    - distritos: Es una lista con los nombres de los distritos.

    Devuelve: Una serie con el porcentaje de tipos de alojamientos
 ↪  en los distritos dados.
    '''
    return
     ↪  alojamientos[alojamientos.distrito.isin(distritos)].tipo_alojamiento.value_cour
     ↪  = True) * 100

# Ejemplo
print(tipos_alojamientos_distritos(alojamientos, ['Arganzuela',
 ↪  'Centro']))
```

4.2.10. Requisito 8

Crear una función que reciba una lista de distritos y devuelva un diccionario con el número de alojamientos que cada anfitrión ofrece en esos distritos, ordenado de más a menos alojamientos.

```python
def alojamientos_anfitriones_distritos(alojamientos, distritos):
    '''
    Función que devuelve una serie con el número de alojamientos de
 ↪  cada anfitrion en unos distritos dados, ordenada de mas a menos
 ↪  alojamientos.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    - distritos: Es una lista con los nombres de los distritos.

    Devuelve: Una serie con el número de alojamientos de cada
 ↪  anfitrion en los distritos dados, ordenada de mas a menos
 ↪  alojamientos.
    '''
    return
     ↪  alojamientos[alojamientos.distrito.isin(distritos)].anfitrion.value_counts().so
     ↪  = False)

# Ejemplo
print(alojamientos_anfitriones_distritos(alojamientos, ['Centro']))
print(alojamientos_anfitriones_distritos(alojamientos,
 ↪  ['Villaverde']))
```

4.2.11. Requisito 9

Crear una función que devuelva un diccionario con el número medio de alojamientos por anfitrión de cada distrito.

```python
def media_alojamientos_distritos(alojamientos):
    '''
    Función que devuelve una serie con el número medio de
    alojamientos por anfitrión de cada distrito.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
    diccionario contiene los datos de un alojamiento.

    Devuelve: Una serie con el número medio de alojamientos por
    anfitrión de cada distrito.

    '''
    return alojamientos.groupby('distrito'
    ).anfitrion.value_counts().unstack(level =
    "distrito").mean()

# Ejemplo
print('Número medio de alojamientos por anfitrión en cada distrito')
print(media_alojamientos_distritos(alojamientos))
```

4.2.12. Requisito 10

Crear una función que reciba una lista de distritos y dibuje un diagrama de sectores con los porcentajes de tipos de alojamientos en esos distritos.

```
import matplotlib.pyplot as plt

def sectores_tipos_alojamientos(alojamientos, distritos):
    '''
    Función que dibuja un diagrama de sectores con los porcentajes
 ↪  de tipos de alojamientos en unos distritos dados.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    - distritos: Es una lista con los nombres de los distritos.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Filtramos los distritos de la lista de distritos dada, después
     ↪  contamos la frecuencias de los tipos de alojamientos y
     ↪  dibujamos el diagrama de sectores

 ↪  alojamientos[alojamientos.distrito.isin(distritos)].tipo_alojamiento.value_counts(n
 ↪  = True).plot(kind = 'pie',  autopct='%1.0f%%', ax = ax)
    # Ponermos el título
    ax.set_title('Distribución del porcentaje de tipos de
 ↪  alojamientos\n Distritos de ' + ', '.join(distritos), loc =
 ↪  "center", fontdict = {'fontsize':14, 'fontweight':'bold',
 ↪  'color':'tab:blue'})
    # Eliminamos la etiqueta del eje y
    ax.set_ylabel('')
    # Guardamos el gráfico.
    plt.show()
    return

#Ejemplo
sectores_tipos_alojamientos(alojamientos, ['Arganzuela', 'Centro'],
 ↪  )
```

4.2.13. Requisito 11

Crear una función que dibuje un diagrama de barras con el número de alojamientos por distritos.

