0.00.5em

0.0.00.5em

0.0.0.00.5em

0em

,

Curso PIB-M 2023: Introducción a Python y a las librerías científicas

Angel Silva

Jose Luis Casado

September 15, 2023

Índice general

Capítulo 1

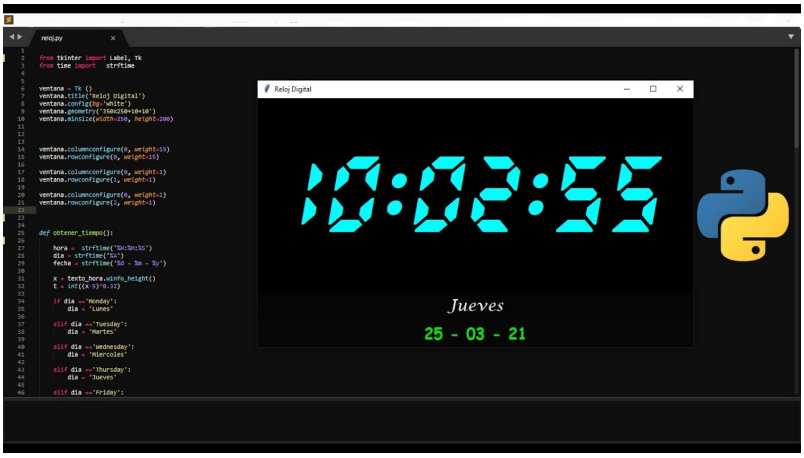
Introducción

Python es un **lenguaje sencillo de leer y escribir** debido a su alta similitud con el lenguaje humano. Además, se trata de un **lenguaje multiplataforma de código abierto** y, por lo tanto, gratuito, lo que permite desarrollar software sin límites. Con el paso del tiempo, **Python ha ido ganando adeptos gracias a su sencillez y a sus amplias posibilidades**, sobre todo en los últimos años, ya que facilita trabajar con inteligencia artificial, big data, machine learning y data science, entre muchos otros campos en auge.

1.1. ¿Por qué Python?

Nuestro motivo fundamental para aprender python:

Ahorrar tiempo



1.1.1. Ventajas

¿Qué características tiene?

- *Multi-paradigma*: permite crear scripts, programas con procedimientos, modular, orientación a objetos...
- *Interpretado*: no hace falta compilar
- Con variadas estructuras de datos
- Sintaxis fácil de entender
- *Multiplataforma*: independiente de la plataforma (Unix, Windows, MacOS), de código abierto, y gratis

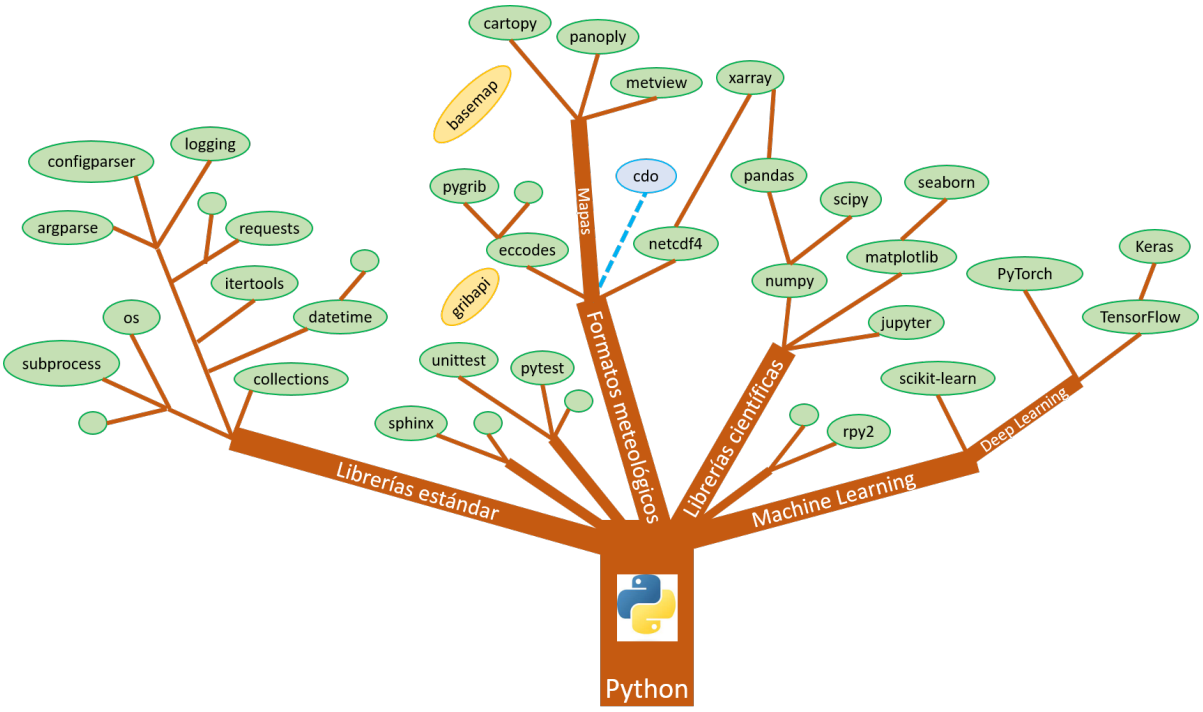
Ventajas

- *Fácil de leer*: tiene una sintaxis más sencilla que otros lenguajes, y al ser interpretado no hace falta compilar o enlazar.
- *Con una gran comunidad de usuarios*: hay miles de paquetes o librerías disponibles gratuitamente, y foros donde resolver tus dudas.
- Útil para muchos ámbitos: científico, técnico, diseño GUI, Web, bases de datos, etc.
- *Permite un flujo unificado*: no es necesario utilizar distintas aplicaciones para diversas tareas, con python se puede hacer prácticamente todo
- Es "open source"

Inconvenientes

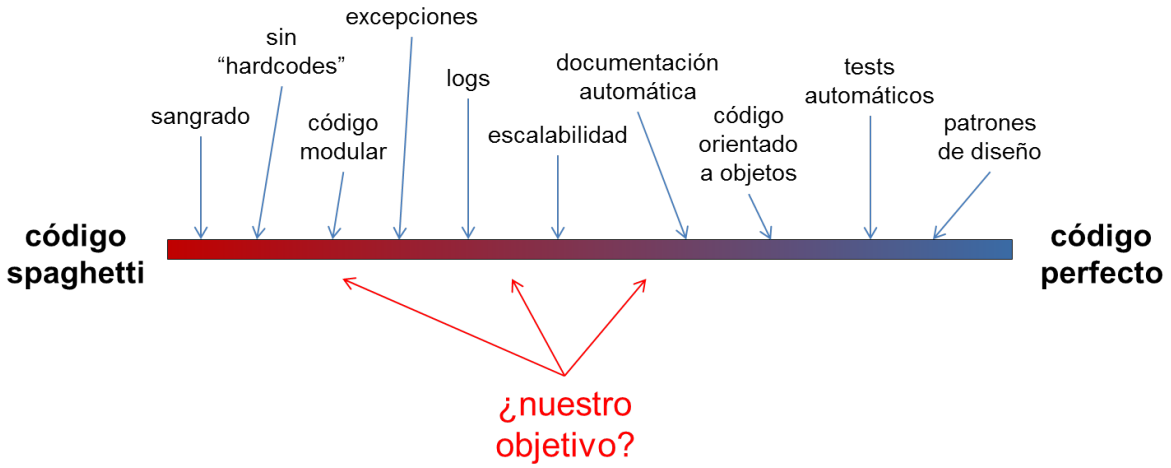
- Al ser interpretado, es más lento que Fortran o C. Esto se puede evitar utilizando paquetes científicos optimizados, escritos en lenguajes más rápidos.
- Menor soporte y documentación más dispersa

Este es un curso de 15 horas, por lo que sólo vamos a poder explicar los fundamentos de python, algunas de las librerías básicas, y varias librerías científicas. Pero el árbol de python es mucho más grande:



Usaremos Python versión 3

Nuestro objetivo será ver de qué manera este lenguaje de programación nos va a permitier evitar tareas rutinarias y así ganar tiempo. Además veremos que ha motivado que cada vez se use más en el mundo científico y en particular porqué en la meteorología nos supone una ventaja.



1.2. Python en meteorología

En meteorología Python nos aporta entre otras :

- Lectura y escritura de datos (GRIB, BUFR, NetCDF, ...)
- Cálculos meteorológicos
- Acceso a datos (API, THREDDS, OpenDAP, ERDDAP, ...)
- Manejo de grandes volúmenes de datos (xarray, ...)
- Integración con modelos meteorológicos (Weather Research and Forecasting model (WRF), wrf-python)
- Remapeo de rejillas
- Visualización
- Datos de satélite
- GIS

1.3. Historia

Python es un lenguaje de programación de propósito general, creado por Guido van Rossum, un informático holandés.

El nombre de Python es debido a que Guido van Rossum es un fan de la famosa serie británica *Monty Python Flying Circus*.

1.4. Consejos para programar

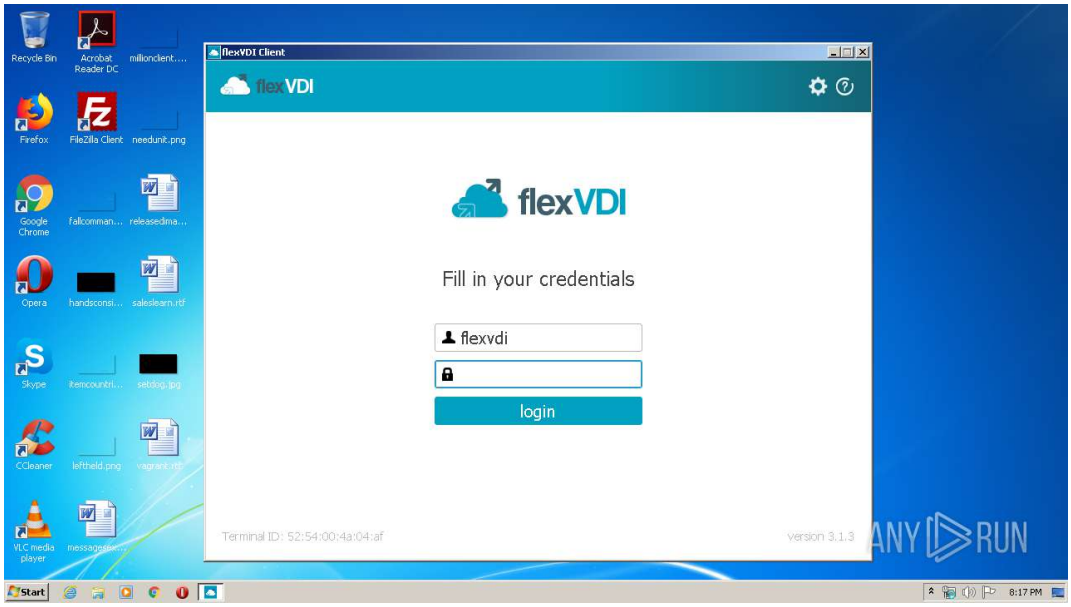
Un listado de consejos muy interesantes cuando nos enfrentamos a la programación, basados en la experiencia de @codewithvoid:

1. Escribir código es el último paso del proceso.
2. Para resolver problemas: pizarra mejor que teclado.
3. Escribir código sin planificar = estrés.

4. Pareces más inteligente siendo claro, no siendo listo.
5. La constancia a largo plazo es mejor que la intensidad a corto plazo.
6. La solución primero. La optimización después.
7. Gran parte de la programación es resolución de problemas.
8. Piensa en múltiples soluciones antes de decidirte por una.
9. Se aprende construyendo proyectos, no tomando cursos.
10. Siempre elije simplicidad. Las soluciones simples son más fáciles de escribir.
11. Los errores son inevitables al escribir código. Sólo te informan sobre lo que no debes hacer.
12. Fallar es barato en programación. Aprende mediante la práctica.
13. Gran parte de la programación es investigación.
14. La programación en pareja te enseñará mucho más que escribir código tu solo.
15. Da un paseo cuando estés bloqueado con un error.
16. Convierte en un hábito el hecho de pedir ayuda. Pierdes cero credibilidad pidiendo ayuda.
17. El tiempo gastado en entender el problema está bien invertido.
18. Cuando estés bloqueado con un problema: sé curioso, no te frustres.
19. Piensa en posibles escenarios y situaciones extremas antes de resolver el problema.
20. No te estreses con la sintaxis de lenguaje de programación. Entiende conceptos.
21. Aprende a ser un buen corrector de errores. Esto se amortiza.
22. Conoce pronto los atajos de teclado de tu editor favorito.
23. Tu código será tan claro como lo tengas en tu cabeza.
24. Gastarás el doble de tiempo en corregir errores que en escribir código.
25. Saber buscar bien en Google es una habilidad valiosa.
26. Lee código de otras personas para inspirarte.
27. Únete a comunidades de desarrollo para aprender con otros/as programadores/as.

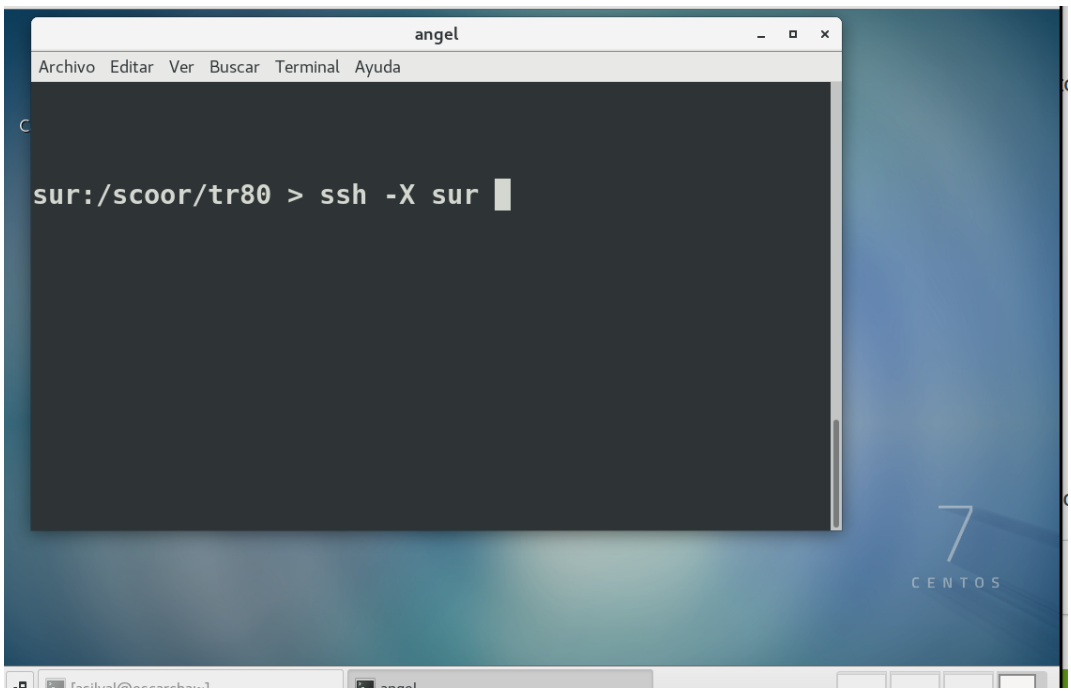
1.5. Usuarios y acceso.

Vamos hacer uso de la aplicación FlexVDI que es una plataforma de virtualización que usaremos para Escritorios Virtuales de Linux.



Una vez dentro del Escritorio virtual de Linux y abierta una terminal, usaremos el siguiente comando para acceder al servidor *sur*.

```
ssh -X sur
# o también podemos usar
ssh -X trXX@sur
```



Por último, vamos a realizar la carga de algunos módulos que nos permita disponer de **Python 3.10** :

```
# lo haremos ejecutando la siguiente orden.
module load aemet/5.0
```

Algunos comandos útiles en LINUX.

```
# Ir al Directorio HOME (inicial)
cd
# Ir al Directorio de descargas
```

```
cd Descargas
# Listar los directorios
ls -lrt
# Abrir un fichero con Gedit
gedit fichero.py
# Ejecutar un fichero en python
python fichero.py
# Abrir un notebook de jupyter
jupyter notebook
```

Capítulo 2

Tipos de datos

2.1. Diferentes tipos de datos

Los programas están formados por código y datos. Pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación, que almacena en la memoria no sólo el puro dato sino distintos metadatos.

Cada «trozo» de memoria contiene realmente un objeto, de ahí que se diga que en Python todo son objetos. Y cada objeto tiene, al menos, los siguientes campos:

- Un tipo del dato almacenado.
- Un identificador único para distinguirlo de otros objetos.
- Un valor consistente con su tipo.

A continuación se muestran los distintos tipos de datos que podemos encontrar en Python, sin incluir aquellos que proveen paquetes externos:

Tabla 1: Tipos de datos en Python

Nombre	Tipo	Ejemplos
Booleano	bool	`True`, `False`
Entero	int	`21`, `34500`, `34_500`
Flotante	float	`3.14`, `1.5e3`
Complejo	complex	`2j`, `3 + 5j`
Cadena	str	`'tfn'`, `'''tenerife - islas canarias'''`
Tupla	tuple	`(1, 3, 5)`
Lista	list	`['Chrome', 'Firefox']`
Conjunto	set	`set([2, 4, 6])`
Diccionario	dict	`{'Chrome': 'v79' , 'Firefox': 'v71'}`

2.1.1. Variables

Las variables son fundamentales ya que permiten definir nombres para los valores que tenemos en memoria y que vamos a usar en nuestro programa.

Figura 2: Uso de un *nombre* de variable

2.1.2. Reglas para nombrar variables

En Python existen una serie de reglas para los nombres de variables:

1. Sólo pueden contener los siguientes caracteres :
   - Letras minúsculas.
   - Letras mayúsculas.
   - Dígitos.
   - Guiones bajos (_).
2. Deben empezar con una letra o un guion bajo, nunca con un dígito.
3. No pueden ser una palabra reservada del lenguaje («keywords»).

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma:

```
>>> help( keywords )
Here is a list of the Python keywords. Enter any keyword to get more help.
False class from or None continue global pass True def if raise and del import return
as elif in try assert else is while async except lambda with await finally nonlocal yield
break for not
```

Importante: Los nombres de variables son «case-sensitive», se distinguen mayúsculas y minúsculas.

- *Por ejemplo*, stuff y Stuff son nombres diferentes.

2.1.3. Convenciones para nombres

snake_case en el que utilizamos caracteres en minúsculas (incluyendo dígitos si procede) junto con guiones bajos –

Constantes:

Un caso especial y que vale la pena destacar son las constantes. Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo la velocidad de la luz.

Asignación:

En Python se usa el símbolo = para asignar un valor a una variable:

Figura 3: Asignación de *valor* a *nombre* de variable

Python nos ofrece la posibilidad de hacer una asignación múltiple de la siguiente manera:

```
>>> tres = three = drei = 3
```

2.1.4. Asignando una variable a otra variable

Las asignaciones que hemos hecho hasta ahora han sido de un valor literal a una variable. Pero nada impide que podamos hacer asignaciones de una variable a otra variable:

```
>>> people = 157503
>>> total_population = people
>>> total_population
    157503
```

2.1.5. Conocer el valor de una variable

```
>>> final_stock = 38934
>>> final_stock
    38934
```

2.1.6. Conocer el tipo de una variable

Para poder descubrir el tipo de un literal o una variable, Python nos ofrece la función type(). Veamos algunos ejemplos de su uso:

```
>>> type(9)
    int
>>> type(1.2)
    float
>>> height = 3718
>>> type(height)
    int
>>> sound_speed = 343.2
>>> type(sound_speed)
```

Documentación oficial de Python: https://docs.python.org/es/3/library/functions.html?highlight=built

2.1.7. Pidiendo ayuda

En Python podemos pedir ayuda con la función help(). Supongamos que queremos obtener información sobre id. Desde el intérprete de Python ejecutamos lo siguiente:

```
>>> help(id)
    Help on built-in function id in module builtins:
    id(obj, /)
    Return the identity of an object.
    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object s memory address.)
    Existe una forma alternativa de obtener ayuda: añadiendo el signo de
    interrogación ? al término de búsqueda:
>>> id?
    Signature: id(obj, /)
    Docstring:
    Return the identity of an object.
    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object s memory address.)
    Type: builtin_function_or_method
```

2.1.8. Operadores Aritméticos

Entre los operadores aritméticos que Python utiliza, podemos encontrar los siguientes:

Símbolo	Significado	Ejemplo	Resultado
+	Suma	a = 10 + 5	a es 15
-	Resta	a = 12 - 7	a es 5
-	Negación	a = -5	a es -5
*	Multiplicación	a = 7 * 5	a es 35
**	Exponente	a = 2 ** 3	a es 8
/	División	a = 12.5 / 2	a es 6.25
//	División entera	a = 12.5 / 2	a es 60
%	Módulo	a = 27 % 4	a es 3

Importante : Con operadores siempre colocar un espacio en blanco, antes y después de un operador

Un *ejemplo* sencillo con variables y operadores aritméticos:

```
>>> monto_bruto = 175
>>> tasa_interes = 12
>>> monto_interes = monto_bruto \* tasa_interes / 100
>>> tasa_bonificacion = 5
>>> importe_bonificacion = monto_bruto * tasa_bonificacion / 100
>>> monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
```

2.1.9. Comentarios

Un archivo, no solo puede contener código fuente. También puede incluir comentarios(notas que como programadores, indicamos en el código para poder comprenderlo mejor).

Los comentarios pueden ser de dos tipos: de una sola línea o multi-línea y se expresan de la siguiente manera:

```
# Esto es un comentario de una sola línea
mi_variable = 15
"""Y este es un comentario
      de varias líneas"""
```

```
mi_variable = 15
mi_variable = 15 # Este comentario es de una línea también
```

En los comentarios, pueden incluirse palabras que nos ayuden a identificar además, el subtipo de comentario:

```
# TODO esto es algo por hacer
# FIXME esto es algo que debe corregirse
# XXX esto también, es algo que debe corregirse
```

2.2. Estructuras de datos

Podríamos hablar de tipos de datos más complejos en Python que se constituyen en estructuras de datos. Si pensamos en estos elementos como átomos, las estructuras de datos que vamos a ver sería moléculas. Es decir, combinamos los tipos básicos de formas más complejas.

2.2.1. Listas

Las listas permiten **almacenar objetos** mediante un orden definido y con posibilidad de duplicados, son además mutables, lo que significa que podemos añadir, eliminar o modificar sus elementos.

Creando listas

```
>>> empty_list = []
>>> languages = [ "Python" , "Ruby" , "Javascript" ]
>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]
>>> data = [ "Tenerife" , { "cielo" : "limpio" , "temp" : 24},
            3718, (28.2933947, -16.5226597)]
```

Importante : Aunque está permitido, NUNCA llames list a una variable porque destruirías la función que nos permite crear listas.

Ejercicio

Cree una lista con las 5 tipos de nubes que le gusten.

Conversión

Para convertir otros tipos de datos en una lista podemos usar la función list():

```
>>> # conversión desde una cadena de texto
>>> list( "Python" )
    [ P , y , t , h , o , n ]
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un carácter de la cadena. Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado (iterables).

Otro ejemplo interesante de conversión puede ser la de los rangos. En este caso queremos obtener una lista explícita con los valores que constituyen el rango [0, 9]:

```
>>> list(range(10))
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Operaciones con listas

Obtener un elemento Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del índice (lugar) que ocupa. Veamos un ejemplo:

```
>>> shopping = [ Agua , Huevos , Aceite ]
>>> shopping[0]
    Agua
>>> shopping[1]
    Huevos
>>> shopping[2]
    Aceite
>>> shopping[-1] # acceso con índice negativo
    Aceite
```

Trocear una lista El troceado de listas funciona de manera totalmente análoga al troceado de cadenas. Veamos algunos ejemplos:

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> shopping[0:3]
    [ Agua , Huevos , Aceite ]
>>> shopping[:3]
    [ Agua , Huevos , Aceite ]
>>> shopping[2:4]
    [ Aceite , Sal ]
>>> shopping[-1:-4:-1]
    [ Limón , Sal , Aceite ]
>>> # Equivale a invertir la lista
>>> shopping[::-1]
    [ Limón , Sal , Aceite , Huevos , Agua ]
```

Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

Conservando la lista original: Opción 1: Mediante troceado de listas con step negativo:

```
>>> shopping
    [ Agua , Huevos , Aceite , Sal , Limón ]
>>> shopping[::-1]
    [ Limón , Sal , Aceite , Huevos , Agua ]
```

Opción 2: Mediante la función reversed():

```
>>> shopping
    [ Agua , Huevos , Aceite , Sal , Limón ]
>>> list(reversed(shopping))
    [ Limón , Sal , Aceite , Huevos , Agua ]
```

Modificando la lista original: Utilizando la función reverse() (nótese que es sin «d» al final):

```
>>> shopping
    [ Agua , Huevos , Aceite , Sal , Limón ]
>>> shopping.reverse()
>>> shopping
    [ Limón , Sal , Aceite , Huevos , Agua ]
```

Añadir al final de la lista

```
>>> shopping
>>> shopping = [ Agua , Huevos , Aceite ]
>>> shopping.append( Atún )
>>> shopping
    [ Agua , Huevos , Aceite , Atún ]
```

Añadir en cualquier posición de una lista

```
>>> shopping = [ Agua , Huevos , Aceite ]
>>> shopping.insert(1, Jamón )
>>> shopping
    [ Agua , Jamón , Huevos , Aceite ]
>>> shopping.insert(3, Queso )
>>> shopping
    [ Agua , Jamón , Huevos , Queso , Aceite ]
```

Nota: El índice que especificamos en la función insert() lo podemos interpretar como la posición delante (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

Modificar una lista

Del mismo modo que se accede a un elemento utilizando su índice, también podemos modificarlo:

```
>>> shopping = [ Agua , Huevos , Aceite ]
>>> shopping[0]
    Agua
>>> shopping[0] = Jugo
>>> shopping
    [ Jugo , Huevos , Aceite ]
```

En el caso de acceder a un índice no válido de la lista, incluso para modificar, obtendremos un error:

```
>>> shopping[100] = Chocolate
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    IndexError: list assignment index out of range
```

Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

Por su índice: Mediante la sentencia del:

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> del shopping[3]
>>> shopping
    [ Agua , Huevos , Aceite , Limón ]
```

Por su valor: Mediante la función remove():

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> shopping.remove( Sal )
>>> shopping
    [ Agua , Huevos , Aceite , Limón ]
```

Advertencia: Si existen valores duplicados, la función remove() sólo borrará la primera ocurrencia.

Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

Conservando lista original: Mediante la función sorted() que devuelve una nueva lista ordenada:

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> sorted(shopping)
    [ Aceite , Agua , Huevos , Limón , Sal ]
```

Modificando la lista original: Mediante la función sort():

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> shopping.sort()
>>> shopping
    [ Aceite , Agua , Huevos , Limón , Sal ]
```

Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función len():

```
>>> shopping = [ Agua , Huevos , Aceite , Sal , Limón ]
>>> len(shopping)
    5
```

2.2.2. Tuplas

El concepto de tupla es muy similar al de lista. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una lista es mutable y se puede modificar, una tupla no admite cambios y por lo tanto, es inmutable.

Creando tuplas

Podemos pensar en crear tuplas tal y como lo hacíamos con listas, pero usando paréntesis en lugar de corchetes:

```
>>> empty_tuple = ()
>>> tenerife_geoloc = (28.46824, -16.25462)
>>> three_wise_men = ( Melchor , Gaspar , Baltasar )
```

Tuplas de un elemento

Hay que prestar especial atención cuando vamos a crear una tupla de un único elemento. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ( Papá Noel )
>>> one_item_tuple
    Papá Noel
>>> type(one_item_tuple)
    str
```

Realmente, hemos creado una variable de tipo str (cadena de texto). Para crear una tupla de un elemento debemos añadir una coma al final:

```
>>> one_item_tuple = ( Papá Noel ,)
>>> one_item_tuple
    ( Papá Noel ,)
>>> type(one_item_tuple)
    tuple
```

Modificar una tupla

Como ya hemos comentado previamente, las tuplas son estructuras de datos inmutables. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

```
>>> three_wise_men = Melchor , Gaspar , Baltasar
>>> three_wise_men[0] = Tom Hanks
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    TypeError: tuple object does not support item assignment
```

Operaciones con tuplas

Con las tuplas podemos realizar todas las operaciones que vimos con listas salvo las que conlleven una modificación «in-situ» de la misma:

- reverse()
- append()
- extend()

- remove()
- clear()
- sort()

Truco: Sí es posible aplicar sorted() o reversed() sobre una tupla ya que no estamos modificando su valor sino creando un nuevo objeto.

Tuplas contra Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

1. Las tuplas ocupan menos espacio en memoria.
2. En las tuplas existe protección frente a cambios indeseados.
3. Las tuplas se pueden usar como claves de diccionarios (son «hashables»).

2.2.3. Diccionarios

Diríamos que en Python un diccionario es también un objeto indexado por claves (las palabras) que tienen asociados unos valores (los significados). Los diccionarios en Python tienen las siguientes características:

- *Mantienen el orden* en el que se insertan las claves.
- *Son mutables*, con lo que admiten añadir, borrar y modificar sus elementos.
- Las *claves deben ser únicas*. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un *acceso muy rápido a sus elementos*, debido a la forma en la que están implementados internamente.

Creando diccionarios

Para crear un diccionario usamos llaves {} rodeando asignaciones clave: valor que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
>>> empty_dict = {}
>>> rae = {
...  bifronte : De dos frentes o dos caras ,
...  anarcoide : Que tiende al desorden ,
...  montuvio : Campesino de la costa
...  }
>>> population_can = {
...  2015: 2_135_209,
...  2016: 2_154_924,
...  2017: 2_177_048,
...  2018: 2_206_901,
...  2019: 2_220_270
...  }
```

Ejercicio

Cree un diccionario con los nombres de 3 países y su capital.
Solución –> `solucion_dicc.py`

Es posible crear un diccionario especificando sus claves y un único valor de «relleno»:

```
>>> dict.fromkeys( aeiou , 0)
    { a : 0, e : 0, i : 0, o : 0, u : 0}
```

Conversión

Para convertir otros tipos de datos en un diccionario podemos usar la función dict():

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict([ a1 , b2 ])
    { a : 1 , b : 2 }
>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(( a1 , b2 ))
    { a : 1 , b : 2 }
>>> # Diccionario a partir de una lista de listas
>>> dict([[ a , 1], [ b , 2]])
    { a : 1, b : 2}
```

Operaciones con diccionarios

Obtener un elemento Para obtener un elemento de un diccionario basta con escribir la clave entre corchetes.

Veamos un ejemplo:

```
>>> rae = {
... bifronte : De dos frentes o dos caras ,
... anarcoide : Que tiende al desorden ,
... montuvio : Campesino de la costa
... }
>>> rae[ anarcoide ]
```

Que tiende al desorden

Si intentamos acceder a una clave que no existe, obtendremos un error:

```
>>> rae[ acceso ]
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    KeyError: acceso
```

Usando get()

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de get() y su comportamiento es el siguiente:

1. Si la clave que buscamos existe, nos devuelve su valor.
2. Si la clave que buscamos no existe, nos devuelve None salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

```
>>> rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa }
```

```
>>> rae.get( bifronte )
    De dos frentes o dos caras

>>> rae.get( programación )
>>> rae.get( programación , No disponible )
    No disponible
```

Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor: * Si la clave ya existía en el diccionario, se reemplaza el valor existente por el nuevo. * Si la clave es nueva, se añade al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

```
>>> rae
>>> rae = {
... bifronte : De dos frentes o dos caras ,
... anarcoide : Que tiende al desorden ,
... montuvio : Campesino de la costa
... }
>>> rae[ enjuiciar ] = Someter una cuestión a examen, discusión y juicio
>>> rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa ,
    enjuiciar : Someter una cuestión a examen, discusión y juicio }
>>> rae[ enjuiciar ] = Instruir, juzgar o sentenciar una causa
>>> rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa ,
    enjuiciar : Instruir, juzgar o sentenciar una causa }
```

Pertenencia de una clave

La forma con Python de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador in:

```
>>> bifronte in rae
    True
>>> almohada in rae
    False
>>> montuvio not in rae
    False
```

Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> bifronte in rae
>>> rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa ,
    enjuiciar : Instruir, juzgar o sentenciar una causa }
>>> # Obtener todas las claves de un diccionario: Mediante la función keys():
>>> rae.keys()
    dict_keys([ bifronte , anarcoide , montuvio , enjuiciar ])
>>> # Obtener todos los valores de un diccionario: Mediante la función values():
>>> rae.values()
    dict_values([
    De dos frentes o dos caras ,
    Que tiende al desorden ,
    Campesino de la costa ,
    Instruir, juzgar o sentenciar una causa
```

```
    ])
>>> # Obtener todos los pares «clave-valor» de un diccionario:
>>> # Mediante la función items():
>>> rae.items()
    dict_items([
    ( bifronte , De dos frentes o dos caras ),
    ( anarcoide , Que tiende al desorden ),
    ( montuvio , Campesino de la costa ),
    ( enjuiciar , Instruir, juzgar o sentenciar una causa )
    ])
```

Longitud de un diccionario

```
>>> rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa ,
    enjuiciar : Instruir, juzgar o sentenciar una causa }
>>> len(rae)
    4
```

Borrar elementos

Una forma para borrar elementos en un diccionario:

Por su clave: Mediante la sentencia del:

```
>>> rae
>>> rae = {
... bifronte : De dos frentes o dos caras ,
... anarcoide : Que tiende al desorden ,
... montuvio : Campesino de la costa
... }
>>> del rae[ bifronte ]
>>> rae
{ anarcoide : Que tiende al desorden , montuvio : Campesino de la costa }
```

Cuidado con las copias

```
>>> original_rae = {
... bifronte : De dos frentes o dos caras ,
... anarcoide : Que tiende al desorden ,
... montuvio : Campesino de la costa
... }
>>> copy_rae = original_rae
>>> original_rae[ bifronte ] = bla bla bla
>>> original_rae
    { bifronte : bla bla bla ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa }
>>> copy_rae
    { bifronte : bla bla bla ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa }
```

Una posible solución a este problema es hacer una «copia dura». Para ello Python proporciona la función copy():

```
>>> original_rae = {
... bifronte : De dos frentes o dos caras ,
... anarcoide : Que tiende al desorden ,
... montuvio : Campesino de la costa
... }
>>> copy_rae = original_rae.copy()
>>> original_rae[ bifronte ] = bla bla bla
```

```
>>> original_rae
    { bifronte : bla bla bla ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa }
>>> copy_rae
    { bifronte : De dos frentes o dos caras ,
    anarcoide : Que tiende al desorden ,
    montuvio : Campesino de la costa }
```

2.2.4. Conjuntos

Un conjunto en Python representa una serie de valores únicos y sin orden establecido, con la única restricción de que sus elementos deben ser «hashables». Mantiene muchas similitudes con el concepto matemático de conjunto.

```
>>> original_rae = {
>>> lottery = {21, 10, 46, 29, 31, 94}
>>> lottery
    {10, 21, 29, 31, 46, 94}
```

2.3. Cadenas de texto

Las cadenas de texto son secuencias de caracteres. También se les conoce como «strings» y nos permiten almacenar información textual de forma muy cómoda. Python 3 codifica todos los caracteres en *Unicode*.

2.3.1. Creando cadenas de texto (strings)

```
>>> 'Mi primera cadena con Python 3'
     Mi primera cadena con Python 3
>>> 'Los llamados "strings" son secuencias de caracteres'
     Los llamados "strings" son secuencias de caracteres
>>> poem = To be, or not to be, that is the question:
... Whether tis nobler in the mind to suffer
... The slings and arrows of outrageous fortune,
... Or to take arms against a sea of troubles

>>> # Podemos convertir un número a string con la función str()
>>> str(10)
    '10'
>>> str(21.7)
    '21.7'

>>> # Salto de línea
>>> msg = Primera línea\nSegunda línea\nTercera línea
>>> print(msg)
    Primera línea
    Segunda línea
    Tercera línea

# Tabulador
>>> msg = Valor = \t40
>>> print(msg)
    Valor = 40

# Comilla simple
>>> msg = Necesitamos \escapar\ la comilla simple
```

```
>>> print(msg)
    Necesitamos escapar la comilla simple

# Barra invertida
>>> msg = Capítulo \\ Sección \\ Encabezado
>>> print(msg)
    Capítulo \ Sección \ Encabezado
```

2.3.2. Operaciones con cadenas de texto

```
>>> # Combinar cadenas
>>> proverb1 = Cuando el río suena
>>> proverb2 = agua lleva
>>> proverb1 + proverb2
    Cuando el río suenaagua lleva
>>> proverb1 + , + proverb2 # incluimos una coma
    Cuando el río suena, agua lleva

>>> # Repetir cadenas.
>>> reaction = Wow
>>> reaction * 4
    WowWowWowWow

>>> # Obtener un carácter: el indexado de una cadena de texto siempre empieza en 0.
>>> sentence = Hola, Mundo
>>> sentence[0]
    H
>>> sentence[-1]
    o
>>> sentence[4]
    ,
>>> sentence[-5]
    M

>>> # Obtener una parte de la cadena.
>>> proverb = Agua pasada no mueve molino
>>> proverb[:]
    Agua pasada no mueve molino
>>> proverb[12:]
    no mueve molino
>>> proverb[:11]
    Agua pasada
>>> proverb[5:11]
    pasada
>>> proverb[5:11:2]
    psd
```

Longitud de una cadena

```
>>> proverb = "En abril aguas mil"
>>> len(proverb)
    18
>>> empty = ""
>>> len(empty)
    0
```

Pertenencia de un elemento

```
>>> proverb = 'Hasta el cuarenta de mayo no te quites el sayo'
>>> mayo in proverb
```

```
    True
>>> sayo in proverb
    True
>>> pones in proverb
    False
```

2.3.3. Sobre print()

Algunos parámetros interesantes:

```
>>> # Podemos imprimir todas las variables que queramos separándolas por comas.
>>> msg1 = ¿Sabes por qué estoy acá?
>>> msg2 = Porque me apasiona
>>> print(msg1, msg2)
    ¿Sabes por qué estoy acá? Porque me apasiona

>>> # El separador por defecto entre las variables es un espacio, podemos cambiar
>>> # el carácter que se utiliza como separador entre cadenas.
>>> print(msg1, msg2, sep= | )
    ¿Sabes por qué estoy acá?|Porque me apasiona

>>> # El carácter de final de texto es un salto de línea, podemos cambiar
>>> # el carácter que se utiliza como final de texto.
>>> print(msg2, end= !! )
    Porque me apasiona!!
```

2.4. Leer datos desde teclado

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función input():

```
>>> # Podemos imprimir todas las variables que queramos separándolas por comas.
>>> nombre = input( Introduzca su nombre: )
    Introduzca su nombre: Ángel
>>> nombre
    Ángel
>>> type(nombre)
    str
>>> edad = input( Introduzca su edad: )
    Introduzca su edad: 41
>>> edad
    41
>>> type(edad)
    str
```

Ejercicio

Escriba un programa en Python que lea por teclado dos números enteros y muestre por pantalla el resultado de realizar las operaciones básicas entre ellos.

Capítulo 3

Control de flujo

Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada. En este capítulo, hablaremos sobre dos estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas

3.1. Sangría *´indentación´*

Una "indentación" o sangrado de 4 (cuatro) espacios en blanco, indicará que las instrucciones sangradas *"indentadas"*, forman parte de una misma estructura de control.

No todos los lenguajes de programación, necesitan de un sangrado o una *"indentación"*, aunque sí, se estila implementarla, a fin de otorgar mayor legibilidad al código fuente. Pero en el caso de Python, la indentación o sangrado es *obligatorio*.

3.2. Codificación "Encoding"

El *"encoding"* (o codificación) es otro de los elementos del lenguaje que no puede omitirse a la hora de hablar de estructuras de control.

El *"encoding"* o codificación no es más que una directiva que se coloca al inicio de un archivo Python, a fin de indicar al sistema, la codificación de caracteres utilizada en el archivo.

```
# -- coding: utf-8 --
```

"utf-8" podría ser cualquier codificación de caracteres. Si no se indica una codificación de caracteres, Python podría producir un error si encontrara caracteres "extraños":

```
print "En el Ñágara encontré un Ñandú"
```

Producirá un error de sintaxis:

```
SyntaxError: Non-ASCII character[...]
```

En cambio, indicando el *"encoding"* correspondiente, el archivo se ejecutará con éxito:

Ejemplo

-- *coding: utf-8* --
print "En el Ñágara encontré un Ñandú"
Produciendo la siguiente salida: # En el Ñágara encontré un Ñandú

3.3. Asignación múltiple

Otra de las ventajas que Python nos provee, es la de poder asignar en una sola instrucción, múltiples variables.

En una sola instrucción, estamos declarando tres variables: a, b y c y asignándoles un valor concreto a cada una:

```
>>> # Estamos declarando tres variables: a, b y c y asignándoles un valor concreto
>>> # a cada una.
>>> a, b, c = 'string', 15, True

>>> # Podemos imprimir todas las variables que queramos separándolas por comas.
>>> print a
    string
>>> print b
    15
>>> print c
    True

>>> # La asignación múltiple de variables, también puede darse utilizando como
>>> # valores, el contenido de una tupla:
>>> mi_tupla = ('hola mundo', 2011)
>>> texto, anio = mi_tupla
>>> print texto
    hola mundo
>>> print anio
    2011

>>> # O también, de una lista:
>>> mi_lista = ['Argentina', 'Buenos Aires']
>>> pais, provincia = mi_lista
>>> print pais
    Argentina
>>> print provincia
    Buenos Aires
```

3.4. Condicionales

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. La evaluación de condiciones, solo puede arrojar 1 de 2 resultados: verdadero o falso (True o False).

Para describir la evaluación a realizar sobre una condición, se utilizan operadores relacionales (o de comparación):

OPERADORES RELACIONALES (DE COMPARACIÓN)

Símbolo	Significado	Ejemplo	Resultado
==	Igual que	5 == 7	Falso
!=	Distinto que	rojo != verde	Verdadero
<	Menor que	8 < 12	Verdadero
>	Mayor que	12 > 7	Falso
<=	Menor o igual que	12 <= 12	Verdadero
>=	Mayor o igual que	4 >= 5	Falso

Y para evaluar más de una condición simultáneamente, se utilizan operadores lógicos:

OPERADORES LÓGICOS

Operador	Ejemplo	Resultado*	
and (y)	5 == 7 and 7 < 12	0 y 0	Falso
	9 < 12 and 12 > 7	1 y 1	Verdadero
	9 < 12 and 12 > 15	1 y 0	Falso
or (o)	12 == 12 or 15 < 7	1 o 0	Verdadero
	7 > 5 or 9 < 12	1 o 1	Verdadero
xor	4 == 4 xor 9 > 3	1 o 1	Falso
(o excluyente)	4 == 4 xor 9 < 3	1 o 0	Verdadero

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje: if (si), elif (sino, sí) y else (sino).

