MÓDULO 1: HERRAMIENTAS BIG DATA

TEMA 4.2. HERRAMIENTAS DE ANALISIS: PROGRAMACIÓN EN PYTHON

Ferran Carrascosa Mallafrè

Licenciado en Matemáticas por la Universidad de Barcelona. Data Scientist

**ÍNDICE**

[PROGRAMACIÓN EN PYTHON 2](#__RefHeading___Toc96682_40828882)

[INTRODUCCIÓN 2](#__RefHeading___Toc96684_40828882)

[ACTIVIDAD GUIADA 2 3](#__RefHeading___Toc96686_40828882)

[ELEMENTOS BÁSICOS DE PYTHON 4](#__RefHeading___Toc96688_40828882)

[GRÁFICOS CON MATPLOTLIB 9](#__RefHeading___Toc96690_40828882)

[CONCEPTOS BASICOS EN MATPLOTLIB 10](#__RefHeading___Toc96692_40828882)

[LIBRERÍA SEABORN 17](#__RefHeading___Toc96694_40828882)

[COLECCIONES DE OBJETOS 21](#__RefHeading___Toc96696_40828882)

[DEFINCIÓN DE LOS OBJETOS 21](#__RefHeading___Toc96698_40828882)

[LISTAS 22](#__RefHeading___Toc96700_40828882)

[TUPLAS 24](#__RefHeading___Toc96702_40828882)

[DICCIONARIOS 25](#__RefHeading___Toc96704_40828882)

[NUMPY 26](#__RefHeading___Toc96706_40828882)

[FILTROS EN NUMPY 30](#__RefHeading___Toc96708_40828882)

[MODIFICAR ARRAYS 33](#__RefHeading___Toc96710_40828882)

[PANDAS 37](#__RefHeading___Toc96712_40828882)

[PANDAS SERIES 38](#__RefHeading___Toc96714_40828882)

[OBJETOS PANDAS DATAFRAME 39](#__RefHeading___Toc96716_40828882)

[FILTROS EN DATAFRAMES 42](#__RefHeading___Toc96718_40828882)

[CONTROL DE FLUJO 48](#__RefHeading___Toc96720_40828882)

[FUNCIONES 49](#__RefHeading___Toc96722_40828882)

[CONDICIONALES 49](#__RefHeading___Toc96724_40828882)

[BUCLES 50](#__RefHeading___Toc96726_40828882)

[GESTIÓN DE DATOS 51](#__RefHeading___Toc96728_40828882)

[IMPORTAR Y EXPORTAR DATOS 52](#__RefHeading___Toc96730_40828882)

[CRUCE ENTRE TABLAS 54](#__RefHeading___Toc96732_40828882)

[RESUMENES DE AGREGADOS 60](#__RefHeading___Toc96734_40828882)

[AGREGADOS POR SUBGRUPOS 66](#__RefHeading___Toc96736_40828882)

[TABLAS PIVOTE 70](#__RefHeading___Toc96738_40828882)

[DATETIME: GESTIÓN DE FECHAS Y HORARIOS 71](#__RefHeading___Toc96740_40828882)

[ANEXO: README DE PYTHON 74](#__RefHeading___Toc96742_40828882)

[PREPARACIÓN DEL ENTORNO COLAB 74](#__RefHeading___Toc96744_40828882)

[PREPARACIÓN ENTORNO LOCAL-JUPYTER (OPCIONAL) 74](#__RefHeading___Toc96746_40828882)

[IMPORTAR DATOS DE STARWARS SWAPI 75](#__RefHeading___Toc96748_40828882)

# PROGRAMACIÓN EN PYTHON

## INTRODUCCIÓN

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py.ipynb)

Python es una herramienta para la programación de propósito general.

En los últimos años, se ha convertido en uno de los lenguajes de referencia para el *Data Science*. El motivo es que ha sabido rodearse de un gran ecosistema como [SciPy](https://www.scipy.org/) con librerías (numpy, pandas, scipy…) y herramientas (Jupyter, Spyder,…) orientadas al análisis y a la programación matemática.