```
def barras_alojamientos_distritos(alojamientos):
    '''
    Función que dibuja un diagrama de barras con el número de
 ↪  alojamientos por distritos.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Contamos la frecuencias de alojamientos por distritos y
     ↪  dibujamos las barras.
    alojamientos.distrito.value_counts().plot(kind = 'bar')
    # Ponemos el título
    ax.set_title('Número de alojamientos por distrito', loc =
 ↪  "center", fontdict = {'fontsize':14, 'fontweight':'bold',
 ↪  'color':'tab:blue'})
    # Ponemos una rejilla
    ax.grid(axis = 'y', color = 'lightgray', linestyle = 'dashed')
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
barras_alojamientos_distritos(alojamientos)
```

4.2.14. Requisito 12

Crear una función que dibuje un diagrama de barras con los porcentajes acumulados de tipos de alojamientos por distritos.

```
def barras_tipos_alojamientos_distritos(alojamientos):
    '''
    Función que dibuja un diagrama de barras con los porcentajes
 ↪  acumulados de tipos de alojamientos por distritos.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Agrupamos el DataFrame por distritos y contamos las
     ↪  frecuencias de los tipos de alojamiento. Después pivotamos
     ↪  el índice de los tipos de alojamientos para pasarlos a
     ↪  columnas y dibujamos las barras acumuladas.
    (alojamientos.groupby('distrito'
 ↪  ).tipo_alojamiento.value_counts(normalize =
 ↪  True)*100).unstack().plot(kind = 'bar', stacked = True, ax = ax)
    # Ponemos el título
    ax.set_title('Tipos de alojamiento por distrito (%)', loc =
 ↪  "center", fontdict = {'fontsize':14, 'fontweight':'bold',
 ↪  'color':'tab:blue'})
    # Eliminamos la etiqueta del eje x
    ax.set_xlabel('')
    # Ponemos una rejilla
    ax.grid(axis = 'y', color = 'lightgray', linestyle = 'dashed')
    # Reducimos el eje x un 30% para que quepa la leyenda
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.7, box.height])
    # Dibujar la leyenda fuera del área del gráfico
    plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
barras_tipos_alojamientos_distritos(alojamientos)
```

4.2.15. Requisito 13

Crear una función reciba una lista de distritos y una lista de tipos de alojamientos, y dibuje un diagrama de sectores con la distribución del número de alojamientos de

ese tipo por anfitrión en esos distritos.

```
def sectores_tipos_alojamientos_anfitrion(alojamientos, distritos,
 ↪  tipos):
    '''
    Función que dibuja un diagrama de sectores con la distribución
 ↪  del número de alojamientos por anfitrión de unos tipos y en unos
 ↪  distritos dados.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    - distritos: Es una lista con los nombres de los distritos.
    - tipos: Es una lista con los nombres de los tipos de
 ↪  alojamientos.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Filtramos los distritos y los tipos de alojamientos
    alojamientos_filtrados =
 ↪  alojamientos[alojamientos.distrito.isin(distritos) &
 ↪  alojamientos.tipo_alojamiento.isin(tipos)]
    # Contamos la frecuencia de alojamientos por anfitrión y
     ↪  dibujamos el diagrama de sectores
    alojamientos_filtrados.anfitrion.value_counts(normalize =
 ↪  True).plot(kind = 'pie', ax = ax)
    # Ponermos el título
    ax.set_title('Distribución del número de alojamientos por
 ↪  anfitrión\nDistritos de ' + ', '.join(distritos) + '\nTipos de
 ↪  alojamiento' + ','.join(tipos), loc = "center", fontdict =
 ↪  {'fontsize':14, 'fontweight':'bold', 'color':'tab:blue'})
    # Eliminamos la etiqueta del eje y
    ax.set_ylabel('')
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
sectores_tipos_alojamientos_anfitrion(alojamientos, ['Villaverde',
 ↪  'Vicálvaro'], ['Entire home/apt', 'Hotel room'])
```

4.2.16. Requisito 14

Crear una función que dibuje un diagrama de barras con los precios medios por persona y día de cada distrito.