```
>>> semaforo = "verde"
>>> if semaforo == "verde":
...     print "Cruzar la calle"
... else:
...     print "Esperar"
...

>>> compra = 200
>>> if compra <= 100:
...     print "Pago en efectivo"
... elif compra > 100 and compra < 300:
...     print "Pago con tarjeta de débito"
... else:
...     print "Pago con tarjeta de crédito"
...

>>> # Si la compra es mayor a $100, obtengo un descuento del 10%
>>> importe_a_pagar = total_compra

>>> if total_compra > 100:
...     tasa_descuento = 10
...
>>> importe_descuento = total_compra * tasa_descuento / 100
>>> importe_a_pagar = total_compra - importe_descuento
```

3.5. Ejercicios con condicionales

Ejercicio 1

Escribir un programa que pregunte al usuario su edad y muestre por pantalla si es mayor de edad o no.

Ejercicio 2

Escribir un programa que almacene la cadena de caracteres contraseña en una variable, pregunte al usuario por la contraseña e imprima por pantalla si la contraseña introducida por el usuario coincide con la guardada en la variable sin tener en cuenta mayúscula y minúscula.

Ejercicio 3

Escribir un programa que pida al usuario dos números y muestre por pantalla su división. Si el divisor es cero el programa debe mostrar un error.

Ejercicio 4

Escribir un programa que pida al usuario un número entero y muestre por pantalla si es par o impar.

Ejercicio 5

Para tributar un determinado impuesto se debe ser mayor de 16 años y tener unos ingresos iguales o superiores a 1000 € mensuales. Escribir un programa que pregunte al usuario su edad y sus ingresos mensuales y muestre por pantalla si el usuario tiene que tributar o no.

Ejercicio 6

Los alumnos de un curso se han dividido en dos grupos A y B de acuerdo al sexo y el nombre. El grupo A está formado por las mujeres con un nombre anterior a la M y los hombres con un nombre posterior a la N y el grupo B por el resto. Escribir un programa que pregunte al usuario su nombre y sexo, y muestre por pantalla el grupo que le corresponde.

Ejercicio 7

Los tramos impositivos para la declaración de la renta en un determinado país son los siguientes: Escribir un programa que pregunte al usuario su renta anual y muestre por pantalla el tipo impositivo que le corresponde.

Renta	Tipo impositivo
Menos de 10000€	5 %
Entre 10000€ y 20000€	15 %
Entre 20000€ y 35000€	20 %
Entre 35000€ y 60000€	30 %
Más de 60000€	45 %

Ejercicio 8

En una determinada empresa, sus empleados son evaluados al final de cada año. Los puntos que pueden obtener en la evaluación comienzan en 0.0 y pueden ir aumentando, traduciéndose en mejores beneficios. Los puntos que pueden conseguir los empleados pueden ser 0.0, 0.4, 0.6 o más, pero no valores intermedios entre las cifras mencionadas. A continuación se muestra una tabla con los niveles correspondientes a cada puntuación. La cantidad de dinero conseguida en cada nivel es de 2.400€ multiplicada por la puntuación del nivel.
Escribir un programa que lea la puntuación del usuario e indique su nivel de rendimiento, así como la cantidad de dinero que recibirá el usuario.

Nivel	Puntuación
Inaceptable	0.0
Aceptable	0.4
Meritorio	0.6 o más

Ejercicio 9

Escribir un programa para una empresa que tiene salas de juegos para todas las edades y quiere calcular de forma automática el precio que debe cobrar a sus clientes por entrar. El programa debe preguntar al usuario la edad del cliente y mostrar el precio de la entrada. Si el cliente es menor de 4 años puede entrar gratis, si tiene entre 4 y 18 años debe pagar 5€ y si es mayor de 18 años, 10€.

Ejercicio 10

La pizzería "Bella Napoli" ofrece pizzas vegetarianas y no vegetarianas a sus clientes. Los ingredientes para cada tipo de pizza aparecen a continuación.

- Ingredientes vegetarianos: Pimiento y tofu.
- Ingredientes no vegetarianos: Peperoni, Jamón y Salmón.

Escribir un programa que pregunte al usuario si quiere una pizza vegetariana o no, y en función de su respuesta le muestre un menú con los ingredientes disponibles para que elija. Solo se puede elegir un ingrediente además de la mozzarella y el tomate que están en todas la pizzas. Al final se debe mostrar por pantalla si la pizza elegida es vegetariana o no y todos los ingredientes que lleva.

3.6. Bucles while y for

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

En Python se dispone de dos estructuras cíclicas:

- El bucle while
- El bucle for

3.6.1. Bucle while

Este bucle, se encarga de ejecutar una misma acción "mientras que" una determinada condición se cumpla:

Ejemplo

Mientras que año sea menor o igual a 2012, imprimir la frase: “Informes del Año año”:
# – coding: utf-8 –
anio = 2001
while anio <= 2012: print(“Informes del Año”, str(anio)) anio += 1

Aquí vemos cómo *parar* o *continuar* con el bucle:

```
>>> MAX_GREETS = 4
>>> num_greets = 0
>>> want_greet = "S"
>>> while want_greet == "S" :
...     print( "Hola qué tal!" )
...     num_greets += 1
...     if num_greets == MAX_GREETS:
...             print( "Máximo número de saludos alcanzado" )
...             break
...     want_greet = input( "¿Quiere otro saludo? [S/N]" )
... print( "Que tenga un buen día" )
...

>>> # Se prodŕia hacer también así
>>> while want_greet == "S" and num_questions < MAX_GREETS:
...     print( "Hola qué tal!" )
...

>>> # COMPROBAR LA RUPTURA
>>> # Si el bucle while finaliza normalmente (sin llamada a break)
>>> #    el flujo de control pasa a la sentencia opcional else.
>>> MAX_GREETS = 4
>>> num_greets = 0
>>> want_greet = "S"
>>> while want_greet == "S" :
...     print( "Hola qué tal!" )
...     num_greets += 1
...     if num_greets == MAX_GREETS:
...             print( "Máximo número de saludos alcanzado" )
...             break
...     want_greet = input( "¿Quiere otro saludo? [S/N]" )
... else:
...     print( "Usted no quiere más saludos" )
... print( "Que tenga un buen día" )

>>> # Hay situaciones en las que, en vez de romper un bucle,
>>> #     nos interesa saltar adelante hacia la siguiente repetición.
>>> want_greet = "S"
>>> valid_options = 0
>>> while want_greet == "S" :
...     print( "Hola qué tal!" )
...     want_greet = input( "¿Quiere otro saludo? [S/N]" )
...     if want_greet not in "SN" :
...             print( "No le he entendido pero le saludo" )
...             want_greet = "S"
...             continue
...     valid_options += 1
... print(f'{valid_options} respuestas válidas' )
... print( "Que tenga un buen día" )
```

Ejemplo

El bucle no tiene que ser numérico, y se puede detener de otra forma, veamos el siguiente ejemplo donde tenemos que introducir un nombre.
– coding: utf-8 –
while True: nombre = raw_input("Indique su nombre: ") if nombre:
break
print("Su nombre es :" , nombre)

El bucle anterior, incluye un condicional anidado que verifica si la variable nombre es verdadera (solo será verdadera si el usuario escribe un texto en pantalla cuando el nombre le es solicitado). Si es verdadera, el bucle para (break). Sino, seguirá ejecutándose hasta que el usuario, ingrese un texto en pantalla.""

Ejercicio

Escriba un programa que encuentre todos los múltiplos de 5 menores que un valor dado:

Bucle infinito

Imaginemos que queremos escribir un programa que ayude a un observador en meteorología a introducir las temperaturas. Si la temperatura no está en el intervalo [-50, 50] mostramos un mensaje de error, en otro caso seguimos pidiendo valores:

```
>>> while True:
...     mark = float(input( "Introduzca una nueva temperatura" ))
...     if not(-50 <= mark <= 50):
...             print( "Temperatura fuera de rango" )
...             break
```

3.6.2. Bucle For

El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

Por cada nombre en mi_lista, imprimir nombre

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
      print nombre
```

Por cada color en mi_tupla, imprimir color

```
mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')
for color in mi_tupla:
       print color
```

En los ejemplos anteriores, nombre y color, son dos variables declaradas en tiempo de ejecución (es decir, se declaran dinámicamente durante el bucle), asumiendo como valor, el de cada elemento de la lista (o tupla) en cada iteración.

Ejercicio

Dada una cadena de texto, indique el número de vocales que tiene.
Ejemplo

- Entrada: "Una nube es un hidrometeoro consistente en diminutas partículas de agua líquida o hielo, o de ambos, suspendidas en la atmósfera y que, por lo general, no tocan el suelo."
- Salida: 61

3.6.3. Sintaxis de Try y Except

La sintaxis básica es la siguiente:

```
try:
        # Código a ejecutar
        # Pero podría haber errores en este bloque

except <tipo de error>:
        # Haz esto para manejar la excepción
        # El bloque except se ejecutara si el bloque try lanza un error

else:
        # Esto se ejecutara si el bloque try se ejecuta sin errores

finally:
        # Este bloque se ejecutara siempre
```

Veamos el uso de cada uno de estos bloques:

- **EL bloque try** es el bloque con las sentencias que quieres ejecutar. Sin embargo, podría llegar a haber errores de ejecución y el bloque se dejará de ejecutarse.
- **El bloque except** se ejecutará cuando el bloque try falle debido a un error. Este bloque contiene sentencias que generalmente nos dan un contexto de lo que salió mal en el bloque try.
- Siempre deberías de mencionar el **tipo de error** que se espera, como una excepción dentro del bloque except dentro de <tipo de error> como lo muestra el ejemplo anterior. Algunos tipos de error pueden ser *´IndexError´, ´KeyError´ y ´FileNotFoundError´*. Y tendrías que añadir un bloque except por cada tipo de error que puedas anticipar, A continuación mencionare algunas:
  - **ImportError**: falla una importación;
  - **IndexError**: una lista es indexada con un número fuera de rango;
  - **NameError**: una variable desconocida es utilizada;
  - **SyntaxError**: el código no puede ser analizado correctamente;
  - **TypeError**: una función es llamada con un valor de un tipo inapropiado;
  - **ValueError**: Una función es llamada con un valor de tipo correcto pero con un valor incorrecto.
- Podrías usar except sin especificar el <tipo de error>. Pero no es una práctica recomendable, ya que no estarás al tanto de los tipos de errores que puedan ocurrir.
- **El bloque else** se ejecutará solo si el bloque try se ejecuta sin errores. Esto puede ser útil cuando quieras continuar el código del bloque try. Por ejemplo si abres un archivo en el bloque try, podrías leer su contenido dentro del bloque else.
- **El bloque finally** siempre es ejecutado sin importar que pase en los otros bloques, esto puede ser útil cuando quieras liberar recursos después de la ejecución de un bloque de código, (try, except o else).

  Nota: Los bloques else y finally son opcionales. En muchos casos puedes solo ocupar el bloque try para tratar de ejecutar algo y capturar los errores como excepciones en el bloque except.

3.6.4. Casos de uso de la sintaxis try

- **ZeroDivisionError**

En caso de que el divisor fuese cero, python arrojaría un error, que lo podemos tener controlado con la sentencia try de la siguiente manera. Lo adecuado será considerar el error de tipo: **ZeroDivisionError**

```
num, div = 10,0
try:
    res = num/div
    print(res)
except ZeroDivisionError:
    print("Trataste de dividir entre cero :( ")
```

- **TypeError**

En caso de intentar sumar un número y una cadena de texto ´String´ también nos daría un error. Lo adecuado será considerar el error de tipo : **TypeError**

```
mi_num = "cinco"
try:
  resultado = 10 + mi_num
  print(resultado)
except TypeError:
  print("El argumento mi_num deberia ser un número")
```

- **IndexError**

Queremos imprimir un elemento de la lista que no existe. Lo adecuado será considerar el error de tipo: **IndexError**

```
mi_lista = ["Python","C","C++","JavaScript"]
buscar_ind = 3
try:
  print( mi_lista[buscar_ind] )
except IndexError:
  print("Lo siento, el índice esta fuera de rango")
```

- **KeyError**

Intentamos imprimir una clave de un diccionario que no existe. Lo adecuado será considerar el error de tipo: **KeyError**

```
mi_dict = {"clave1":"valor1", "clave2":"valor2", "clave3":"valor3"}
buscar_clave = "clave no existente"
try:
  print( mi_dict[buscar_clave] )
except KeyError as msg_error::
  print(f"¡Lo siento, {error_msg} no es una clave valida!")
```

- **FileNotFoundError**

Intentamos abrir un fichero. Lo adecuado será considerar el error de tipo: **FileNotFoundError**

```
try:
    mi_archivo = open("contenido/datos_meteorologicos/temperaturas.csv")
except FileNotFoundError:
    print(f"Lo siento, el archivo no existe")
else:
    contenido = mi_archivo.read()
    print(contenido)
finally:
    mi_archivo.close()
```

3.6.5. Secuencias de números

Una función *range()* que devuelve un flujo de números en el rango especificado.

range(start, stop, step)

- **start**: Es opcional y tiene valor por defecto 0.

- **stop**: Es obligatorio (siempre se llega a 1 menos que este valor).
- **step**: Es opcional y tiene valor por defecto 1.

range() devuelve un objeto iterable, así que iremos obteniendo los valores paso a paso con una sentencia for ... in

```
>>> for i in range(0, 3):
... print(i)
...
0
1
2

>>> for i in range(3): # No hace falta indicar el inicio si es 0
... print(i)
...
0
1
2

>>> for i in range(1, 6, 2):
... print(i)
...
1
3
5

>>> for i in range(2, -1, -1):
... print(i)
...
2
1
0
```

Hay situaciones en las que no necesitamos usar la variable. Para estos casos se suele recomendar usar el guion bajo _ como nombre de variable.

```
>>> for _ in range(10):
...     print( "Repeat me 10 times!" )
...
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
```

Ejercicio

Determine si un número dado es un número primo.
No es necesario implementar ningún algoritmo en concreto. La idea es probar los números menores al dado e ir viendo si las divisiones tienen resto cero o no.

Capítulo 4

Módulos y la biblioteca estándar

4.1. Introducción a los módulos en python

Un módulo en Python es un fichero ".py" que contiene un conjunto de funciones o variables y que puede ser usado por otros módulos y así se puede reutilizar código y organizarlo mejor

Por ejemplo, podemos definir un módulo mimodulo.py con dos funciones suma() y resta().

```
# mimodulo.py
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b
```

Una vez definido, dicho módulo puede ser usado o importado en otro fichero. "otromodulo.py"

```
# otromodulo.py
import mimodulo

print(mimodulo.suma(4, 3))    # 7
print(mimodulo.resta(10, 9)) # 1
```

También podemos importar únicamente los componentes que nos interesen

```
# otromodulo.py
from mimodulo import suma, resta

print(suma(4, 3))    # 7
print(resta(10, 9)) # 1
```

Por último, podemos importar todo el módulo haciendo uso de *. Pero esta práctica no está recomendada

```
from mimodulo import *

print(suma(4, 3))    # 7
print(resta(10, 9)) # 1
```

Podemos asignar un nombre más corto a un módulo haciendo uso de 'as'.

```
import modulodenombresuperlargoydificil as breve
```

La función dir() nos permite ver los nombres (variables, funciones, clases, etc) existentes en nuestro namespace. Por ejemplo, __file__ es creado automáticamente y alberga el nombre del fichero .py.

```
print(__file__)
#/tu/ruta/tufichero.py
```

Dependiendo de la situación, puede ser importante especificar que únicamente queremos que se ejecute el código si el módulo es el __main__. Con la siguiente modificación

```
# modulo.py
def suma(a, b):
```

```
    return a + b

# Con esto si hacemos import modulo desde otro módulo,
#    este fragmento ya no se ejecutará al ser el módulo main otro.
if (__name__ == '__main__'):
    c = suma(1, 2)
    print("La suma es:", c)
```

4.2. La biblioteca estándar

La librería estándar es un conjunto de módulos y paquetes que vienen por defecto con Python. Muchas de las operaciones más comunes de la programación diaria ya están implementadas en ella, de modo que podemos concentrarnos en lo que realmente nos ocupa. Encontrarás la lista de todos los módulos y paquetes junto con su documentación en docs.python.org.

Por ejemplo, el *módulo "math"* contiene una colección de operaciones matemáticas comunes.

```
>>> import math

>>> # Raíz cuadrada.
>>> print(math.sqrt(16))
```

O bien el *módulo "random"*, que implementa funciones para trabajar con números aleatorios.

```
>>> from random import randint, choice

>>> # Número aleatorio entre 1 y 10.
>>> print(randint(1, 10))
>>> # Retorna un elemento aleatorio de la lista.
>>> print(choice(["Python", "C", "C++", "Java"]))
```

Recuerda que tienes a tu disposición la función help().

4.3. Módulos *sys*

El módulo "sys" es un módulo incorporado por defecto en Python que proporciona acceso a algunas variables y funciones que interactúan con el intérprete de Python y con el SO subyacente.

Algunas de las funciones del módulo "sys" son:

- *sys.argv:* una lista que contiene los argumentos de línea de comandos pasados al script Python.
- *sys.version*: Retorna el número de versión de Python con información adicional
- *sys.executable:* Retorna el path absoluto del binario ejecutable del intérprete de Python
- *sys.platform*: una cadena que indica la plataforma en la que se está ejecutando Python (por ejemplo, "linux2" en Linux).
- *sys.exit([arg]):* una función que permite salir del programa Python con un código de estado opcional.
- *sys.stdin, sys.stdout y sys.stderr:* objetos que representan la entrada estándar, la salida estándar y la salida de error estándar, respectivamente.
- *sys.path*: una lista de cadenas que especifica las rutas de búsqueda para los módulos de Python.

Ejemplo con sys.argv "*ejemplo_argv.py*"

Se ejecutaría como '$ ejemplo_argv.py 12 hola 34 verano 983'
import sys
for x in range(1,int(sys.argv[1])): print(x)
Imprime todos los argumentos print("Argumento pasado:",sys.argv)

Ejemplo con sys.platform

Tenemos a continuación otro ejemplo con sys.platform, si es una plataforma "Windows" se saldrá sin hacer nada. Se deja como ejercicio adaptarlo para que se ejecute siendo una plataforma Linux.
# – coding: utf-8 –
import sys
if sys.platform == 'windows': # CHECK ENVIRONMENT print("Perfecto estoy en el entorno adecuado")
pass
else: print("This script is intended to run only on Windows, Detected platform: ", sys.platform)
sys.exit("Failed")

4.4. Módulo os

El módulo os nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular la estructura de directorios.

- *os.access(path, modo_de_acceso):* Saber si se puede acceder a un archivo o directorio.
- *os.getcwd()*: Conocer el directorio actual.
- *os.chdir(nuevo_path)*: Cambiar de directorio de trabajo.
- *os.chroot()*: Cambiar al directorio de trabajo raíz.
- *os.chmod(path, permisos)*:Cambiar los permisos de un archivo o directorio.
- *os.chmod(path, permisos)*: Cambiar el propietario de un archivo o directorio.
- *os.mkdir(path[, modo])*: Crear un directorio.
- *os.mkdirs(path[, modo])*: Crear directorios recursivamente.
- *os.remove(path)*: Eliminar un archivo.
- *os.rmdir(path)*: Eliminar un directorio.
- *os.removedirs(path)*: Eliminar directorios recursivamente.
- *os.rename(actual, nuevo)*: Renombrar un archivo.
- *os.symlink(path, nombre_destino)*: Crear un enlace simbólico.

"os.path" permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios que resultan muy útiles:

- *os.path.abspath(path)*: Ruta absoluta
- *os.path.basename(path)*: Directorio base
- *os.path.exists(path)*: Saber si un directorio existe
- *os.path.getatime(path)*: Conocer último acceso a un directorio

- *os.path.getsize(path)*: Conocer tamaño del directorio
- *os.path.isabs(path)*: Saber si una ruta es absoluta
- *os.path.isfile(path)*: Saber si una ruta es un archivo
- *os.path.isdir(path)*: Saber si una ruta es un directorio
- *os.path.islink(path)*: Saber si una ruta es un enlace simbólico
- *os.path.ismount(path)*: Saber si una ruta es un punto de montaje

Veamos un ejemplo.

```
>>> import os

>>> # Para cambiar de directorio en python
>>> os.chdir('/home/tr19/cursoPython')
>>> os.getcwd()

>>> # Creación de carpetas
>>> os.makedirs('/home/tr19/cursoPython/dir_nuevo')

>>> # Dividir una ruta de archivo, en la ruta y el nombre del archivo,
>>> path = '/home/tr19/cursoPython/mi_python.py'
>>> os.path.basename(path)
"mi_python.py"
>>> os.path.dirname(path)
"/home/tr19/cursoPython/"

>>> # Eliminación de carpetas
>>> os.rmdir("/home/tr19/cursoPython/borrar_dir")

>>> # Tamaño de archivos y carpetas
>>> os.path.getsize('/home/tr19/cursoPython/dir_nuevo')

>>> # Saber si una ruta es valida o no
>>> os.path.exists('/home/tr19/cursoPython/dir_nuevo')
>>> os.path.exists('/home/tr19/cursoPython/dir_no_existe')
>>> os.path.isdir('/home/tr19/cursoPython/dir_nuevo')
>>> os.path.isfile('/home/tr19/cursoPython/existe.txt')
>>> os.path.isdir('/home/tr19/cursoPython/dir_nuevo_no')
>>> os.path.isfile('/home/tr19/cursoPython/dir_nuevo/no_existe.txt')
```

4.4.1. Objeto os.environ

El módulo *OS* de Python proporciona funciones para interactuar con el Sistema Operativo (SO).

1. *os.environ* es un objeto de mapeo que representa las variables del entorno del usuario y devuelve un diccionario que tiene la variable de entorno del usuario como clave y sus valores como valor.
2. *os.environ* se comporta como un diccionario, por lo que se pueden realizar todas las operaciones comunes del diccionario, como get y set. También podemos modificar os.environ pero cualquier cambio será efectivo solo para el proceso actual donde fue asignado y no cambiará el valor de forma permanente

A continuación vemos como usar os.eviron para acceder a las variables de entrono.

```
# Programa Pyton para explicar el objeto os.environ

# importamos el módulo os
import os
import pprint
```

```
# Obtenemos la lista de
# variables de entorno del usuario
env_var = os.environ

# Imprimimos la lista de
# variables de entorno del usuario
print("User's Environment variable:")
pprint.pprint(dict(env_var), width = 1)
```

Ahora vemos como acceder a una variable de entorno determinada.

```
# Programa Pyton para explicar el objeto os.environ

# importamos el módulo os
import os

# Obtenemos el valor de
# 'HOME' variable de entorno
home = os.environ['HOME']

# imprimimos el valor de
# la variable de entorno 'HOME'
print("HOME:", home)

# Obtenemos el valor de
# la variable de entorno 'JAVA_HOME'
# usando la operación get del diccionario.
java_home = os.environ.get('JAVA_HOME')

# Imprime el valor de
# la variable de entorno 'JAVA_HOME'
print("JAVA_HOME:", java_home)
```

También podemos exportar una variable con un script en BASH que además lance nuestro Python y usarla.

```
#!/bin/sh
# Script en bash
export fichero_entrada="/home/tr25/fichero.txt"
python ./entorno.py
```

```
# python --> entorno.py
import os
fich = os.environ.get('fichero_entrada')
```

4.5. Módulo datetime

Este módulo nos permite la manipulación de valores de fecha y hora,

Clase time de datetime

Los valores de tiempo se representan con la clase time.

```
import datetime

# Se crea un objeto de tipo time
t = datetime.time(1, 2, 3) # instancia de time
print(t)

# Veamos algunos atributos de la instancia de time "t"
#    que nos permite acceder a los elementos de time por separado.
```

```
print('hour        :', t.hour)
print('minute      :', t.minute)
print('second      :', t.second)
print('microsecond:', t.microsecond)
print('tzinfo      :', t.tzinfo)

# Si queremos almacenar una hora en concreto.
hora = datetime.time(14,30,20,7577)
# Con los tres primeros sería suficiente.
hora = datetime.time(14,30,20)
```

La resolución para time está limitada a micro segundos.

Clase date del módulo datetime

Los valores de la fecha del calendario se representan con date.

```
import datetime

# A este objeto, le pasaremos como primer argumento, el año.
# Como segundo el mes y como tercero el día del mes.
fecha = datetime.date(2027,12,20)
print(fecha)

# Algunos atributos para separar sus elementos.
print(fecha.year)
print(fecha.month)
print(fecha.day)

# Obtener la fecha actual.
fecha_hoy = datetime.date.today()
print(fecha_hoy)

# Obtener la hora actual
hora_actual = datetime.datetime.now()
print(hora_actual)
```

Formateando nuestra fecha en un string.

Tenemos una serie de códigos de formateo de fechas, con los que podemos obtener en un string, las partes que nos interesan

Código	Significado	Ejemplo de uso
%a	Nombre abreviado del día de la semana.	Fry, Sat
%A	Día entero de la semana.	Fryday, Saturday
%w	Día de la semana (numérico) 0 - domingo.	0, 1, 2, 3, 4, 5, 6
%d	Día del mes con el prefijo 0 (solo del 01 al 09, el resto es 10, 11, 12...).	01, 02, 03 ... 10, 11, 12...
%b	Nombre del mes abreviado.	Dec, Jan, Feb
%m	Mes del año (numérico) con el prefijo 0 hasta el mes 09.	01, 02, 03 ... 10, 11, 12
%B	Nombre del mes entero.	December, January, ...
%y	Número del año abreviado a dos dígitos.	00, 01, 02, ... 98, 99
%Y	Número del año completo.	2021, 2022, 2023 ...

Podemos imprimir la fecha como un string de la siguiente forma.

```
import datetime
# Obtenemos la fecha hora actual
now = datetime.datetime.now()
format = now.strftime('Día :%d, Mes: %m, Año: %Y, Hora: %H, Minutos: %M, Segundos: %S')
print(format)

# Importamos el módulo de una forma diferente
```

```
from datetime import datetime
# Obtenemos la fecha hora actual
now = datetime.now()
format = now.strftime('Día :%d, Mes: %m, Año: %Y, Hora: %H, Minutos: %M, Segundos: %S')
print(format)
```

Convertir una strong a datetime

```
from datetime import datetime

fecha_y_hora_em_texto = '01/09/2020 12:30'
fecha_y_hora = datetime.strptime(fecha_y_texto, '%d/%m/%Y%H:%M')
```

Operaciones

En ocasiones tendremos la necesidad de realizar ciertas operaciones con fechas, ya sea agregar días, restar años.

```
from datetime import datetime
from datetime import timedelta

# Sumar dos días a la fecha actual
now = datetime.now()
new_date = now + timedelta(days=2)
print(new_date)

#Comparación
if now < new_date:
    print("La fecha actual es menor que la nueva fecha")
```

El problema de la zona horaria

Lo resolvemos con el módulo pytz

```
from datetime import datetime
import pytz

fecha_y_hora_actuales = datetime.now()

# Aqui podemos definir una zona horaria
zona_horaria = pytz.timezone('America/Bogota')
zona_horaria = pytz.timezone('America/New_York')
zona_horaria = pytz.timezone('Europe/Madrid')
zona_horaria = pytz.timezone('CET')
# hora utc
zona_horaria = pytz.timezone('Zulu')
zona_horaria = pytz.timezone("UTC")

# Definimos la zona horaria
fecha_y_hora_bogota = fecha_y_hora_actuales.astimezone(huso_horario)
# La imprimimos como un "string"
fecha_y_hora_bogota_en_texto = fecha_y_hora_bogota.strftime('%d/%m/%Y %H:%M')

print(fecha_y_hora_bogota_en_texto)
```

Como podemos cambiar de una zona horaria a otra ...

```
import datetime
import pytz

# Definimos la hora actual en UTC
dt_ini = datetime.datetime.now( pytz.utc )

# Definimos la zona horaria que nos interesa
zona_horaria = pytz.timezone('America/Bogota')
```

```
# Cambiamos la hora a la nueva zona horaria
fecha_resultado = dt_ini.astimezone(zona_horaria)

# Imprimimos la fecha por pantalla
print(fecha_resultado)
```

4.6. Módulo request

Es posible que tengas que hacer peticiones web, ya sea para consumir un API, extraer información de una página o enviar el contenido de un formulario de manera automatizada. El módulo "request" te facilitará todas estas tareas.

Cómo hacer una petición GET

De este modo podemos obtener el contenido de una web o realizar una petición a un API.

```
>>> import requests
>>> resp = requests.get('https://www.aemet.es/')
>>> resp.encoding
'ISO-8859-1'

# Contenido de la respuesta

# Si se trata de un texto
>>> resp.text
'<!doctype html><html itemscope="" itemtype="h...'

# Si se trata de una imagen
>>> i = Image.open(StringIO(resp.content))
```

Para archivos como los *.json*, la librería posee una función especial resp.json() que decodifica los datos para mostrarlo en formato .json.

```
>>> import requests
>>> resp = requests.get('https://www.aemet.es/')
>>> j_son = resp.json()
```

Código de estado de la respuesta

```
>>> import requests
>>> resp = requests.get('https://www.aemet.es/')
>>> resp.status_code
```

Cabeceras de la respuesta

```
>>> import requests
>>> resp = requests.get('https://www.aemet.es/')
>>> resp.headers
```

Cómo hacer una petición POST

Para enviar los datos de un formulario por ejemplo.

```
>>> import requests
>>> auth_data = {'email': 'unocualquiera@aemet.es', 'pass': 'agencia'}
>>> resp = requests.post('https://aemet.agencia.com/login/', data=auth_data)
```

Se pueden realizar más peticiones además de las GET y POST:

```
>>> r = requests.put("https://una_web.org/put")
>>> r = requests.delete("https://una_web.org/delete")
>>> r = requests.head("https://una_mas.org/get")
>>> r = requests.options("https://ultima_web.org/get")
```

4.7. Módulo ConfigParser

Permite gestionar archivos de configuración editables por el usuario.

Formato de archivo de configuración

1. Las secciones del archivo de configuración se identifican buscando líneas que comiencen con [y terminen con].
2. La línea comienza con el nombre de la opción, que está separada del valor por dos puntos (:) o un signo igual (=).
3. Las líneas que comienzan con punto y coma (;) o numeral (#) son tratadas como comentarios

```
# Este es un ejemplo con comentario simple.config
[bug_tracker]
url = https://aemet.es
username = Hill
; No debes guardar "password" en texto plano.
; fichero de configuración
password = SECRETO
```

Lectura de archivos de configuración

Lo habitual es hacer que la aplicación lea el archivo de configuración, lo analice y actúe en función de su contenido.

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('simple.config')

print(parser.get('bug_tracker', 'url'))
```

Puede que sea necesario abrirlo con la codificación adecuada.

```
from configparser import ConfigParser

parser = ConfigParser()
# Open the file with the correct encoding
parser.read('unicode.ini', encoding='utf-8')

print(parser.get('bug_tracker', 'url'))
```

Acceso a los ajustes de configuración

Tenemos ahora 'inicial.config' como fichero de configuración para nuestro Python.

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET

[wiki]
url = http://localhost:8080/wiki/
username = dhellmann
password = SECRET
```

Y vamos a ver como podemos acceder a cada sección y opción; de la siguiente forma nos devolverá listas.

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

for section_name in parser.sections():
```

```
    print('Section:', section_name)
    print('  Options:', parser.options(section_name))
    for name, value in parser.items(section_name):
        print('  {} = {}'.format(name, value))
    print()
```

Pero también admite la misma interfaz de programación de mapeo que dict, con el ConfigParser actuando como un diccionario que contiene diccionarios separados para cada sección.

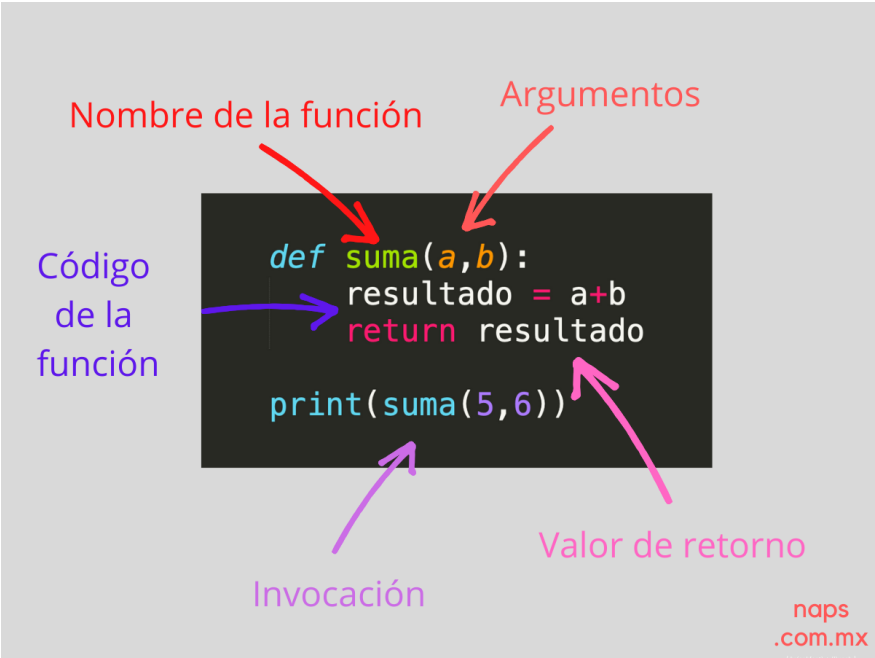
```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

for section_name in parser:
    print('Section:', section_name)
    section = parser[section_name]
    print('  Options:', list(section.keys()))
    for name in section:
        print('  {} = {}'.format(name, section[name]))
    print()
```

Capítulo 5

Funciones

El concepto de función es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

1. **No repetir** fragmentos de código en un programa.
2. **Reutilizar** el código en distintos escenarios.



5.1. Primera función

Vamos *invocar* a esta función y obtendremos un mensaje por pantalla, fruto de la ejecución del cuerpo de la función.

```
>>> def say_hello():
...     print( "Hello!" )
...
>>> # Esta sería la forma de invocar una función.
>>> say_hello()
Hello!
```

Las funciones pueden devolver un valor.

```
>>> def one():
...     return 1
...
```

```
>>> one()
1
```

Podemos integrar las funciones de otras estructuras como en condicionales:

```
>>> if one() == 1:
...     print( "¡¡¡ Funciona !!!" )
... else:
...     print( "Algo va mal :( " )
...
¡¡¡ Funciona !!!
```

Una función puede devolver más de un valor.

```
>>> def multiple():
...     return 0, 1 # es una tupla!
...

>>> result = multiple()
>>> result
(0, 1)
>>> type(result)
tuple

>>> # Podemos obtener los valores por separado de esta forma
>>> a, b = multiple()
>>> a
0
>>> b
1
```

5.2. Parámetros y argumentos

Una función sin valores de entrada estaría muy limitada. Sin embargo los parámetros nos permiten variar los datos que usa una función para obtener distintos resultados.

```
>>> def sqrt(valor): # valor es un parámetro.
...     raiz = valor ** (1/2)
... return raiz
...
>>> sqrt(4) # 4 sería un argumento.
2.0

>>> # En la definición "a" y "b" son parámetros.
>>> def minimo(a, b):
...     if a < b:
...         return a
...     else:
...         return b
...
>>> # Cuando "invoco" a la función son argumentos
>>> minimo(7, 9) # 7 y 9 son argumentos.
7

>>> # También puedo escribir la función mínimo así:
>>> def minimo(a, b):
...     if a < b:
...         return a
... return b
...
```

Tenemos dos tipos de argumentos:

- **Posicionales**: Se copian en sus correspondientes parámetros en orden.
- **Nominales**: Se asignan por nombre a cada parámetro.

Ejemplo Posicionales

```
>>> def build_cpu(vendor, num_cores, freq):
...     return dict(
...             vendor=vendor,
...             num_cores=num_cores,
...             freq=freq
...             )
...
>>> diccionario_cpu = build_cpu( AMD , 8, 2.7)
>>> diccionario_cpu
{ vendor : AMD , num_cores : 8, freq : 2.7}
```

Ejemplo Nominales

```
>>> build_cpu(vendor= "AMD" , num_cores=8, freq=2.7)
{ vendor : AMD , num_cores : 8, freq : 2.7}

>>> # Se puede ver como el orden de los argumentos no influye en el resultado.
>>> build_cpu(num_cores=8, freq=2.7, vendor= "AMD" )
{ vendor : AMD , num_cores : 8, freq : 2.7}
```

Python **permite mezclar argumentos** posicionales y nominales en la llamada a una función:

```
>>> build_cpu( "INTEL" , num_cores=4, freq=3.1)
{ vendor : INTEL , num_cores : 4, freq : 3.1}
```

Los argumentos posicionales deben ir siempre antes.

```
>>> build_cpu(num_cores=4, INTEL , freq=3.1)
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Es posible especificar **valores por defecto en los parámetros de una función**. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

```
>>> def build_cpu(vendor, num_cores, freq=2.0):
...     return dict(
...     vendor=vendor,
...     num_cores=num_cores,
...     freq=freq
... )
...

>>> # Llamada a la función sin especificar frecuencia de «cpu»:
>>> build_cpu( "INTEL" , 2)
{ vendor : "INTEL" , num_cores : 2, freq : 2.0}

>>> # Llamada a la función indicando una frecuencia concreta de «cpu»:
>>> build_cpu( "INTEL" , 2, 3.4)
{ vendor : "INTEL" , num_cores : 2, freq : 3.4}
```

Ejercicio

Escriba un programa que pida la anchura y altura de un rectángulo y lo dibuje con caracteres producto (*):

Anchura del rectángulo: 5
Altura del rectángulo: 3
* * * * *
* * * * *
* * * * *

Ejercicio

Escriba un programa que pida un año y que escriba si es bisiesto o no.

Se recuerda que los años bisiestos son múltiplos de 4, pero los múltiplos de 100 no lo son, aunque los múltiplos de 400 sí.
Estos son algunos ejemplos de posibles respuestas: 2012 es bisiesto, 2010 no es bisiesto, 2000 es bisiesto, 1900 no es bisiesto.
COMPROBADOR DE AÑOS BISIESTOS
Escriba un año y le diré si es bisiesto: 2000
El año 2000 es un año bisiesto.

5.3. Documentación de funciones

Ya hemos visto que en Python podemos incluir comentarios para explicar mejor determinadas zonas de nuestro código. Del mismo modo podemos (y en muchos casos debemos) adjuntar documentación a la definición de una función incluyendo una cadena de texto (docstring) al comienzo de su cuerpo:

```
>>> def sqrt(value):
...     'Returns the square root of the value'
...      return value ** (1/2)
...
```

La forma más ortodoxa de escribir un docstring es utilizando triples comillas:

```
>>> def closest_int(value):
...     ''' Returns the closest integer to the given value.
...     The operation is:
...         1. Compute distance to floor.
...         2. If distance less than a half, return floor.
...     Otherwise, return ceil.
...     '''
...     floor = int(value)
...     if value - floor < 0.5:
...             return floor
...     else:
...             return floor + 1
...
```

Para ver el docstring de una función, basta con utilizar help:

```
>>> help(closest_int)
Help on function closest_int in module __main__:
closest_int(value)
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
```

```
    2. If distance less than a half, return floor.
Otherwise, return ceil.
```

También es posible extraer información usando el símbolo de interrogación:

```
>>> closest_int?
Signature: closest_int(value)
Docstring:
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
    2. If distance less than a half, return floor.
Otherwise, return ceil.
```

5.4. Tipos de funciones.

5.4.1. Funciones anónimas «lambda»

Una función lambda tiene las siguientes propiedades:

1. Se escribe en una única sentencia (línea).
2. No tiene nombre (anónima).
3. Su cuerpo conlleva un *"return"* implícito.
4. Puede recibir cualquier número de parámetros.

Aquí podemos ver un esquema de cómo funcionan.

```
def a(x, y):
    return x + y

b = lambda x, y: x + y
```

Veamos un primer ejemplo de función «lambda» que nos permite contar el número de palabras en una cadena de texto dada. La transformación de su versión clásica en su versión anónima sería la siguiente:

```
>>> num_words = lambda t: len(t.split())
>>> type(num_words)
function
>>> num_words
<function __main__.<lambda>(t)>
>>> num_words( hola socio vamos a ver )
5
```

5.4.2. Lambdas como argumentos

Las funciones «lambda» son bastante utilizadas como argumentos a otras funciones. Un ejemplo claro de ello es la función sorted que recibe un parámetro opcional key donde se define la clave de ordenación. Veamos cómo usar una función anónima «lambda» para ordenar una tupla de pares longitud-latitud:

```
>>> geoloc = (
... (15.623037, 13.258358),
... (55.147488, -2.667338),
```

```
... (54.572062, -73.285171),
... (3.152857, 115.327724),
... (-40.454262, 172.318877)
)
>>> # Ordenación por longitud (primer elemento de la tupla)
>>> sorted(geoloc)
[(-40.454262, 172.318877),
(3.152857, 115.327724),
(15.623037, 13.258358),
(54.572062, -73.285171),
(55.147488, -2.667338)]
>>> # Ordenación por latitud (segundo elemento de la tupla)
>>> sorted(geoloc, key=lambda t: t[1])
[(54.572062, -73.285171),
(55.147488, -2.667338),
(15.623037, 13.258358),
(3.152857, 115.327724),
(-40.454262, 172.318877)]
```

5.5. Consejos para programar

1. Las funciones deberían hacer una única cosa.
2. Utiliza nombres descriptivos y con significado.
3. Mantenerlo regularmente y con sentido.
4. No utilices «números mágicos» o valores «hard-codeados».
5. Escribe lo que necesites ahora, no lo que pienses que podrías necesitar en el futuro.
6. Usa comentarios para explicar el «por qué» y no el «qué». (Contextualizar)

Capítulo 6

Trabajar con ficheros de texto

Todo lo que hemos visto hasta ahora no perdura, es decir, una vez cerrado el programa, toda interacción con el usuario no tendrá efecto en las posteriores ejecuciones.