Los desarrolladores de Python buscan hacer un lenguaje vivo y atractivo para el programador. Prueba de ello es que su nombre es un tributo a la compañia de humor británica *Monthy Python*.

El núcleo de su filosofía de programación se resume en el “*Zen de Python*”, formado por 20 aforismos escritos por Tim Peters, 19 de los cuales se pueden leer más abajo importando this. El veinte, dijo Tim Peters que lo diría el creador de Python, Guido van Rossum, pero parece que aún no se ha pronunciado…

import this

## The Zen of Python, by Tim Peters  
##   
## Beautiful is better than ugly.  
## Explicit is better than implicit.  
## Simple is better than complex.  
## Complex is better than complicated.  
## Flat is better than nested.  
## Sparse is better than dense.  
## Readability counts.  
## Special cases aren't special enough to break the rules.  
## Although practicality beats purity.  
## Errors should never pass silently.  
## Unless explicitly silenced.  
## In the face of ambiguity, refuse the temptation to guess.  
## There should be one-- and preferably only one --obvious way to do it.  
## Although that way may not be obvious at first unless you're Dutch.  
## Now is better than never.  
## Although never is often better than \*right\* now.  
## If the implementation is hard to explain, it's a bad idea.  
## If the implementation is easy to explain, it may be a good idea.  
## Namespaces are one honking great idea -- let's do more of those!

Ser *Pythoniano* o *Pythónico* significa utilizar correctamente el código, es decir, programar con un lenguaje simple y fácil de leer.

## ACTIVIDAD GUIADA 2

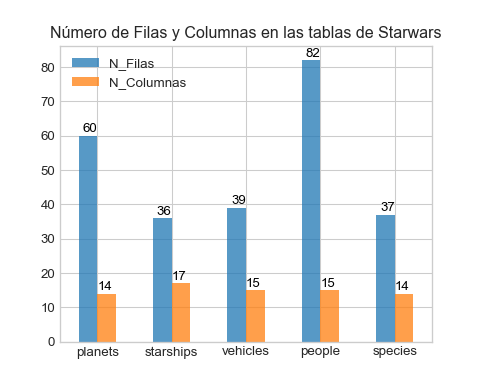
La actividad guiada, que te proponemos, en consonancia con los Pythonianos, es que te diviertas analizando datos relacionados con la saga de Star Wars: Planetas, Naves, Vehículos, Personajes y Especies.

Para ello, contamos con los datos de [SWAPI](https://swapi.dev/), acrónimo de STAR WARS API, que nos da acceso libre a una colección de datos de la saga.

Estos datos se han descargado y preparado expresamente para este curso. Puedes encontrar el código utilizado en el [Anexo: README de Python](modulo1_tema4_Py_60_anexo.ipynb), capítulo “IMPORTAR DATOS DE STARWARS SWAPI”.

Para cargar los datos y visualizar su contenido.

import pandas as pd  
import matplotlib.pyplot as plt  
#import seaborn as sns  
plt.style.use('seaborn-whitegrid')  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
entidades\_count = {k: v.shape for k,v in entidades\_df.items()}  
entidades\_count\_df = pd.DataFrame.from\_dict(entidades\_count, orient='index', columns=['N\_Filas', "N\_Columnas"])  
  
g = entidades\_count\_df.plot.bar(alpha=0.75, rot=0)  
plt.title("Número de Filas y Columnas en las tablas de Starwars")  
#g.title = 'Entidad de Starwars'  
for i, (k, row) in enumerate(entidades\_count\_df.iterrows()):  
 g.text(i-0.1, row['N\_Filas']+1, row['N\_Filas'], color='black', ha="center")  
 g.text(i+0.1, row['N\_Columnas']+1, row['N\_Columnas'], color='black', ha="center")  
plt.show()



Vemos que los datos están formados por 5 conjuntos. Por ejemplo, people, contienen 82 personajes descritos mediante 16 variables.