```
def barras_precios_medios_persona(alojamientos):
    '''
    Función que dibuja un diagrama de barras con los precios medios
 ↪  por persona y día de cada distrito.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Agrupamos por distrito, calculamos la media de precio por
     ↪  persona y dibujamos las barras.
    alojamientos.groupby('distrito').precio_persona.mean().plot(kind
 ↪  = 'bar', ax = ax)
    # Ponemos el título
    ax.set_title('Precio medio por persona y noche', loc = "center",
 ↪  fontdict = {'fontsize':14, 'fontweight':'bold',
 ↪  'color':'tab:blue'})
    # Eliminamos la etiqueta del eje x
    ax.set_xlabel('')
    # Ponemos una rejilla
    ax.grid(axis = 'y', color = 'lightgray', linestyle = 'dashed')
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
barras_precios_medios_persona(alojamientos)
```

4.2.17. Requisito 15

Crear una función que reciba una lista de distritos y dibuje un gráfico de dispersión con el precio por por noche y persona y la puntuación en esos distritos.

```python
def precios_puntuacion_distritos(alojamientos, distritos):
    '''
    Función que dibuja un diagrama de dispersión con el precio por
 ↪  noche y persona y la puntuación en unos distritos dados.

    Parámetros:
    - alojamientos: Es una lista de diccionarios, donde cada
 ↪  diccionario contiene los datos de un alojamiento.
    - distritos: Es una lista con los nombres de los distritos.
    '''
    # Definimos la figura y los ejes del gráfico
    fig, ax = plt.subplots()
    # Filtramos los distritos
    alojamientos_filtrados =
 ↪  alojamientos[alojamientos.distrito.isin(distritos)]
    # Creamos una nueva columna con el precio por persona
     ↪  multiplicando el precio diario por el número mínimo de
     ↪  noches, sumando los gastos de limpieza y finalmente
     ↪  dividiendo por el número mínimo de noches y el número de
     ↪  plazas.
    alojamientos['precio_persona'] = (alojamientos.precio *
 ↪  alojamientos.noches_minimas + alojamientos.gastos_limpieza) /
 ↪  (alojamientos.noches_minimas + alojamientos.plazas)
    # Dibujamos el diagrama de sipersión
    ax.scatter(alojamientos['precio_persona'],
 ↪  alojamientos['puntuacion'])
    # Ponemos el título
    ax.set_title('Precios vs Puntuación\nDistritos de ' + ',
 ↪  '.join(distritos), loc = "center", fontdict = {'fontsize':14,
 ↪  'fontweight':'bold', 'color':'tab:blue'})
    # Ponemos las etiquetas de los ejes
    ax.set_xlabel('Precio en €')
    ax.set_ylabel('Puntuación')
    # Guardamos el gráfico.
    plt.show()
    return

# Ejemplo
precios_puntuacion_distritos(alojamientos, ['Arganzuela', 'Centro'])
```

Abrir la solución en Google Colab

Parte II.

Retos de programación

5. Ajedrez

5.1. Tarea 1

La primera tarea consiste en escribir un programa que guarde en un fichero la secuencia de tableros de una partida de ajedrez. Partiremos del tablero inicial donde las filas del tablero están separadas por cambios de línea y las columnas por tabuladores.

El programa debe guardar el tablero inicial en un fichero con el nombre que elija el usuario. Después debe preguntar al usuario si quiere hacer un movimiento o terminar la partida. Cada vez que el usuario quiera hacer un nuevo movimiento debe preguntar la fila y la columna de la pieza que quiere mover y la fila y la columna a la que la quiere mover. Tras ello añadirá el tablero resultante al final del fichero anterior.

El fichero partida-ajedrez.txt contiene un ejemplo con el fichero resultante de una partida con 3 movimientos.

5.2. Tarea 2

Una vez generado el fichero con los tableros sucesivos de una partida de ajedrez, el programa preguntará por un movimiento y mostrará por pantalla el tablero correspondiente ese movimiento. Por ejemplo, utilizando el fichero partida-ajedrez.txt, si el usuario introduce el movimiento 2, debería mostrar por pantalla el siguiente tablero:

Solución

5.2.1. Tarea 1

```
def partida_ajedrez(nombre_fichero):
    ''' Función que guarda los tableros de una partida de ajedrez en
     ↪  un fichero.