La ruta o ("path") de un archivo es lo que nos indica su ubicación en el ordenador y esta puede ser de tipo relativa o absoluta.

6.1. Abrir y cerrar un archivo de texto

Esto lo podemos hacer con la función open(), que recibe como parámetros la ruta donde se encuentra el archivo y el modo (lectura o escritura). Después de usar un archivo (ya sea por lectura o escritura) es necesario cerrarlo, porque si no lo hacemos, a veces los archivos solo pueden ser abiertos por un programa a la vez o puede que lo que se haya escrito no quede guardado hasta que el archivo se haya cerrado o incluso porque el límite en la cantidad de archivos que puede manejar un programa puede ser bajo. Para cerrar un archivo basta con utilizar la función *close()*.

```
mi_archivo = open ('<nombre del archivo >', '<modo >')
mi_archivo.close()
```

Existen distintos modos de leer un archivo:

- 'r' solo lectura
- 'w' modo escritura
- 'a' modo 'append'

Otra opción para abrir y cerrar archivos es la siguiente:

```
>>> with open ('<nombre del archivo >', '<modo >') as mi_archivo :
>>>     # Aquí va, intentado , la lectura y/o escritura del archivo
>>>     # Leemos el archivo
>>>     lineas = mi_archivo.readlines() # SE VERÁ A CONTINUACIÓN.
```

6.2. Leer un archivo de texto

Vamos a abrir un fichero en modo lectura 'r', y vamos a ver como leer su contenido.

6.2.1. Readline

La función *readline* lee la primera línea del archivo cargado (identifica esta línea según los saltos de línea) y la retorna como *"string"* (generalmente con el fin de almacenarlo en una variable), además, esta línea se elimina del archivo cargado (no del archivo original).

Supongamos que tenemos el siguiente fichero *"aemet.txt"*:

```
Como Servicio Meteorológico Nacional y Autoridad Meteorológica del Estado,
El objetivo básico de AEMET es contribuir a la protección de vidas y bienes.
A través de la adecuada predicción y vigilancia de fenómenos meteorológicos adversos.
Como soporte a las actividades sociales y económicas en España.
Mediante la prestación de servicios meteorológicos de calidad.
```

Vamos a leerlo:

```
>>> # Abrimos el archivo
>>> mi_archivo = open ('aemet.txt', 'r')
>>>
>>> # Leemos
>>> linea_1 = mi_archivo.readline()
>>> print( linea_1 )
>>> linea_2 = mi_archivo.readline()
>>> print( linea_2 )
>>> linea_3 = mi_archivo.readline()
>>> print( linea_3 )
>>>
>>> # Cerramos el archivo
>>> mi_archivo.close()
```

Podemos notar que el programa no ha podido leer las tildes, esto tiene que ver con la manera en que se codifican/decodifican los archivos en tu ordenador, por lo que puede que no tengas este problema. Pero en caso de que suceda, se puede solucionar de la siguiente forma: agregando el parámetro *"encoding"* a la función open(), con valor 'utf-8'

6.2.2. Readlines

La función *readlines* lee todas líneas y las retorna como una lista de *"strings"*.

```
>>> # Abrimos el archivo
>>> mi_archivo = open ('aemet.txt','r', encoding ='utf -8')
>>>
>>> # Leemos
>>> lineas = mi_archivo.readlines()
>>> print( lineas )
>>>
>>> # Cerramos el archivo
>>> mi_archivo .close()
```

Vemos que líneas es una lista donde cada elemento corresponde a una línea del archivo.

Ejercicio

Imprime en consola, como un solo string y sin saltos de línea, el contenido del archivo dado archivo ejemplo.txt:
Hola , soy un archivo .

6.3. Escribir un archivo de texto

Para escribir un archivo de texto, necesitamos abrir un archivo en modo escritura, lo que se puede denotar con **'w' (write) o con 'a' (append).**

6.3.1. Modo Write

Si el archivo no existe, Python lo creará. Si el archivo ya existe, se borrará su contenido. Usaremos la función *write()*, que recibe como parámetro un *"string"* con el contenido que queremos escribir en el archivo.

```
>>> mi_archivo = open('nombre_del_archivo.txt','w')
>>> mi_archivo.write('String con contenido a escribir')
>>> mi_archivo.close()

>>> # Otra forma equivalente de escribirlo:
>>> with open ('nombre_del_archivo.txt','w') as mi_archivo :
...     mi_archivo.write('String con contenido a escribir')

>>> # Veamos su aplicación:
>>> mi_archivo = open ('archivo.txt', 'w', encoding='utf-8')
>>> mi_archivo.write('Escribe en el doc \nEsta es otra linea\nY esta otra')
>>> mi_archivo.close()

>>>  # Si ejecutamos esto a continuación se sobrescribirá el fichero.
>>>  mi_archivo = open('archivo.txt', 'w', encoding ='utf-8')
>>>  mi_archivo.write('Esto es nuevo')
>>>  mi_archivo.close()
```

6.3.2. Modo Append

Si el archivo no existe, Python lo creará, pero si el archivo ya existe, agregará el contenido en la línea siguiente a la ultima escrita.

```
>>> mi_archivo = open('nombre_del_archivo.txt','a')
>>> mi_archivo.write('String con contenido a escribir')
>>> mi_archivo.close()

>>> # Otra forma equivalente de escribirlo:
>>> with open ('nombre_del_archivo.txt','a') as mi_archivo :
...     mi_archivo.write('String con contenido a escribir')

>>> # Veamos su aplicación:
>>> mi_archivo = open ('archivo.txt', 'a', encoding='utf-8')
>>> mi_archivo.write('Escribe en el doc \nEsta es otra linea\nY esta otra')
>>> mi_archivo.close()

>>>  # Hasta ahora todo era igual al modo "write"
>>>  #    pero al ejecutar esto ahora no se sobreescribirá
>>>  mi_archivo = open('archivo.txt', 'a', encoding ='utf-8')
>>>  mi_archivo.write('Esto es nuevo')
>>>  mi_archivo.close()
```

6.4. Ejercicios

Ejercicio 5.1

Crear un archivo llamado "observacioens.txt" que contendrá una lista de observaciones con los siguientes campos:

- temperatura
- precipitación
- humedad
- presión

Leer cada línea y colocar cada campo un diccionario y mostrar los datos del diccionario.

Ejercicio 5.2

Debe realizar un programa que lea del usuario una palabra. Luego, vaya a buscar esta palabra en un archivo de texto llamado mi_texto.txt. En esta búsqueda su programa debe hacer lo siguiente:

1. Contar cuantas ocurrencias de la palabra hay en el archivo de texto (después de haber leído todo su contenido) y desplegar en pantalla esa cantidad.
2. Crear otro archivo llamado "resultado.txt", el cual contenga solamente las líneas en donde se encuentra la palabra buscada, desplegando al principio de la línea, su número de línea. Es decir que si fuera la primera línea, el número es 1, si es la tercera el número es 3, etc.

Capítulo 7

Librería Pandas

(Basado en el tutorial "10 minutes to pandas" https://pandas.pydata.org/pandas-docs/stable/10min.html)

7.1. Introducción

Pandas (http://pandas.pydata.org) es una librería de acceso abierto (open source), con licencia BSD que proporciona estructuras y herramientas de análisis de datos de altas prestaciones y fáciles de usar escritas en lenguaje Python. Es parte del ecosistema Scipy (https://projects.scipy.org).

Numpy es la librería básica de cálculo en python, con la matriz multidimensional np.array como su estructura fundamental.

Sin embargo nosotros estudiaremos directamente Pandas, que es una librería especializada en el análisis de datos que puede sernos más útil, aunque nos referiremos en alguna ocasión a Numpy. Pandas tiene dos características que la hacen especialmente interesante: permite utilizar datos heterogéneos, y utilizar etiquetas para identificar datos tabulados fácilmente.

Observar, en cualquier caso, que ambas librerías tienen muchos métodos comunes, por lo que gran parte de la funcionalidad aprendida en Pandas será aplicable también en Numpy (aunque con una sintaxis diferente).

Características de Pandas

Ventajas:

- Más versatilidad que numpy en el manejo de arrays: permite tablas no homogéneas, nombres de las columnas
- Maneja muy bien series temporales
- Gran cantidad de funciones "off-the-shelf": implementadas, testeadas, y listas para usar
- Código corto y fácil de leer

Desventajas:

- Ocupa más memoria
- Código mucho más difícil de escribir si esa funcionalidad no la hace ya pandas
- Curva de aprendizaje dura

- R dispone de herramientas más potentes de análisis estadístico, aunque Pandas es parte de un lenguaje de propósito general.

7.2. Creación de series y dataframes

Objetos más importantes:

- Series (1 dimensión)
- DataFrame (2 dimensiones)

También está el Panel (3 dimensiones), pero se usa poco: en general es mejor utilizar un dataframe más complicado (o un array de Xarray).

```
import pandas as pd
import numpy as np
from datetime import datetime
```

Creación de una serie a partir de una lista:

```
s = pd.Series(['angel',3,5,np.nan,6,8])
```

```
s
```

```
0    angel
1        3
2        5
3      NaN
4        6
5        8
dtype: object
```

Un dataframe se puede crear de varias formas distintas. En primer lugar se puede crear a partir de un diccionario. Tomamos datos de precipitación de tres estaciones del CENAOS, el Centro de Estudios Atmosféricos, Oceanográficos y Sísmicos de Honduras, como ejemplo:

```
df = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                   'Date': [datetime(2023,8,27,12),
                            datetime(2023,8,27,12),
                            datetime(2023,8,27,12)],
                   'PCP 1h':[0.0, 0.0, 0.0],
                   'PCP 6h':[7.6, 0.1, 0.0],
                   'PCP 12h':[11.7, 0.1, 7.3],
                   'PCP 24h':[11.7, 19.2, 18.8]})
```

```
df
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8

Observar que las columnas pueden tener distintos tipos de datos:

```
df.dtypes
```

```
Estacion            object
Date        datetime64[ns]
PCP 1h             float64
PCP 6h             float64
PCP 12h            float64
PCP 24h            float64
dtype: object
```

También se puede crear un dataframe especificando cada una de sus filas:

```
df = pd.DataFrame([['Los Platos', datetime(2023,8,27,12), 0.0, 7.6, 11.7, 11.7],
                   ['Atima', datetime(2023,8,27,12), 0.0, 0.1, 0.1, 19.2],
                   ['Bomberos-Tegucigalpa', datetime(2023,8,27,12), 0.0, 0.0, 7.3, 18.8]],
                  columns=['Estacion', 'Date', 'PCP 1h', 'PCP 6h', 'PCP 12h', 'PCP 24h'])
df
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8

Otra alternativa es cargar los datos a partir de un fichero csv (lo estudiaremos en detalle más adelante). Por ejemplo, estos datos de Buenos Aires tomados de la web del Servicio Meteorológico Nacional de Argentina:

```
df = pd.read_csv("buenos_aires_20230813.txt", sep=",", encoding = "ISO-8859-1")
df
```

	Fecha	Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad
0	14/08/2023	0:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km
1	13/08/2023	23:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km
2	13/08/2023	22:00	Nublado	13.2	No se calcula	43	Norte	7 km/h	1015.6 hPa	10 km
3	13/08/2023	21:00	Mayormente nublado	13.8	No se calcula	38	Norte	11 km/h	1015 hPa	10 km
4	13/08/2023	20:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km
5	13/08/2023	19:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km
	...	...	...	...	...	...	...	...	...	...
21	13/08/2023	3:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km

O también a partir de un fichero Excel:

```
df = pd.read_excel("buenos_aires_20230813.xlsx")
```

7.3. Indices

Hay una cierta equivalencia entre DataFrames y tablas de una base de datos.

Los índices pueden ser muy útiles cuando haya que seleccionar datos en un gran dataframe, o si hay que unir dos de ellos.

Para crear un índice usaremos la función `set_index`:

```
df = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                   'Date': [datetime(2023,8,27,12),
                            datetime(2023,8,27,12),
                            datetime(2023,8,27,12)],
                   'PCP 1h':[0.0, 0.0, 0.0],
                   'PCP 6h':[7.6, 0.1, 0.0],
                   'PCP 12h':[11.7, 0.1, 7.3],
                   'PCP 24h':[11.7, 19.2, 18.8]})
df
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8

```
df.set_index('Date', inplace=True)
df
```

	Estacion	PCP 1h	PCP 6h	PCP 12h	PCP 24h
Date					
2023-08-27 12:00:00	Los Platos	0	7.6	11.7	11.7
2023-08-27 12:00:00	Atima	0	0.1	0.1	19.2
2023-08-27 12:00:00	Bomberos-Tegucigalpa	0	0	7.3	18.8

El índice también se puede definir simultáneamente al crear el dataframe:

```
df = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                   'PCP 1h':[0.0, 0.0, 0.0],
                   'PCP 6h':[7.6, 0.1, 0.0],
                   'PCP 12h':[11.7, 0.1, 7.3],
                   'PCP 24h':[11.7, 19.2, 18.8]},
                  index = [datetime(2023,8,27,12),
                           datetime(2023,8,27,12),
                           datetime(2023,8,27,12)])
```

```
df
```

	Estacion	PCP 1h	PCP 6h	PCP 12h	PCP 24h
2023-08-27 12:00:00	Los Platos	0	7.6	11.7	11.7
2023-08-27 12:00:00	Atima	0	0.1	0.1	19.2
2023-08-27 12:00:00	Bomberos-Tegucigalpa	0	0	7.3	18.8

Para eliminar el índice:

```
df.reset_index(inplace=True)
```

Observar que muchas funciones, como `set_index`, devuelven un dataframe como salida:

```
df.set_index('Date')
```

	Estacion	PCP 1h	PCP 6h	PCP 12h	PCP 24h
2023-08-27 12:00:00	Los Platos	0	7.6	11.7	11.7
2023-08-27 12:00:00	Atima	0	0.1	0.1	19.2
2023-08-27 12:00:00	Bomberos-Tegucigalpa	0	0	7.3	18.8

pero no modifican el dataframe original:

```
df
```

Date	Estacion	PCP 1h	PCP 6h	PCP 12h	PCP 24h
2023-08-27 12:00:00	Los Platos	0	7.6	11.7	11.7
2023-08-27 12:00:00	Atima	0	0.1	0.1	19.2
2023-08-27 12:00:00	Bomberos-Tegucigalpa	0	0	7.3	18.8

Para modificar el dataframe original hay que incluir la opción *inplace* o modificar el propio dataframe:

```
df.set_index('Date', inplace=True)
# o:
df = df.set_index('Date', inplace=False) # inplace toma el valor False por defecto
```

Es posible crear multiíndices, índices de más de un campo:

```
df = df.set_index(['Date', 'Estacion'])
df
```

	PCP 1h	PCP 6h	PCP 12h	PCP 24h
(Timestamp('2023-08-27 12:00:00'), 'Los Platos')	0	7.6	11.7	11.7
(Timestamp('2023-08-27 12:00:00'), 'Atima')	0	0.1	0.1	19.2
(Timestamp('2023-08-27 12:00:00'), 'Bomberos-Tegucigalpa')	0	0	7.3	18.8

7.4. Manejo de series temporales

La función `date_range` resulta muy útil para crear rangos de fechas:

```
pd.date_range("20230801", "20230803")
```

```
DatetimeIndex(['2023-08-01', '2023-08-02', '2023-08-03'], dtype='datetime64[ns]', freq='D')
```

o en un formato string:

```
pd.date_range(start="20230801", end="20230803").format()
```

```
['2023-08-01', '2023-08-02', '2023-08-03']
```

Entre otras cosas, se puede modificar la frecuencia:

```
pd.date_range("20230801", "20230803", freq="6H").format()
```

```
['2023-08-01 00:00:00',
 '2023-08-01 06:00:00',
 '2023-08-01 12:00:00',
 '2023-08-01 18:00:00',
 '2023-08-02 00:00:00',
 '2023-08-02 06:00:00',
 '2023-08-02 12:00:00',
 '2023-08-02 18:00:00',
 '2023-08-03 00:00:00']
```

En meteorología, es muy común tener datos ordenados por fechas. Pandas se adapta muy bien a esta necesidad, y nos permite utilizar la fecha como índice de nuestros DataFrames. De hecho dispone de un objeto específico para ello, el DataTimeIndex:

```
df = pd.DataFrame({'Tmax':[20.3, 21.5, 20.1, 22.0]},
                  index = pd.date_range("20230801", "20230804"))
df
```

	Tmax
2023-08-01 00:00:00	20.3
2023-08-02 00:00:00	21.5
2023-08-03 00:00:00	20.1
2023-08-04 00:00:00	22

El índice es un objeto DataTimeIndex:

```
df.index
```

```
DatetimeIndex(['2023-08-01', '2023-08-02', '2023-08-03', '2023-08-04'],
              dtype='datetime64[ns]', freq='D')
```

que tiene sus métodos específicos, por ejemplo:

```
print(df.index.month)
print(df.index.day)
```

```
Int64Index([8, 8, 8, 8], dtype='int64')
Int64Index([1, 2, 3, 4], dtype='int64')
```

```
print(df.index)
```

```
DatetimeIndex(['2023-08-01', '2023-08-02', '2023-08-03', '2023-08-04'],
              dtype='datetime64[ns]', freq='D')
```

7.5. Visualización de las características de un DataFrame

Para manejar adecuadamente un dataframe debemos conocer sus características: qué columnas tiene, qué tipo de datos almacena, etc. Hay varias propiedades y métodos que nos dan esa información.

Creemos un dataframe a partir del fichero csv de Buenos Aires, tomando como índice la fecha y la hora (más adelante veremos cómo se combinan estos dos campos en uno):

```
df = pd.read_csv("buenos_aires_20230813.txt", parse_dates=[['Fecha', 'Hora']],
                 sep=",", encoding = "ISO-8859-1")
df = df.set_index('Fecha_Hora')
df
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
2023-08-14 00:00:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km
2023-08-13 23:00:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km
2023-08-13 22:00:00	Nublado	13.2	No se calcula	43	Norte	7 km/h	1015.6 hPa	10 km
2023-08-13 21:00:00	Mayormente nublado	13.8	No se calcula	38	Norte	11 km/h	1015 hPa	10 km
2023-08-13 20:00:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km
2023-08-13 19:00:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km
...	...	...	...	...	...	...	...	...
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km

Como ya hemos visto, para mostrar el índice del dataframe escribiremos:

```
df.index
```

```
DatetimeIndex(['2023-08-14 00:00:00', '2023-08-13 23:00:00',
               '2023-08-13 22:00:00', '2023-08-13 21:00:00',
               '2023-08-13 20:00:00', '2023-08-13 19:00:00',
               '2023-08-13 18:00:00', '2023-08-13 17:00:00',
               '2023-08-13 16:00:00', '2023-08-13 15:00:00',
               '2023-08-13 14:00:00', '2023-08-13 13:00:00',
               '2023-08-13 12:00:00', '2023-08-13 11:00:00',
               '2023-08-13 10:00:00', '2023-08-13 09:00:00',
               '2023-08-13 08:00:00', '2023-08-13 07:00:00',
               '2023-08-13 06:00:00', '2023-08-13 05:00:00',
               '2023-08-13 04:00:00', '2023-08-13 03:00:00'],
              dtype='datetime64[ns]', name='Fecha_Hora', freq=None)
```

columns devuelve una lista con el nombre de cada columna (por cierto, es un objeto índice). Observar que el índice ya no se considera una columna:

```
print(df.columns)
print(df.columns[3])
```

```
Index(['Estado', 'T (°C)', 'ST (°C)', 'HR (%)', 'Viento dirección',
       'Viento velocidad', 'Presión', 'Visibilidad'],
      dtype='object')
HR (%)
```

dtypes da información sobre el tipo de dato de cada columna:

```
df.dtypes
```

```
Estado               object
T (°C)              float64
ST (°C)              object
HR (%)                int64
Viento dirección     object
Viento velocidad     object
Presión              object
Visibilidad          object
dtype: object
```

values devuelve el dataframe en forma de array de numpy

```
print(df.values)
print(type(df.values))
```

```
[['Nublado' 12.8 'No se calcula' 37 'Noroeste' '9 km/h' '1014.8 hPa'
  '10 km']
 ['Nublado' 13.0 'No se calcula' 37 'Noroeste' '16 km/h' '1015.2 hPa'
  '10 km']
 ['Nublado' 13.2 'No se calcula' 43 'Norte' '7 km/h' '1015.6 hPa' '10 km']
 ['Mayormente nublado' 13.8 'No se calcula' 38 'Norte' '11 km/h'
  '1015 hPa' '10 km']
 ['Mayormente nublado' 14.9 'No se calcula' 33 'Norte' '9 km/h'
  '1015.2 hPa' '10 km']
 ['Mayormente nublado' 14.7 'No se calcula' 32 'Norte' '13 km/h'
  '1015.6 hPa' '10 km']
 ...
 ['Despejado' 3.7 'No se calcula' 72 'Calma' nan '1023.7 hPa' '10 km']]
<class 'numpy.ndarray'>
```

`describe` es un método que da un resumen de varios estadísticos para el dataframe:

```
df.describe()
```

	T (°C)	HR (%)
count	22	22
mean	11.3273	41.6364
std	5.11954	18.1647
min	3.1	23
25 %	6.125	26.25
50 %	13.1	35.5
75 %	15.125	55.25
max	17.2	72

Cada uno de estos estadísticos es un método del dataframe:

```
df.count()
```

```
Estado              22
T (°C)              22
ST (°C)             22
HR (%)              22
Viento dirección    22
Viento velocidad    18
Presión             22
Visibilidad         22
dtype: int64
```

Hay más estadísticos disponibles:

```
df.quantile(0.5)
```

```
T (°C)    13.1
HR (%)    35.5
Name: 0.5, dtype: float64
```

```
df.quantile(0.666, axis=1)
```

```
Fecha_Hora
2023-08-14 00:00:00    28.9172
2023-08-13 23:00:00    28.9840
2023-08-13 22:00:00    33.0468
2023-08-13 21:00:00    29.9172
2023-08-13 20:00:00    26.9546
2023-08-13 19:00:00    26.2218
2023-08-13 18:00:00    22.5264
2023-08-13 17:00:00    22.0608
2023-08-13 16:00:00    22.3948
2023-08-13 15:00:00    21.6286
2023-08-13 14:00:00    20.7622
2023-08-13 13:00:00    21.7268
2023-08-13 12:00:00    22.7916
2023-08-13 11:00:00    24.2886
2023-08-13 10:00:00    26.2512
2023-08-13 09:00:00    32.1764
2023-08-13 08:00:00    41.0642
2023-08-13 07:00:00    47.9560
2023-08-13 06:00:00    48.3214
2023-08-13 05:00:00    48.3548
2023-08-13 04:00:00    47.7556
2023-08-13 03:00:00    49.1878
Name: 0.666, dtype: float64
```

Hay otras operaciones sencillas que nos pueden resultar útiles. La traspuesta (`T`):

```
df.T
```

	2023-08-14 00:00:00	**2023-08-13 23:00:00**	**2023-08-13 22:00:00**	**...**	**2023-08-13 04:00:00**	**2023-08-13 03:00:00**
Estado	Nublado	Nublado	Nublado	...	Despejado	Despejado
T (°C)	12.8	13.0	13.2	...	3.4	3.7
ST (°C)	No se calcula	No se calcula	No se calcula	...	No se calcula	No se calcula
HR (%)	37	37	43	...	70	72
Viento dirección	Noroeste	Noroeste	Norte	...	Calma	Calma
Viento velocidad	9 km/h	16 km/h	7 km/h	...	nan	nan
Presión	1014.8 hPa	1015.2 hPa	1015.6 hPa	...	1023.3 hPa	1023.7 hPa
Visibilidad	10 km	10 km	10 km	...	10 km	10 km

Y ordenar, ya sea por índice o por columna (`sort_index` y `sort_values` respectivamente). Recordar que sort_index no cambia el dataframe, a no ser que añadamos la opción inplace=True

```
df.sort_index()
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km
2023-08-13 04:00:00	Despejado	3.4	No se calcula	70	Calma	nan	1023.3 hPa	10 km
2023-08-13 05:00:00	Despejado	3.2	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 06:00:00	Despejado	3.1	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 07:00:00	Despejado	4	No se calcula	70	Oeste	3 km/h	1023.2 hPa	10 km
2023-08-13 08:00:00	Ligeramente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km
...	...	...	...	...	...	...	...	...
2023-08-13 23:00:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km
2023-08-14 00:00:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km

```
df.sort_values(by="T (°C)")
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad
2023-08-13 06:00:00	Despejado	3.1	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 05:00:00	Despejado	3.2	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 04:00:00	Despejado	3.4	No se calcula	70	Calma	nan	1023.3 hPa	10 km
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km
2023-08-13 07:00:00	Despejado	4	No se calcula	70	Oeste	3 km/h	1023.2 hPa	10 km
2023-08-13 08:00:00	Ligeramente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km
...	...	...	...	...	...	...	...	...
2023-08-13 15:00:00	Algo nublado	16.9	No se calcula	24	Norte	16 km/h	1017.7 hPa	10 km
2023-08-13 16:00:00	Algo nublado	17.2	No se calcula	25	Norte	16 km/h	1016.6 hPa	10 km

Notar que podemos ordenar por filas, pero también por columnas:

```
df.sort_index(axis=1)
```

Fecha_Hora	Estado	HR (%)	Presión	ST (°C)	T (°C)	Viento dirección	Viento velocidad	Visibilidad
2023-08-14 00:00:00	Nublado	37	1014.8 hPa	No se calcula	12.8	Noroeste	9 km/h	10 km
2023-08-13 23:00:00	Nublado	37	1015.2 hPa	No se calcula	13	Noroeste	16 km/h	10 km
2023-08-13 22:00:00	Nublado	43	1015.6 hPa	No se calcula	13.2	Norte	7 km/h	10 km
2023-08-13 21:00:00	Mayormente nublado	38	1015 hPa	No se calcula	13.8	Norte	11 km/h	10 km
2023-08-13 20:00:00	Mayormente nublado	33	1015.2 hPa	No se calcula	14.9	Norte	9 km/h	10 km
2023-08-13 19:00:00	Mayormente nublado	32	1015.6 hPa	No se calcula	14.7	Norte	13 km/h	10 km
...	...	...	...	...	...	...	...	...
2023-08-13 04:00:00	Despejado	70	1023.3 hPa	No se calcula	3.4	Calma	nan	10 km
2023-08-13 03:00:00	Despejado	72	1023.7 hPa	No se calcula	3.7	Calma	nan	10 km

`rename` permite cambiar de nombre a las columnas elegidas. Tampoco se modifica el dataframe, a no ser que se utilice la opción inplace=True:

```
df.rename(columns={'T (°C)': 'Temperatura', 'HR (%)': 'Humedad relativa'})
```

Fecha_Hora	Estado	Temperatura	ST (°C)	Humedad relativa	Viento dirección	Viento velocidad	Presión	Visibilidad
2023-08-14 00:00:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km
2023-08-13 23:00:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km
2023-08-13 22:00:00	Nublado	13.2	No se calcula	43	Norte	7 km/h	1015.6 hPa	10 km
2023-08-13 21:00:00	Mayormente nublado	13.8	No se calcula	38	Norte	11 km/h	1015 hPa	10 km
2023-08-13 20:00:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km
2023-08-13 19:00:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km
...	...	...	...	...	...	...	...	
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km

7.6. Seleccionar y modificar valores

Partimos de nuestro dataframe habitual:

```
df = pd.read_csv("buenos_aires_20230813.txt", parse_dates=[['Fecha', 'Hora']],
                 sep=",", encoding = "ISO-8859-1")
df = df.set_index('Fecha_Hora')
```

Para seleccionar una columna de un dataframe podemos utilizar el operador []. Esto nos devolverá una serie (reducimos su dimensión):

```
s = df['Estado']   # También podría ponerse s = df.Estado
s
```

```
Fecha_Hora
2023-08-14 00:00:00                Nublado
2023-08-13 23:00:00                Nublado
2023-08-13 22:00:00                Nublado
2023-08-13 21:00:00      Mayormente nublado
2023-08-13 20:00:00      Mayormente nublado
2023-08-13 19:00:00      Mayormente nublado
2023-08-13 18:00:00    Parcialmente nublado
2023-08-13 17:00:00    Parcialmente nublado
2023-08-13 16:00:00            Algo nublado
2023-08-13 15:00:00            Algo nublado
2023-08-13 14:00:00     Ligeramente nublado
2023-08-13 13:00:00     Ligeramente nublado
2023-08-13 12:00:00     Ligeramente nublado
2023-08-13 11:00:00     Ligeramente nublado
2023-08-13 10:00:00     Ligeramente nublado
2023-08-13 09:00:00     Ligeramente nublado
2023-08-13 08:00:00     Ligeramente nublado
2023-08-13 07:00:00               Despejado
2023-08-13 06:00:00               Despejado
2023-08-13 05:00:00               Despejado
2023-08-13 04:00:00               Despejado
2023-08-13 03:00:00               Despejado
Name: Estado, dtype: object
```

Para obtener un dataframe tendríamos que usar [[]]:

```
df[['T (°C)', 'HR (%)']]
```

Fecha_Hora	T (°C)	HR (%)
2023-08-14 00:00:00	12.8	37
2023-08-13 23:00:00	13	37
2023-08-13 22:00:00	13.2	43
2023-08-13 21:00:00	13.8	38
2023-08-13 20:00:00	14.9	33
2023-08-13 19:00:00	14.7	32
...	...	...
2023-08-13 03:00:00	3.7	72

De la misma forma, aplicando este operador al índice de una serie, nos devuelve un valor:

```
s['2023-08-13 12:00:00']
```

```
14.4
```

También podemos seleccionar varias filas:

```
df['2023-08-13 12:00:00':'2023-08-13 14:00:00']
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
2023-08-13 14:00:00	Ligeramente nublado	16.3	No se calcula	23	Norte	13 km/h	1019.4 hPa	10 km
2023-08-13 13:00:00	Ligeramente nublado	15.2	No se calcula	25	Norte	14 km/h	1020.6 hPa	10 km
2023-08-13 12:00:00	Ligeramente nublado	14.4	No se calcula	27	Norte	7 km/h	1021.7 hPa	10 km

o usando la posición, como se hace en python estándar

```
df[10:13]
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
2023-08-13 14:00:00	Ligeramente nublado	16.3	No se calcula	23	Norte	13 km/h	1019.4 hPa	10 km
2023-08-13 13:00:00	Ligeramente nublado	15.2	No se calcula	25	Norte	14 km/h	1020.6 hPa	10 km
2023-08-13 12:00:00	Ligeramente nublado	14.4	No se calcula	27	Norte	7 km/h	1021.7 hPa	10 km

De todas formas, en un programa es mejor utilizar los métodos *loc* e *iloc* que están más optimizados. Además estos métodos nos evitan problemas relacionados con vistas y copias.

loc permite seleccionar subconjuntos de nuestra estructura (serie o dataframe), según etiquetas. Para seleccionar columnas:

```
df[['T (°C)', 'Estado']]
```

Fecha_Hora	T (°C)	Estado
2023-08-14 00:00:00	12.8	Nublado
2023-08-13 23:00:00	13	Nublado
2023-08-13 22:00:00	13.2	Nublado
2023-08-13 21:00:00	13.8	Mayormente nublado
2023-08-13 20:00:00	14.9	Mayormente nublado
2023-08-13 19:00:00	14.7	Mayormente nublado
...	...	...
2023-08-13 03:00:00	3.7	Despejado

```
df.loc[:, ('T (°C)', 'Estado')]
```

Fecha_Hora	T (°C)	Estado
2023-08-14 00:00:00	12.8	Nublado
2023-08-13 23:00:00	13	Nublado
2023-08-13 22:00:00	13.2	Nublado
2023-08-13 21:00:00	13.8	Mayormente nublado
2023-08-13 20:00:00	14.9	Mayormente nublado
2023-08-13 19:00:00	14.7	Mayormente nublado
...	...	...
2023-08-13 03:00:00	3.7	Despejado

y para seleccionar filas:

```
df.loc['2023-08-13 12:00:00', :]
```

```
Estado              Ligeramente nublado
T (°C)                             14.4
ST (°C)                 No se calcula
HR (%)                             27
Viento dirección                 Norte
Viento velocidad                 7 km/h
Presión                       1021.7 hPa
Visibilidad                        10 km
Name: 2023-08-13 12:00:00, dtype: object
```

o ambos a la vez:

```
df.loc['2023-08-13 12:00:00':'2023-08-13 14:00:00',
       ('T (°C)', 'Estado')]
```

Fecha_Hora	T (°C)	Estado
2023-08-13 14:00:00	16.3	Ligeramente nublado
2023-08-13 13:00:00	15.2	Ligeramente nublado
2023-08-13 12:00:00	14.4	Ligeramente nublado

Nota: observar que en la selección por etiquetas, tanto el elemento inicial como el final se incluyen (al contrario de lo que ocurre en una lista)

iloc selecciona subconjuntos según su posición. De forma equivalente a los ejemplos anteriores:

```
df.iloc[2]
```

```
Estado                    Nublado
T (°C)                       13.2
ST (°C)             No se calcula
HR (%)                         43
Viento dirección            Norte
Viento velocidad           7 km/h
Presión                 1015.6 hPa
Visibilidad                 10 km
Name: 2023-08-13 22:00:00, dtype: object
```

o también:

```
df.iloc[2, :]
```

```
Estado                    Nublado
T (°C)                       13.2
ST (°C)             No se calcula
HR (%)                         43
Viento dirección            Norte
Viento velocidad           7 km/h
Presión                 1015.6 hPa
Visibilidad                 10 km
Name: 2023-08-13 22:00:00, dtype: object
```

```
df = df.reset_index()
df.loc[2:4, ('T (°C)', 'Estado')]
```

	T (°C)	Estado
2	13.2	Nublado
3	13.8	Mayormente nublado
4	14.9	Mayormente nublado

```
df.iloc[2:5, 1:3]
```

	Estado	T (°C)
2	Nublado	13.2
3	Mayormente nublado	13.8
4	Mayormente nublado	14.9

También se pueden hacer selecciones discontinuas, como haríamos en python estándar:

```
df.iloc[[1,2,4], [2,5]]
```

	T (°C)	Viento dirección
1	13	Noroeste
2	13.2	Norte
4	14.9	Norte

E incluso un elemento:

```
df.iloc[0,0]
```

```
Timestamp('2023-08-14 00:00:00')
```

Aunque para seleccionar elementos es mejor utilizar el método *at* (o el *iat*), que son más rápidos y eficientes:

```
df = df.set_index('Fecha_Hora')
df.at['2023-08-13 23:00:00', 'HR (%)']
37
df.iat[1,4]
'Noroeste'
```

Otra forma de realizar selecciones es mediante la **indexación booleana**, que nos permite seleccionar valores de acuerdo a una condición relativa al dataframe.

Por ejemplo, si queremos seleccionar las filas para las que la GHI es mayor de 500, tenemos que usar esta condición:

```
df.index.hour==15
array([False, False, False, False, False, False, False, False, False,
        True, False, False, False, False, False, False, False, False,
       False, False, False, False])
```

y si introducimos esta serie de booleanos en nuestro dataframe, imponemos la condición:

```
df[df.index.hour==15]
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad
2023-08-13 15:00:00	Algo nublado	16.9	No se calcula	24	Norte	16 km/h	1017.7 hPa	10 km

Se pueden utilizar operadores booleanos (&, |, ~) para construir condiciones más complicadas:

```
df[(df['T (°C)']>14.0) & (df['HR (%)']>30.0)]
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad
2023-08-13 20:00:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km
2023-08-13 19:00:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km

No solo se pueden realizar selecciones sobre filas o columnas, también sobre valores. En este caso se da el valor np.nan a los elementos donde no se cumpla la condición:

```
df_num = df[['T (°C)', 'HR (%)']]
(df_num>15) & (df_num<40)
```

Fecha_Hora	T (°C)	HR (%)
2023-08-14 00:00:00	False	True
2023-08-13 23:00:00	False	True
2023-08-13 22:00:00	False	False
2023-08-13 21:00:00	False	True
2023-08-13 20:00:00	False	True
2023-08-13 19:00:00	False	True
2023-08-13 18:00:00	True	True
2023-08-13 17:00:00	True	True
2023-08-13 16:00:00	True	True
2023-08-13 15:00:00	True	True
2023-08-13 14:00:00	True	True
2023-08-13 13:00:00	True	True
2023-08-13 12:00:00	False	True
2023-08-13 11:00:00	False	True
2023-08-13 10:00:00	False	True
2023-08-13 09:00:00	False	False
2023-08-13 08:00:00	False	False
2023-08-13 07:00:00	False	False
2023-08-13 06:00:00	False	False
2023-08-13 05:00:00	False	False
2023-08-13 04:00:00	False	False
2023-08-13 03:00:00	False	False

```
df_num[(df_num>15) & (df_num<40)]
```

Fecha_Hora	T (°C)	HR (%)
2023-08-14 00:00:00	nan	37
2023-08-13 23:00:00	nan	37
2023-08-13 22:00:00	nan	nan
2023-08-13 21:00:00	nan	38
2023-08-13 20:00:00	nan	33
2023-08-13 19:00:00	nan	32
2023-08-13 18:00:00	15.6	26
2023-08-13 17:00:00	16.2	25
2023-08-13 16:00:00	17.2	25
2023-08-13 15:00:00	16.9	24
2023-08-13 14:00:00	16.3	23
2023-08-13 13:00:00	15.2	25
2023-08-13 12:00:00	nan	27
2023-08-13 11:00:00	nan	30
2023-08-13 10:00:00	nan	34
2023-08-13 09:00:00	nan	nan
2023-08-13 08:00:00	nan	nan
2023-08-13 07:00:00	nan	nan
2023-08-13 06:00:00	nan	nan
2023-08-13 05:00:00	nan	nan
2023-08-13 04:00:00	nan	nan
2023-08-13 03:00:00	nan	nan

Se pueden dar valores a filas, columnas o elementos determinados aplicando la sintaxis y métodos anteriores:

```
df.loc['2023-08-13 09:00:00.0':'2023-08-13 11:00:00.0',
       ('T (°C)', 'HR (%)')] = 5.0
df
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
...	...	...	...	...	...	...	...	...
2023-08-13 13:00:00	Ligeramente nublado	15.2	No se calcula	25	Norte	14 km/h	1020.6 hPa	10 km
2023-08-13 12:00:00	Ligeramente nublado	14.4	No se calcula	27	Norte	7 km/h	1021.7 hPa	10 km
2023-08-13 11:00:00	Ligeramente nublado	5	No se calcula	5	Noroeste	5 km/h	1023.2 hPa	10 km
2023-08-13 10:00:00	Ligeramente nublado	5	No se calcula	5	Norte	9 km/h	1023.3 hPa	10 km
2023-08-13 09:00:00	Ligeramente nublado	5	7.5	5	Noroeste	7 km/h	1022.9 hPa	10 km
2023-08-13 08:00:00	Ligeramente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km
...	...	...	...	...	...	...	...	...

También se pueden crear nuevas columnas:

```
df['T*2']=df['T (°C)']**2
```

Para aplicar una función a toda una columna usaremos la función *apply*:

```
from numpy import sqrt
df['T (°C)'].apply(np.sqrt)
```

```
Fecha_Hora
2023-08-14 00:00:00    3.577709
2023-08-13 23:00:00    3.605551
2023-08-13 22:00:00    3.633180
2023-08-13 21:00:00    3.714835
2023-08-13 20:00:00    3.860052
2023-08-13 19:00:00    3.834058
2023-08-13 18:00:00    3.949684
2023-08-13 17:00:00    4.024922
2023-08-13 16:00:00    4.147288
2023-08-13 15:00:00    4.110961
2023-08-13 14:00:00    4.037326
2023-08-13 13:00:00    3.898718
2023-08-13 12:00:00    3.794733
2023-08-13 11:00:00    2.236068
2023-08-13 10:00:00    2.236068
2023-08-13 09:00:00    2.236068
2023-08-13 08:00:00    2.302173
2023-08-13 07:00:00    2.000000
2023-08-13 06:00:00    1.760682
2023-08-13 05:00:00    1.788854
2023-08-13 04:00:00    1.843909
2023-08-13 03:00:00    1.923538
Name: T (°C), dtype: float64
```

Hay una gran cantidad de funciones para modificar columnas. Por ejemplo, si queremos formatear la velocidad del viento, y convertirla en un valor numérico para poder operar con ella, podemos hacer lo siguiente:

```
df['Viento velocidad'].str.slice(stop=-5).apply(float)
```

```
Fecha_Hora
2023-08-14 00:00:00     9.0
2023-08-13 23:00:00    16.0
2023-08-13 22:00:00     7.0
2023-08-13 21:00:00    11.0
2023-08-13 20:00:00     9.0
```

```
2023-08-13 19:00:00    13.0
2023-08-13 18:00:00     7.0
2023-08-13 17:00:00    14.0
2023-08-13 16:00:00    16.0
2023-08-13 15:00:00    16.0
2023-08-13 14:00:00    13.0
2023-08-13 13:00:00    14.0
2023-08-13 12:00:00     7.0
2023-08-13 11:00:00     5.0
2023-08-13 10:00:00     9.0
2023-08-13 09:00:00     7.0
2023-08-13 08:00:00     3.0
2023-08-13 07:00:00     3.0
2023-08-13 06:00:00     NaN
2023-08-13 05:00:00     NaN
2023-08-13 04:00:00     NaN
2023-08-13 03:00:00     NaN
Name: Viento velocidad, dtype: float64
```

7.7. Entrada / salida

Aunque es posible definir un dataframe directamente, como hemos hecho en apartados anteriores, es más habitual leer los datos de uno varios ficheros. Para leer ficheros csv (el caso más habitual) utilizaremos la función *read_csv*.

Esta función tiene una gran cantidad de argumentos opcionales posibles, aunque en su versión más simple, solo necesita un argumento, el nombre de fichero. Otros argumentos que usaremos a menudo son: *sep*, *header*, *skiprows*, *index_col*, *names*, *parse_dates* y *usecols*.

Por ejemplo, si queremos leer la columna de la temperatura (T (°C)) del fichero *buenos_aires_20230813.txt*:

```
!head buenos_aires_20230813.txt
```

```
Fecha,Hora,Estado,T (C),ST (C),HR (%),Viento direccion,Viento velocidad,Presion,Visibilidad
14/08/2023,0:00,Nublado,12.8,No se calcula,37,Noroeste,9 km/h,1014.8 hPa,10 km
13/08/2023,23:00,Nublado,13,No se calcula,37,Noroeste,16 km/h,1015.2 hPa,10 km
13/08/2023,22:00,Nublado,13.2,No se calcula,43,Norte,7 km/h,1015.6 hPa,10 km
13/08/2023,21:00,Mayormente nublado,13.8,No se calcula,38,Norte,11 km/h,1015 hPa,10 km
13/08/2023,20:00,Mayormente nublado,14.9,No se calcula,33,Norte,9 km/h,1015.2 hPa,10 km
13/08/2023,19:00,Mayormente nublado,14.7,No se calcula,32,Norte,13 km/h,1015.6 hPa,10 km
13/08/2023,18:00,Parcialmente nublado,15.6,No se calcula,26,Norte,7 km/h,1015.6 hPa,10 km
13/08/2023,17:00,Parcialmente nublado,16.2,No se calcula,25,Norte,14 km/h,1016 hPa,10 km
13/08/2023,16:00,Algo nublado,17.2,No se calcula,25,Norte,16 km/h,1016.6 hPa,10 km
```

escribiríamos, por ejemplo:

```
df = pd.read_csv("buenos_aires_20230813.txt",
                 sep=",", encoding = "ISO-8859-1", index_col=0, header=0, usecols=[0,3])
df
```

Fecha	T (°C)
14/08/2023	12.8
13/08/2023	13
13/08/2023	13.2
13/08/2023	13.8
13/08/2023	14.9
13/08/2023	14.7
...	...
13/08/2023	3.7

- *encoding="ISO-8859-1"* indica que no estamos utilizando las letras del inglés, sino las del español, (así podemos usar tildes).
- *header=0* indica que la primera línea es la cabecera.
- *usecols=[0,3]* indica que tomamos la primera (la fecha) y la cuarta columna (la temperatura).

Notar que solo ha tomado la primera columna, la fecha, como índice

Observar que read_csv nos permite hacer conversiones automáticamente, por ejemplo nos permite crear el índice a la vez que leemos el fichero. Pero podemos hacer esto mismo en dos pasos:

```
df = pd.read_csv("buenos_aires_20230813.txt",
                 sep=",", encoding = "ISO-8859-1", header=0, usecols=[0,3])
df = df.set_index('Fecha')
```

Aunque read_csv trata de convertir al tipo adecuado cada columna (por ejemplo para *floats*), no hace esta conversión para objetos tipo datetime. Para ello podemos usar el argumento *parse_dates*, que además nos permite combinar el campo de la fecha y de la hora y así obtener un campo datetime bien formado:

```
df = pd.read_csv("buenos_aires_20230813.txt", parse_dates=[[0,1]], sep=",",
                 index_col=0, encoding = "ISO-8859-1")
# También podríamos haber tomado parse_dates=[['Fecha', 'Hora']]
df
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
2023-08-14 00:00:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km
2023-08-13 23:00:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km
2023-08-13 22:00:00	Nublado	13.2	No se calcula	43	Norte	7 km/h	1015.6 hPa	10 km
2023-08-13 21:00:00	Mayormente nublado	13.8	No se calcula	38	Norte	11 km/h	1015 hPa	10 km
2023-08-13 20:00:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km
2023-08-13 19:00:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km
...	...	...	...	...	...	...	...	...
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km

Otra alternativa es hacer esta conversión separadamente, utilizando la función *to_datetime*:

```
df = pd.read_csv("buenos_aires_20230813.txt", sep=",", encoding = "ISO-8859-1")
df["Fecha_Hora"] = pd.to_datetime(df["Fecha"] + " " + df["Hora"], format="%d/%m/%Y %H:%M")
df
```

	Fecha	Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Presión	Visibilidad	Fecha_Hora
0	14/08/2023	0:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km	2023-08-14 00:00:00
1	13/08/2023	23:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km	2023-08-13 23:00:00
2	13/08/2023	22:00	Nublado	13.2	No se calcula	43	Norte	7 km/h	1015.6 hPa	10 km	2023-08-13 22:00:00
3	13/08/2023	21:00	Mayormente nublado	13.8	No se calcula	38	Norte	11 km/h	1015 hPa	10 km	2023-08-13 21:00:00
4	13/08/2023	20:00	Mayormente nublado	14.9	No se calcula	33	Norte	9 km/h	1015.2 hPa	10 km	2023-08-13 20:00:00
5	13/08/2023	19:00	Mayormente nublado	14.7	No se calcula	32	Norte	13 km/h	1015.6 hPa	10 km	2023-08-13 19:00:00
	...	...	...	...	...	...	...	...	...	...	...
21	13/08/2023	3:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km	2023-08-13 03:00:00

Ahora ya tiene el formato datetime:

```
print(df.dtypes)
```

```
Fecha                          object
Hora                           object
Estado                         object
T (°C)                        float64
ST (°C)                        object
HR (%)                          int64
Viento dirección               object
Viento velocidad               object
Presión                        object
Visibilidad                    object
Fecha_Hora          datetime64[ns]
dtype: object
```

También disponemos del argumento *converters* para forzar cambios de tipo que *read_csv* no haga apropiadamente.