Una muestra de los datos de los personajes.

entidades\_df['people'].head()

## height ... url  
## name ...   
## Luke Skywalker 172.0 ... http://swapi.dev/api/people/1/  
## C-3PO 167.0 ... http://swapi.dev/api/people/2/  
## R2-D2 96.0 ... http://swapi.dev/api/people/3/  
## Darth Vader 202.0 ... http://swapi.dev/api/people/4/  
## Leia Organa 150.0 ... http://swapi.dev/api/people/5/  
##   
## [5 rows x 15 columns]

Observa que hay variables numéricas como la altura, peso, edad (en años ABY Antes de la Batalla de Yavin). Otras categóricas, como el género del personaje. Incluso variables en formato de lista, como las películas en las que salió el personaje y los vehículos y naves que condujo…

"Yoda habló de otra."   
"La otra de quién habló es tu hermana gemela.»   
― Luke Skywalker y Obi-Wan Kenobi

Te acordabas que Luke y Leia eran gemelos? ¿Sabías que su padre (Anakin, posteriormente Darth Vader tenía 22 años cuando los tuvo?

Podrás analizar todo esto y mucho más, en los datos y así convertirte en el auténtico Jedi que sabemos que llevas dentro.

Consulta la [Documentación de SWAPI](https://swapi.dev/documentation) sobre sus tablas y campos.

*Y que la fuerza te acompañe…*

## ELEMENTOS BÁSICOS DE PYTHON

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_10_elem_bas.ipynb)

Aunque se presuponen unos conocimientos iniciales de Python, a continuación, se hace un repaso de Python, Anaconda y notebooks de Jupyter-Colab. El objetivo es reforzar aquellos elementos más orientados al análisis de datos.

#### INTERFAZ DE USUARIO

Python se puede ejecutar de forma interactiva des de consola de texto. Para abrir la consola, ejecutar el comando Python desde un terminal Windows o Linux. El temario utiliza notebooks ejecutados des de Jupyter y/o Colab.

Prueba a ejecutar la siguiente línea directamente en este notebook.

6 / (4 - 1.5)

## 2.4

**Recuerda**: Para editar una celda del notebook, aprieta Enter. Para ejecutarla, Ctrl+Enter (sólo ejecuta), o bien, Mayús+Enter (ejecuta y avanza celda).

#### FICHEROS \*.py

Una forma habitual de programar en Python es mediante ficheros de texto con extensión .py. Existen múltiples entornos para estos ficheros: Spyder (parecido a RStudio), PyCharm, Visual Studio Code o cualquier editor de texto como Notepad++.

Este tipo de ficheros se utilizan habitualmente para generar librerías de funciones y ejecutar desarrollo productivo de aplicaciones en Python.

#### NOTEBOOKS JUPYTER (\*.ipynb)

Los notebooks, como éste que estás leyendo, son la solución de la comunidad de SciPy para la programación interactiva. El proyecto nació con el nombre [IPython](https://ipython.org/) (o Python interactivo) y actualmente se orquestra en el paraguas de la plataforma [Jupyter](https://jupyter.org/). Jupyter permite ejecutar de forma interactiva docenas de lenguajes como Python, R…

Los notebooks, al igual que Rmardown, permiten combinar texto en formato Markdown (lenguaje de marcas ligero), con código con resultados en forma de texto o gráficos. Esta forma de funcionar resulta óptima para el análisis de datos y la formación.

Su uso se ha extendido de la mano del proyecto [Anaconda](https://www.anaconda.com/), la mayor plataforma de gestión de librerías open source enfocadas a la computación científica.

Para iniciar el entorno Jupyter sigue la guía del [Anexo: README de Python](modulo1_tema4_Py_60_anexo.ipynb) apartado “PREPARACIÓN ENTORNO LOCAL-JUPYTER (OPCIONAL)”.

##### COLABORATORY o COLAB

Actualmente la comunidad de Python cuenta con la plataforma [Colaboratory o Colab](https://colab.research.google.com/) que es un entorno libre para ejecutar notebooks Jupyter y almacenarlos en Google Drive. Este entorno sólo permite abrir ficheros .ipynb (no .py), pero facilita en gran medida su uso para desarrollar formación.