    Parámetros:
        - nombre_fichero: Es una cadena con el nombre del fichero.
    '''
    # Representamos un tablero como una cadena separando las filas
     ↪  por cambios de línea y las columnas por tabuladores.
    tablero_inicial = ' \t \t \t \t \t \t \t \n \t \t \t \t \t \t \t
↪  \n\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\n\t\t\t\t\t\t\t\n
↪   \t \t \t \t \t \t \t \n \t \t \t \t \t \t \t '
    # Creamos una lista vacía para añadir las filas del tablero-
    tablero =  []
    # Bucle iterativo para recorrer las filas del
     ↪  tablero.partida_ajedrez.
    # i toma como valores los elementos las subcadenas resultantes
     ↪  de partir la cadena del tablero por el caracter de cambio de
     ↪  línea.
    for i in tablero_inicial.split('\n'):
        # Dividimos al tablero la lista que resulta de dividir la
         ↪  subcadena i por el caracter de tabulación.
        tablero.append(i.split('\t'))
    # Abrimos el fichero en modo escritura
    f = open(nombre_fichero, 'w')
    # Bucle iterativo para recorrer las filas del tablero.
    # i toma como valores cada una de las listas (filas) del
     ↪  tablero.
    for i in tablero:
        # Escribirmos cada fila en una línea concatenando los
         ↪  caracteres que contiene.
        f.write('\t'.join(i) + '\n')
    # Cerramos el fichero.
    f.close()
    # Empieza la partida inicializando un contador de movimientos a
     ↪  cero.
    movimiento = 0
    # Bucle condicional para realizar movimientos en la partida
     ↪  hasta que el usario decida terminar.
    while True:
        # Preguntamos al usuario si quiere realizar otro movimiento.
        continuar = input('Deseas hacer otro movimiento (s/n): ')
        # Condicional para ver si el usuario ha respondido si.
        if continuar != 's':
            # Si el usuario no ha contestado que si, salimos del
             ↪  bucle condicional para terminar la partida.
            break
        else:
            # Si el usuario quiere hacer otro movimiento preguntamos
             ↪  por las coordenadas de las casillas de origen y
             ↪  destino.
            fila_origen = int(input('Introduce la fila de la pieza a
↪  mover: '))
            columna_origen = int(input('Introduce la columna de la
↪  pieza a mover: '))
            fila_destino = int(input('Introduce la fila de destino:
```

5.2.2. Tarea 2

```
def tablero(nombre_fichero, n):
    ''' Función que muestra por pantalla el tablero n de una partida
     ↪  de ajedrez.

    Parámetros:
        - nombre_fichero: Es una cadena con el nombre del fichero
 ↪  que contiene la sucesión de tableros de la partida de ajedrez.
        - n: Es un entero con el número del tablero que se tiene que
 ↪  mostrar.
    '''

    # Abrimos el fichero en modo lectura
    f = open(nombre_fichero, 'r')
    # Leemos el contenido del fichero en una cadena y la dividimos
     ↪  por el caracter de cambio de línea.
    tableros = f.read().split('\n')
    # Bucle para imprimir las lineas correspondientes al tablero.
     ↪  Cada tablero empieza siempre en una línea múltiplo de 9.
    for i in tableros[n*9:n*9+8]:
        print(i)
    return