O, un poco más corto:

Nota: En general *read_csv* espera que el fichero csv tenga siempre el número de columnas correcto. De todas formas, el argumento *on_bad_lines* y *warn_bad_lines* puede ayudarnos a controlar ficheros mal formateados, hasta cierto punto.

También existe la función *read_fwf*, que permite leer ficheros donde no hay separadores, sino que cada columna tiene una longitud fija.

Para volcar un dataframe a un fichero utilizaremos el método *to_csv*. Tiene argumentos parecidos.

Además de ficheros csv, Pandas es capaz de leer directamente muchos otros formatos, como HDF (*read_hdf*), json (*read_json*) o SQL (*read_sql*), o tablas de html (*read_html*).

En particular, es capaz de leer ficheros Excel también (utiliza el módulo xlrd, que no es parte de la librería estándar).

7.8. Datos nulos

Pandas utiliza el valor *np.nan* para representar datos nulos o no conocidos. Por ejemplo en nuestro dataframe con datos de Buenos Aires, no hay dato de velocidad del viento entre las 3 y las 6 de la madrugada:

```
df = pd.read_csv("buenos_aires_20230813.txt", parse_dates=[[0,1]], sep=",",
                 index_col=0, encoding = "ISO-8859-1")
df
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad
...	...	...	...	...	...	...	...	...
2023-08-13 08:00:00	Ligeramente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km
2023-08-13 07:00:00	Despejado	4	No se calcula	70	Oeste	3 km/h	1023.2 hPa	10 km
2023-08-13 06:00:00	Despejado	3.1	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 05:00:00	Despejado	3.2	No se calcula	71	Calma	nan	1022.8 hPa	10 km
2023-08-13 04:00:00	Despejado	3.4	No se calcula	70	Calma	nan	1023.3 hPa	10 km
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	nan	1023.7 hPa	10 km

Por defecto estos valores no son incluidos en las operaciones, aunque es conveniente examinar cada función para estar seguro. Por ejemplo:

```
len(df)
```

```
22
```

pero por otra parte:

```
df.count()
```

```
Estado              22
T (°C)              22
ST (°C)             22
HR (%)              22
Viento dirección    22
Viento velocidad    18
Presión             22
Visibilidad         22
dtype: int64
```

Algunas funciones, como `mean`, tienen un argumento (*skipna* en este caso), para decidir si se incluyen o no los NaN (por defecto skipna=True, y no los incluirá).

Por ejemplo, si convertimos la velocidad del viento a un valor numérico, y calculamos su media, vemos que:

```
df['Vel.Viento']=df['Viento velocidad'].str.slice(stop=-5).apply(float)
df['Vel.Viento'].mean()
```

```
9.944444444444445
```

```
df['Vel.Viento'][0:18].mean()
```

```
9.944444444444445
```

```
df['Vel.Viento'].mean(skipna=False)
```

```
nan
```

Las funciones `isnull` y `notnull` nos dirán si un valor es igual a NaN. ¡No hagáis comparaciones del tipo x==np.nan!

```
pd.isnull(df)
```

Se pueden eliminar las filas con valores NaN con el método `dropna`:

```
df.dropna()  # Notar que para que se guarden los cambios debemos incluir inplace=True
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad	Vel.Viento
2023-08-14 00:00:00	Nublado	12.8	No se calcula	37	Noroeste	9 km/h	1014.8 hPa	10 km	9
2023-08-13 23:00:00	Nublado	13	No se calcula	37	Noroeste	16 km/h	1015.2 hPa	10 km	16
...	...	...	...	...	...	...	...	...	...
2023-08-13 09:00:00	Ligera-mente nublado	8.6	7.5	44	Noroeste	7 km/h	1022.9 hPa	10 km	7
2023-08-13 08:00:00	Ligera-mente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km	3
2023-08-13 07:00:00	Despejado	4	No se calcula	70	Oeste	3 km/h	1023.2 hPa	10 km	3

Nota: Para comprobar si un dataframe está vacío utilizaremos el atributo `empty`

Para rellenar con otros valores utilizaremos el método `fillna`. Podemos rellenar con un valor concreto,

```
df.fillna(-999.0)
```

Fecha_Hora	Estado	T (°C)	ST (°C)	HR (%)	Viento dirección	Viento velocidad	Pre-sión	Visi-bilidad	Vel.Viento
...	...	...	...	...	...	...	...	...	...
2023-08-13 08:00:00	Ligera-mente nublado	5.3	No se calcula	59	Norte	3 km/h	1023.2 hPa	10 km	3
2023-08-13 07:00:00	Despejado	4	No se calcula	70	Oeste	3 km/h	1023.2 hPa	10 km	3
2023-08-13 06:00:00	Despejado	3.1	No se calcula	71	Calma	-999.0	1022.8 hPa	10 km	-999
2023-08-13 05:00:00	Despejado	3.2	No se calcula	71	Calma	-999.0	1022.8 hPa	10 km	-999
2023-08-13 04:00:00	Despejado	3.4	No se calcula	70	Calma	-999.0	1023.3 hPa	10 km	-999
2023-08-13 03:00:00	Despejado	3.7	No se calcula	72	Calma	-999.0	1023.7 hPa	10 km	-999

o utilizar un método; por ejemplo rellenando con el próximo valor no nulo:

```
df.fillna(method='bfill')
```

También se puede interpolar entre los valores no nulos más cercanos, utilizando el método `interpolate`.

Ejercicio

El reindexado es un método que suele producir muchos valores nulos, que después habrá que rellenar de forma adecuada. Puedes practicar los métodos anteriores en el dataframe *dfmissing*, que obtenemos con este código que previamente ha leído datos de Buenos Aires cada 3 horas, y ha rellenado las horas faltantes con nulos:

```
df = pd.read_csv("buenos_aires_20230813_3h.txt", parse_dates=[[0,1]], sep=",",
                 index_col=0, encoding = "ISO-8859-1")[['T (°C)', 'HR (%)']].sort_index()
```

```
dfmissing = df.reindex(index=pd.date_range("20230813 03:00", "20230814 00:00", freq="H"))
dfmissing
```

	T (°C)	HR (%)
2023-08-13 03:00:00	3.7	72
2023-08-13 04:00:00	nan	nan
2023-08-13 05:00:00	nan	nan
2023-08-13 06:00:00	3.1	71
2023-08-13 07:00:00	nan	nan
2023-08-13 08:00:00	nan	nan
2023-08-13 09:00:00	8.6	44
...	...	...
2023-08-13 21:00:00	13.8	38
2023-08-13 22:00:00	nan	nan
2023-08-13 23:00:00	nan	nan
2023-08-14 00:00:00	12.8	37

7.9. Merge, concat, append

Pandas permite combinar varios dataframes (o series), de dos formas básicas: - Concatenar dos (o más) dataframes con las mismas columnas simplemente "pegándolos", para lo que usaremos la función *concat*. Otra función relacionada es *append*. - Unir dos dataframes por medio de un campo común, de forma que el nuevo dataframe tendrá las columnas de ambos. Esta operación se realiza por medio de la función *merge*.

Si tenemos estos dataframes, con datos del CENAOS:

```
# Observaciones en dos horas distintas:
dfobs1 = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                       'Date': [datetime(2023,8,27,12),
                                datetime(2023,8,27,12),
                                datetime(2023,8,27,12)],
                       'PCP 1h':[0.0, 0.0, 0.0],
                       'PCP 6h':[7.6, 0.1, 0.0],
                       'PCP 12h':[11.7, 0.1, 7.3],
                       'PCP 24h':[11.7, 19.2, 18.8]})

dfobs2 = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                       'Date': [datetime(2023,8,28,2),
                                datetime(2023,8,28,2),
                                datetime(2023,8,28,2)],
                       'PCP 1h':[1.5, 0.0, 0.0],
                       'PCP 6h':[1.5, 0.4, 0.0],
                       'PCP 12h':[1.5, 8.3, 64.0],
                       'PCP 24h':[13.2, 8.4, 71.3]})
# Predicciones (hipotéticas) en una hora determinada:
dfpred = pd.DataFrame({'Estacion':['Los Platos', 'Atima', 'Bomberos-Tegucigalpa'],
                       'Date': [datetime(2023,8,27,12),
                                datetime(2023,8,27,12),
                                datetime(2023,8,27,12)],
                       'PCPpred 1h':[0.0, 0.0, 0.0],
                       'PCPpred 6h':[5.0, 0.0, 0.0],
                       'PCPpred 12h':[10.0, 0.0, 5.0],
                       'PCPpred 24h':[15.0, 19.2, 25.0]})
```

Los podemos concatenar de esta forma:

```
dfobs = pd.concat([dfobs1, dfobs2])
dfobs
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8
0	Los Platos	2023-08-28 02:00:00	1.5	1.5	1.5	13.2
1	Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4
2	Bomberos-Tegucigalpa	2023-08-28 02:00:00	0	0	64	71.3

También se pueden concatenar según el otro eje (en este caso no tiene mucho sentido):

```
pd.concat([dfobs1, dfobs2], axis=1)
```

La segunda forma consiste en unir los dos dataframes por medio de un campo común, de forma similar a como se hace en el lenguaje SQL, utilizado habitualmente en bases de datos relacionales. Indicaremos los dos dataframes a unir, y el campo (o campos) respectivo que usaremos como nexo de unión:

```
df = pd.merge(dfobs, dfpred, left_on=['Estacion', 'Date'], right_on=['Estacion', 'Date'])
```

o, más sencillo en este caso:

```
df = pd.merge(dfobs, dfpred, on=['Estacion', 'Date'])
df
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25

Ahora podríamos calcular por ejemplo el error de la predicción fácilmente:

```
df["PCPerror 24h"] = abs(df["PCPpred 24h"] - df["PCP 24h"])
df
```

	Estacion	Date	PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h	PCPerror 24h
0	Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15	3.3
1	Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2	0
2	Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25	6.2

También se pueden usar índices para realizar la unión. Este es un método más rápido si los dataframes son grandes (notar que esto ocurre también en bases de datos):

```
dfobs = dfobs.set_index(['Estacion', 'Date'])
dfpred = dfpred.set_index(['Estacion', 'Date'])
df = pd.merge(dfobs, dfpred, left_index=True, right_index=True)
df
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h
Estacion	Date								
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25

Observar que las uniones cuando se usan índices únicos serán uno-a-uno, pero al utilizar columnas pueden ser muchos-a-muchos. Si no escogemos bien la unión, podemos obtener el producto cartesiano por error:

```
pd.merge(dfobs, dfpred, on='Estacion')
```

Estacion	PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCPpred 1h	PCPpred 6h	PCPpred 12h	PCPpred 24h
Los Platos	0	7.6	11.7	11.7	0	5	10	15
Los Platos	1.5	1.5	1.5	13.2	0	5	10	15
Atima	0	0.1	0.1	19.2	0	0	0	19.2
Atima	0	0.4	8.3	8.4	0	0	0	19.2
Bomberos-Tegucigalpa	0	0	7.3	18.8	0	0	5	25
Bomberos-Tegucigalpa	0	0	64	71.3	0	0	5	25

En los ejemplos anteriores vemos que al realizar el cruce, tomamos la intersección de los registros, o sea, cogemos solo aquellos donde coincide la columna unión. El argumento *how* nos permite escoger la intersección (con el valor *"inner"*, que es el usado por defecto), la unión (valor *"outer"*), o tomar uno de los dataframes como dominante (para los valores *"left"* y *"right"*). Para estos casos se pone el valor nulo en aquellos registros que no existían.

```
pd.merge(dfobs, dfpred, left_index=True, right_index=True, how='inner')
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h
Estacion	Date								
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25

```
pd.merge(dfobs, dfpred, left_index=True, right_index=True, how='outer')
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h
Estacion	Date								
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2
Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4	nan	nan	nan	nan
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25
Bomberos-Tegucigalpa	2023-08-28 02:00:00	0	0	64	71.3	nan	nan	nan	nan
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15
Los Platos	2023-08-28 02:00:00	1.5	1.5	1.5	13.2	nan	nan	nan	nan

```
pd.merge(dfobs, dfpred, left_index=True, right_index=True, how='left')
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h	PCP-pred 1h	PCP-pred 6h	PCP-pred 12h	PCP-pred 24h
Estacion	Date								
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7	0	5	10	15
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2	0	0	0	19.2
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8	0	0	5	25
Los Platos	2023-08-28 02:00:00	1.5	1.5	1.5	13.2	nan	nan	nan	nan
Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4	nan	nan	nan	nan
Bomberos-Tegucigalpa	2023-08-28 02:00:00	0	0	64	71.3	nan	nan	nan	nan

7.10. Agrupar

Agrupar datos es un proceso con tres partes diferenciadas:

- División de los datos en grupos según un cierto criterio
- Aplicación de una función a cada uno de los grupos de forma independiente
- Combinación de los grupos para formar una nueva estructura de datos

Esta operación proviene también del lenguaje SQL, donde se utiliza la expresión "group by" para hacer lo mismo.

Partimos de nuestro dataframe de ejemplo con 6 observaciones (2 observaciones para 3 estaciones):

```
dfobs
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h
Estacion	Date				
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8
Los Platos	2023-08-28 02:00:00	1.5	1.5	1.5	13.2
Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4
Bomberos-Tegucigalpa	2023-08-28 02:00:00	0	0	64	71.3

Para realizar el primer paso, la división en grupos, usaremos el método `groupby`. Se pueden utilizar muchos criterios o claves (*keys*) para dividir.

La clave más habitual es una columna (o varias) del dataframe:

```
grouped_st = dfobs.groupby('Estacion')
```

Nota: En realidad esto es "azúcar sintáctico", creado para facilitarnos la escritura de código. Los grupos no se crean hasta que no son utilizados.

Es posible iterar el objeto *groupby* para ver qué contiene cada grupo:

```
for name, group in grouped_st:
    print("*** Grupo %s ***"%str(name))
    print(group)
```

```
* Grupo Atima *
                             PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion Date
Atima    2023-08-27 12:00:00     0.0     0.1      0.1     19.2
         2023-08-28 02:00:00     0.0     0.4      8.3      8.4
* Grupo Bomberos-Tegucigalpa *
                                      PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion             Date
Bomberos-Tegucigalpa 2023-08-27 12:00:00     0.0     0.0      7.3     18.8
                     2023-08-28 02:00:00     0.0     0.0     64.0     71.3
* Grupo Los Platos *
                               PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion   Date
Los Platos 2023-08-27 12:00:00     0.0     7.6     11.7     11.7
           2023-08-28 02:00:00     1.5     1.5      1.5     13.2
```

También podemos dividir según el índice:

```
grouped_dt = dfobs.groupby(dfobs.index)
for name, group in grouped_dt:
    print("*** Grupo %s ***"%str(name))
    print(group)
```

```
* Grupo ('Atima', Timestamp('2023-08-27 12:00:00')) *
                             PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion Date
Atima    2023-08-27 12:00:00     0.0     0.1      0.1     19.2
* Grupo ('Atima', Timestamp('2023-08-28 02:00:00')) *
                             PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion Date
Atima    2023-08-28 02:00:00     0.0     0.4      8.3      8.4
* Grupo ('Bomberos-Tegucigalpa', Timestamp('2023-08-27 12:00:00')) *
                                      PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion             Date
Bomberos-Tegucigalpa 2023-08-27 12:00:00     0.0     0.0      7.3     18.8
```

```
* Grupo ('Bomberos-Tegucigalpa', Timestamp('2023-08-28 02:00:00')) *
                                        PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion            Date
Bomberos-Tegucigalpa 2023-08-28 02:00:00     0.0     0.0     64.0     71.3
* Grupo ('Los Platos', Timestamp('2023-08-27 12:00:00')) *
                                PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion   Date
Los Platos 2023-08-27 12:00:00     0.0     7.6     11.7     11.7
* Grupo ('Los Platos', Timestamp('2023-08-28 02:00:00')) *
                                PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion   Date
Los Platos 2023-08-28 02:00:00     1.5     1.5      1.5     13.2
```

O incluso aplicando una función:

```
def PCP_higher_than_10(df):
    return df['PCP 12h']>10

grouped_pcp10 = dfobs.groupby(PCP_higher_than_10(dfobs))
for name, group in grouped_pcp10:
    print("*** Grupo %s ***"%str(name))
    print(group)
```

```
* Grupo False *
                                        PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion            Date
Atima               2023-08-27 12:00:00     0.0     0.1      0.1     19.2
Bomberos-Tegucigalpa 2023-08-27 12:00:00     0.0     0.0      7.3     18.8
Los Platos          2023-08-28 02:00:00     1.5     1.5      1.5     13.2
Atima               2023-08-28 02:00:00     0.0     0.4      8.3      8.4
* Grupo True *
                                        PCP 1h  PCP 6h  PCP 12h  PCP 24h
Estacion            Date
Los Platos          2023-08-27 12:00:00     0.0     7.6     11.7     11.7
Bomberos-Tegucigalpa 2023-08-28 02:00:00     0.0     0.0     64.0     71.3
```

Además de las filas, también se pueden dividir las columnas, utilizando el argumento *axis*. Se puede encontrar una lista detallada de las claves posibles en https://pandas.pydata.org/pandas-docs/stable/groupby.html

El atributo *groups* es un diccionario cuyas claves son los nombres de los grupos, y los valores las etiquetas del eje correspondiente a cada grupo:

```
grouped_st.groups
```

```
{'Atima': [('Atima', 2023-08-27 12:00:00),
           ('Atima', 2023-08-28 02:00:00)],
 'Bomberos-Tegucigalpa': [('Bomberos-Tegucigalpa', 2023-08-27 12:00:00),
                          ('Bomberos-Tegucigalpa', 2023-08-28 02:00:00)],
 'Los Platos': [('Los Platos', 2023-08-27 12:00:00),
                ('Los Platos', 2023-08-28 02:00:00)]}
```

Se selecciona un grupo determinado con el método `get_group`:

```
grouped_st.get_group('Atima')
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h
Estacion	Date				
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4

Después de agrupar, podemos aplicar una función a cada grupo, aplicándola directamente al objeto *groupby*:

```
grouped_st.sum()
```

	PCP 1h	PCP 6h	PCP 12h	PCP 24h
Estacion				
Atima	0	0.5	8.4	27.6
Bomberos-Tegucigalpa	0	0	71.3	90.1
Los Platos	1.5	9.1	13.2	24.9

Podemos elegir una columna únicamente:

```
grouped_st['PCP 24h'].mean()
```

```
Estacion
Atima                   13.80
Bomberos-Tegucigalpa    45.05
Los Platos              12.45
Name: PCP 24h, dtype: float64
```

Y también aplicar diferentes funciones a distintas columnas, por medio del método `agg` (o `aggregate`):

```
grouped_st.agg({'PCP 12h': np.mean, 'PCP 24h': np.std})
```

	PCP 12h	PCP 24h
Estacion		
Atima	4.2	7.63675
Bomberos-Tegucigalpa	35.65	37.1231
Los Platos	6.6	1.06066

o si renombramos para que quede más bonito:

```
grouped_st.agg({'PCP 12h': np.mean, 'PCP 24h': np.std}).rename(
                  columns={'PCP 12h':'mean_PCP 12h', 'PCP 24h': 'std_PCP 24h'})
```

	mean_PCP 12h	std_PCP 24h
Estacion		
Atima	4.2	7.63675
Bomberos-Tegucigalpa	35.65	37.1231
Los Platos	6.6	1.06066

Ejercicio

Para este ejercicio utilizaremos datos de un fichero csv de la estación de Atlacomulco, extraído de la web del Servicio Meteorológico Nacional de México. Realice las siguientes operaciones:

1. Cargue el fichero en un dataframe de Pandas, convirtiendo las fechas a Timestamps, y tomando la fecha local como índice (pista: tendrá que saltarse las primeras líneas de metadatos con la opción *skiprows*).
2. Seleccione los registros con una temperatura mayor de 20 °C y humedad relativa menor de 40 %
3. Seleccione los registros que corresponden a las 12 horas hora local únicamente (para cada día).
4. Calcule la media y desviación estandar de la radiación solar
5. Cree dos dataframes nuevos, el primero incluyendo solo los datos correspondientes al día 25 de agosto y el segundo los datos del día 27 de agosto.
6. Una los dos dataframes en uno (con *concat*)
7. Divida el dataframe en tres más pequeños, de forma que cada uno contenga información únicamente sobre la temperatura, humedad relativa y radiación solar, respectivamente.
8. Vuelva a unir los tres dataframes en uno (con *merge*).
9. Calcule la temperatura máxima y mínima para cada día, del 24 al 28 de agosto (pista: tendrá que agrupar con groupby).

7.11. Reshaping: stack, pivot

Partimos de este dataframe (con su índice):

```
dfobs
```

		PCP 1h	PCP 6h	PCP 12h	PCP 24h
Estacion	Date				
Los Platos	2023-08-27 12:00:00	0	7.6	11.7	11.7
Atima	2023-08-27 12:00:00	0	0.1	0.1	19.2
Bomberos-Tegucigalpa	2023-08-27 12:00:00	0	0	7.3	18.8
Los Platos	2023-08-28 02:00:00	1.5	1.5	1.5	13.2
Atima	2023-08-28 02:00:00	0	0.4	8.3	8.4
Bomberos-Tegucigalpa	2023-08-28 02:00:00	0	0	64	71.3

El método `stack` "comprime" una de las columnas de un dataframe. El método inverso es `unstack`.

```
dfobs.stack()
```

```
Estacion             Date
Los Platos           2023-08-27 12:00:00  PCP 1h      0.0
                                          PCP 6h      7.6
                                          PCP 12h    11.7
                                          PCP 24h    11.7
Atima                2023-08-27 12:00:00  PCP 1h      0.0
                                          PCP 6h      0.1
                                          PCP 12h     0.1
                                          PCP 24h    19.2
Bomberos-Tegucigalpa 2023-08-27 12:00:00  PCP 1h      0.0
                                          PCP 6h      0.0
                                          PCP 12h     7.3
                                          PCP 24h    18.8
Los Platos           2023-08-28 02:00:00  PCP 1h      1.5
                                          PCP 6h      1.5
                                          PCP 12h     1.5
                                          PCP 24h    13.2
Atima                2023-08-28 02:00:00  PCP 1h      0.0
                                          PCP 6h      0.4
                                          PCP 12h     8.3
                                          PCP 24h     8.4
Bomberos-Tegucigalpa 2023-08-28 02:00:00  PCP 1h      0.0
                                          PCP 6h      0.0
                                          PCP 12h    64.0
                                          PCP 24h    71.3
dtype: float64
```

Para "pivotar" un dataframe usaremos la función `pivot_table`:

```
dfobs = dfobs.reset_index()
df_pivot = pd.pivot_table(dfobs, values='PCP 12h', index='Date', columns='Estacion')
df_pivot
```

Date	Atima	Bomberos-Tegucigalpa	Los Platos
2023-08-27 12:00:00	0.1	7.3	11.7
2023-08-28 02:00:00	8.3	64	1.5

Se puede pivotar más de una columna:

```
pd.pivot_table(dfobs, values=['PCP 1h', 'PCP 6h', 'PCP 12h', 'PCP 24h'],
               index='Date', columns='Estacion')
```

	PCP12h			PCP1h			PCP24h			PCP6h		
Estacion	Atima	Bomberos-Tegucigalpa	Los Platos	Atima	Bomberos-Tegucigalpa	Los Platos	Atima	Bomberos-Tegucigalpa	Los Platos	Atima	Bomberos-Tegucigalpa	Los Platos
Date												
2023-08-27 12:00:00	0.1	7.3	11.7	0	0	0	19.2	18.8	11.7	0.1	0	6
2023-08-28 02:00:00	8.3	64	1.5	0	0	1.5	8.4	71.3	13.2	0.4	0	5

La función `melt` es la función inversa de `pivot_table`:

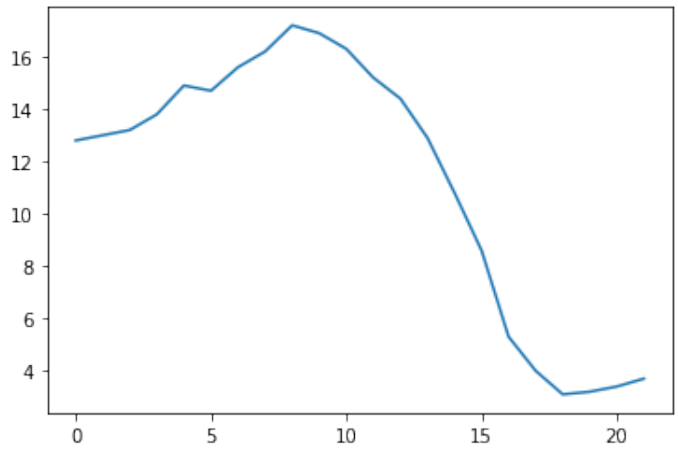
```
pd.melt(df_pivot.reset_index(), id_vars = 'Date', value_name = 'PCP 24h')
```

	Date	Estacion	PCP 24h
0	2023-08-27 12:00:00	Atima	0.1
1	2023-08-28 02:00:00	Atima	8.3
2	2023-08-27 12:00:00	Bomberos-Tegucigalpa	7.3
3	2023-08-28 02:00:00	Bomberos-Tegucigalpa	64
4	2023-08-27 12:00:00	Los Platos	11.7
5	2023-08-28 02:00:00	Los Platos	1.5

7.12. Gráficos

El objeto dataframe dispone de un método `plot` para realizar gráficos. Este método es simplemente un wrapper de la función `plot` de `matplolib.pyplot`, y se usa de forma similar:

```
df = pd.read_csv("buenos_aires_20230813.txt", parse_dates=[['Fecha', 'Hora']],
                 sep=",", encoding = "ISO-8859-1")
df["T (°C)"].plot()
```



Capítulo 8

Librería Matplotlib

(Basado en el tutorial https://matplotlib.org/stable/tutorials/introductory/quick_start.html)

Matplotlib es un paquete python para la generación de gráficos. Permite obtener resultados de calidad de forma rápida en gran variedad de formatos.

La idea de este breve repaso a matplotlib es tratar de comprender su funcionamiento básico. Entender los cambios que podemos realizar sobre los diferentes elementos para poder entender y modificar ejemplos más complejos.

Antes de empezar es buena idea echar un vistazo a la galería http://matplotlib.org/gallery.html para ver las posibilidades que nos ofrece.

8.1. Primera gráfica

Matplotlib posee un montón de módulos que pueden resultar confusos al principio. La forma más recomendable de empezar es centrarse en el módulo `pyplot`.

El módulo `pyplot` es el encargado de juntar toda la potencia de matplotlib. Se trata del punto de partida para:

- Preparar las figuras
- Pintar las gráficas
- Hacer modificaciones sobre las gráficas

Con este módulo, cada función que invoquemos realizará un cambio sobre la figura: pintar una línea, añadir una etiqueta, cambiar los límites de un eje,...

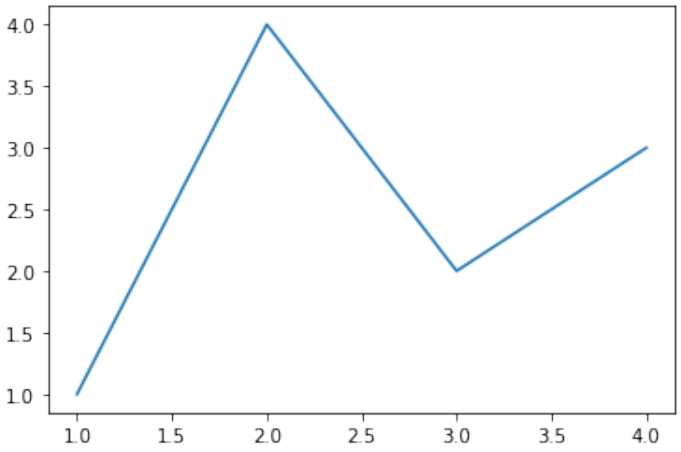
Las funciones que pintan los gráficos en general utilizarán un array de numpy o un objeto similar como entrada. Las series y dataframes de pandas suelen funcionar (aunque no para todas las funciones). En todo caso se pueden convertir a un array de numpy antes de plotearlos.

Importamos los módulos que utilizaremos:

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

Un ejemplo muy sencillo que podemos realizar con matplotlib es el siguiente:

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.show()
```

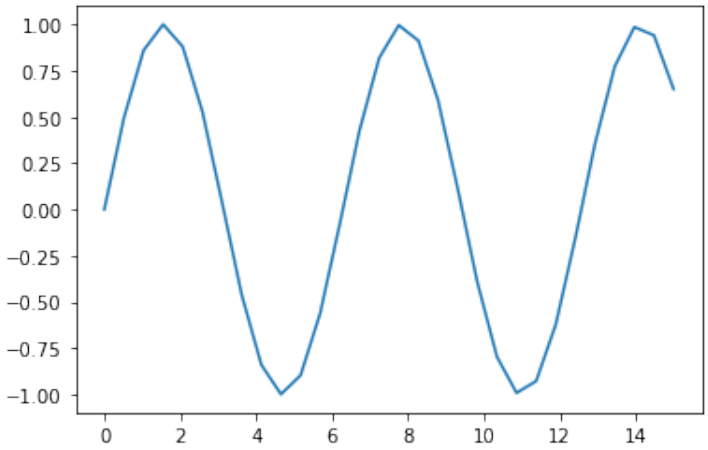


Simplemente hemos especificado las coordenadas x e y de los puntos que queremos conectar con líneas utilizando dos listas y todo lo demás ha funcionado solo.

Matplotlib ha realizado una serie de acciones por defecto como crear la figura (lienzo), crear una gráfica en el lienzo, asignar valores al eje X, etc.

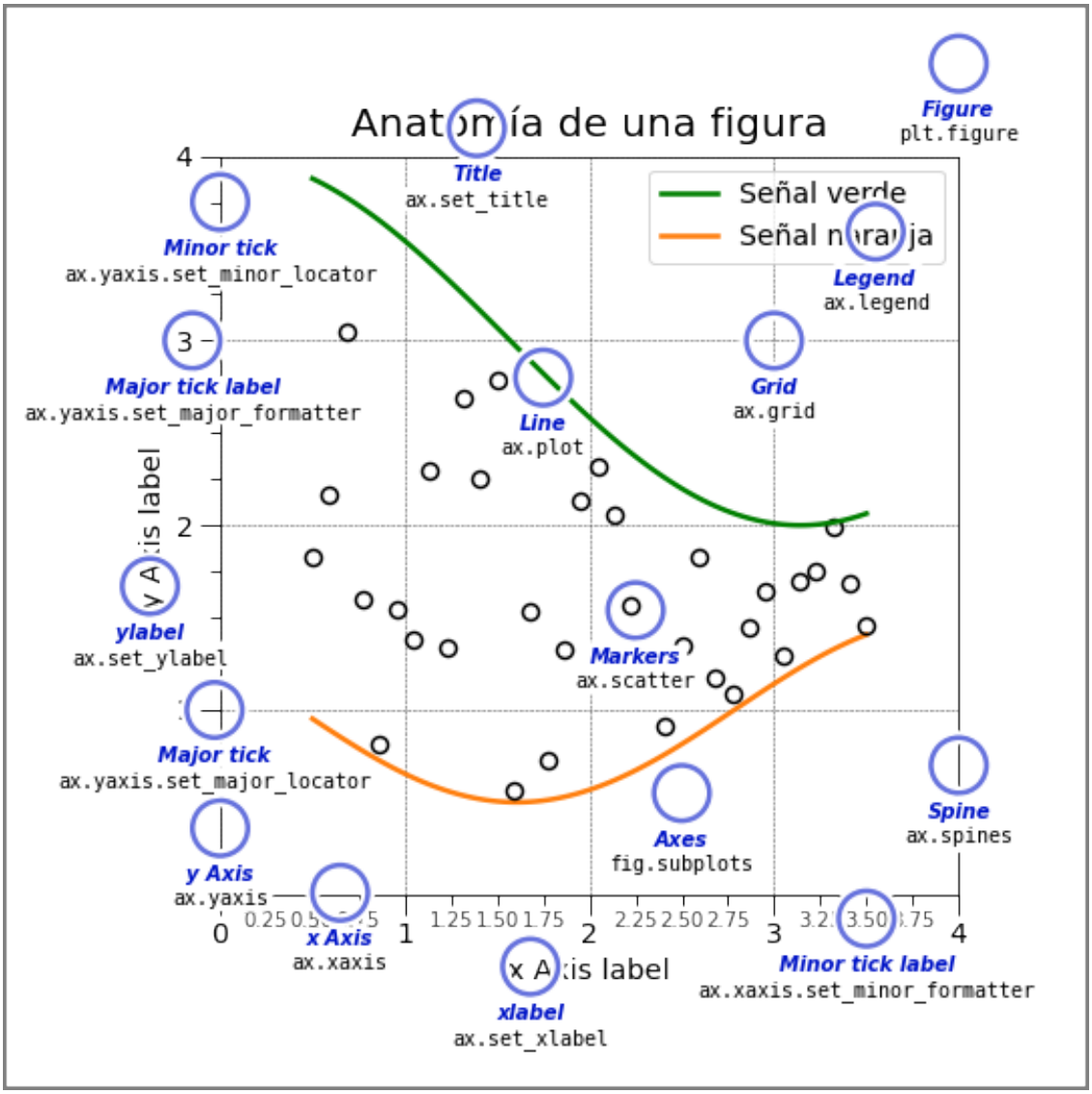
Podemos especificar estas coordenadas como una función:

```
x = np.linspace(0,15,30)
y = np.sin(x)
plt.plot(x,y)
```



8.2. Partes de una figura

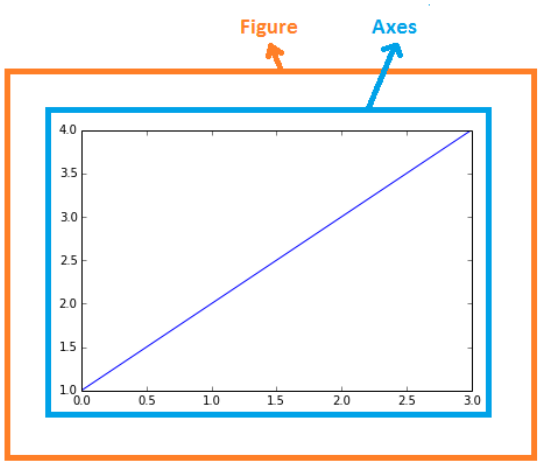
En una figura podemos encontrar una gran variedad de objetos diferentes que la componen:



El pintado se realiza a través del objeto `Figure` (http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure).

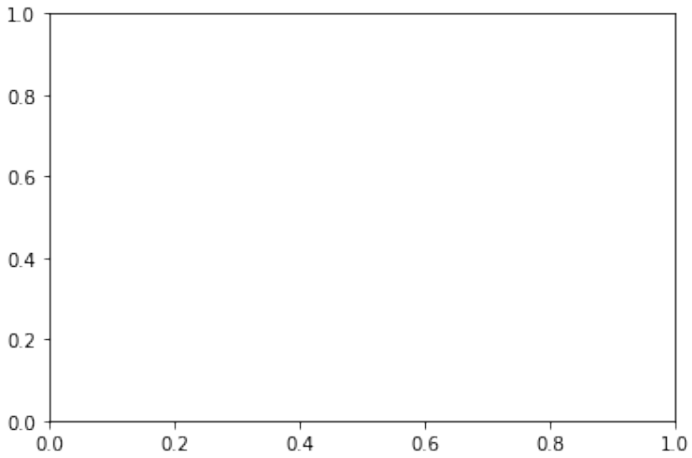
Se puede ver como el lienzo donde vamos a pintar nuestra/s gráfica/s. Es aquí donde especificaremos cosas como el tamaño, la resolución en dpi o su aspect ratio.

Cada `Figure` puede contener uno o varios `Axes`:



Un `Axes` es lo que identificamos como una gráfica. Puede ser una línea, un histograma, etc. A través de este objeto podemos acceder y modificar la apariencia, los ejes, etiquetas, título, leyenda, etc, por medio de distintos métodos.