Puedes abrir los notebooks de este tema con Colab clicando en: [| Abre en Colab |](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_10_elem_bas.ipynb)

En la parte superior de los notebooks, encontrarás un script que inicializa el entorno Colab con las versiones de los paquetes de Python correctas y descarga el repositorio Github con el material del curso.

##### ACTIVIDAD GUIADA 2.1.

Crea tu primer notebook. En Jupyter/Colab, accede al menú File > New Notebook.

En Jupyter, selecciona el Kernel mbdds\_rpy20 que has creado como se indica en el [Anexo: README de Python](modulo1_tema4_Py_60_anexo.ipynb).

Inserta el siguiente código Python y ejecútalo:

print("Hello Wold!")

Guarda el documento con el nombre 21\_actividad.

##### EDITAR NOTEBOOKS JUPYTER

Un notebook está compuesto por un conjunto de casillas que pueden ser de distinto tipo:

* *Code*: Código Python que se puede ejecutar interactivamente y puede tener un salida o output.
* *Markdown*: Inserta texto con este formato. No tiene output.

Una casilla, puede tener dos modos:

* Edición: Aprieta Enter o doble clic. Puedes modificar su contenido Python o Markdown
* Visión: Tecla Esc o clic barra izquierda (en Jupyter). Selecciona una o varias casillas con el ratón o con las teclas Mayús + Cursos.

Para mostrar una casilla Markdown en formato impresión (sin marcas), ejecútala del mismo modo que si fuera código: - Sólo ejecutar: Ctrl+Enter. - Ejecutar y avanzar casilla: Botón Run, o bien, Mayús+Enter.

###### ATAJOS DE TECLADO

Puedes gestionar las celdas des del teclado en el modo de Visión (**teclaEsc**):

* Cambiar tipo celda:
  + Modo Markdown: Tecla M, Ctr+M+M (Colab).
  + Modo Code: Tecla Y, Ctr+M+Y (Colab).
* Insertar celdas:
  + Antes: Tecla A, Ctr+M+A (Colab).
  + Después: Tecla B, Ctr+M+B (Colab).
* Copiar, pegar, desplazar arriba, abajo…

###### MARKDOWN

* **Parágrafo de texto**: Deja una línea en blanco (Enter) antes y después de un parágrafo de texto.
* **Encabezados**: Utiliza # seguido de espacio en blanco para los títulos de las secciones:

# TITULO NIVEL 1  
## TITULO NIVEL 2  
...

Deja también una línea en blanco (Enter) antes y después del título.

* **Emphasis**: Negrita con \*\*string\*\* . Cursiva con \*string\*.
* **Código látex**: $x^2$.
* **Fuente ancho fijo**: `texto` (usa tilde abierta o con orientación izquierda).
* **Bloque de código (sin que se ejecute)**: Tres abiertas, código (nueva línea), tres tildes abiertas. Ejemplo:

```

Soy codigo1  
Soy código2

```

* **Línea en blanco**: Usa una líena nueva con dos espacios o <br>.
* **Lista de puntos**: Al inicio de línea, usa guion - seguido de espacio y texto. Para una sublista de puntos pon tres espacios o tabulado y guion. Puedes usar también asterisco.
* **Lista numerada**: Al inicio, pon 1. seguido de espacio y texto. No es necesario cambiar el 1. en cada nueva línea.
* **Link externo**: [texto enlace](http://url)
* **Link interno**: Crea un ID con html <a id="seccion\_ID"></a> antes de la sección. Ahora, para enlazar con la sección, utiliza [titulo sección](#seccion\_ID).
* **Imagen**: Arrastrar y soltar en la celda markdown. También con ![texto alternativo](url imagen o ruta del a imagen), o también, <img src="url imagen o ruta del a imagen" alt="texto alternativo" title="texto descriptivo" /> .
* **Sangrado (Indented)**: Al incio de línea, usa símbolo > seguido de espacio y texto. Ejemplo:

Texto sangrado

* **Línea horizontal**: Tres asteriscos al inicio de línea, \*\*\* .
* **Colores**: Usa <font color=blue|red|green|pink|yellow>Texto</font> .