tablero('partida1.txt', 2)
```

Abrir la solución en Google Colab

6. Laberinto

6.1. Tarea 1

La primera tarea consiste en construir un laberinto como el de la siguiente figura.

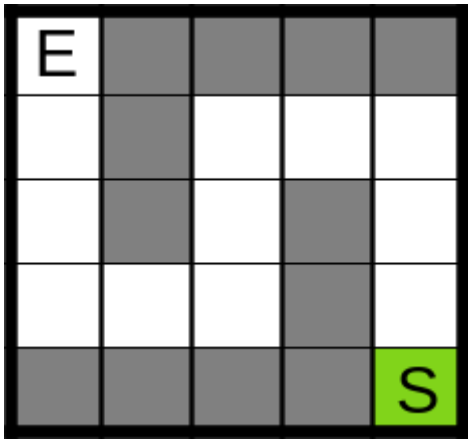

Figura 6.1.: Laberinto

El laberinto se representará como una una lista de listas, donde cada lista es una fila del laberinto y cada casilla se representará con un espacio ' ' si hay paso o con la letra 'X' si hay un muro, tal y como se muestra a continuación:

```
laberinto = [
    [' ', 'X', 'X', 'X', 'X'],
    [' ', 'X', ' ', ' ', ' '],
    [' ', 'X', ' ', 'X', ' '],
    [' ', ' ', ' ', 'X', ' '],
    ['X', 'X', 'X', 'X', 'S']
    ]
```

El laberinto se debe crear a partir de una tupla con las coordenadas de las casillas donde hay muro, como la siguiente:

```
muro = ((0,1), (0,2), (0,3), (0,4), (1,1), (2,1), (2,3), (3,3), (4,0),
 ↪  (4,1), (4,2), (4,3))
```

6.2. Tarea 2

La segunda tarea a resolver consiste en construir un programa para recorrer el laberinto desde la entrada a la salida. La entrada siempre está en la esquina superior izquierda y la salida en la esquina inferior derecha.

El programa debe devolver una lista con la secuencia de movimientos para ir de la entrada a la salida del laberinto, tal y como se muestra a continuación:

```
['Abajo', 'Abajo', 'Abajo', 'Abajo', 'Derecha', 'Derecha', 'Arriba',
 ↪  'Arriba', 'Derecha', 'Derecha', 'Abajo', 'Abajo', 'Abajo']
```

Solución

6.2.1. Tarea 1

```
def laberinto(dimension, muros):
    ''' Función que construye un laberinto cuadrado de una dimensión
     ↪  dada poniendo.

    Parámetros:
        - dimension: Es un entero con la dimensión del laberinto.
        - muros: Es una lista de tuplas con las coordenadas (x,y)
 ↪  donde hay muros.

    Salida:
        Una matriz (lista de listas) que representa el laberinto.
    '''

    # Creamos una lista vacía para añadir las filas del laberinto.
    laberinto = []
    # Bucle iterativo para añadir las filas del laberinto.
    # i toma valores de 0 a dimension-1
    for i in range(dimension):
        # Creamos una lista vacía para añadir las casillas de la
         ↪  fila.
        fila = []
        # Bucle iterativo para recorrer las columnas del laberinto.
        # j toma valores de 0 a dimension-1.
        for j in range(dimension):
            # Condicional para comprobar si la tupla está en el la
             ↪  lista de muros.
            if tuple([i, j]) in muro:
                # Si la tupla está en la lista de muros ponemos una
                 ↪  X en la casilla.
                fila.append('X')
            else:
                # Si la tupla no está en el muro ponemos un espacio
                 ↪  en blanco en la casilla.
                fila.append(' ')
        laberinto.append(fila)
    return laberinto

# Tupla de coordenadas de las celdas con muro en el laberinto
muro = ((0,1), (0,2), (0,3), (0,4), (1,1), (2,1), (2,3), (3,3),
 ↪  (4,0), (4,1), (4,2), (4,3))
lab = laberinto(5, muro)

# Mostrar el laberinto por pantalla
for i in lab:
    print(''.join(i))
```

6.2.2. Tarea 2

```
def recorre_laberinto(laberinto):
    '''Función que busca la salida de un laberinto.

    Parámetros:
        - laberinto: Es una matriz cuadrada (lista de listas) que
 ↪  representa el laberinto con el caracter X donde hay un muro.