```
fig = plt.figure()     # creamos un objeto Figure
ax = plt.subplot()     # cremos un objeto Axes
```



O, en una sola línea:

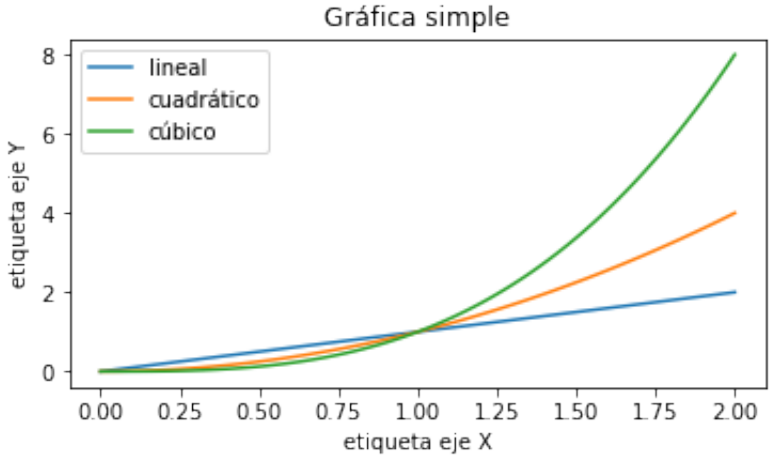
```
fig, ax = plt.subplots(figsize=(6, 4))  # aquí damos el tamaño de la figura
```

8.3. Explícito o implícito

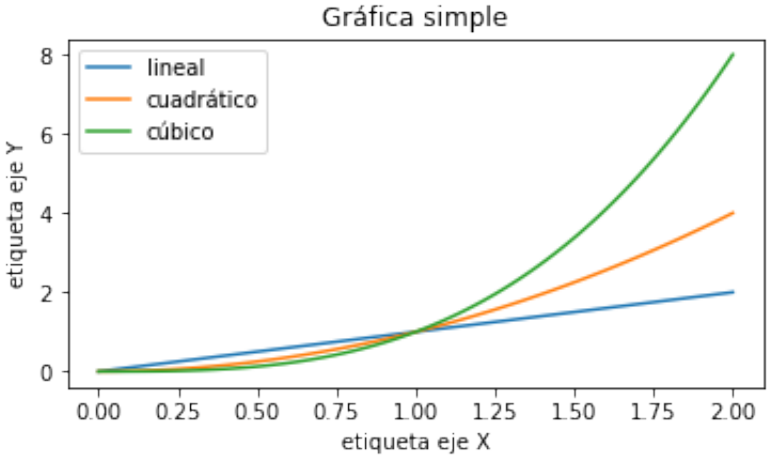
Hay dos formas diferentes de pintar una misma gráfica:

```
x = np.linspace(0, 2, 100)
# Método implícito

plt.figure(figsize=(5, 3), layout='constrained')

plt.plot(x, x, label='lineal')  # Plot some data on the (implicit) axes.
```

```
plt.plot(x, x**2, label='cuadrático')  # etc.
plt.plot(x, x**3, label='cúbico')

plt.xlabel('etiqueta eje X')
plt.ylabel('etiqueta eje Y')
plt.title("Gráfica simple")
plt.legend()
```




```
# Método explícito

fig, ax = plt.subplots(figsize=(5, 3), layout='constrained')

ax.plot(x, x, label='lineal')  # Plot some data on the axes.
ax.plot(x, x**2, label='cuadrático')  # Plot more data on the axes...
ax.plot(x, x**3, label='cúbico')  # ... and some more.

ax.set_xlabel('etiqueta eje X')  # Add an x-label to the axes.
ax.set_ylabel('etiqueta eje Y')  # Add a y-label to the axes.
ax.set_title("Gráfica simple")  # Add a title to the axes.
ax.legend()  # Add a legend.
```



El primer ejemplo, el método implícito, parece más claro y sencillo. Lo que está ocurriendo es que todos los métodos del objeto `Axes` están disponibles directamente desde `pyplot`. Así pues cuando ejecutamos `plt.plot()`, `pyplot` llama internamente a `ax.plot()`. Esto da un nivel extra de sencillez pero reduce en parte la versatilidad. En cambio en el segundo ejemplo se usa el método explícito, llamando directamente a un objeto `Axes`.

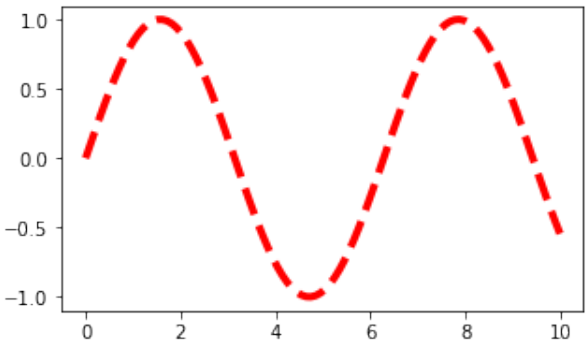
Conclusión: para ejemplos sencillos y rápidos podemos recurrir a la versión "abreviada" o implícita, más fácil de usar. Pero cuando queramos realizar gráficas más elaboradas es recomendable utilizar la versión explícita, que nos da más control sobre los diferentes elementos individuales.

Cuando copiemos código de los ejemplos nos encontraremos en este segundo caso. Con objeto de poder entender y reutilizar dicho código, en el resto de este tutorial usaremos la forma explícita.

8.4. Estilos

Usaremos `plot` para dibujar una línea. Podemos modificar el estilo cambiando el estilo de línea, su grosor o su color, por ejemplo. Esto se puede hacer con argumentos de la función plot:

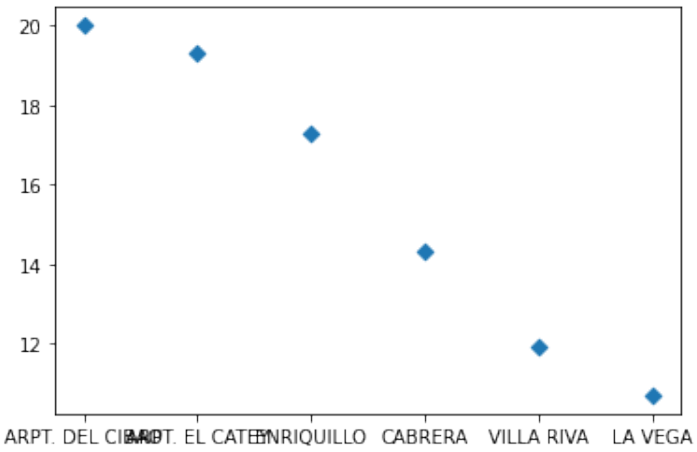
```
x = np.linspace(0, 10, 100)
fig, ax = plt.subplots(figsize=(5, 3))
ax.plot(x, np.sin(x), linestyle='--', linewidth=4, color='red')
# Esto es idéntico, usando RGB:
#ax.plot(x, np.sin(x), linestyle='--', linewidth=4, color=(1.0, 0.0, 0.0))
```



Hay mucha más información sobre colores en https://matplotlib.org/stable/gallery/color/index.html y https://matplotlib.org/stable/tutorials/colors/colors.html

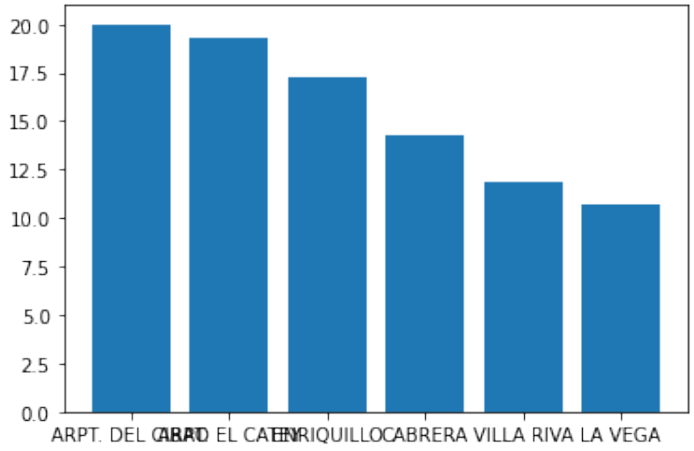
Además de líneas, hay otros muchos gráficos que se pueden utilizar en matplotlib. Por ejemplo, aquí representamos algunos datos de estaciones extraídos de la web de ONAMET (Oficina Nacional Meteorológica de la República Dominicana) por medio de marcadores (markers) en forma de rombos, gracias al método `ax.scatter`:

```
df = pd.read_csv("onamet_20230824.csv", sep=",", skiprows=4, encoding = "utf-8")
df =df[4:10] # Nos quedamos con seis filas para simplificar

fig, ax = plt.subplots()
ax.scatter(df["ESTACIÓN"], df["LLUVIA (mm)"], marker="D")
plt.show()
```



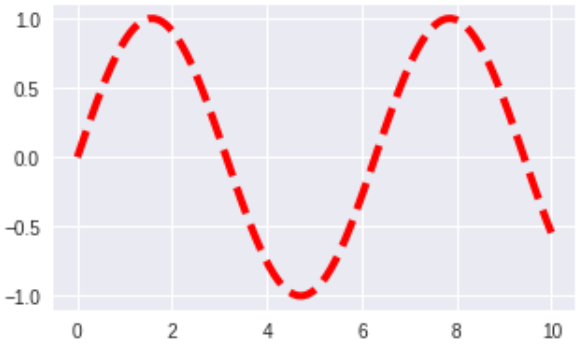
O quizás mejor, usando un diagrama de barras (`ax.bar`):

```
fig, ax = plt.subplots()
ax.bar(df["ESTACIÓN"], df["LLUVIA (mm)"])
plt.show()
```



Se puede modificar también el estilo de dibujo. Por ejemplo para usar el estilo seaborn:

```
plt.style.use(['seaborn'])
x = np.linspace(0, 10, 100)
fig, ax = plt.subplots(figsize=(5, 3))
ax.plot(x, np.sin(x), linestyle='--', linewidth=4, color='red')
```

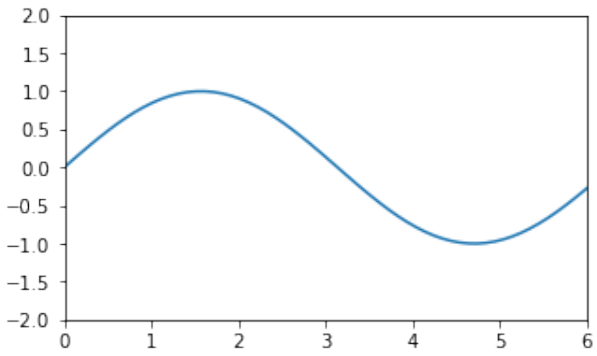


seaborn tiene además su propia librería: https://seaborn.pydata.org/

8.5. Límites. Marcas (ticks) y etiquetas

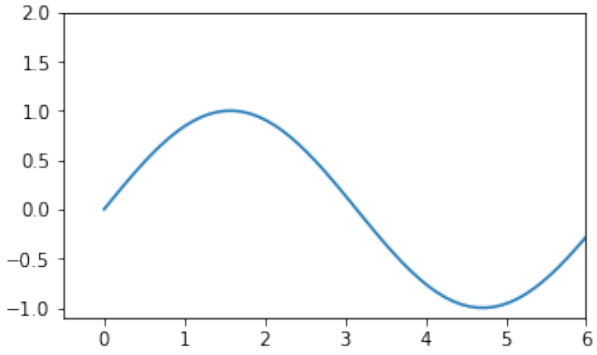
Matplotlib se encarga de por sí solo de establecer los límites de las gráficas. Sin embargo en muchas ocasiones podemos querer controlar este aspecto. Lo haremos mediante los métodos `ax.set_ylim` y `ax.set_ylim`:

```
x = np.linspace(0, 10, 100)
fig, ax = plt.subplots(figsize=(5, 3))
ax.plot(x, np.sin(x))
ax.set_xlim(0, 6)
ax.set_ylim(-2, 2)
```



Podemos establecer únicamente el límite inferior o superior y dejar que matplotlib se encargue del resto:

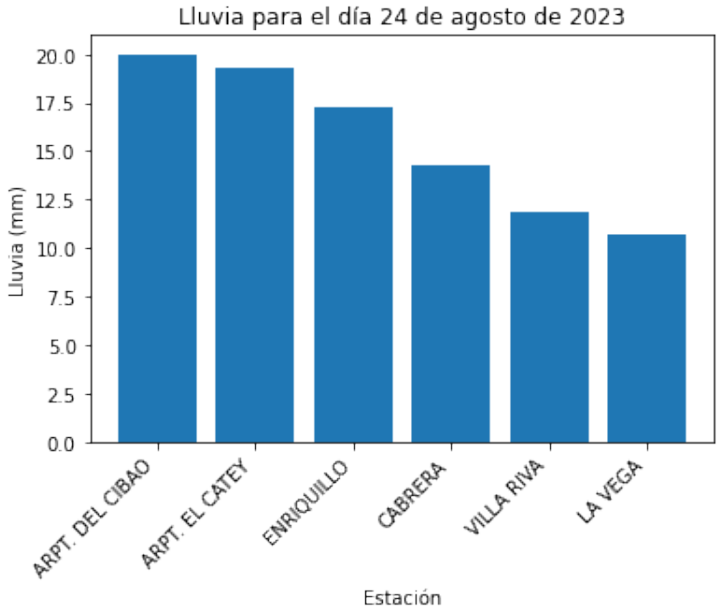
```
fig, ax = plt.subplots(figsize=(5, 3))
ax.plot(x, np.sin(x))
ax.set_xlim(right=6)
ax.set_ylim(top=2)
```



Matplotlib nos da también métodos para dar título de la gráfica (`ax.set_title`), a las etiquetas de los ejes (`ax.set_xlabel` y `ax.set_ylabel`), y a la frecuencia de las marcas y sus etiquetas (`ax.xticks` y `ax.yticks`). Alternativamente se puede usar el objeto axis, con la función `ax.axis.set_ticks`.

Si los incluímos en el diagrama de barras del apartado anterior:

```
fig, ax = plt.subplots()
ax.bar(df["ESTACIÓN"], df["LLUVIA (mm)"])
ax.set_ylabel('Lluvia (mm)')
ax.set_xlabel('Estación')
ax.set_xticks(range(0, len(df["ESTACIÓN"])), df["ESTACIÓN"], rotation=45, ha="right")
# o también:
# ax.xaxis.set_ticks(range(0, len(df["ESTACIÓN"])), df["ESTACIÓN"], rotation=45, ha="right")
ax.set_title("Lluvia para el día 24 de agosto de 2023")
plt.show()
```



En el caso de las etiquetas de las marcas (los "ticks") observar que el primer argumento indica la posición de cada marca, y el segundo el texto de cada etiqueta:

```
list(range(0, len(df["ESTACIÓN"])))  #  Las barras están en las posiciones 0 a 5
```

```
[0, 1, 2, 3, 4, 5]
```

```
df["ESTACIÓN"]
```

```
4    ARPT. DEL CIBAO
5     ARPT. EL CATEY
6          ENRIQUILLO
7             CABRERA
8          VILLA RIVA
9             LA VEGA
Name: ESTACIÓN, dtype: object
```

Además hemos girado las etiquetas para evitar que se pisen unas a otras (argumento opcional rotation), y las hemos desplazado a la derecha (argumento opcional ha) para que estén bien situadas bajo su marca.

Hay una gran variedad de formatos de ticks posibles (por ejemplo se pueden dibujar ticks mayores y menores). En la documentación se puede encontrar una explicación más detallada (por ejemplo, para usar fechas como etiquetas se puede consultar https://matplotlib.org/stable/gallery/text_labels_and_annotations/date.html).

8.5.1. Expresiones matemáticas

Matplotlib puede utilizar el lenguaje de markup TeX para escribir expresiones matemáticas (en el título, una etiqueta, la leyenda, etc). Para ello basta con poner la expresión en código TeX dentro de signos de dólar($).

Por ejemplo, la expresión de TeX $e^{i\pi}+1=0$ queda así:

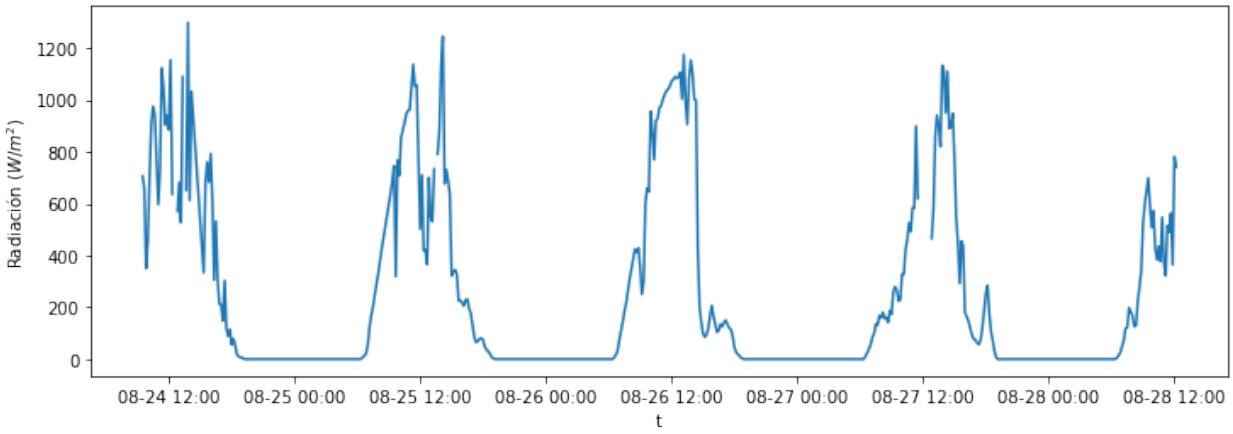
$$e^{i\pi} + 1 = 0$$

Veamos un caso práctico en el que tomamos datos de una semana de la estación de Atlacomulco de la web del Servicio Meteorológico Nacional de México:

```
df = pd.read_csv("Estacion_ATLACOMULCO_1_semana.csv", parse_dates=[0], skiprows=9,
                 sep=",", encoding = "ISO-8859-1", index_col=[0])
df = df.sort_index()
```

Si queremos representar la radiación, que tiene como unidad watios por metro cuadrado, podemos incluir el símbolo del cuadrado fácilmente con TeX:

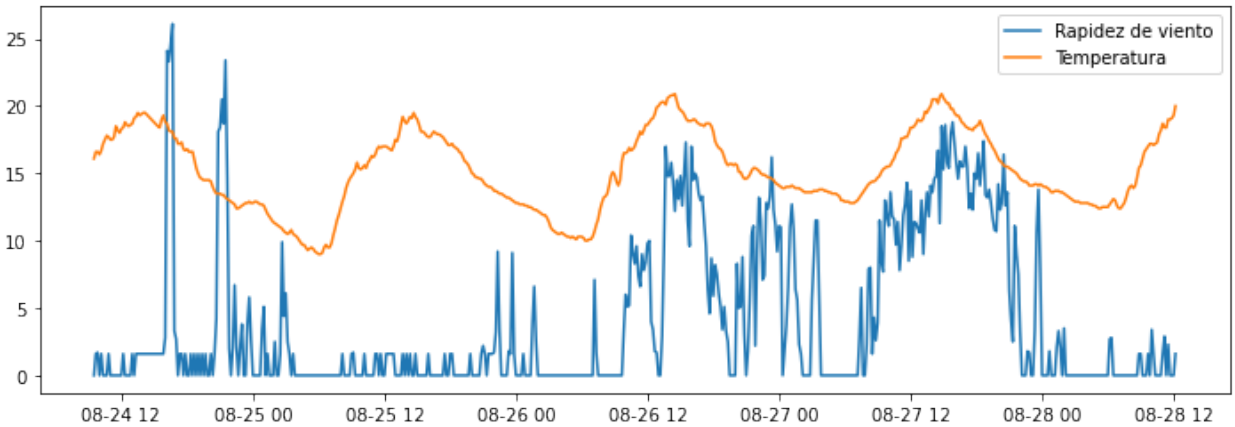
```
import matplotlib.dates as mdates

fig, ax = plt.subplots(figsize=(12,4))
ax.plot(df.index, df['Radiación Solar (W/m2)'])
ax.set_ylabel('Radiación ($W/m^2$)') # Con los $, escribe el cuadrado correctamente
ax.set_xlabel('t')
myFmt = mdates.DateFormatter('%m-%d %H:%M') # Ejemplo de formateo de fechas para xticks
ax.xaxis.set_major_formatter(myFmt)
```



8.6. Leyenda

El método `ax.legend` nos da la opción de incluir una leyenda. Hay distintos argumentos para modificar el contenido de la leyenda y su posición en el gráfico (poniéndola incluso fuera de él). Usando el ejemplo anterior:

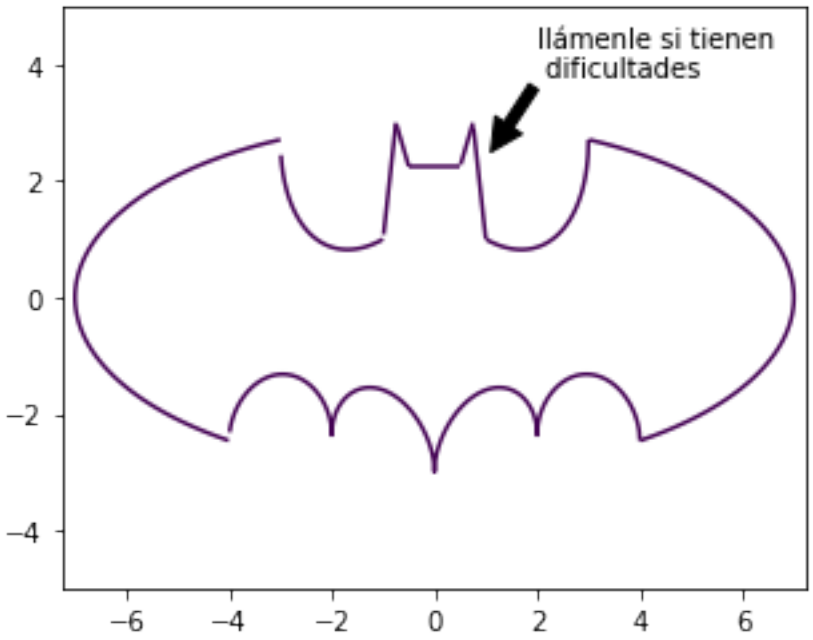
```
fig, ax = plt.subplots(figsize=(12,4))
ax.plot(df.index, df['Rapidez de viento (km/h)'], label='Rapidez de viento')
ax.plot(df.index, df['Temperatura del Aire (°C)'], label='Temperatura')
ax.legend()
plt.show()
```



8.7. Anotaciones

Para añadir una anotación sobre la propia gráfica se puede utilizar `ax.annotate`:

```
import matplotlib.pyplot as plt
from numpy import sqrt, meshgrid, arange

xs = arange(-7.25, 7.25, 0.01)
ys = arange(-5, 5, 0.01)
x, y = meshgrid(xs, ys)

eq1 = ((x/7)**2*sqrt(abs(abs(x)-3)/(abs(x)-3))
      +(y/3)**2*sqrt(abs(y+3/7*sqrt(33))/(y+3/7*sqrt(33)))-1)
eq2 = (abs(x/2)-((3*sqrt(33)-7)/112)*x**2-3+sqrt(1-(abs(abs(x)-2)-1)**2)-y)
eq3 = (9*sqrt(abs((abs(x)-1)*(abs(x)-.75))/((1-abs(x))*(abs(x)-.75)))-8*abs(x)-y)
eq4 = (3*abs(x)+.75*sqrt(abs((abs(x)-.75)*(abs(x)-.5))/((.75-abs(x))*(abs(x)-.5)))-y)
eq5 = (2.25*sqrt(abs((x-.5)*(x+.5))/((.5-x)*(.5+x)))-y)
eq6 = (6*sqrt(10)/7+(1.5-.5*abs(x))*sqrt(abs(abs(x)-1)/(abs(x)-1))
      -(6*sqrt(10)/14)*sqrt(4-(abs(x)-1)**2)-y)

fig, ax = plt.subplots(figsize=(5, 4))

for f in [eq1,eq2,eq3,eq4,eq5,eq6]:
    ax.contour(x, y, f, [0])

ax.annotate('llámenle si tienen\n dificultades', xy=(1.1, 2.5), xytext=(2, 3.8),
            arrowprops=dict(facecolor='black'), fontsize=10)

plt.show()
```

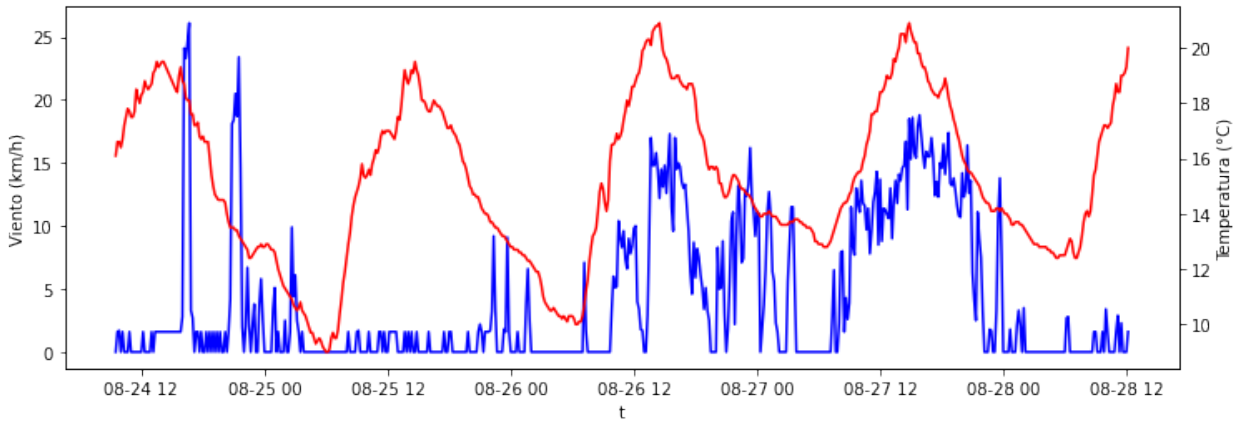


8.8. Compartiendo ejes

Volviendo al ejemplo de la estación de Atlacomulco, no parece adecuado representar la temperatura y la rapidez del viento en el mismo eje, dado que tienen escalas distintas.

Podemos construir una gráfica con dos `Axes` diferentes, uno para cada parámetro, pero que compartan el eje X. Para ello usaremos `ax.twinx`:

```
fig, ax1 = plt.subplots(figsize=(12,4))
ax1.plot(df.index, df['Rapidez de viento (km/h)'], color=(0,0,1))
ax2 = ax1.twinx()
ax2.plot(df.index, df['Temperatura del Aire (°C)'], color=(1,0,0))
ax1.set_ylabel("Viento (km/h)")
ax2.set_ylabel("Temperatura (°C)")
ax1.set_xlabel("t")
plt.show()
```



Ejercicio

Utilizar el fichero de Buenos Aires (buenos_aires_20230813.txt) para representar en una misma gráfica la temperatura y la humedad relativa, compartiendo el eje X (eje de tiempo). Incluir una leyenda con los dos parámetros.

8.9. Subplots

Cada `Figure` puede contener varios `Axes`, se conocen como subaxes o subplots.

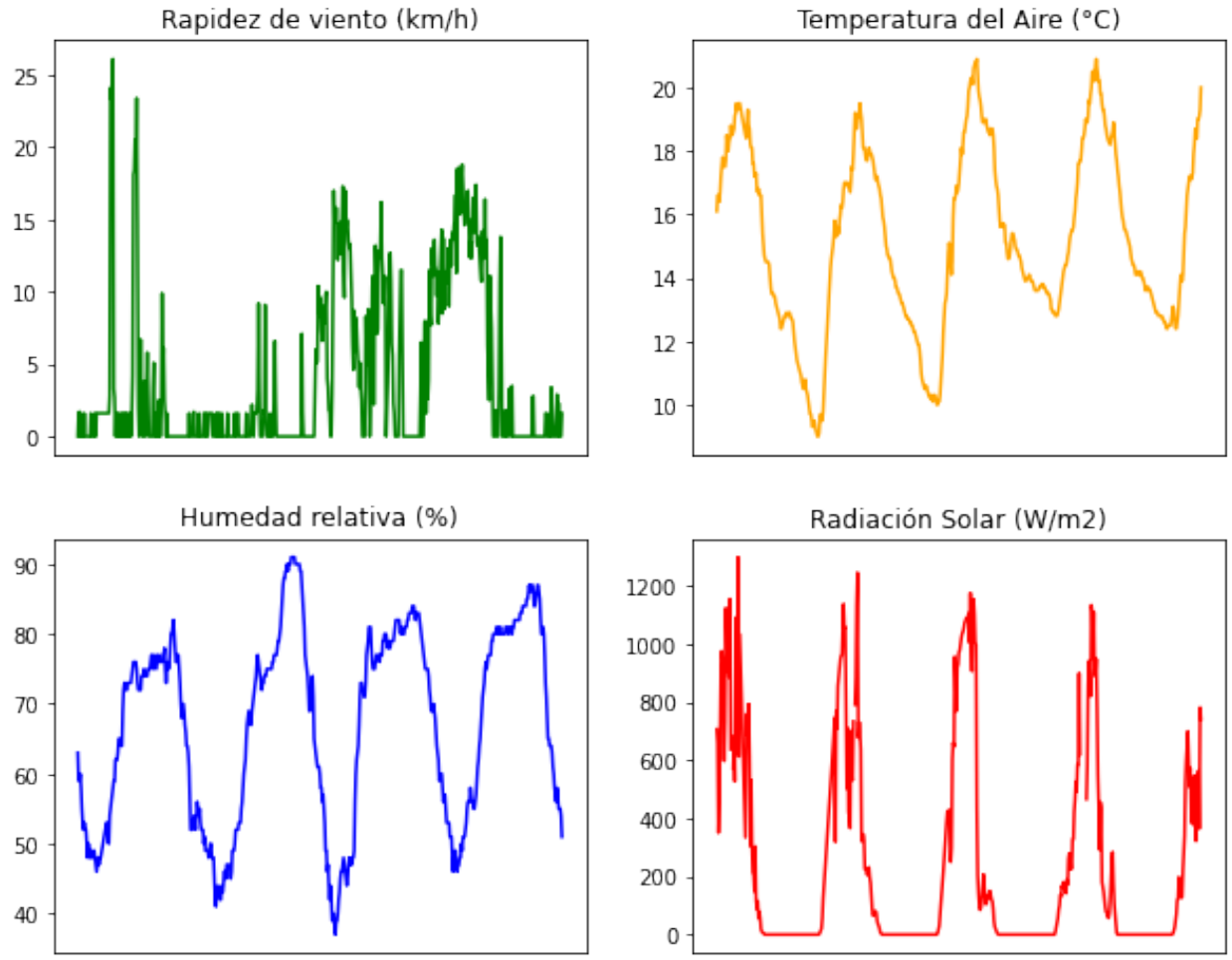
Cuando añadimos una gráfica a la figura podemos especificar cuantas filas y columnas queremos, y que posición ocupa la gráfica actual, con los argumentos de subplot(ncol, nrow, n).

Hay varias maneras de añadir y modificar gráficas en una figura. Veamos las más comunes:

```
fig, axes = plt.subplots(1, 3)    # devuelve una Figure y una 3-tupla de Axes
    ...

ax = plt.subplot(2, 2, 1)         # divide la figura en 4 partes y devuelve el primer axes

ax = fig.add_subplot(2, 2, 1)     # equivalente al anterior

ax = fig.add_axes([.6, .6, .2, .2]) # añade Axe con geometria [left, bottom, width, height]
```

Veamos un ejemplo de dibujo de subplots con el fichero de Atlacomulco:

```
df = pd.read_csv("Estacion_ATLACOMULCO_1_semana.csv", parse_dates=[0], skiprows=9,
                      sep=",", encoding = "ISO-8859-1", index_col=[0])
df = df.sort_index()
fig, axes = plt.subplots(2, 2, figsize=(10,8))    # devuelve una Figure y un array de Axes
parameter = [["Rapidez de viento (km/h)","Temperatura del Aire (°C)"],
             ["Humedad relativa (%)", "Radiación Solar (W/m2)"]]
parcolors = [["green", "orange"], ["blue", "red"]]

for i in [0,1]:
    for j in [0,1]:
        axes[i,j].plot(df[parameter[i][j]], color=parcolors[i][j])
        axes[i,j].set_title(parameter[i][j])
        axes[i,j].set_xticks([])  # quito las etiquetas del eje X para que no molesten
```



Otras instrucciones útiles: `plt.subplots_adjust` nos permite jugar con el espaciado entre gráficas. Si tenemos problemas con el ajuste de espacios podemos usar `plt.tight_layout`, que realiza un auto ajuste.

8.10. Guardar en fichero

Podemos guardar la figura por medio de `fig.savefig`, indicando el nombre y la extensión:

```
fig.savefig('atlacomulco.png')
fig.savefig('atlacomulco.pdf')
```

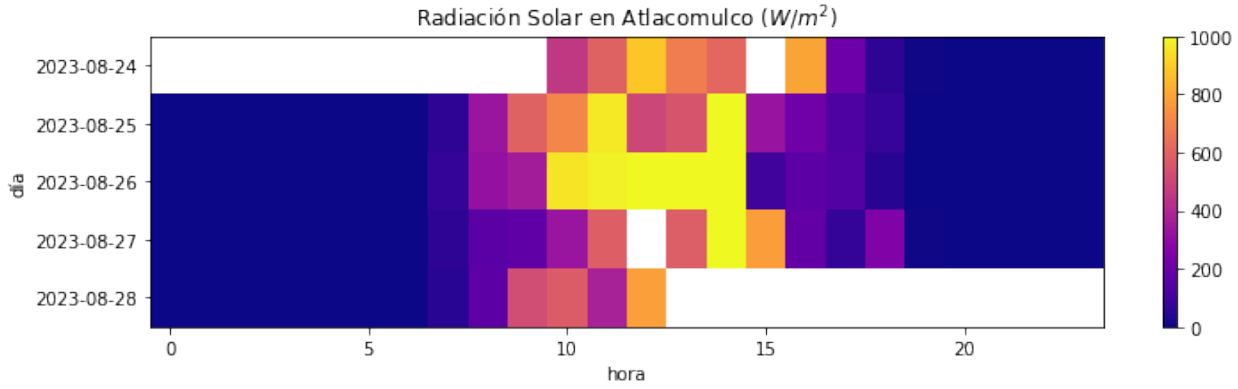
```
fig.savefig('atlacomulco.ps')
```

8.11. Mapas de color

Es posible que a veces queramos una gráfica con una tercera dimensión representada por colores diferentes en un mapa de colores. Matplotlib dispone de varias funciones (`ax.pcolormesh`, `ax.contourf`, `ax.imshow`) para conseguir esto.

Por ejemplo, supongamos que queremos representar la radiación solar dependiendo del día del mes (en el eje Y) y de la hora del día (en el eje X), de forma que cada color represente una diferente insolación, desde una radiación nula en color azul oscuro a una radiación máxima en color amarillo. Este código nos da la respuesta:

```
parameter = 'Radiación Solar (W/m2)'
df = pd.read_csv("Estacion_ATLACOMULCO_1_semana.csv", parse_dates=[0], skiprows=9,
                 sep=",", encoding = "ISO-8859-1", index_col=[0])
df = df.sort_index()

df = df[[parameter]]      # Nos quedamos con el parametro elegido
df['dia'] = df.index.date
df['hora'] = df.index.hour
df_hour = df[df.index.minute==0] # Eliminamos observaciones que no son a la hora en punto
df_rad = pd.pivot_table(df_hour, values='Radiación Solar (W/m2)',
                        index='dia', columns='hora')
import matplotlib as mpl

fig, ax = plt.subplots(figsize=(12,3))
pc = ax.imshow(df_rad, cmap='plasma', aspect='auto',
               norm=mpl.colors.Normalize(vmin=0, vmax=1000))
ax.set_yticks(range(0, len(df_rad.index)),df_rad.index)
ax.set_xlabel("hora")
ax.set_ylabel("día")
ax.set_title("Radiación Solar en Atlacomulco ($W/m^2$)")
fig.colorbar(pc)
```

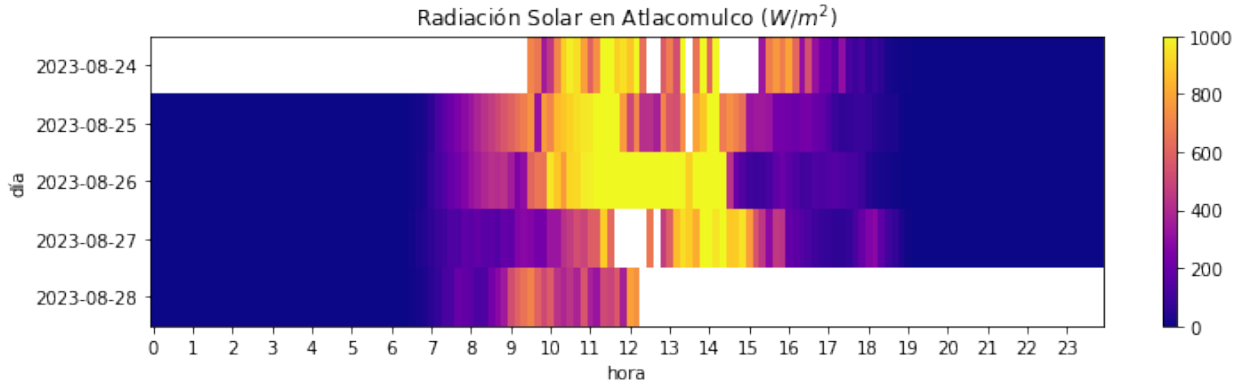


Warning: Notar que en este caso hemos cometido un error, al tomar la observación de la hora en punto como representativa de toda la hora.

Dado que tenemos observaciones cada 10 minutos, podemos afinar incluso más:

```
df['minuto'] = df.index.minute
df_rad_min = pd.pivot_table(df, values='Radiación Solar (W/m2)',
                index='dia', columns=['hora', 'minuto'])
```

```
import matplotlib.dates as mdates
# Indice superior (horas) en el multiindice:
xtickvalues = df_rad_min.columns.get_level_values(0)

fig, ax = plt.subplots(figsize=(12,3))
pc = ax.imshow(df_rad_min, cmap='plasma', aspect='auto',
               norm=mpl.colors.Normalize(vmin=0, vmax=1000))
xFmt = mdates.DateFormatter('%H')
yFmt = mdates.DateFormatter('%Y-%m-%d')
ax.xaxis.set_major_formatter(xFmt)
ax.yaxis.set_major_formatter(yFmt)
ax.set_xticks(range(0, len(xtickvalues),6),xtickvalues[range(0, len(xtickvalues),6)])
ax.set_yticks(range(0, len(df_rad_min.index)),df_rad_min.index)
ax.set_xlabel("hora")
ax.set_ylabel("día")
ax.set_title("Radiación Solar en Atlacomulco ($W/m^2$)")
fig.colorbar(pc)
```



8.12. Consejos útiles

8.12.1. ¿Qué formato de salida debo escoger?

Como diría un buen gallego, depende. Depende del gráfico que estemos generando, pero en general para empezar escoged PNG. Una breve explicación sobre formatos que puede aclarar algunas ideas:

- El formato GIF está limitado a 256 colores y es un formato de compresión sin pérdida de información, una elección habitual para su uso en la Web. GIF es una buena opción para guardar dibujos de líneas, texto o iconos con un tamaño de fichero pequeño.
- El formato PNG es también un formato de compresión sin pérdida de información, una elección habitual para su uso en la Web. PNG es una buena opción para guardar dibujos de líneas, texto o iconos con un tamaño de fichero pequeño.
- El formato JPG es un formato de compresión con pérdida de información. Esto lo hace útil para almacenar fotografías con un tamaño menor que un BMP. JPG es una opción habitual en la Web porque está comprimido. Para guardar dibujos de líneas, texto, e iconos con un tamaño de fichero menor GIF o PNG son mejores opciones porque no tienen pérdidas.

JPEGs son para fotografías e imágenes realistas. PNGs son para dibujos con líneas, imágenes con una gran cantidad de texto e imágenes con pocos colores. GIFs no sirven en general.

8.12.2. Producción masiva: acelerando Matplotlib

Este consejo va más enfocado al dibujo de mapas, donde la generación de gráficos puede llevar bastante tiempo. Si tenemos que crear una gran cantidad de gráficos o mapas veremos que los procesos llevan bastante tiempo. Si dichos gráficos tienen características en común, por ejemplo el mapa de fondo (proyección, líneas de costa, colores, extensión) es adecuado reutilizar dicho trabajo para los gráficos siguientes.

Una opción es limpiar parte de la figura que hemos realizado en lugar de construirla de nuevo. De este modo únicamente cambiamos los datos sobre el gráfico de fondo ya creado.

Cabe también pensar en la serialización de objetos a través del módulo pickle, para ser reutilizados posteriormente.

En general, no solo para la generación de gráficos, lo primero es hacer que nuestro programa funcione, lo segundo es optimizarlo.

8.12.3. Un último consejo

Matplotlib está muy bien, pero no nos obsesionemos ;-)

Capítulo 9

Librería Xarray

Pandas es una librería muy útil para tratar datos tabulados, en dos dimensiones, pero no es tan conveniente para tres dimensiones o más. Sin embargo en meteorología es muy habitual tratar con datos multidimensionales, por ejemplo en campos meteorológicos que dependen de las tres coordenadas espaciales y del tiempo.

Xarray (https://docs.xarray.dev) es una librería construida sobre Numpy que permite tratar datos multidimensionales con etiquetas, y comparte muchas de las características de Pandas. Hace que los datos sean más legibles tratándolos de forma sencilla y eficiente.

Además está estrechamente relacionado con Cartopy, otra librería especializada en el dibujo de mapas.

9.1. Lectura de datos. Estructuras Dataset y DataArray

Cargamos el módulo xarray:

```
import xarray as xr
import matplotlib.pyplot as plt
# Deshabilitamos los avisos
import warnings
warnings.filterwarnings('ignore')
```

`open_dataset` es una función de xarray que nos permite abrir ficheros netCDF (o URLs de un servidor remoto OpenDAP). El formato netCDF se utiliza comúnmente en meteorología para guardar información de los modelos meteorológicos (junto al formato grib), y xarray está especialmente adaptado para leerlo.

Para ilustrar xarray vamos a utilizar aquí varios ficheros del reanálisis ERA5 (https://www.ecmwf.int/en/forecasts/dataset/ecmwf-reanalysis-v5), descargados del Climate Data Store del C3S (Climate Change Service) del programa Copernicus. Se puede encontrar más información en https://climate.copernicus.eu/climate-reanalysis.