###### AYUDA DES DEL NOTEBOOK

En Python se pide ayuda de un comando con:

help(len)

## Help on built-in function len in module builtins:  
##   
## len(obj, /)  
## Return the number of items in a container.

En Jupyter, ejecuta ?comando para que se abra un menú contextual de ayuda.

?len

También con el tabulador tea ayuda a autocompletar código, ver opciones, parámetros…

###### COMANDOS MÁGICOS

Los comandos mágicos, aumentan las capacidades interactivas de los notebooks. Son de uso común:

Ejecutar script .py

%run holaMundo.py

## Hello World!

Historial de comandos ejecutados.

# vemos los primerso 3:  
%history 1-3

## if 'google.colab' in str(get\_ipython()):  
## !git clone https://github.com/griu/mbdds\_fc20.git /content/mbdds\_fc20  
## !git -C /content/mbdds\_fc20 pull  
## %cd /content/mbdds\_fc20/Python  
## !python -m pip install -r requirementsColab.txt  
6 / (4 - 1.5)  
## help(len)

Cambiar de directorio de trabajo.

%cd nueva\_ruta

Tiempo de ejecución de una línea.

%time L = [n \*\* 2 for n in range(1000)]

## CPU times: user 566 µs, sys: 43 µs, total: 609 µs  
## Wall time: 616 µs

De toda una celda

%%time  
L1 = [n \*\* 2 for n in range(1000)]  
L2 = [n \*\* 2 for n in range(1000)]  
L3 = [n \*\* 2 for n in range(1000)]

## CPU times: user 1.99 ms, sys: 0 ns, total: 1.99 ms  
## Wall time: 2 ms

###### COMANDOS SHELL

Los comandos de sistema se pueden ejecutar con !comando

!dir

## holaMundo.py modulo1\_tema4\_Py\_40\_contr\_flujo.ipynb  
## modulo1\_tema4\_Py\_00\_indice.ipynb modulo1\_tema4\_Py\_50\_gest\_dat.ipynb  
## modulo1\_tema4\_Py\_10\_elem\_bas.ipynb modulo1\_tema4\_Py\_60\_anexo.ipynb  
## modulo1\_tema4\_Py\_20\_matplotlib.ipynb modulo1\_tema4\_Py.ipynb  
## modulo1\_tema4\_Py\_30\_colec\_obj.ipynb README.md  
## modulo1\_tema4\_Py\_31\_numpy.ipynb requirementsColab.txt  
## modulo1\_tema4\_Py\_32\_pandas.ipynb www

#### FUNCIONES

Para definir la función valor absoluto:

def valor\_absoluto(num):  
 """devuelve el valor absoluto"""  
  
 if num >= 0:  
 return num  
 else:  
 return -num

* Se inicia la definción con def, seguido del nombre de la función
* A continuación los parámetros entre paréntesis y :
  + Admite definir valores por defecto
* El texto sirve como ayuda cuando se llama a ?valor\_absoluto
* El cuerpo del código con sangrado
* Puede devolver un objeto añadiendo return objecto
* Si se quiere devolver valores por pantalla hay que usar print

Para llamar a la función.

valor\_absoluto(-6)

## 6

#### PAQUETES

El indexador de paquetes de Python es **PyPI**. igual que **CRAN** en R. Actualmente indexa más de 235.000 librearías.

Para instalarlos hay distintos métodos. El más general es mediante pip desde Shell.

pip install nombre\_paquete

**Sabías que**: Anaconda tiene su propio gestor de paquetes llamado conda. Funciona de forma similar a pip. Además conda, aporta la capacidad de gestionar *Environments*, que a su vez, permiten fijar un conjunto de versiones de paquetes de Python. Los *Environments* facilitan reproducir en otras máquinas todo el entorno de ejecución y la replicabilidad de los desarrollos. Revisa un ejemplo en [Anexo: README de Python](modulo1_tema4_Py_60_anexo.ipynb).

Desde Jupyter/Colab también se pueden instalar paquetes con !pip y !conda (fíjate que ahora llevan ! delante.