    Salida:
        Una lista de tuplas con las coordenadas (x,y) de los pasos
 ↪  que hay que dar para recorrer el laberinto.
    '''
    # Fila y columnas iniciales
    fila = columna = 0
    # Lista de movimientos
    movimientos = ['Abajo']
    while (fila < n-1 and columna < n-1):
        if movimientos[-1] != 'Arriba' and fila + 1 < n and
         ↪  laberinto[fila + 1][columna] != 'X':
            fila += 1
            movimientos.append('Abajo')
        elif movimientos[-1] != 'Abajo' and fila - 1 > 0 and
         ↪  laberinto[fila - 1][columna] != 'X':
            fila -= 1
            movimientos.append('Arriba')
        elif movimientos[-1] != 'Izquierda' and columna + 1 < n and
         ↪  laberinto[fila][columna + 1] != 'X':
            columna += 1
            movimientos.append('Derecha')
        elif movimientos[-1] != 'Derecha' and columna - 1 > 0 and
         ↪  laberinto[fila][columna - 1] != 'X':
            columna -= 1
            movimientos.append('Izquierda')
        else:
            movimientos.append('No hay salida')
    return movimientos

# Mostrar por pantalla la lista de movimientos
print('Solución: ', recorre_laberinto(lab))
```

Abrir la solución en Google Colab

7. Estrellas regulares

7.1. Tarea

El reto consiste en programar una función que dibuje estrellas como la que aparece a continuación con un número de puntas dado.

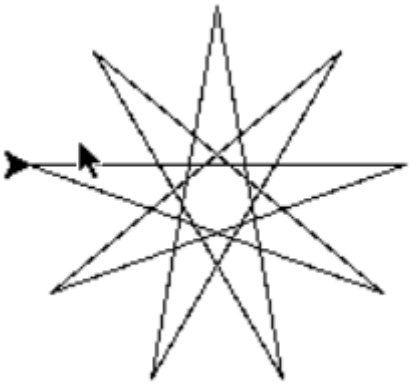

Figura 7.1.: Estrella

Para ello hay que utilizar el módulo de Python Turtle que permite realizar trazos sencillos en una ventana gráfica.

En el siguiente video hay una introducción básica al módulo Turtle.

Ver el vídeo de introducción a Turtle en YouTube

Y también dispones de una hoja de resumen con las principales funciones de Turtle.

Solución

```
def estrella(puntas, lado=150):
    ''' Función que dibuja una estrella regular de un número de
    ↪ puntas dado haciendo uso del módulo Turtle.

    Parámetros:
        - puntas: Es un entero con el número de puntas de la
↪   estrella.
        - lado: Es un entero con el tamaño de los lados de la
↪   estrella.
    '''

    import turtle

    # Función para calcular el máximo común divisor de dos números
    ↪ enteros por el algoritmo de euclides. Se necesita para
    ↪ calcular el ángulo de rotación.
    def mcd(a, b):
        while b != 0:
            a, b = b, a % b
        return a

    # Condicional para avisar de que el número de puntas debe ser
    ↪ mayor que 4. Si no no es posible dibujar la estrella.
    if puntas <= 4:
        print('El número de puntas es demasiado pequeño. Prueba de
        ↪ nuevo con un número de puntas mayor.')
        return
    # El ángulo de rotación en los vértices de la estrella se
    ↪ calcula en fución del primer número entero coprimo con el
    ↪ número de puntas
    # Bucle iterarivo para buscar el mayor coprimo con el número de
    ↪ puntas
    for i in range(puntas // 2, 1, -1):
        # Condicional para ver si i y el número de puntas son
        ↪ números coprimos
        if mcd(puntas, i) == 1:
            angle = 360.0 / puntas * i
            break
    # Bucle iterativo para dibujar los trazos de cada punta.
    for _ in range(puntas):
        # Dibujamos el lado
        turtle.forward(lado)
        # Rotamos el cursor
        turtle.left(angle)

    return

estrella(20)
```

Abrir la solución en Google Colab