```
ds = xr.open_dataset('curso_PIBM/era5_america_2022.nc')
```

`open_dataset` abre un fichero netcDF y nos devuelve una de las dos estructuras básicas de datos de xarray, el Dataset. El Dataset es una especie de diccionario de DataArrays (otra estructura que veremos después):

Si se quiere abrir varios ficheros netCDF simultáneamente y cargarlos en un único dataset (por ejemplo, si tenemos 24 ficheros netCDF, uno por hora de predicción) podemos usar la función `open_mfdataset`. Para liberar los recursos usaremos `close`.

Nota: Por otra parte, si se quieren abrir ficheros con formato grib con xarray es necesario instalar previamente la librería cfgrib y dar al argumento *engine* de `open_dataset` el valor 'cfgrib' para que abra el fichero correctamente.

Veamos que contiene esta estructura de datos:

```
ds
```

```
<xarray.Dataset>
Dimensions:    (longitude: 361, latitude: 373, time: 568)
Coordinates:
  * longitude  (longitude) float32 -120.0 -119.8 -119.5 ... -30.5 -30.25 -30.0
  * latitude   (latitude) float32 33.0 32.75 32.5 32.25 ... -59.5 -59.75 -60.0
  * time       (time) datetime64[ns] 2022-01-05 ... 2022-12-30T21:00:00
Data variables:
    t2m        (time, latitude, longitude) float32 ...
    sst        (time, latitude, longitude) float32 ...
Attributes:
    Conventions:  CF-1.6
    history:      2023-09-02 16:18:24 GMT by grib_to_netcdf-2.25.1: /opt/ecmw...
```

Dentro de un dataset tenemos distintos atributos que nos informan sobre su contenido: variables, coordenadas, y dimensiones (los ejes de nuestros datos):

```
ds.data_vars
```

```
Data variables:
    t2m      (time, latitude, longitude) float32 ...
    sst      (time, latitude, longitude) float32 ...
```

```
ds.coords
```

```
Coordinates:
  * longitude  (longitude) float32 -120.0 -119.8 -119.5 ... -30.5 -30.25 -30.0
  * latitude   (latitude) float32 33.0 32.75 32.5 32.25 ... -59.5 -59.75 -60.0
  * time       (time) datetime64[ns] 2022-01-05 ... 2022-12-30T21:00:00
```

```
ds.dims
```

```
Frozen({'longitude': 361, 'latitude': 373, 'time': 568})
```

Vemos que este fichero contiene información sobre dos variables, la temperatura a 2 metros (t2m) y la temperatura superficial del agua del mar (sst), para la región comprendida entre las longitudes -120 y 30 W, y entre las latitudes 33 N y 60 S (Centroamérica y Sudamérica, aproximadamente), para el año 2022.

Las variables del dataset son datos de tipo DataArray y se puede acceder ellos de la misma forma que se accede a las columnas en Pandas:

```
ds['t2m']
```

o, de forma equivalente:

```
ds.t2m
```

```
<xarray.DataArray 't2m' (time: 568, latitude: 373, longitude: 361)>
[76482904 values with dtype=float32]
Coordinates:
  * longitude  (longitude) float32 -120.0 -119.8 -119.5 ... -30.5 -30.25 -30.0
  * latitude   (latitude) float32 33.0 32.75 32.5 32.25 ... -59.5 -59.75 -60.0
  * time       (time) datetime64[ns] 2022-01-05 ... 2022-12-30T21:00:00
Attributes:
    units:      K
    long_name:  2 metre temperature
```

En este caso tenemos dos dimensiones espaciales (latitud y longitud), y la temporal

```
ds.t2m.dims
```

```
('time', 'latitude', 'longitude')
```

Este es el tamaño del datarray en cada dimensión:

```
ds.t2m.shape
```

```
(568, 373, 361)
```

Podemos acceder también a cada variable coordenada:

```
ds.t2m.latitude
```

```
<xarray.DataArray 'latitude' (latitude: 373)>
array([ 33.  ,  32.75,  32.5 , ..., -59.5 , -59.75, -60.  ], dtype=float32)
Coordinates:
  * latitude  (latitude) float32 33.0 32.75 32.5 32.25 ... -59.5 -59.75 -60.0
Attributes:
    units:      degrees_north
    long_name:  latitude
```

Dentro de un datarray tenemos metadatos y datos. Además de los metadatos anteriores podemos guardar otros metadatos adicionales en attrs:

```
ds.t2m.attrs
```

```
{'units': 'K', 'long_name': '2 metre temperature'}
```

Los datos están contenidos en data, que es un array de Numpy:

```
ds.t2m.data
```

```
array([[[286.97794, 287.05383, 287.10287, ..., 292.75677, 292.79648,
         292.82333],
        [287.1227 , 287.12857, 287.1227 , ..., 292.74976, 292.7848 ,
         292.814  ],
        [287.1846 , 287.1846 , 287.21265, ..., 292.71942, 292.75327,
         292.78247],
        ...,
        [275.75888, 275.73203, 275.65494, ..., 272.94827, 272.91092,
         272.84552],
        [275.74136, 275.77054, 275.79974, ..., 272.77896, 272.74744,
         272.72992],
        [275.62576, 275.6456 , 275.66547, ..., 272.68906, 272.67505,
         272.6692 ]],

       [[287.1986 , 287.2278 , 287.36444, ..., 292.67505, 292.69022,
         292.70773],
        [287.2465 , 287.3539 , 287.38196, ..., 292.6692 , 292.67035,
         292.67853],
        [287.26517, 287.35626, 287.36792, ..., 292.68204, 292.68204,
         292.69604],
        ...,
        [275.61176, 275.6596 , 275.71567, ..., 273.01248, 273.01248,
         272.97513],
        [275.51834, 275.55338, 275.58838, ..., 272.8759 , 272.88406,
         272.90155],
        [275.4658 , 275.48914, 275.51248, ..., 272.82684, 272.83502,
         272.8525 ]],

       [[287.25116, 287.28503, 287.4777 , ..., 292.61316, 292.6073 ,
         292.60147],
        [287.3002 , 287.4847 , 287.56995, ..., 292.6108 , 292.619  ,
         292.6225 ],
        [287.36676, 287.55008, 287.5851 , ..., 292.66803, 292.67972,
         292.69254],
        ...,
        [275.328  , 275.4004 , 275.4915 , ..., 273.0195 , 273.0055 ,
         272.9296 ],
        [275.2007 , 275.24744, 275.29413, ..., 272.86536, 272.8607 ,
```

```
       272.8537 ],
      [275.15402, 275.19138, 275.22992, ..., 272.814  , 272.8245 ,
       272.83618]],

     ...,

     [[288.24603, 287.69022, 287.5863 , ..., 293.02185, 292.75677,
       292.68555],
      [288.12692, 287.71356, 287.6762 , ..., 292.9284 , 292.94827,
       293.1094 ],
      [288.225  , 287.77194, 287.69604, ..., 293.24487, 293.40134,
       293.5473 ],
      ...,
      [274.76636, 274.7605 , 274.7465 , ..., 273.41418, 273.413  ,
       273.39316],
      [274.63907, 274.6449 , 274.65076, ..., 273.40134, 273.399  ,
       273.36395],
      [274.62738, 274.63907, 274.6484 , ..., 273.20517, 273.18063,
       273.1386 ]],

     [[288.4644 , 287.97046, 287.81284, ..., 292.0842 , 292.5466 ,
       292.8023 ],
      [288.37097, 287.87704, 287.8023 , ..., 293.07437, 293.3161 ,
       293.36163],
      [288.4387 , 287.99966, 287.84317, ..., 293.45505, 293.24603,
       293.13626],
      ...,
      [274.6986 , 274.79437, 274.87027, ..., 273.4527 , 273.44922,
       273.42703],
      [274.61456, 274.73364, 274.84924, ..., 273.46674, 273.45856,
       273.41766],
      [274.5445 , 274.64023, 274.73364, ..., 273.31958, 273.29507,
       273.26355]],

     [[288.43637, 288.1246 , 287.9833 , ..., 290.2766 , 290.3817 ,
       290.56387],
      [288.45972, 288.03   , 287.9693 , ..., 290.66547, 290.8231 ,
       291.0146 ],
      [288.60217, 288.2332 , 288.06738, ..., 291.15704, 291.4443 ,
       291.7514 ],
      ...,
      [276.2236 , 276.36957, 276.52487, ..., 273.32077, 273.31375,
       273.29974],
      [275.86047, 275.99707, 276.12436, ..., 273.3383 , 273.3266 ,
       273.2799 ],
      [275.56738, 275.61057, 275.65378, ..., 273.20398, 273.18063,
       273.15378]]], dtype=float32)
```

9.2. Selección e indexado

Vamos a guardar el datarray anterior en grados Celsius:

```
t2m = ds['t2m'] - 273.15
```

Se puede seleccionar un subconjunto de este darray de varias formas. La primera es utilizando la posición en el array, como haríamos con Numpy:

```
t2m[1, 10:13, 20:25]
```

Xarray dispone de dos operadores muy fáciles de usar que permiten hacer selecciones:

- *isel* : selecciona por posición
- *sel* : selecciona por valores o etiquetas

```
# Selecciona los datos que corresponden al segundo valor del tiempo
# (para el primero sería time=0)
t2m.isel(time=1)
```

```
# Lo mismo que antes, pero seleccionando por valor
tt='2022-01-05T03:00:00'
t2m.sel(time=tt)
```

Se pueden seleccionar varias dimensiones. Y también rangos, por medio de la función `slice`:

```
# Selección de una región geográfica
# (en ERA5 la latitud se almacena invertida)
t2m.sel(time=tt, latitude=slice(30.0, 20.0), longitude=slice(-90.0,-80.0))
```

```
# También pueden encadenarse. El resultado es el mismo:
t2m.sel(time=tt).sel(latitude=slice(30.0,20.0).sel(longitude=slice(-90.0,-80.0))
```

Si el valor específico de la dimensión no está en los datos se puede usar *method* para que busque el valor más próximo que sí esté en los datos:

```
t2m.sel(time=tt).sel(latitude=20.34, method='nearest')
```

method no solo tiene la opción *nearest*, también tiene otras (*pad*, *ffill*, *backfill*, *bfill*).

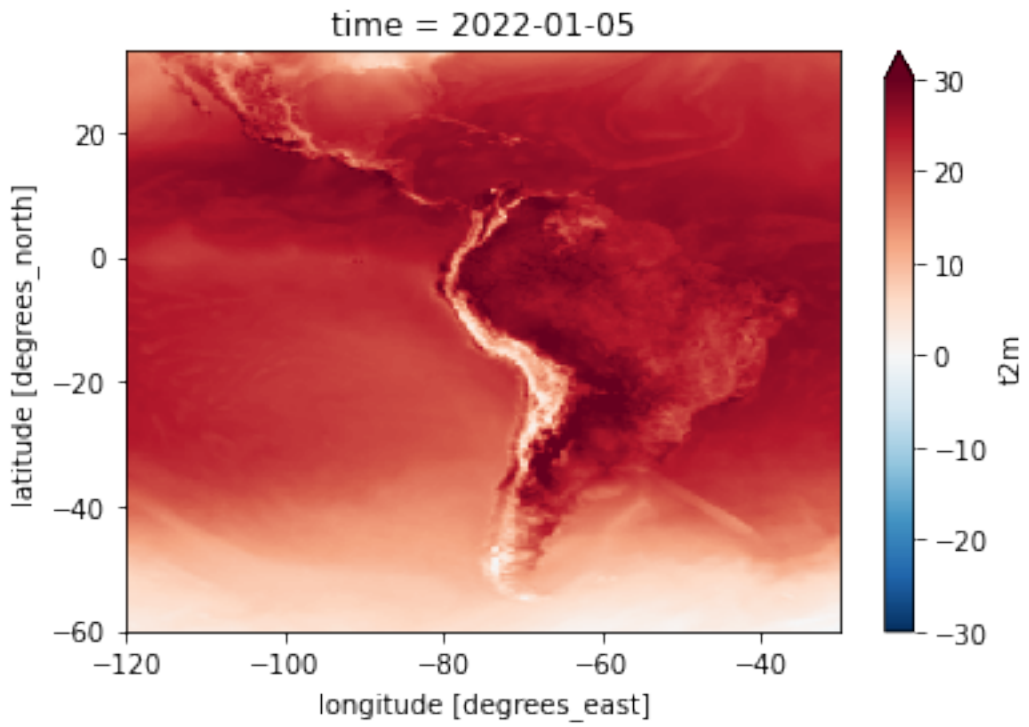
9.3. Gráficos

El método `plot` nos permite dibujar un datarray. Aquí vamos a dar unas cuantas "pinceladas" y mostrar algunas de sus capacidades, pero xarray puede hacer muchas cosas más. Si se quiere profundizar más se puede consultar esta página: https://docs.xarray.dev/en/stable/user-guide/plotting.html.

Xarray tiene una propiedad muy interesante a la hora de pintar un gráfico: intenta deducir a partir de las dimensiones de los datos qué tipo de dibujo queremos.

Por ejemplo si eliminamos la dimensión temporal, `plot` representará un mapa:

```
t2m.isel(time=0).plot(size=4, vmax=30)
```

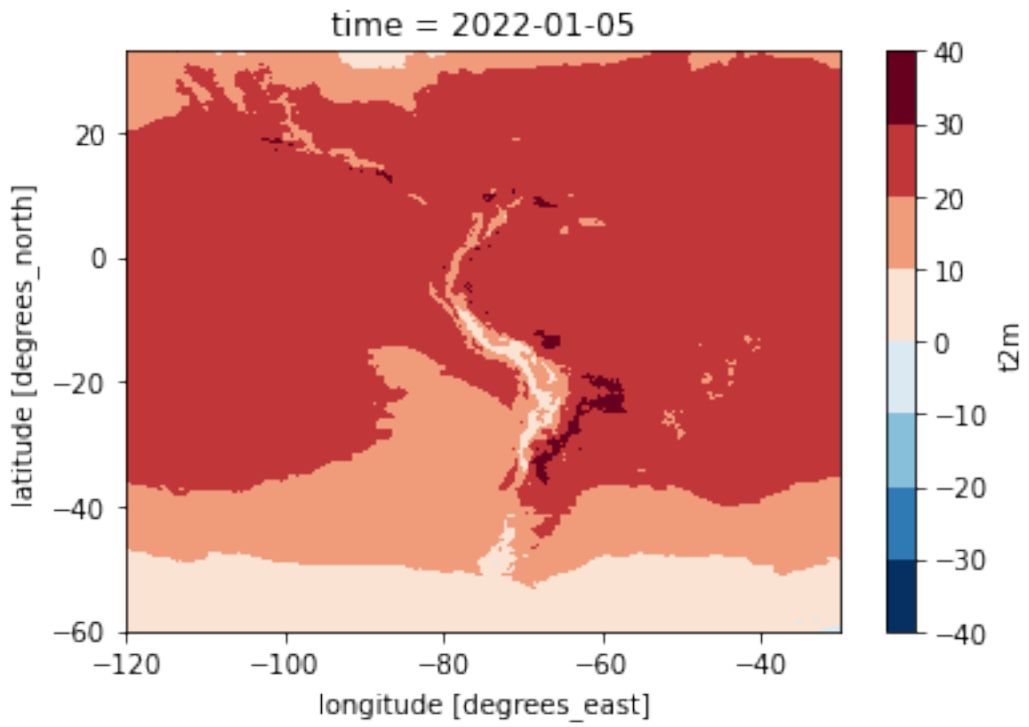


xarray hace buena parte del trabajo por nosotros (añadiendo por defecto etiquetas y la barra de colores). El ajuste de la escala de color es automático, pero podemos usar los argumentos *vmin/vmax* para ajustarlos.

Observar que al ajustar el nivel de temperatura máxima vmax=30, la barra de color indica que hay puntos por encima de ella acabando en un triángulo, en lugar de con un rectángulo

xarray ofrece una gran versatilidad a la hora de pintar gráficos. Por ejemplo, con el argumento *levels* podemos elegir pintar intervalos de forma discreta:

```
t2m.isel(time=0).plot(size=4, levels=8)
```



No solo podemos eliminar la dimensión tiempo, también las dimensiones espaciales.

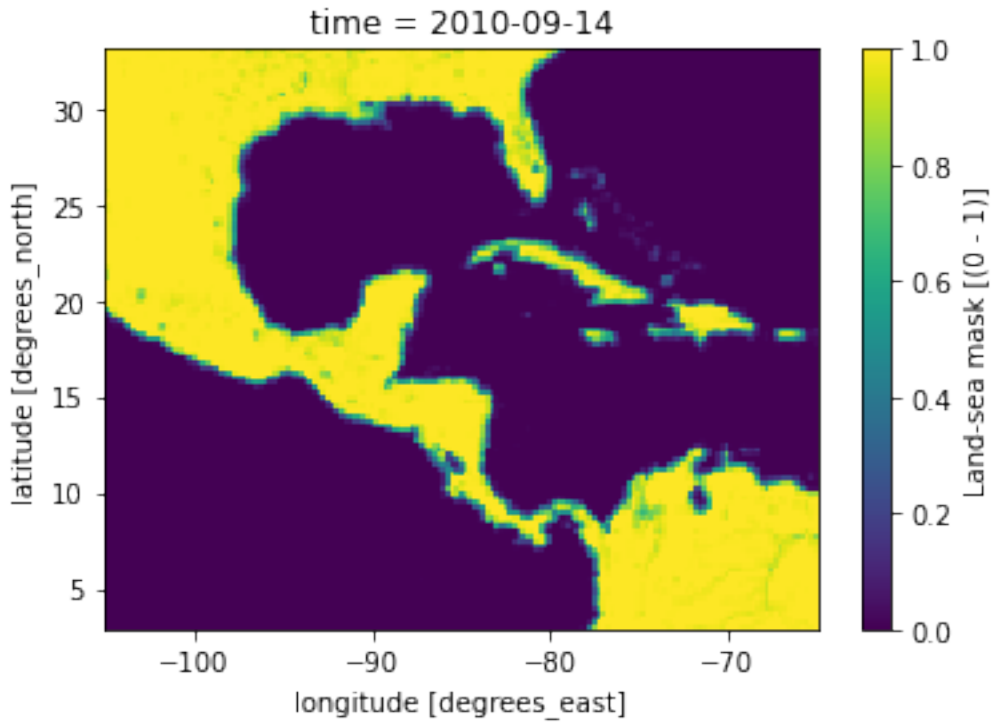
Vamos a ilustrar esta funcionalidad con dos nuevos ficheros de ERA5, que contienen información sobre el huracán Karl (https://es.wikipedia.org/wiki/Hurac%C3%A1n_Karl), que afectó a México entre el 14 y el 18 de septiembre de 2010. El primer fichero contiene información de superficie, y el segundo tiene información en la vertical, en las tres coordenadas espaciales (latitud, longitud y nivel), y en el tiempo:

```
ds_karl_sfc = xr.open_dataset('curso_PIBM/era5_karl2010_sfc.nc')
ds_karl_lev = xr.open_dataset('curso_PIBM/era5_karl2010_levels.nc')
```

```
ds_karl_lev
```

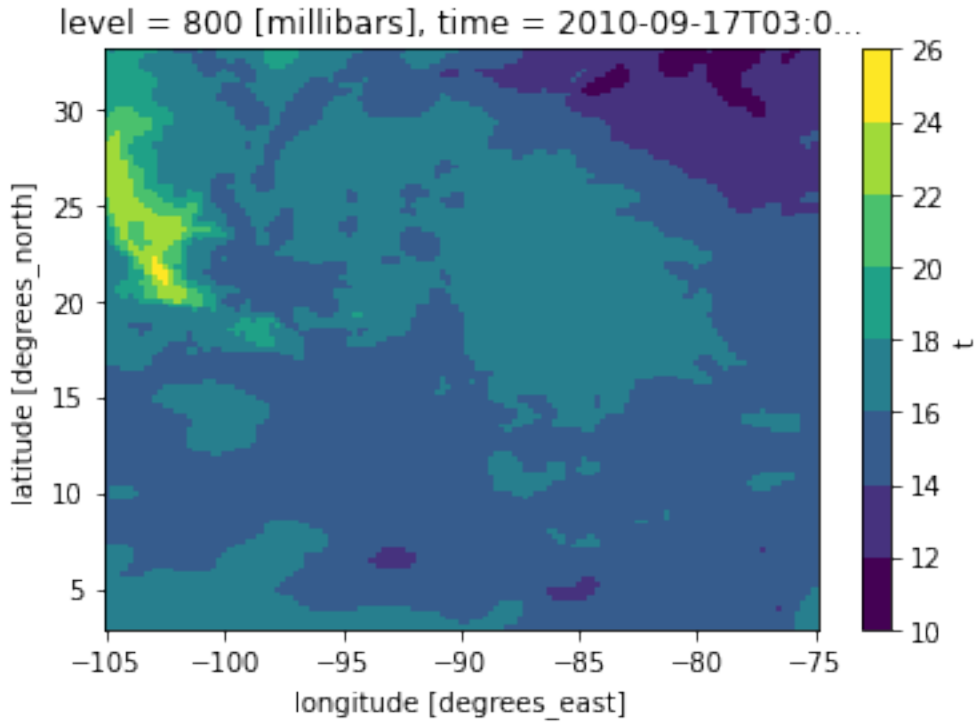
Esta es la zona representada:

```
# Pintamos el parámetro lsm, la máscara de tierra
ds_karl_sfc.lsm.isel(time=0).plot(size=4)
```



Si seleccionamos la dimensión tiempo y la coordenada vertical en el fichero con niveles volvemos a obtener un mapa:

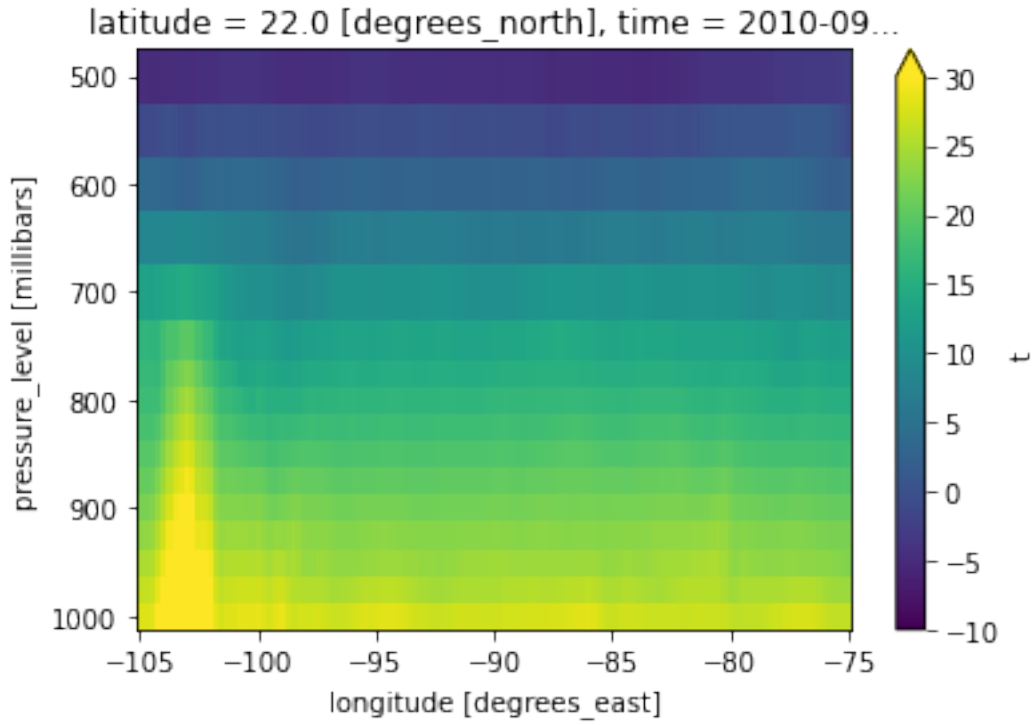
```
tem_karl = ds_karl_lev.t - 273.15
tem_karl.sel(time='2010-09-17T03:00:00').sel(level=800).plot(levels=9)
```



¿Qué ocurre si eliminamos otra coordenada, por ejemplo la latitud, en lugar del nivel? En ese caso obtenemos un corte

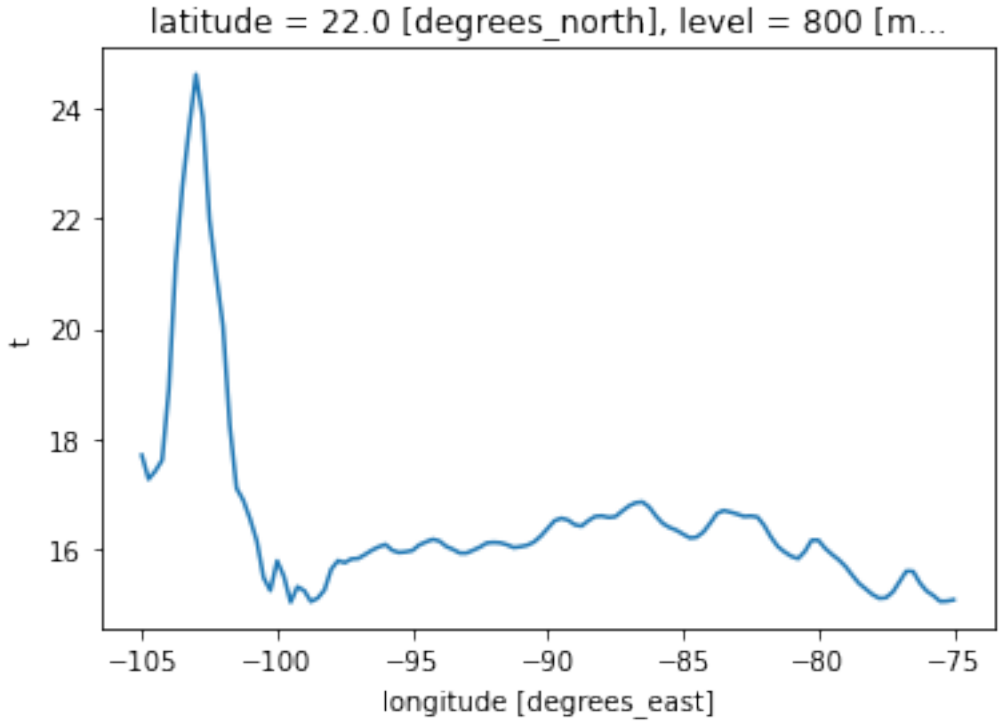
vertical a lo largo del paralelo elegido:

```
tem_karl.sel(time='2010-09-17T03:00:00',
             latitude=22.0).plot(size=4,yincrease=False,vmin=-10,vmax=30)
```



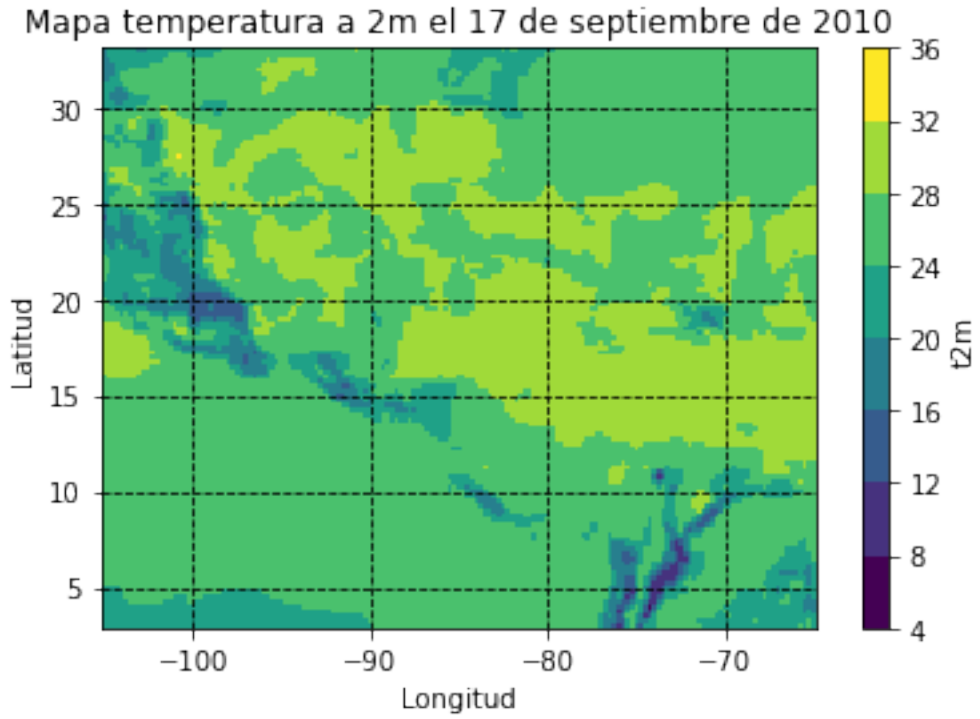
Si eliminamos otra dimensión más, por ejemplo el nivel:

```
tem_karl.sel(time='2010-09-17T03:00:00', latitude=22.0, level=800).plot(size=4)
```



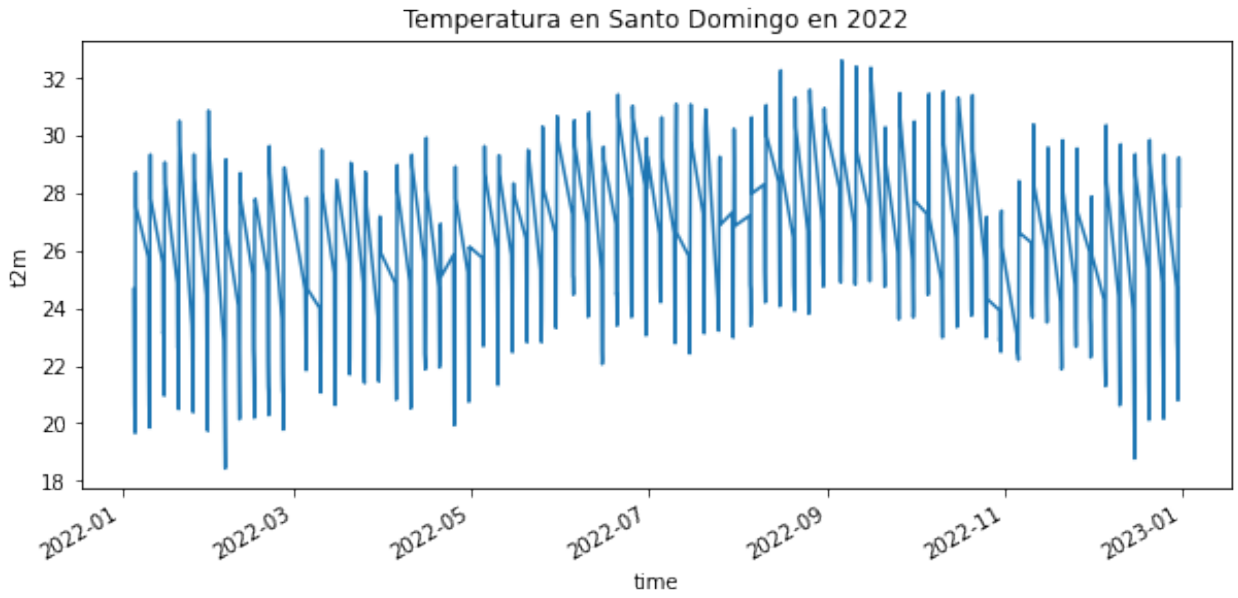
Podemos utilizar los objetos Figure y Axes como hacíamos en matplotlib si queremos tener un mayor control sobre el gráfico:

```
fig, ax = plt.subplots()

# Pasamos los ejes como argumento:
t2m_karl = ds_karl_sfc.t2m - 273.15
t2m_karl.sel(time='2010-09-17T03:00:00').plot(ax=ax, levels=9)

# Personalizamos el gráfico:
ax.grid(True, color='black', linestyle='--')
ax.set_xlabel('Longitud')
ax.set_ylabel('Latitud')
ax.set_title('Mapa temperatura a 2m el 17 de septiembre de 2010')
```



También podemos variar la dimensión tiempo y dejar el resto constantes. Vamos a hacer esto con nuestro fichero original del continente, tomando las coordenadas de Santo Domingo (aproximadamente 18.5N 70W):

```
fig, ax = plt.subplots(figsize=(10,4))
t2m.sel(latitude=18.5, longitude=-69.9, method='nearest').plot(ax=ax)
ax.set_title("Temperatura en Santo Domingo en 2022")
```

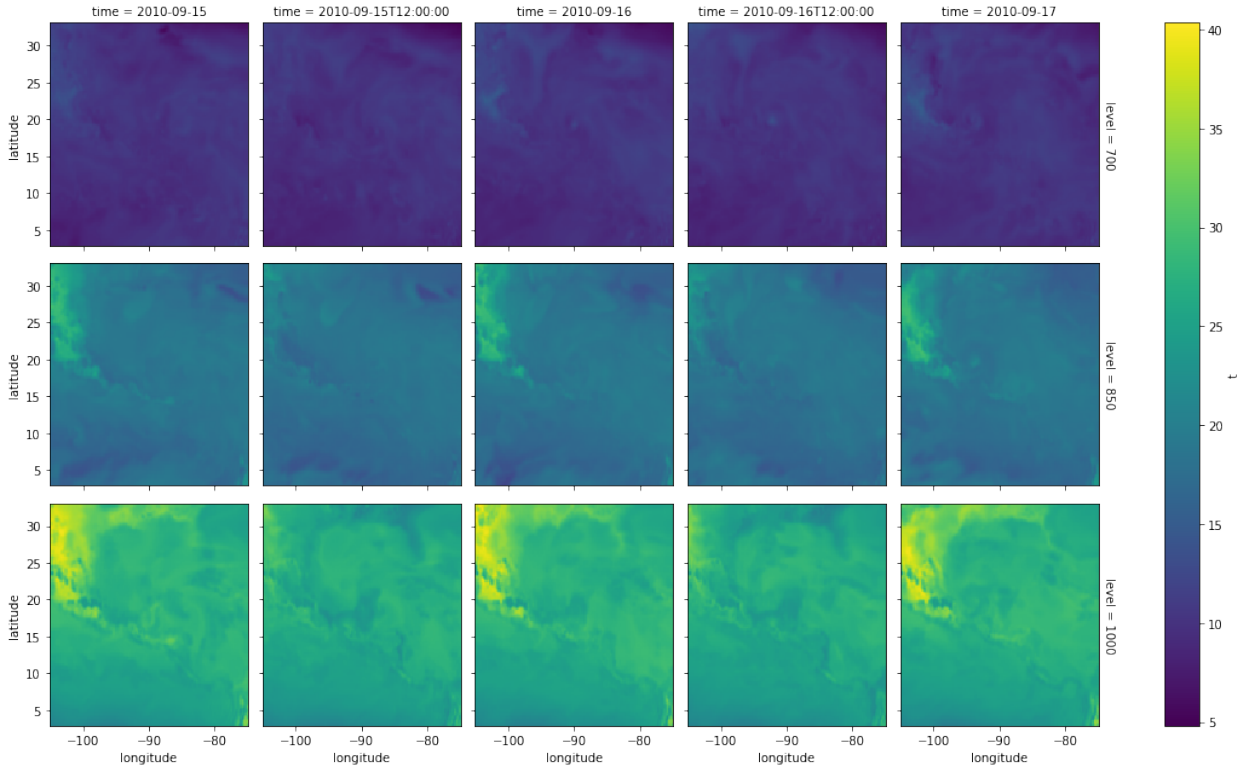


9.3.1. Facetas

Las facetas nos permiten dividir un array según una dimensión y pintar cada grupo resultante, reduciendo en uno la dimensión a dibujar. Por ejemplo, si tenemos como dimensiones la latitud, la longitud y el tiempo, podemos hacer "faceting" en el tiempo, pintando un mapa por cada instante de tiempo. No es recomendable utilizar esta opción si van a crearse demasiadas gráficas.

En el siguiente ejemplo tenemos como dimensiones la latitud, longitud, nivel de presión y tiempo. Hacemos "faceting" en nivel de presión y tiempo (argumentos *col* y *row*). Como tenemos 5 niveles y 3 tiempos diferentes, nos salen 5 * 3 = 15 gráficas:

```
ds_karl_sfc
tem_karl = ds_karl_lev.t - 273.15
tem_karl.isel(time=[8,12,16,20,24]).sel(level=[700,850,1000]).plot(col='time', row='level')
```

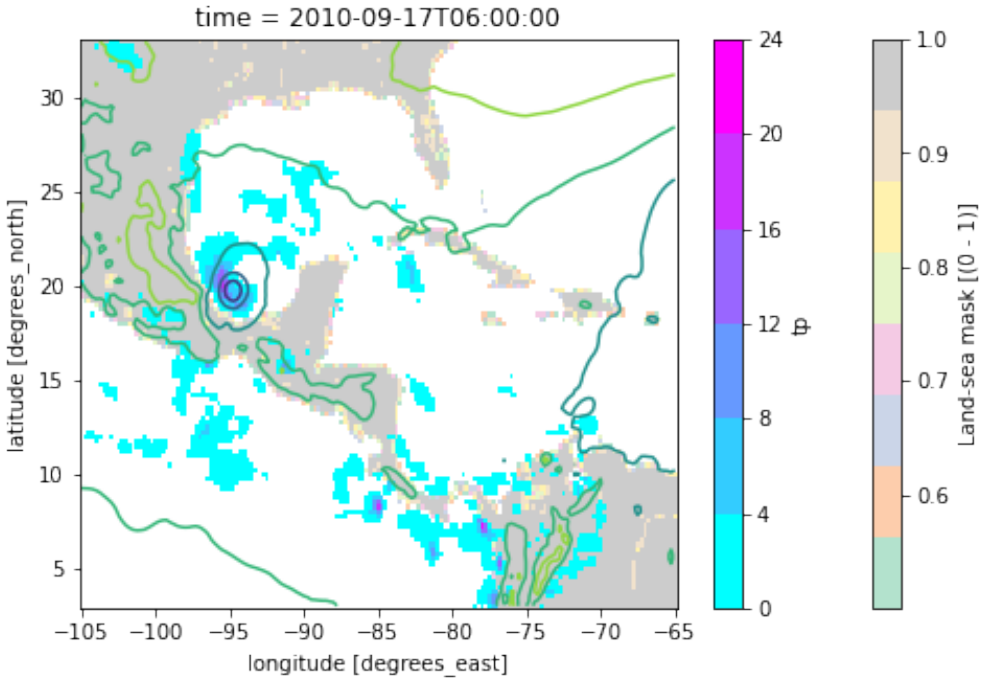


9.4. Máscaras

Las máscaras permiten elegir un subconjunto de los datos queremos usar. A diferencia de las funciones isel() y sel(), una máscara preserva las dimensiones originales de los datos. La función `where` nos permite hacer esta selección, aceptando una o más condiciones. Esta función puede encadenarse a otras (como suele suceder en general). Cuando se encadenen varias funciones es importante tener en cuenta el orden para optimizar recursos.

La condición de la máscara puede depender de los valores de los datos, o de los valores de las coordenadas. En el ejemplo siguiente queremos superponer varios campos, y evitar que la lluvia nos tape el campo de la máscara de tierra-mar (land sea mask), que está debajo. Para ello imponemos que solo se seleccionen los puntos con una precipitación mayor de 1mm:

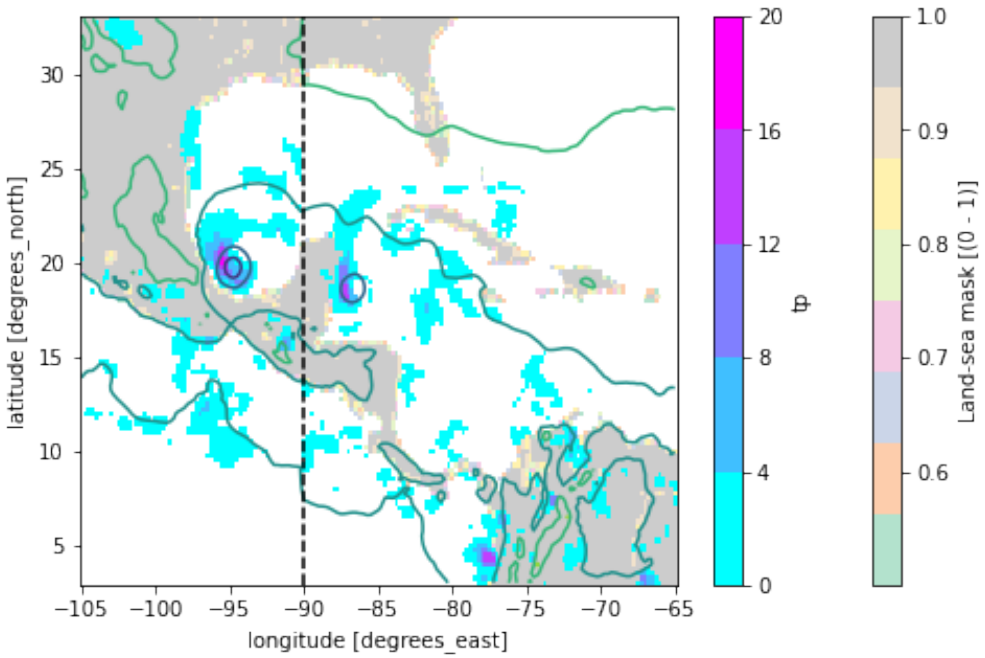
```
tp = ds_karl_sfc['tp']*1000.0  # Precipitación en mm
tpval = tp.where(tp > 1)
lsm = ds_karl_sfc['lsm']
lsm = lsm.where(lsm > 0.5)  # Puntos de tierra
msl = ds_karl_sfc['msl']
lsm.sel(time='2010-09-17T06:00:00').plot(cmap='Pastel2', figsize=(8,5))
tpval.sel(time='2010-09-17T06:00:00').plot(cmap='cool', levels=6)
msl.sel(time='2010-09-17T06:00:00').plot.contour()
```



Se puede rellenar la zona enmascarada con valores preestablecidos. Esto se indica con el segundo argumento de `where`.

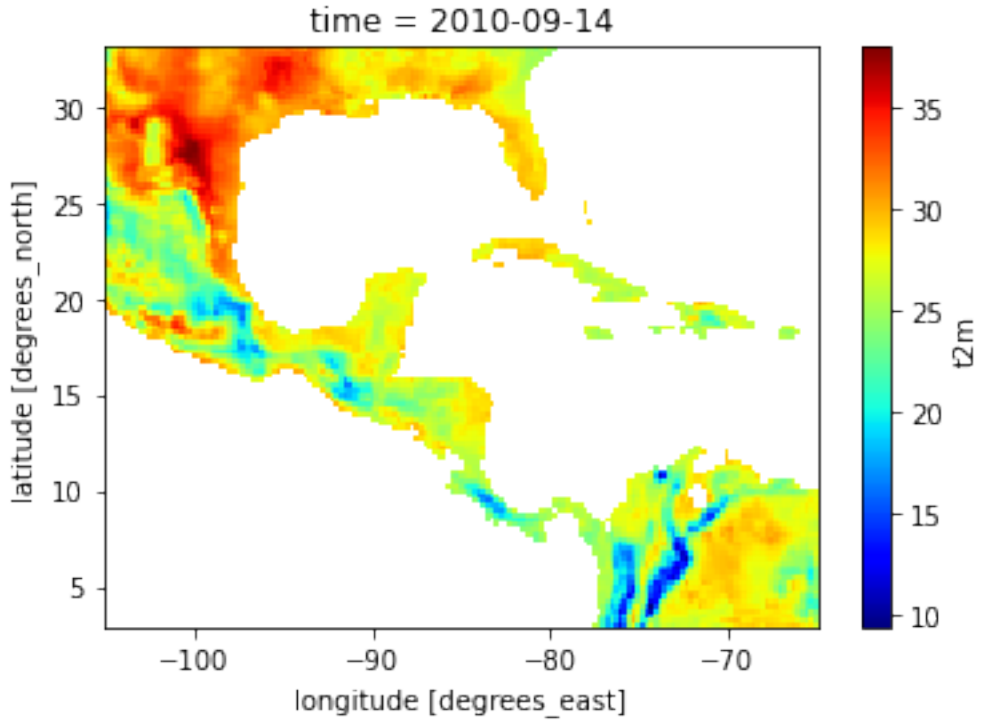
Las máscaras pueden depender también de las coordenadas. En el siguiente ejemplo se pinta la precipitación y el contorno de la presión a nivel del mar para las 6 UTC del día 17 de septiembre, pero solo para zonas al oeste del meridiano 90W, y se completa la gráfica pintando la precipitación y la presión para las 9 UTC del día 15 de septiembre para el resto de zonas (al este del meridiano 90W). El efecto que produce es el de "dos huracanes Karl": antes y después de cruzar la península del Yucatán (se muestra la gráfica solo para mostrar las capacidades de xarray, pero esto probablemente no tenga mucho sentido).