!pip install nombre\_paquete

Una vez instalados, en cada nueva sesión de Jupyter, hay que importarlos para que sean accesibles.

import nombre\_paquete

## GRÁFICOS CON MATPLOTLIB

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_20_matplotlib.ipynb)

La librería gráfica [Matplotlib](https://matplotlib.org/), concebida por John Hunter en 2002, fué construida sobre objetos numpy (arrays N-dimensionales) y posteriormente adaptada a objetos Pandas (matrices con vectores datos de distintos tipos).

Actualmente, se apoya en otras librerías, más simples y de aspecto gráfico modernizado com Seaborn y los mismos Pandas.

Los objetos numpy y pandas se expondrán ampliamente en el siguiente apartado, no obstante, dado que el curso presupone ciertos conocimientos de Python, se muestran ahora los gráficos para poder utilizarlos en los siguientes apartados como herramienta.

Para importar la librería, habitualmente se importa el submódulo pyplot.

import matplotlib.pyplot as plt

##### ACTIVIDAD GUIADA 2.2

Consiste en conocer mejor a las especies mediante gráficos. En concreto se puede analizar su altura, años de vida, clase de especie (mamífero, reptil,…) y en cuantas películas ha salido esa especie.

«¡No puedes llevar a Su Alteza Real allí! Los Hutts son gangsters ...»

―Quarsh Panaka.

El primer paso consiste en cargar los datos de las especies y extraer las variables.

import numpy as np  
import pandas as pd  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
entidades\_df['species']["num\_peliculas"] = entidades\_df['species'].films.apply(lambda x: len(x)) # numero de peliculas en las que aparece  
species\_df = entidades\_df['species'][["classification","average\_height","average\_lifespan","num\_peliculas"]].dropna()  
  
  
nombre = species\_df.index.values  
clasificacion = species\_df.classification.values  
altura\_media = species\_df.average\_height.values  
vida\_media = species\_df.average\_lifespan.values  
num\_peliculas = species\_df.num\_peliculas.values  
  
species\_df.head()

## classification average\_height average\_lifespan num\_peliculas  
## name   
## Human mammal 180.0 120.0 6  
## Wookie mammal 210.0 400.0 4  
## Hutt gastropod 300.0 1000.0 2  
## Yoda's species mammal 66.0 900.0 5  
## Toydarian mammal 120.0 91.0 2

Observa como cada fila representa una especie que viene caracterizada por 5 vectores numpy. Para saber más de los numpy ve al capítulo [Colecciones de objetos: numpy](modulo1_tema4_Py_31_numpy.ipynb):

* **nombre**: Nombre de la especie.
* **clasificacion**: Clasificación de la especie.
* **altura\_media**: Altura media (en cm).
* **vida\_media**: Vida media en años.
* **num\_peliculas**: Número de películas en las que aparece esa especie.

### CONCEPTOS BASICOS EN MATPLOTLIB

Se ajusta el estilo que regula el aspecto general del gráfico al seaborn-whitegrid.

plt.style.use('seaborn-whitegrid');

Para definir un gráfico librería pyplot cuenta con muchos métodos.

Por ejemplo, para construir un gráfico de dispersión se utiliza la función plt.scatter().

Veamos la dispersión entre la altura media de cada especie.

altura\_media # vector x

## array([180., 210., 300., 66., 120., 80., 180., 240., 180., 180., 178.,  
## 220., 190., 180., 170., 190.])

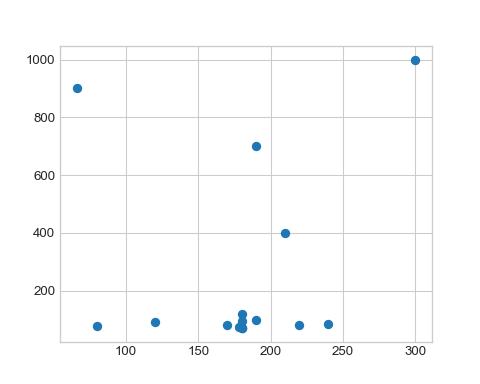
Y el tiempo de vida medio de cada especie.

vida\_media # vector y

## array([ 120., 400., 1000., 900., 91., 79., 70., 86., 70.,  
## 70., 75., 80., 100., 94., 80., 700.])

Se especifica tanto el vector x como el y.

plt.scatter(x=altura\_media,y=vida\_media)  
plt.show();



Observa que matplotlib ha añadido un conjunto de elementos como son los ejes con sus marcas o ticks. Estos elemntos se configuran en el estilo.