```
post_t = '2010-09-17T06:00:00'
pre_t = '2010-09-15T09:00:00'

lsm.sel(time=post_t).plot(cmap='Pastel2', figsize=(8,5))
tpval.sel(time=post_t).where(tpval.longitude<-90,
                             tpval.sel(time=pre_t)).plot(cmap='cool', levels=6)
msl.sel(time=post_t).where(msl.longitude<-90, msl.sel(time=pre_t)).plot.contour()
plt.axvline(x=-90, color='k', linestyle='--')
```



Notar que el campo máscara de tierra-mar toma valores un valor 0 en el mar y 1 en tierra, lo que nos puede servir para discriminar qué coordenadas corresponden a tierra o mar. En este ejemplo pintamos la temperatura a 2m, pero solo para los puntos de tierra:

```
t2m_karl_sfc = ds_karl_sfc['t2m'] - 273.15
t2m_karl_sfc.isel(time=0).where(lsm.isel(time=0)>0.5).plot(cmap='jet')
```



9.5. Estadísticas

Xarray puede calcular todo tipo de estadísticos, como Pandas. Por ejemplo, si queremos calcular la media:

```
t2m.mean()
```

Notar que es un datarray de un solo elemento, porque hemos hecho la media de todas las dimensiones (tiempo, latitud y longitud). Podemos extraer el valor con `item`:

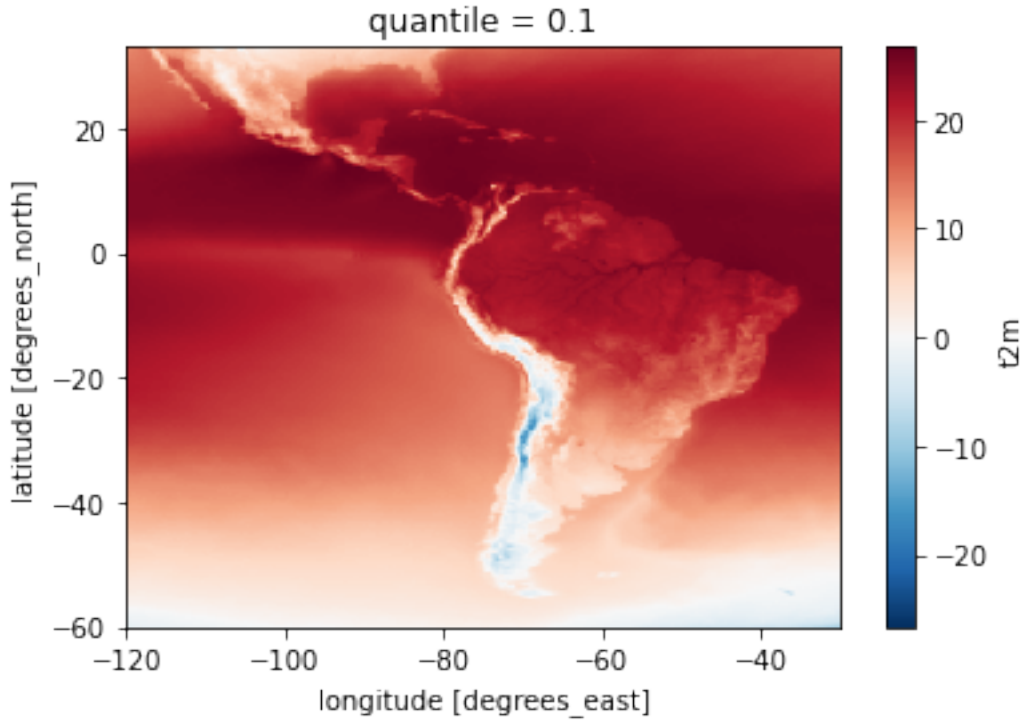
```
t2m.mean().item()
```

```
19.10647964477539
```

Si se quiere calcular el estadístico solo sobre algunas dimensiones las podemos especificar con el argumento *dim*.

Por ejemplo, si se quiere calcular el percentil 10 para la dimensión tiempo (o sea, el valor que solo tiene por debajo un 10 % de los datos para cada latitud y longitud), y pintar el mapa resultante:

```
t2m.quantile(0.1, dim='time').plot(size=4)
```



Recordar que podemos ver las coordenadas del datarray con *coords*:

```
t2m.coords
```

```
Coordinates:
  * longitude  (longitude) float32 -120.0 -119.8 -119.5 ... -30.5 -30.25 -30.0
  * latitude   (latitude) float32 33.0 32.75 32.5 32.25 ... -59.5 -59.75 -60.0
  * time       (time) datetime64[ns] 2022-01-05 ... 2022-12-30T21:00:00
```

Ejercicio

Calcular y pintar un mapa con el máximo intervalo de temperaturas para cada latitud y longitud con los datos de ERA5 de 2022.
Calcular el mínimo de la temperatura en tierra durante el episodio del huracán Karl.
Solucion –> `solucion_intervalo.py`

9.6. Computación

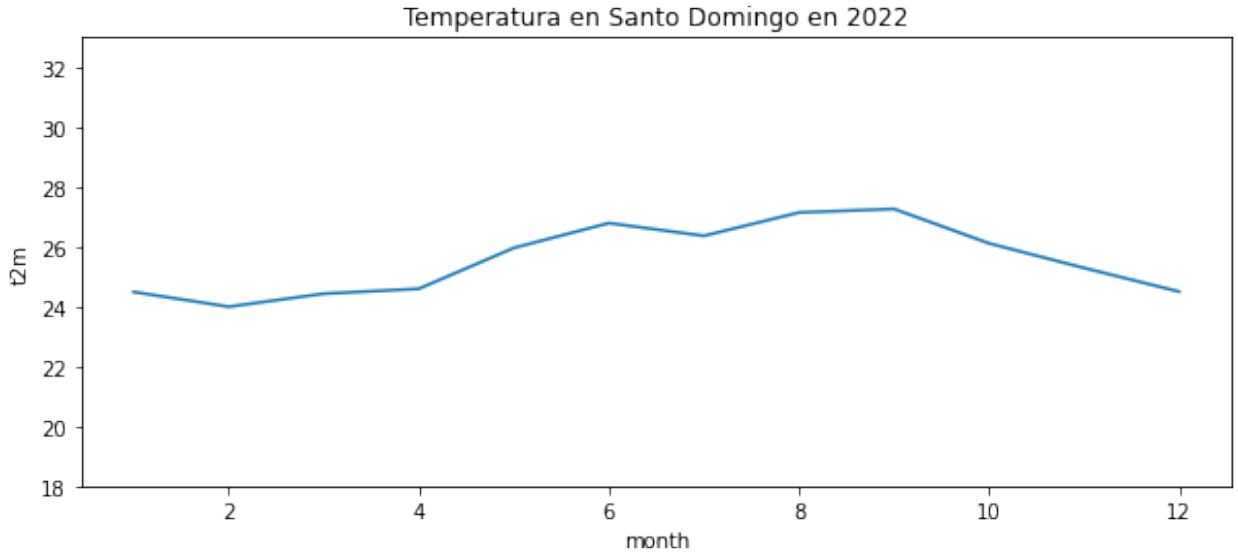
Xarray nos da varios métodos para hacer cálculos más avanzados. A modo de ejemplo, vamos a explicar uno de ellos, las agrupaciones (hay varios más).

9.6.1. Agrupaciones

De la misma forma que en pandas, podemos separar y combinar datos de acuerdo a un cierto criterio. Para ello se pueden usar los métodos `groupby` y `groupby_bins`. Vamos a mostrarlo con dos ejemplos.

En el primer ejemplo, vamos a agrupar los datos de temperatura para Santo Domingo por cada mes y calculamos la media:

```
t2m_stdom = t2m.sel(latitude=18.5, longitude=-69.9, method='nearest')

fig, ax = plt.subplots(figsize=(10,4))
t2m_stdom.groupby('time.month').mean().plot(ax=ax)
ax.set_title("Temperatura en Santo Domingo en 2022")
ax.set_ylim([18, 33])
```



Vemos que de esta forma eliminamos el ruido y vemos la variación de la temperatura durante todo el año más claramente.

En el segundo ejemplo, agrupamos los datos de temperatura del continente por estación del año (season), creando un objeto DataArrayGroupBy:

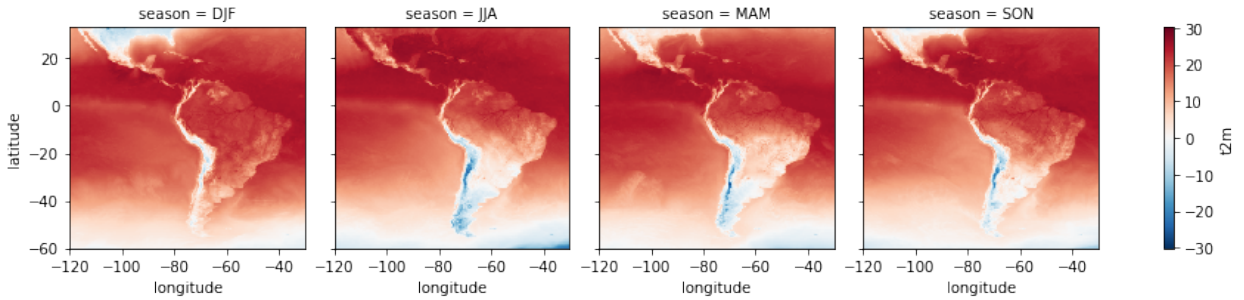
```
t2m.groupby("time.season")
```

```
DataArrayGroupBy, grouped over 'season'
4 groups with labels 'DJF', 'JJA', 'MAM', 'SON'.
```

Y después calculamos la mínima temperatura para cada estación:

```
seasonal_min = t2m.groupby("time.season").min()
seasonal_min
```

```
seasonal_min.plot(col="season")
```



9.7. Escritura de ficheros netCDF

Para escribir un dataset como un fichero netCDF en disco utilizaremos el método `to_netcdf`. Este método funciona también con un datarray, guardándolo como un netCDF.

Veamos un ejemplo: vamos a guardar únicamente la precipitación como variable en un nuevo fichero netCDF:

```
ds_karl_sfc = xr.open_dataset('curso_PIBM/era5_karl2010_sfc.nc')
ds_karl_sfc.data_vars
```

```
Data variables:
    t2m      (time, latitude, longitude) float32 ...
    lsm      (time, latitude, longitude) float32 ...
    msl      (time, latitude, longitude) float32 ...
    tp       (time, latitude, longitude) float32 ...
```

```
# Borramos estos tres datarrays, dejando solo tp como variable en el dataset
new_ds = ds_karl_sfc.drop(['t2m','lsm', 'msl'])
new_ds.to_netcdf('era5_karl_precipitacion.nc')
ds_precip = xr.open_dataset('era5_karl_precipitacion.nc')
ds_precip.data_vars
```

```
Data variables:
    tp       (time, latitude, longitude) float32 ...
```

Efectivamente, solo la precipitación está en el nuevo fichero.

9.8. Unos comentarios sobre Cartopy

Xarray está muy integrada con Cartopy (https://scitools.org.uk/cartopy/docs/latest/), una librería creada por el MetOffice para la generación de mapas y procesamiento de datos geoespaciales.

Con Cartopy podemos incluir varios elementos esenciales en un mapa, como proyecciones, líneas de costa, accidentes geográficos, fronteras, etc.

En este curso no vamos a tener tiempo de explicar Cartopy, pero se incluyen a continuación varios ejemplos que pueden dar idea de su utilidad. Se invita al alumno a continuar profundizando en esta librería:

```
import cartopy.crs as ccrs
import cartopy.feature as cfeature
```

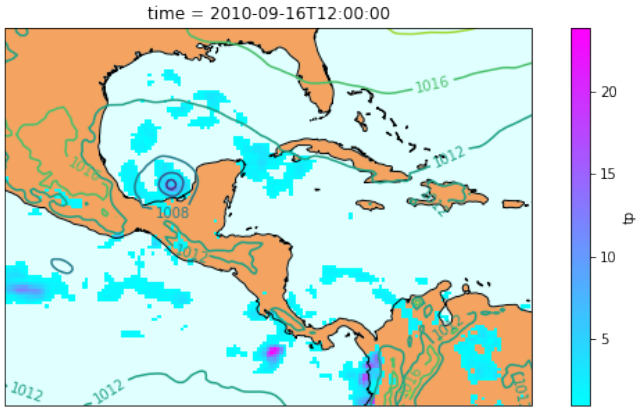
Mapa de Centroamérica y Sudamérica en proyección Ortográfica:

```
fig = plt.figure(figsize=[10, 5])
ax = fig.add_subplot(projection=ccrs.Orthographic(central_longitude=-80,
                                                  central_latitude=0))
ax.add_feature(cfeature.LAND, color='sandybrown')
ax.add_feature(cfeature.OCEAN, color='lightcyan')
```

```
ax.add_feature(cfeature.COASTLINE, edgecolor='black', lw=1)
plt.show()
```

Huracán Karl el día 17 de septiembre de 2010, incluyendo líneas de costa:

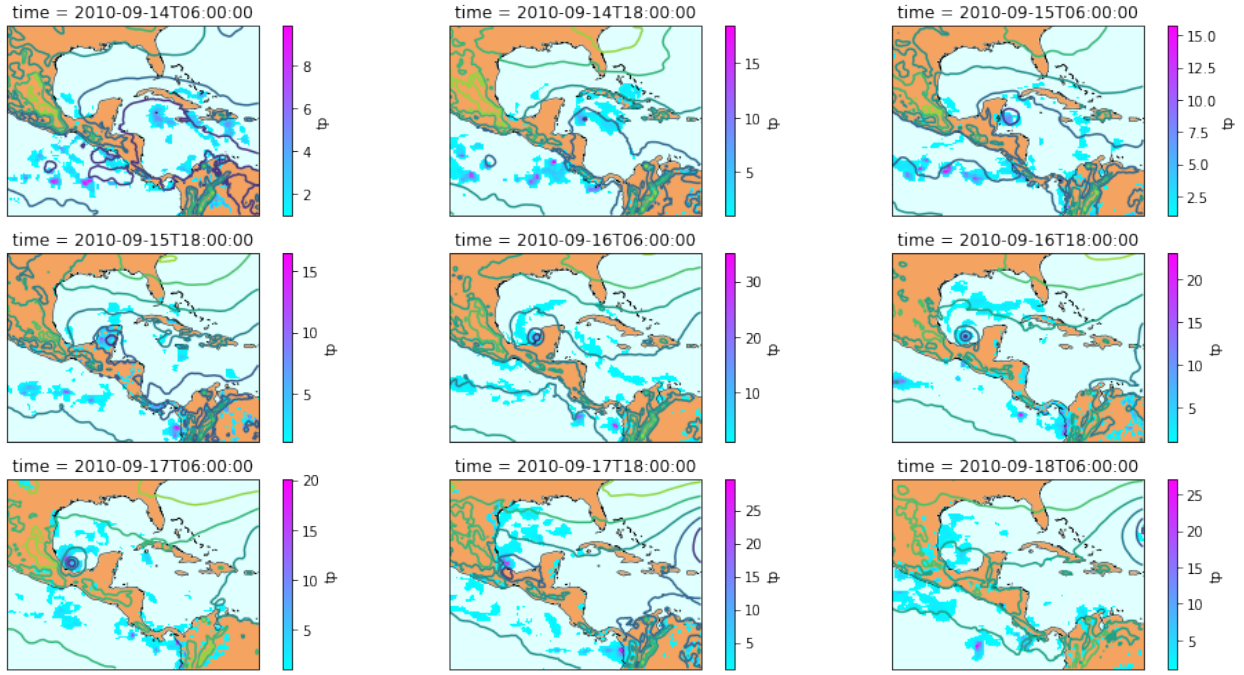
```
fig = plt.figure(figsize=[10, 5])
ax = fig.add_subplot(projection=ccrs.PlateCarree())
ax.set_extent((-105,-65,3,30))
ax.add_feature(cfeature.LAND, color='sandybrown')
ax.add_feature(cfeature.OCEAN, color='lightcyan')
ax.add_feature(cfeature.COASTLINE, edgecolor='black', lw=1)
tpval.isel(time=20).plot(cmap='cool')
msl = msl/100   # Pasamos de pascales a hectopascales
cont_msl = msl.isel(time=20).plot.contour()
ax.clabel(cont_msl)
plt.show()
```



Evolución del huracán Karl entre el 14 y el 18 de septiembre de 2010:

```
fig = plt.figure(figsize=[14, 7])

lst_axes = []

numx = 3
numy = 3

for i in range(0,numx*numy):
```

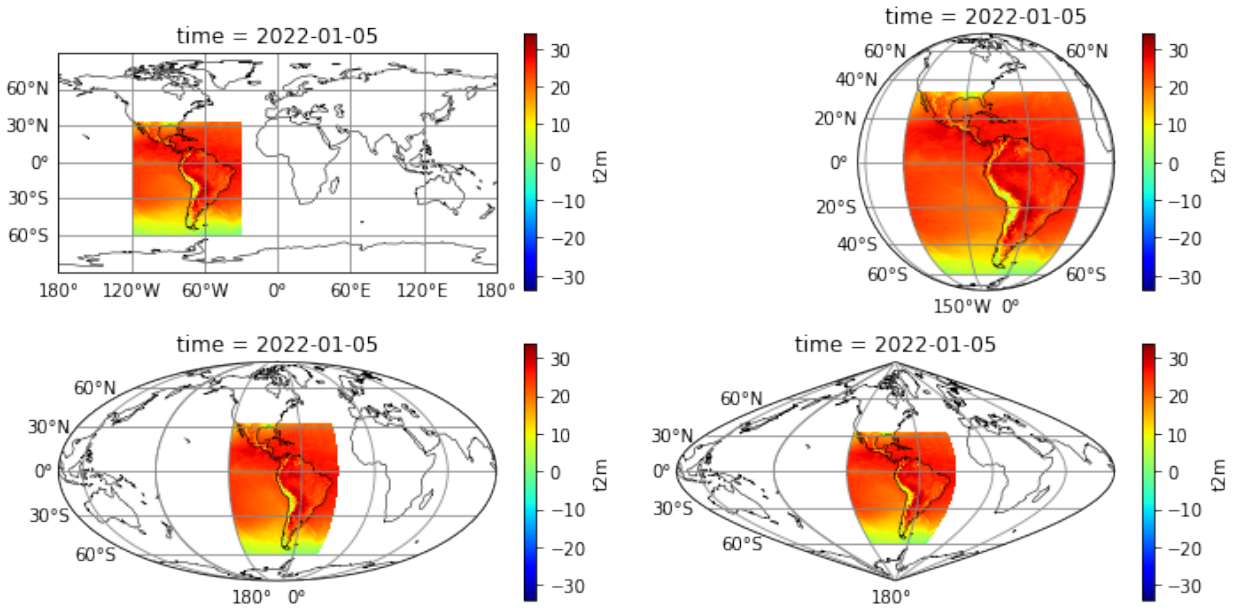
```
    id_time = 2+4*i

    lst_axes.append(fig.add_subplot(numx,numy,i+1,projection=ccrs.PlateCarree()))
    ax = lst_axes[i]

    tpval.isel(time=id_time).plot(ax=ax, transform=ccrs.PlateCarree(), cmap='cool')
    msl.isel(time=id_time).plot.contour(ax=ax)

    ax.add_feature(cfeature.LAND, color='sandybrown')
    ax.add_feature(cfeature.OCEAN, color='lightcyan')
    ax.add_feature(cfeature.COASTLINE, edgecolor='black', lw=0.5)

plt.tight_layout()
plt.show()
```



Mapa de temperatura a 2m en Centroamérica y Sudamérica usando cuatro proyecciones diferentes:

```
fig = plt.figure(figsize=[10, 5])

ax1 = fig.add_subplot(2,2,1,projection=ccrs.PlateCarree())
ax2 = fig.add_subplot(2,2,2,projection=ccrs.Orthographic(central_latitude=0,
                                                         central_longitude=-80))
ax3 = fig.add_subplot(2,2,3,projection=ccrs.Mollweide(central_longitude=-80))
ax4 = fig.add_subplot(2,2,4,projection=ccrs.Sinusoidal(central_longitude=-80))

temp = t2m.isel(time=0)

for ax in (ax1,ax2,ax3,ax4) :

   temp.plot(ax=ax, transform=ccrs.PlateCarree(),cmap='jet')
   ax.set_global()
   gl = ax.gridlines(color='#888888', draw_labels=True, x_inline=False, y_inline=False)
   gl.top_labels=False
   gl.right_labels=False
   ax.add_feature(cfeature.COASTLINE, edgecolor='black', lw=0.5)
```

```
plt.tight_layout()

plt.show()
```



Capítulo 10

Fuentes de datos disponibles

A continuación se enumeran algunas fuentes gratuitas de datos meteorológicos que pueden ser de interés:

- Datos del modelo del Centro Europeo de Predicciones a Medio Plazo (ECMWF):

  Web: https://www.ecmwf.int/en/forecasts/datasets/wmo-essential, https://www.ecmwf.int/en/forecasts/datasets/wmo-additional

  El ECMWF ofrece a los servicios meteorológicos miembros de la Organización Meteorológica Mundial (OMM) varios productos de su modelo de forma gratuita (algunos de ellos también disponibles para el público).

- Datos de CAMS del ECMWF (predicciones de aerosoles, contaminantes, gases de efecto invernadero, ozono estratosférico e índice ultravioleta):

  Web: https://atmosphere.copernicus.eu/global-forecast-plots

  Además de las gráficas disponibles se pueden descargar datos automáticamente a travé de su API.

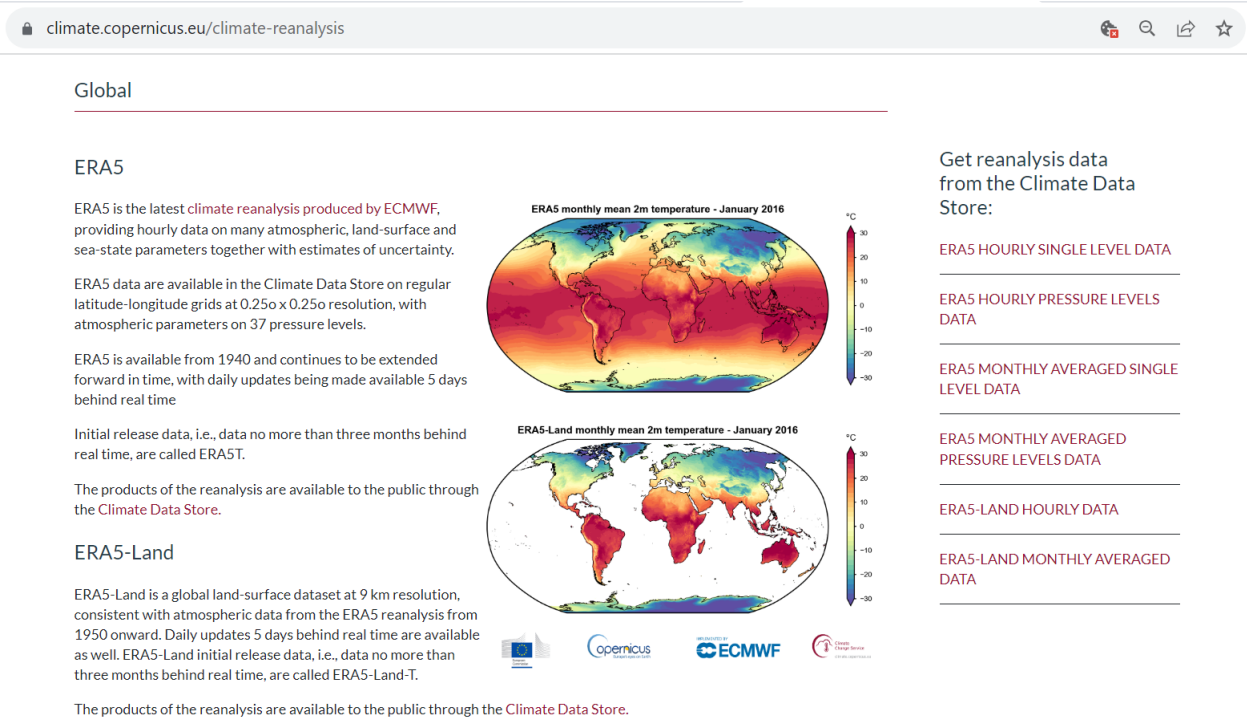
- Datos del reanálisis ERA5 del ECMWF, explicado en más detalle en el siguiente apartado. Parte del C3S de Copernicus.

- Datos de predicción estacional del C3S de Copernicus

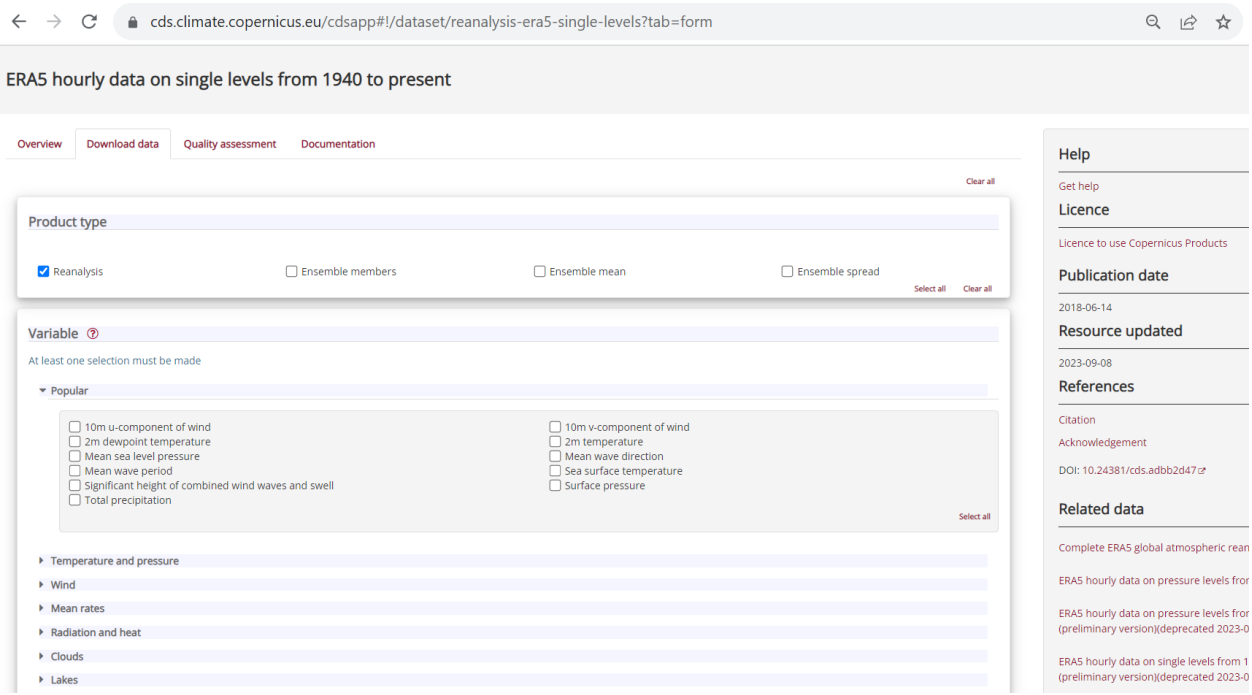
  Web: https://climate.copernicus.eu/seasonal-forecasts

10.1. Descarga de datos de la base de datos ERA5 de reanálisis

Se puede consultar en https://climate.copernicus.eu/climate-reanalysis:



Hay que darse de alta como usuario previamente. Se pueden descargar datos de superficie o de niveles de presión, entre otros. Por ejemplo, en esta página se descargan los datos de superficie:



Basta con indicar qué variables, periodo de tiempo y región se quieren descargar, así como el formato (grib o netCDF).

Capítulo 11

Prácticas

Para poder asimilar y afianzar los conocimientos adquiridos, en esta unidad vamos a realizar algunos prácticas.

11.1. Práctica 1: AEMET OpenData

Objetivo: Recuperar datos meteorológicos desde internet escribiendo un programa en *Python* para ello.

Vamos a realizar una práctica guiada haciendo uso de AEMET OpenData.

AEMET OpenData es una API REST desarrollado por AEMET que permite la difusión y la reutilización de la información meteorológica y climatológica de la Agencia, en el sentido indicado en la Ley 18/2015, de 9 de julio, por la que se modifica la Ley 37/2007, de 16 de noviembre, sobre reutilización de la información del sector público.

AEMET OpenData permite descargar gratuitamente los datos explicitados en el Anexo II de la resolución de 30 de diciembre de 2015 de AEMET, por la que se establecen los precios públicos que han de regir la prestación de servicios meteorológicos y climatológicos

Paso 1

Lo primero que necesitamos es una *api_key* para poder utilizar este Servicio de AEMET y descargarnos datos meteorológicos de la Agencia.

1. Acceder a *AEMET OpenData* –> *obtención de API Key* –> Solicitar
2. Introducir un correo electrónico.

1. Nos llegará un correo a nuestra bandeja de entrada y debemos confirmar que somos nosotros
2. Nos llegará un segundo correo con nuestra *api_key*
3. Ya tenemos nuestra *api_key* que tendrá un aspecto similar a este:

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqbWudGVyb2dAYWVtZXQuZXMiLCJqdGkiOi...
```

Paso 2

1. Visitaremos en AEMET OpenData –> *Acceso a desarrolladores* –> *Documentación HATEOAS* la documentación, o podemos ir directamente a *HATEOAS*

2. Buscaremos la API de *Datos de observación. Tiempo actual.* que son *Datos de observación horarios de las últimas 24 horas todas las estaciones meteorológicas de las que se han recibido datos en ese período. Frecuencia de actualización: continuamente.*, como nos indica AEMET OpenData.
3. Copiaremos la API, que es :

```
/api/observacion/convencional/todas
```

Paso 3

Estamos en disposición de crear nuestro programa en Python para recuperar los datos de observación.

Notar que son datos que se están actualizando constantemente y por tanto disponer de un programa que nos permita recuperarlos es muy útil.

Para realizar una consulta a AEMET OpenData la URL completa sería :

```
/api/observacion/convencional/todas + ?api_key= +  nuestra_api_key
```

Prueba a ver que te da la URL en un navegador, te dará algo parecido a lo siguiente:

```
{
  "descripcion": "exito",
  "estado": 200,
  "datos": "https://opendata.aemet.es/opendata/sh/81a9116b",
  "metadatos": "https://opendata.aemet.es/opendata/sh/55c2971b"
}
```

El *estado 200* quiere decir que nuestra consulta a AEMET OpenData ha tenido éxito y tenemos un enlace con los datos para que nos los descarguemos. Comprobar en un programa que nuestro estado es un 200 es una más que recomendable.

Paso 4

Ejercicio, opendata.py

Escribe un programa que recupere los datos de observación y los guarde en un fichero cuyo nombre contenga al menos la fecha en la que se han consultado.
Avanzado, guarda tu api_key en una variable de entorno, así como el directorio donde quieres que se guarde el fichero con los datos.

Pistas:

1. Guarda tu api_key en una variable.
2. Necesitas importar el módulo *"request"*, "datetime" y "os".
3. AEMET OpenData dispone de programas ejemplo que te pueden ayudar.

Paso 4

Este paso es para explicar cómo resolver el ejercicio.

Escribimos con un editor de texto, nuestras primeras líneas, importando los módulos necesarios y guardando en variables algunas constantes.

```
# -- coding: utf-8 --
import os
import datetime
import request

url = "https://opendata.aemet.es/opendata/api/observacion/convencional/todas/"
akey = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqbWudGVyb2dAYWVtZXQuZXMiL..."
```

Ahora con *"request"* tenemos que descargar el JSON con el enlace de descarga de los datos. Este sería el modo de hacerlo:

```
querystring = {"api_key": akey}

headers = {'cache-control': "no-cache"}

#Hacemos la llamada
response = requests.request("GET", url, headers=headers, params=querystring, verify=False)

print(response.text)
```

Para poder continuar y como pista, también necesitamos el módulo JSON que nos permitirá cargar la respuesta *response.text* como un diccionario, de la siguiente forma:

```
import json
#Obtenemos el json de la respuesta
data = json.loads(response.text)
```

Para terminar, terminar con nuestro programa tenemos que comprobar que el estado es un 200 y recuperar los datos en caso de que así sea, veamos como :

```
estado = data["estado"]

#Si el estado es 200 (éxito) obtenemos los datos y los guardamos en un fichero
if (estado == 200):

    urlDatos = data["datos"]
    datos = requests.request("GET", urlDatos, headers=headers,
                             params=querystring, verify=False)

    #Insertamos los datos en un fichero
    fich_datos = open('fich_productos','w')
    fich_datos.write(datos.content)
    fich_datos.close()
    print( "El fichero de datos esta en: $HOME/scripts_python/bdp/fich_productos")
else:
    print( "No se han podido obtener los datos. Código de estado: ", estado)

exit(0)
```

Si queremos introducir la fecha actual al nombre del fichero con los datos de observación 'fich_productos', tendremos:

```
import datetime
d = datetime.datetime.now()
ft = "%Y%m%dT%H%M%S"
t = datetime.datetime.now().strftime(ft)
# 20220505T090424
fich_name = "observacion_" + t + ".json"
```

Y también habrá que cambiar la línea

```
# fich_datos = open('fich_productos','wb')
fich_datos = open(fich_name,'wb')
```

11.2. Práctica 2: lectura y escritura de ficheros JSON

Objetivo: Leer un fichero JSON con datos de observación, trabajar con sus variables, seleccionar campos de nuestro interés y por último escribir un fichero JSON.

Vamos a realizar una práctica guiada haciendo uso de un fichero JSON , en concreto usaremos el siguiente:

Fichero de entrada con nuestros datos de observación `datos_observacion.json`

Un fichero JSON tiene la siguiente estructura:

```
{
  "nubes": [
    {
      "tipo": "Cirrus",
      "descripcion": "Nubes altas y delgadas en forma de filamentos o hebras.",
      "altura": "Alta"
    },
    {
      "tipo": "Cumulus",
      "descripcion": "Nubes blancas y esponjosas con bordes definidos.",
      "altura": "Media"
    },
    {
      "tipo": "Stratus",
      "descripcion": "Nubes bajas y grises que cubren todo el cielo.",
      "altura": "Baja"
    }
  ],
  "meteoros": [
    {
      "tipo": "Lluvia",
      "descripcion": "Precipitación en forma de gotas de agua líquida.",
      "intensidad": "Moderada"
    },
    {
      "tipo": "Nieve",
      "descripcion": "Precipitación en forma de cristales de hielo.",
      "intensidad": "Débil"
    },
    {
      "tipo": "Granizo",
      "descripcion": "Precipitación en forma de bolas de hielo.",
      "intensidad": "Fuerte"
    }
  ]
}
```

Paso 1

Crear un python que nos permita leer nuestro fichero JSON , definiremos una función para ello, cuidado con la codificación.

Importamos los módulos y creamos la función, intenta hacerlo.

```
# -- coding: utf-8 --

import json

fichero = 'datos_observacion.json'

def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)
    return data

leer_archivo_json()
```

Paso 2

Pasar el nombre del fichero de entrada como un argumento en la ejecución del python.

```
leer_observacion.py datos_observacion.json
```

Se haría lo siguiente:

```
# -- coding: utf-8 --

import pytz

fichero = sys.argv[1]

def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)

    return data

leer_archivo_json()
```

Paso 3

Ejercicio, vamos a ampliar las funciones del python que tenemos.

1. Agrega una función que lea todos los campos que tenemos disponibles.
2. Agrega una función que imprima todos los campos y nos permita seleccionar algunos de ellos.
3. Por último imprime los campos seleccionados.

Paso 4

Ahora disponemos de un Python que lee un fichero JSON, que nos imprime todos los campos que encuentra y nos permite seleccionar un grupo de ellos.

Ejercicio, vamos adaptarlo de la siguiente forma.

1. Queremos los campos de "lon", "lat", "alt", "idema" y "fint"; estén siempre por defecto entre los campos seleccionados.
2. Por tanto estos campos no los podrá seleccionar el usuario.

Paso 5

Llegados a este punto solo nos queda escribir un JSON con nuestra selección:

Ejercicio,

1. Escribimos un JSON que contenga todas las estaciones con los datos seleccionados.
2. Queremos que el nombre del fichero de salida contenga la fecha actual.

Paso 6

Por último, queremos almacenar las fechas como una lista de objetos fecha.

11.3. Practica 3: verificación de predicciones

Objetivo: En esta práctica vamos a usar las librerías pandas, matplotlib y xarray en un posible caso real.

Disponemos de predicciones para el día 13 de agosto de 2023 realizadas con el modelo WRF para el cono sur del continente, cortesía del Servicio Meteorológico Nacional de Argentina. Vamos a verificar estas predicciones frente a las observaciones medidas durante ese día en algunas estaciones meteorológicas de INUMET, el Servicio Meteorológico de Uruguay.

Utilizaremos las librerías anteriores para formatear y manejar todos estos datos, así como para mostrar resultados numéricos y gráficas y mapas que nos ayuden a entender esta situación.

En la vida real no tiene sentido hacer una verificación de solo un día. Un periodo más largo, por ejemplo, un año, sería más apropiado. Sin embargo el procedimiento a seguir sería muy similar, y solo variaría la cantidad de datos a procesar, por lo que este puede ser un ejercicio interesante y útil.

Podemos distinguir tres tareas diferentes para conseguir nuestro objetivo, que especificamos en los siguientes apartados:

Paso 1: extraer y procesar las predicciones del modelo meteorológico

El Servicio Meteorológico Nacional de Argentina dispone de un estupendo repositorio en la Web donde se pueden descargar las predicciones de su modelo WRF en formato netCDF, ya sea de forma manual, o automáticamente (esto último sería preferible si fueramos a realizar las verificaciones periódicamente de forma automática, sin intervención humana).

En la dirección https://odp-aws-smn.github.io/documentation_wrf_det/Acceso_a_los_datos/ se explican los pasos para descargar los ficheros que se deseen. Entre otras alternativas, es posible utilizar una librería de python llamada s3fs para realizar la descarga.

Como paso previo, se han descargado ya los ficheros a utilizar (instalando previamente la librería s3fs), por lo que en la práctica ya disponemos de ellos en nuestro directorio local. A modo de curiosidad, este es el código que se ha utilizado para ello (siguiendo las indicaciones de la página anterior):

```
import os
import s3fs

s3_dir = 's3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/'    # Directorio remoto
aemet_dir = '.'                                           # Directorio local

fs = s3fs.S3FileSystem(anon=True)

# Descargamos las predicciones entre las 9 y las 24 UTC (de h+9 a h+24)
for f in ['WRFDETAR_01H_20230813_00_%s.nc'%str(h).zfill(3) for h in range(9,25)]:
    print("Descargando "+os.path.join(s3_dir, f))
    fs.get(rpath=os.path.join(s3_dir, f),
           lpath=os.path.join(aemet_dir, f))
```

```
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_009.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_010.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_011.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_012.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_013.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_014.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_015.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_016.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_017.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_018.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_019.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_020.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_021.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_022.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_023.nc
Descargando s3://smn-ar-wrf/DATA/WRF/DET/2023/08/13/00/WRFDETAR_01H_20230813_00_024.nc
```

Se propone al alumno realizar estas tareas:

1. Mostrar mapas en mosaico de la temperatura a 2 metros, la humedad relativa a 2 metros, y la velocidad del viento a 10 metros para las 9, 15 y 21 UTC.

2. Mostrar gráficas en mosaico del corte para la latitud xx de la temperatura a 2 metros, la humedad relativa a 2 metros, y la velocidad del viento a 10 metros para las 9, 15 y 21 UTC.

3. Mostrar mapas con la temperatura media, máxima y mínima durante el día 13.

4. Extraer las predicciones de temperatura a 2 metros para la estación "Aeropuerto Melilla G3" (por ejemplo). Pista: el alumno necesitará seguir las indicaciones de https://odp-aws-smn.github.io/documentation_wrf_det/Tutoriales/

Paso 2: formateo de las observaciones

Se han descargado observaciones correspondientes al día 13 de agosto de 2023 de 24 estaciones de INUMET desde su página web, y se ha creado un fichero Excel con nombre *inumet_20230813.xlsx* para guardarlas.

Este fichero Excel consta de una hoja llamada "metadatos" que contiene información sobre cada estación (entre otros, su nombre, longitud y latitud), y de 15 hojas más, una para cada hora del día (desde las 6 a las 21 hora local), con las observaciones de todas las estaciones para esa hora.

Note: Recordar que en Uruguay se usa el huso horario GMT-3.

En este apartado cargaremos en memoria estos datos con una estructura de datos (un dataframe de pandas) que nos permita manejarlos cómodamente. Para ello el alumno debe seguir estos pasos:

1. Cargar el fichero Excel en un DataFrame. Pista: habrá que utilizar el argumento *sheet_name* en la función `pd.read_excel` para identificar cada hoja. Los valientes pueden intentar usar la opción sheet_name=None, que carga todas las hojas de golpe.

2. Crear una columna de tipo fecha-hora ("datetime" o "timestamp"), para que así cada registro este identificado por su estación y su fecha y hora. Pista: se puede utilizar el módulo datetime, o alternativamente, la función `pd.Timestamp` de pandas, que es muy similar.

3. Concatenar todas las horas en un gran dataframe

Paso 3: verificación de las predicciones

Ahora que ya tenemos observaciones y predicciones listas, podemos compararlas. El alumno tiene que realizar estas tareas:

1. Crear un gráfico que muestre la evolución de la temperatura a 2 metros observada y predicha para la estación "Aeropuerto Melilla G3" (por ejemplo) durante el día 13.

2. Calcular el sesgo y el error cuadrático medio de la predicción para esta estación.

Recordar que el sesgo y el error cuadrático medio son dos índices de verificación que se definen de esta forma:

- Sesgo (o bias):

$$sesgo = \frac{1}{n} \sum_{i=1}^{n} f_i - o_i$$

- Error cuadrático medio (o rmse):

$$rmse = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (f_i - o_i)^2}$$

donde f_i es cada una de las predicciones, o_i cada una de las observaciones, y n el número de parejas observación/predicción

Capítulo 12

Soluciones de las prácticas

12.1. Solución práctica 1

```
# -*- coding: utf-8 -*-
import json
import os
import datetime
import requests

url = "https://opendata.aemet.es/opendata/api/observacion/convencional/todas/"
akey = "<clave que corresponda>"

# fecha actual en el nombre del fichero con los datos de observación

d = datetime.datetime.now()
ft = "%Y%m%dT%H%M%S%z"
t = datetime.datetime.now().strftime(ft)
# 20220505T090424+0000
fich_name = "observacion_" + t + ".json"

querystring = {"api_key": akey}

headers = {'cache-control': "no-cache"}

#Hacemos la llamada
response = requests.request("GET", url, headers=headers, params=querystring, verify=False)

print(response.text)

#Obtenemos el json de la respuesta
data = json.loads(response.text)

estado = data["estado"]

#Si el estado es 200 (exito) obtenemos los datos y los guardamos en un fichero
if (estado == 200):

    urlDatos = data["datos"]
    datos = requests.request("GET", urlDatos, headers=headers,
                             params=querystring, verify=False)

    #Insertamos los datos en un fichero
    #fich_datos = open('fich_productos','w')
    fich_datos = open(fich_name,'wb')
    fich_datos.write(datos.content)
    fich_datos.close()
```

```
    print( "El fichero de datos esta en: $HOME/scripts_python/bdp/fich_productos")
else:
    print("No se han podido obtener los datos. Codigo de estado: ", estado)

exit(0)
```

12.2. Solución práctica 2

Paso 3

```
# -*- coding: utf-8 -*-

import sys
import json

fichero = sys.argv[1]

def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)

    return data

def obtener_campos_disponibles(data):
    campos_todos = []
    for item in data:
        campos = list(item.keys())
        for c in campos :
            if c not in campos_todos:
                campos_todos.append(c)
    return campos_todos

def seleccionar_campos(campos):
    print("Campos disponibles:")
    for i, campo in enumerate(campos, 1):
            print(f"{i}. {campo}")

    seleccionados = []
    opcion = input("Selecciona los campos que te interesen (separados por coma): ")
    indices_seleccionados = opcion.split(",")

    for indice in indices_seleccionados:
        indice = int(indice.strip()) - 1
        if indice >= 0 and indice < len(campos):
            seleccionados.append(campos[indice])

    return seleccionados

# Uso de las funciones
data = leer_archivo_json()
campos_disponibles = obtener_campos_disponibles(data)
campos_seleccionados = seleccionar_campos(campos_disponibles)
print(campos_seleccionados)
```

Paso 4

```
# -*- coding: utf-8 -*-

import sys
```

```
import json

fichero = sys.argv[1]

def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)

    return data

def obtener_campos_disponibles(data):
    campos_todos = []
    for item in data:
        campos = list(item.keys())
        for c in campos :
            if c not in campos_todos:
                campos_todos.append(c)
    return campos_todos

def seleccionar_campos(campos):
    print("Campos disponibles:")
    for i, campo in enumerate(campos, 1):
        if campo not in ['lon','lat','alt','idema','fint']:
            print(f"{i}. {campo}")

    seleccionados = []
    opcion = input("Selecciona los campos que te interesen (separados por coma): ")
    indices_seleccionados = opcion.split(",")

    for indice in indices_seleccionados:
        indice = int(indice.strip()) - 1
        if indice >= 0 and indice < len(campos):
            seleccionados.append(campos[indice])

    seleccionados_mas = seleccionados + ['lon','lat','alt','idema','fint']

    return seleccionados_mas


# Uso de las funciones
data = leer_archivo_json()
campos_disponibles = obtener_campos_disponibles(data)
campos_seleccionados = seleccionar_campos(campos_disponibles)
print(campos_seleccionados)
```

Paso 5

```
# -*- coding: utf-8 -*-

import sys
import json
from datetime import datetime

fichero = sys.argv[1]
fich_salida = "resultado_" + datetime.now().strftime('%Y%m%d_%H%M%S') + ".json"


def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)
```

```
    return data

def obtener_campos_disponibles(data):
    campos_todos = []
    for item in data:
        campos = list(item.keys())
        for c in campos :
            if c not in campos_todos:
                campos_todos.append(c)
    return campos_todos

def seleccionar_campos(campos):
    print("Campos disponibles:")
    for i, campo in enumerate(campos, 1):
        if campo not in ['lon','lat','alt','idema','fint']:
            print(f"{i}. {campo}")

    seleccionados = []
    opcion = input("Selecciona los campos que te interesen (separados por coma): ")
    indices_seleccionados = opcion.split(",")

    for indice in indices_seleccionados:
        indice = int(indice.strip()) - 1
        if indice >= 0 and indice < len(campos):
            seleccionados.append(campos[indice])

    return seleccionados

def guardar_campos_seleccionados(data, campos_seleccionados):
    l_obs = []
    nuevos_datos = {"obs":l_obs}

    for item in data:
        nuevo_item = {campo: item.get(campo) for campo in campos_seleccionados}
        nuevo_item['lon'] = item.get('lon')
        nuevo_item['lat'] = item.get('lat')
        nuevo_item['idema'] = item.get('idema')
        nuevo_item['fint'] = item.get('fint')
        l_obs.append(nuevo_item)

    with open(fich_salida, 'w') as file:
        json.dump(nuevos_datos, file, indent=4)

# Uso de las funciones
data = leer_archivo_json()
campos_disponibles = obtener_campos_disponibles(data)
campos_seleccionados = seleccionar_campos(campos_disponibles)
guardar_campos_seleccionados(data, campos_seleccionados)
```

Paso 6

```
# -*- coding: utf-8 -*-

import sys
import json
from datetime import datetime
import pytz

fichero = sys.argv[1]
```

```
fich_salida = "resultado_" + datetime.now().strftime('%Y%m%d_%H%M%S') + ".json"


def leer_archivo_json():
    with open(fichero,'r', encoding='latin-1') as file:
        data = json.load(file)

    fecha_objetos = []
    for item in data:
        fint_str = item.get('fint', '')
        if fint_str:
            fecha_utc = datetime.strptime(fint_str, '%Y-%m-%dT%H:%M:%S').replace(tzinfo=pytz.UTC)
            if fecha_utc not in fecha_objetos:
                fecha_objetos.append(fecha_utc)
#            item['fint'] = fecha_utc

    return data,fecha_objetos

def obtener_campos_disponibles(data):
    campos_todos = []
    for item in data:
        campos = list(item.keys())
        for c in campos :
            if c not in campos_todos:
                campos_todos.append(c)
    return campos_todos

def seleccionar_campos(campos):
    print("Campos disponibles:")
    for i, campo in enumerate(campos, 1):
        if campo not in ['lon','lat','alt','idema','fint']:
            print(f"{i}. {campo}")

    seleccionados = []
    opcion = input("Selecciona los campos que te interesen (separados por coma): ")
    indices_seleccionados = opcion.split(",")

    for indice in indices_seleccionados:
        indice = int(indice.strip()) - 1
        if indice >= 0 and indice < len(campos):
            seleccionados.append(campos[indice])

    return seleccionados

def guardar_campos_seleccionados(data, campos_seleccionados):
    l_obs = []
    nuevos_datos = {"obs":l_obs}

    for item in data:
        nuevo_item = {campo: item.get(campo) for campo in campos_seleccionados}
        nuevo_item['lon'] = item.get('lon')
        nuevo_item['lat'] = item.get('lat')
        nuevo_item['idema'] = item.get('idema')
        nuevo_item['fint'] = item.get('fint')
        l_obs.append(nuevo_item)

    with open(fich_salida, 'w') as file:
        json.dump(nuevos_datos, file, indent=4)
```

```
# Uso de las funciones
data,fecha_objetos = leer_archivo_json()
campos_disponibles = obtener_campos_disponibles(data)
campos_seleccionados = seleccionar_campos(campos_disponibles)
guardar_campos_seleccionados(data, campos_seleccionados)
print( "Tenemos " + str( len(data)) + " observaciones"
       " de las siguientes fechas: \n", fecha_objetos)
```

12.3. Solución práctica 3

Paso 1: extraer y procesar las predicciones del modelo meteorológico

1. Mostrar un mapa de la humedad relativa a 2 metros para las 18 UTC

```
import xarray as xr
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Abrimos el fichero de las 18 UTC con `open_dataset`. En la página de descarga se recomienda utilizar los argumentos *decode_coords* y *engine*, así que lo hacemos así:

```
ds_18utc = xr.open_dataset('WRFDETAR_01H_20230813_00_018.nc',
                           decode_coords = 'all', engine = 'h5netcdf')
ds_18utc.HR2.sel(time='2023-08-13T18:00:00').plot()
```

2. Mostrar mapas en mosaico de la temperatura a 2 metros, la humedad relativa a 2 metros, y la velocidad del viento a 10 metros para las 9, 15 y 21 UTC.

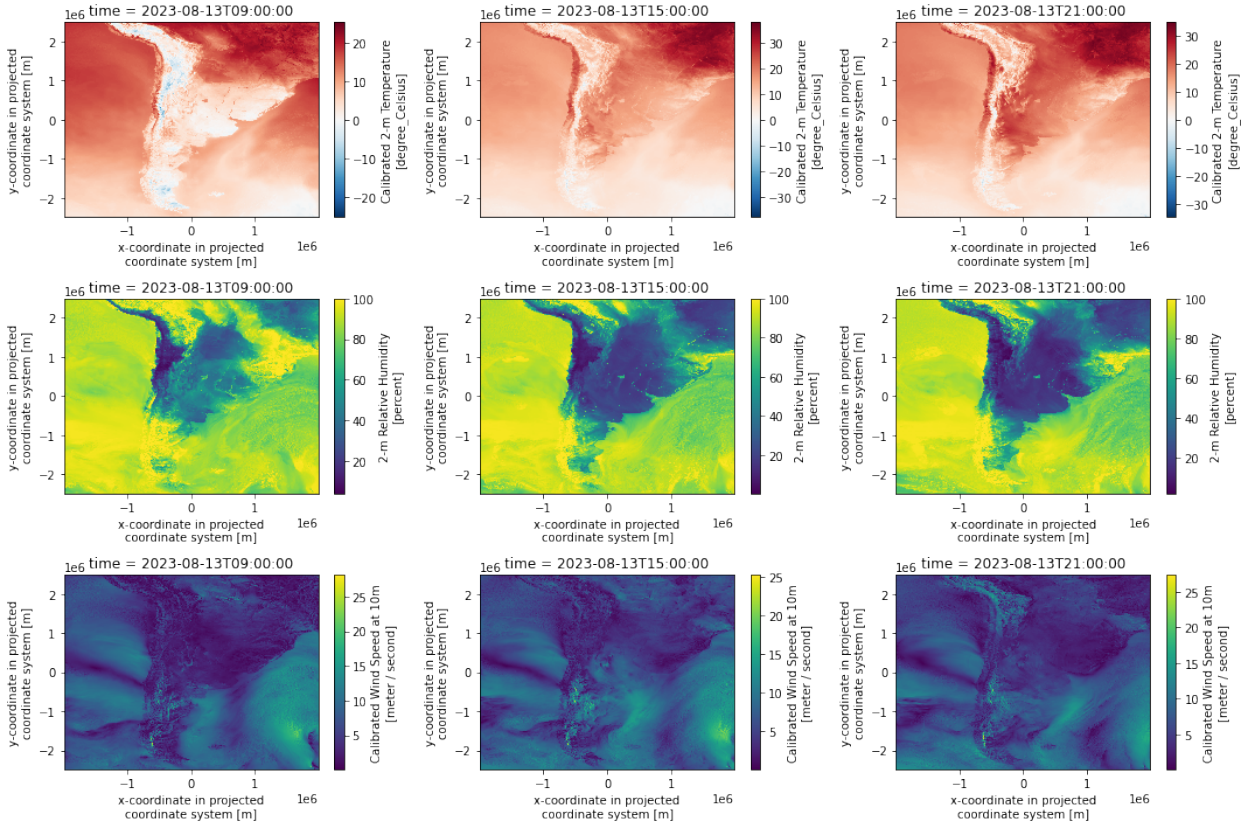
Podemos cargar primero todos los ficheros netCDF y concatenarlos para formar un dataset que contenga todos los datos (open_mfdataset parece fallar aquí, no conozco el motivo):

```
for step in [str(i).zfill(3) for i in range(9,25)]:
    if step == "009":
        ds = xr.open_dataset('WRFDETAR_01H_20230813_00_%s.nc'%step,
                             decode_coords = 'all', engine = 'h5netcdf')
    else:
        ds_step = xr.open_dataset('WRFDETAR_01H_20230813_00_%s.nc'%step,
                                  decode_coords = 'all', engine = 'h5netcdf')
        ds = xr.concat([ds, ds_step], dim="time")
        ds_step.close()
ds
```

Una forma de pintar todos los mapas de golpe es mediante un bucle que vaya escogiendo cada objeto Axes, de forma que cada Axes corresponda a una hora y variable distintas:

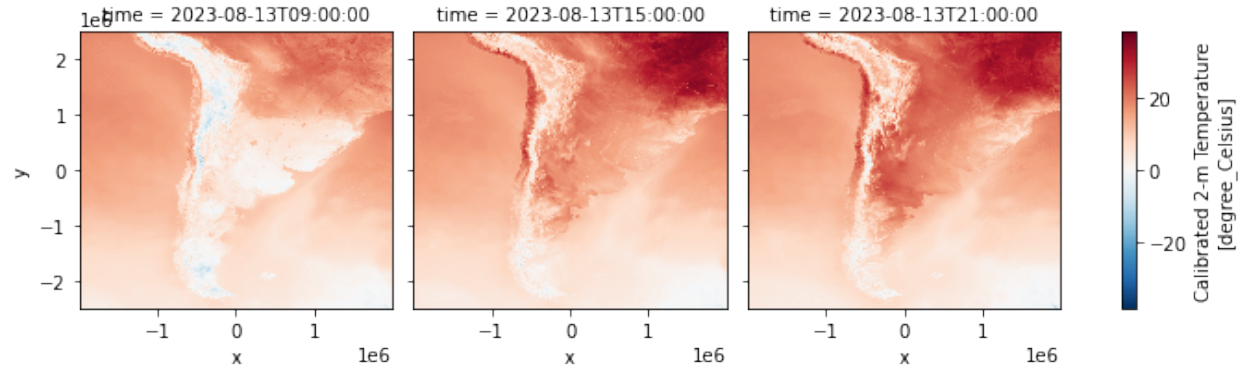
```
fig = plt.figure(figsize=[15, 10])

lst_axes = []
numx = 3
numy = 3

variables = ['T2', 'HR2', 'magViento10']
id_times = ['2023-08-13T'+h+':00:00' for h in ['09','15','21']]

for i in range(0,numx*numy):

    lst_axes.append(fig.add_subplot(numx,numy,i+1))
    ax = lst_axes[i]

    variable = ds[variables[int(i/3)]]
```

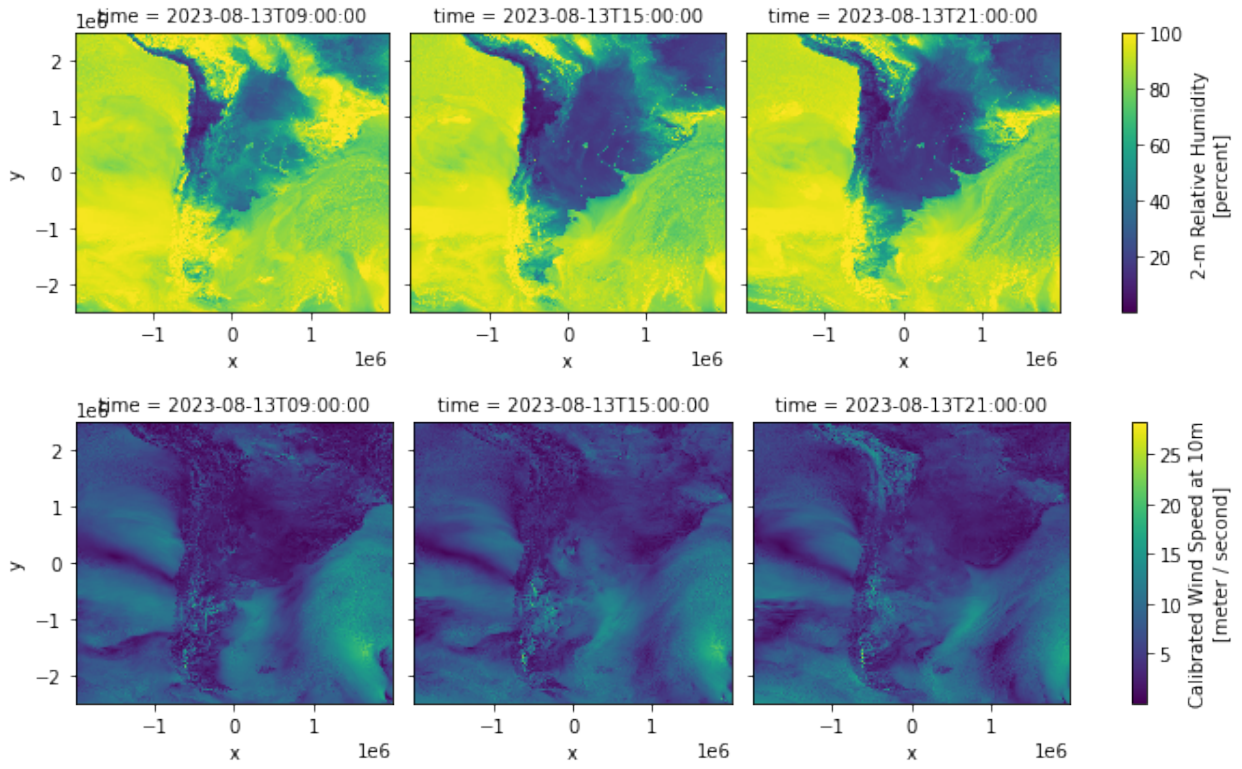
```
    variable.sel(time=id_times[i%3]).plot(ax=ax)

plt.tight_layout()
plt.show()
```



Otra alternativa es hacer un bucle más simple, que solo recorra las variables, y utilizar *Faceting* para recorrer la dimensión tiempo:

```
variables = ['T2', 'HR2', 'magViento10']

for var in variables:
    ds[var].isel(time=slice(0, 13, 6)).plot(col='time')

plt.show()
```

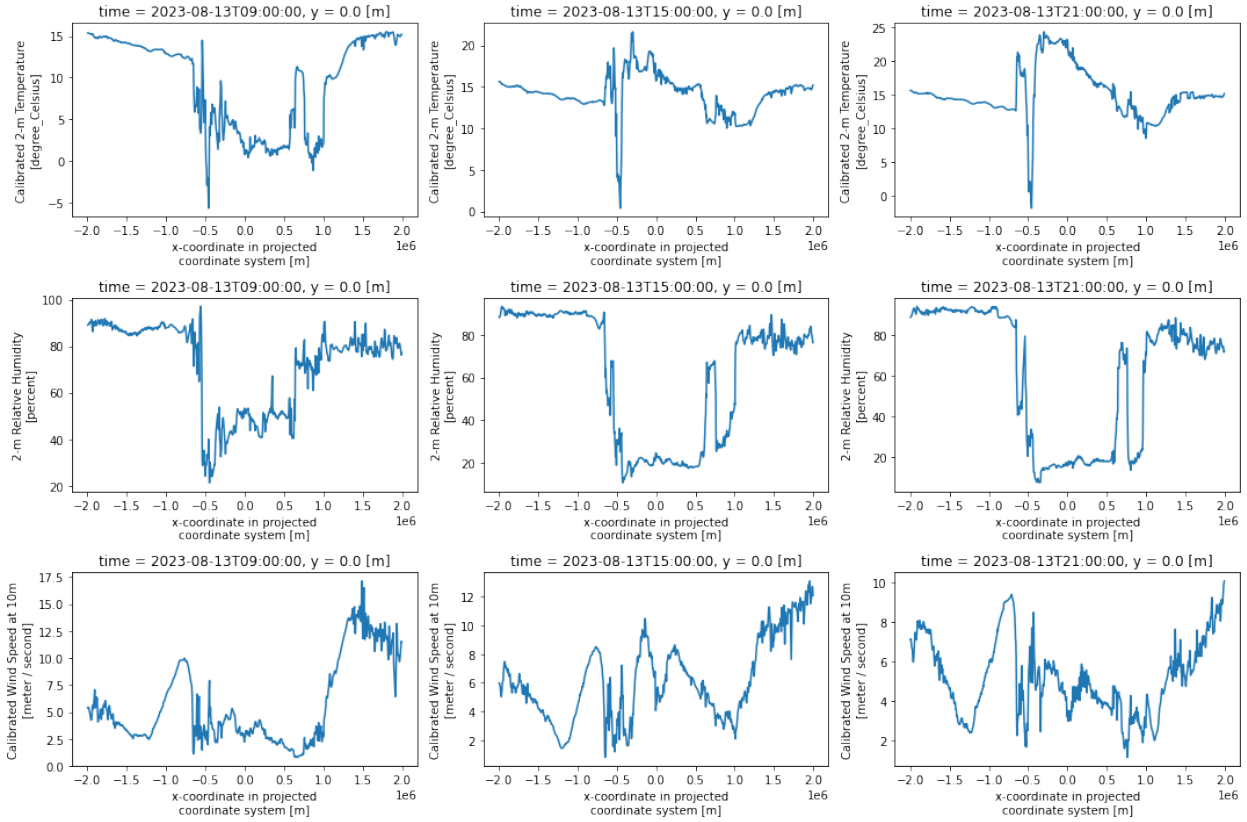




3. Mostrar gráficas en mosaico del corte para la latitud xx de la temperatura a 2 metros, la humedad relativa a 2 metros, y la velocidad del viento a 10 metros para las 9, 15 y 21 UTC.

Basta con eliminar la dimensión latitud, y ya sacamos las gráficas:

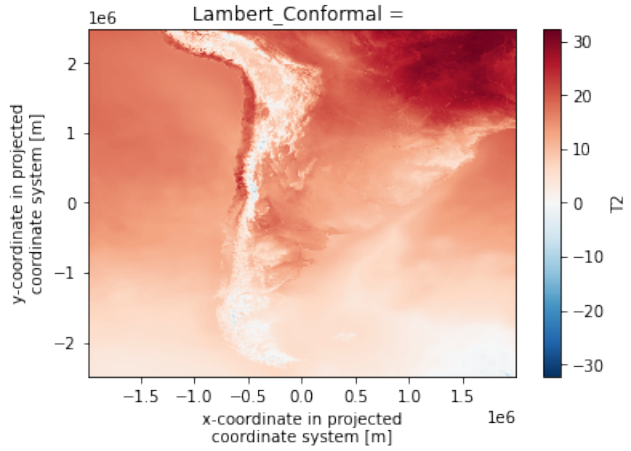
```
fig = plt.figure(figsize=[15, 10])

lst_axes = []
numx = 3
numy = 3

variables = ['T2', 'HR2', 'magViento10']
id_times = ['2023-08-13T'+h+':00:00' for h in ['09','15','21']]

for i in range(0,numx*numy):

    lst_axes.append(fig.add_subplot(numx,numy,i+1))
    ax = lst_axes[i]

    variable = ds[variables[int(i/3)]]
    variable.sel(time=id_times[i%3]).sel(y=100, method='nearest').plot(ax=ax)

plt.tight_layout()
plt.show()
```

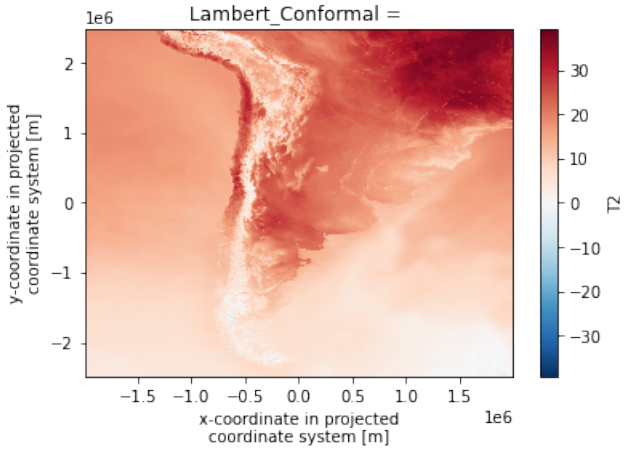


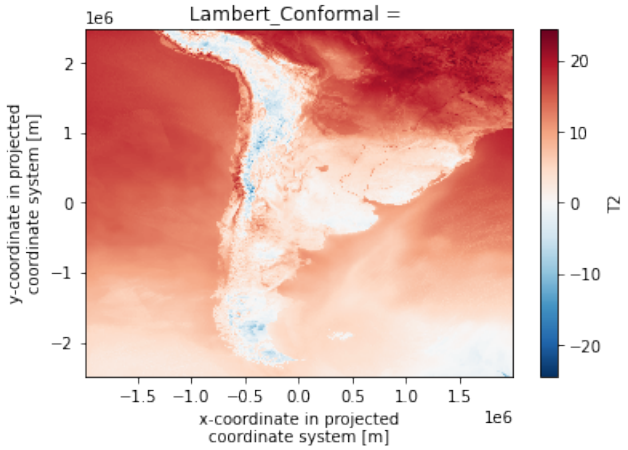
4. Mostrar mapas con la temperatura media, máxima y mínima durante el día 13.

Podemos ejecutar cada estadística individualmente, pintando cada una por separado:

```
ds.T2.mean(dim='time').plot(size=4)
ds.T2.max(dim='time').plot(size=4)
ds.T2.min(dim='time').plot(size=4)
```

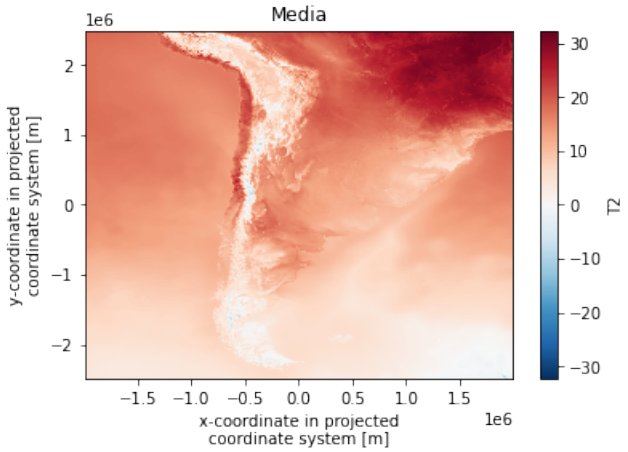


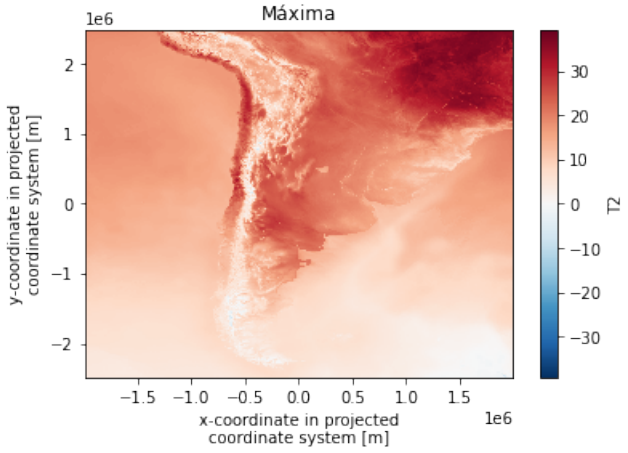


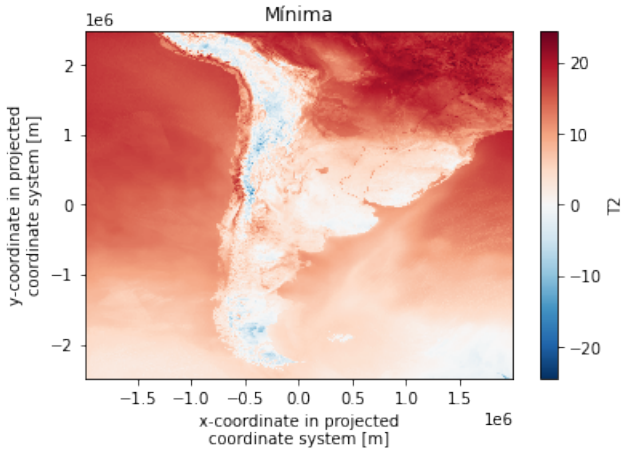


Una alternativa más general (aunque quizás un poco extravagante) sería crear una función general que tenga como argumento la función que queremos pintar:

```
def plot_statistic(name, func):
    '''
    Función que pinta una estadística

    Parameters
    ----------
    name : str
           Título de la gráfica
    func : function
           Función a pintar
    '''

    func(dim='time').plot(size=4)
    plt.title(name)
all_stats = {"Media": ds.T2.mean, "Máxima": ds.T2.max, "Mínima": ds.T2.min}
for st in all_stats:
    plot_statistic(st, all_stats[st])
```







4. Extraer las predicciones de temperatura a 2 metros para la estación “Aeropuerto Melilla G3” (por ejemplo)

Hay un problema para extraer las predicciones: las coordenadas son x e y, los metros desde un determinado punto, no la latitud y la longitud. Afortunadamente en uno de los tutoriales de la página de descarga (https://odp-aws-smn.github.io/documentation_wrf_det/Get_lat_lon_fecha/) se da un ejemplo de cómo transformar las coordenadas x/y a latitud/longitud.

Para ello hay que usar Cartopy, dado que el modelo meteorológico utiliza una proyección distinta, la proyección Lambert, y con Cartopy podemos hacer la transformación. El código (copiado de esa página) es este:

```
import cartopy.crs as ccrs
from datetime import datetime
init_year = 2023
init_month = 8
init_day = 13
init_hour = 0
INIT_DATE = datetime(init_year, init_month, init_day, init_hour)
latitude = -34.78999
longitude = -56.26628
var = 'T2'
# Buscamos la ubicación del punto más cercano a la latitud y longitud solicitada
# We search the closest gridpoint to the selected lat-lon
data_crs = ccrs.LambertConformal(
    central_longitude=ds['Lambert_Conformal'].attrs['longitude_of_central_meridian'],
    central_latitude=ds['Lambert_Conformal'].attrs['latitude_of_projection_origin'],
    standard_parallels=ds['Lambert_Conformal'].attrs['standard_parallel'])
x, y = data_crs.transform_point(longitude, latitude, src_crs=ccrs.PlateCarree())

# Seleccionamos el dato mas cercano a la latitud, longitud y fecha escogida
# We extract the value at the chosen gridpoint

forecast = ds.sel(dict(x = x, y = y), method = 'nearest')[var]
```

forecast es un array de Numpy donde ya tenemos las temperaturas de nuestra estación. Además *forecast.time* nos informa de la hora a la que corresponde cada temperatura:

```
forecast.time
```

```
forecast.values
```

```
array([ 5.207428 ,  4.976166 ,  4.6733093,  5.8909607,  8.144379 ,
       10.019928 , 11.533813 , 12.767242 , 14.006592 , 14.976318 ,
       15.535248 , 15.827026 , 15.614471 , 14.165314 , 13.019592 ,
       12.150909 ], dtype=float32)
```

Transformamos las horas de tipo cadena (string) a datetime y restamos 3 horas (dado que Uruguay tiene el huso horario GMT-3):

```
lead_times = pd.to_datetime(forecast.time, "%Y-%m-%dT%H:%M%s") - pd.Timedelta(hours=3)
lead_times
```

```
DatetimeIndex(['2023-08-13 06:00:00', '2023-08-13 07:00:00',
               '2023-08-13 08:00:00', '2023-08-13 09:00:00',
               '2023-08-13 10:00:00', '2023-08-13 11:00:00',
               '2023-08-13 12:00:00', '2023-08-13 13:00:00',
               '2023-08-13 14:00:00', '2023-08-13 15:00:00',
               '2023-08-13 16:00:00', '2023-08-13 17:00:00',
               '2023-08-13 18:00:00', '2023-08-13 19:00:00',
               '2023-08-13 20:00:00', '2023-08-13 21:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Y emparejamos fechas con temperaturas en un dataframe, para manejarlo más cómodamente:

```
df_fcst = pd.DataFrame({'Fecha': lead_times, 't2m pred': forecast.values})
df_fcst
```

Paso 2: formateo de las observaciones

```
import pandas as pd
import matplotlib.pyplot as plt
import xarray as xr
import cartopy as ccrs
```

Cargamos el fichero Excel en un dataframe de pandas. Notar que d_all es un diccionario de dataframes (uno por cada pestaña del fichero Excel):

```
d_all = pd.read_excel("inumet_20230813.xlsx", sheet_name=None)
type(d_all)
dict
d_all.keys()
dict_keys(['metadatos', '06', '07', '08', '09', '10', '11', '12',
           '13', '14', '15', '16', '17', '18', '19', '20'])
```

d_all.metadatos (la primera pestaña) es un dataframe de pandas con los metadatos de las estaciones:

```
d_all['metadatos'][0:5]
```

Tenemos otros 15 dataframes más, uno por cada hora:

```
d_all['06'][0:6]
```

Añadimos una columna llamada Fecha tomando la hora de cada pestaña:

```
date_file = '2023-08-13'
for hour in [str(i).zfill(2) for i in range(6,21)]:
    d_all[hour]['Fecha'] = pd.Timestamp(date_file + 'T' + hour)
d_all['08']
```

Ahora concatenamos todos los dataframes creando uno más grande, mejor estructurado que antes. Así tenemos la información en un único dataframe, no en un diccionario de muchos más pequeños:

```
df = pd.DataFrame()
for hour in [str(i).zfill(2) for i in range(6,21)]:
    df = pd.concat([df, dfall[hour]])
```

Unimos el dataframe con los metadatos:

```
df = pd.merge(df, dfall['metadatos'], left_on='Estación', right_on='Nombre')
# Excel los lee como enteros en este caso (problema con el punto como separador)
df.Latitud = df.Latitud/100000.0
df.Longitud = df.Longitud/100000.0
df[10:20]
df.dtypes
```

```
Estación                                    object
Viento [Dir./(Km/h)]                        object
Temperatura del Aire [°C]                  float64
Temperatura de Punto de Rocío [°C]         float64
Humedad relativa [%]                       float64
Precipitación Acumulada Horaria [mm]       float64
Presión Atmosférica a Nivel del Mar [hPa]   object
Fecha                                 datetime64[ns]
Nombre                                      object
Depto.                                      object
Latitud                                    float64
Longitud                                   float64
Altitud                                     object
dtype: object
```

Paso 3: verificación de las predicciones

1. Crear un gráfico que muestre la evolución de la temperatura a 2 metros observada y predicha para la estación "Aeropuerto Melilla G3" (por ejemplo) durante el día 13.

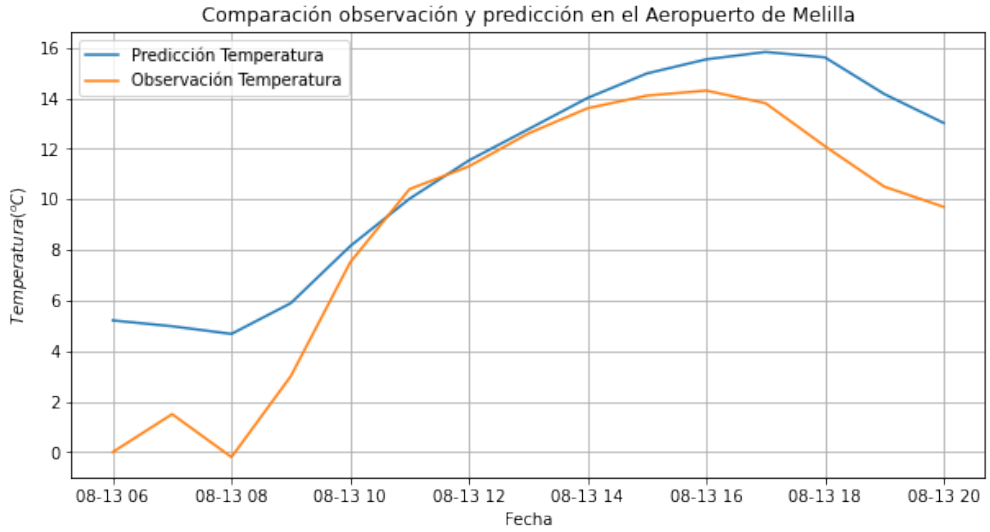
```
df_obs = df[df['Estación']=='Aeropuerto Melilla G3']
```

Nos basta con unir los dataframes de predicciones y observaciones obtenidos antes:

```
df_verif = pd.merge(df_obs, df_fcst, on="Fecha")
df_verif
```

Ya podemos dibujar la gráfica:

```
fig, ax = plt.subplots(figsize=(10,5))
ax.set_ylabel('$Temperatura (^{o}C)$')
ax.set_xlabel('Fecha')
ax.set_title("Comparación observación y predicción en el Aeropuerto de Melilla")
ax.plot(df_verif['Fecha'],
        df_verif['t2m pred'],
        label='Predicción Temperatura')
ax.plot(df_verif['Fecha'],
        df_verif['Temperatura del Aire [°C]'],
        label='Observación Temperatura')
ax.grid()
ax.legend()
```



Observamos que el modelo infraestima la temperatura máxima y mínima. Esto es bastante habitual en un modelo meteorológico: notar que cada punto de un modelo representa un área bastante grande, mientras que una observación es puntual. Suele haber problemas de representatividad. Esto se puede corregir en gran medida con un programa de postproceso.

2. Calcular el sesgo y el error cuadrático medio de la predicción para esta estación.

```
df['error'] = df_verif['t2m pred'] - df_verif['Temperatura del Aire [°C]']
sesgo = df.error.mean()
rmse = np.sqrt((df.error**2).mean())
print("sesgo=%f"%sesgo)
print("rmse=%f"%rmse)
```

```
sesgo=2.143853
rmse=2.772770
```

Efectivamente el sesgo es de 2 grados (como ya intuíamos al ver la gráfica). De todas formas esto ha ocurrido un día concreto: para sacar buenas conclusiones habría que verificar un periodo mucho más largo, y para más estaciones.

Capítulo 13

Agradecimientos

Queremos expresar nuestro agradecimiento por los datos de observaciones tomados de las páginas web del Servicio Meteorológico Nacional de Argentina, CENAOS (Centro de Estudios Atmosféricos, Oceanográficos y Sísmicos de Honduras), INUMET (Servicio Meteorológico de Uruguay), ONAMET (Oficina Nacional Meteorológica de la República Dominicana),y del Servicio Meteorológico Nacional de México, que se han usado en estas prácticas. También agradecemos los datos de predicciones del modelo WRF del Servicio Meteorológico Nacional de Argentina y los datos de reanálisis tomados del servicio C3S de Copernicus.

Asimismo queremos reconocer la ayuda de nuestros compañeros de AEMET, especialmente de Marcos Gómez y Ernesto Barrera. Y por supuesto nuestro agradecimiento a XKCD (https://xkcd.com) y sus cómics.