Observa también que se ha llamado a la función plt.show(). Esta función ordena la salida del gráfico por pantalla. Es habitual su uso en scripts de python con extensión .py.

En notebooks se acostumbra obviar la llamda a plt.show(), configurando su salida en pantalla, de forma automática, en cada celda. Para ello, se incluye el comando mágico %matplotlib inline, al inicio del notebook.

%matplotlib inline

Se puede guardar el gráfico con.

plt.scatter(x=altura\_media,y=vida\_media)  
plt.savefig('www/especies.png');

**Recuerda**: Puedes abrir un fichero .png local con la función Image().

from IPython.display import Image  
Image('www/especies.png');

#### DOS INTERFACES DE GRÁFICOS

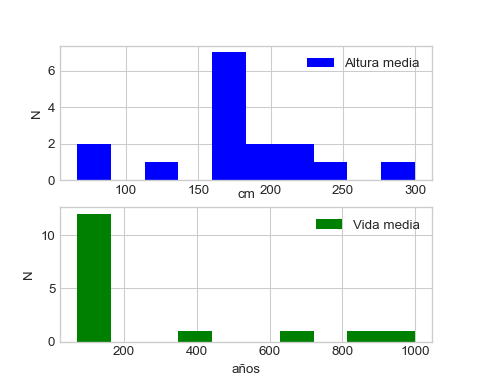
La definición de un gráfico se puede hacer mediante dos interfaces.

##### EL ESTILO DE MATLAB

Pytplot, en un inicio, se diseñó a imagen y semejanza de la librería gráfica de MATLAB.

plt.figure(); # crea la figura del plot  
  
# crea el primer panel y lo configura lo ejes.  
plt.subplot(2, 1, 1) # (filas, columnas, número de panel)  
plt.hist(altura\_media, color="blue", label="Altura media")

plt.xlabel("cm",labelpad=-6) # labelpad desplaça el titulo arriba  
plt.ylabel("N")  
plt.legend();  
  
# crea el segundo panel y lo configura lo ejes  
plt.subplot(2, 1, 2)  
plt.hist(vida\_media, color="green", label="Vida media");  
plt.xlabel("años")  
plt.ylabel("N")  
plt.legend();  
plt.show();

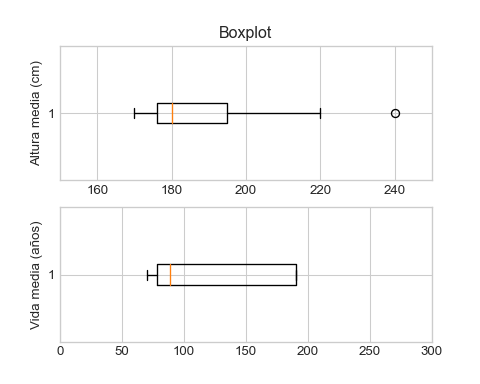


Observa que plt va añadiendo elementos al gráfico actual. Una vez se pasa al segundo panel, volver al primer panel se vuelve un trabajo complejo. Para esto se definió la siguiente interfaz.

##### INTERFAZ ORIENTADA A OBJETOS

Se separa en figura y ejes. Lo que permite hacer referencia a cualquier gráfico del panel.

# Se crea la parrilla de plots  
# ax será un vector de 2 objetos Axes  
fig, ax = plt.subplots(2);  
  
  
# Llama el método plot() sobre el objeto apropiado  
ax[0].boxplot(altura\_media, vert=False);  
ax[1].boxplot(vida\_media, vert=False);  
  
ax[0].set(xlim=(150, 250),  
 ylabel='Altura media (cm)',  
 title='Boxplot');  
ax[1].set(xlim=(0, 300),  
 ylabel='Vida media (años)');  
plt.show();

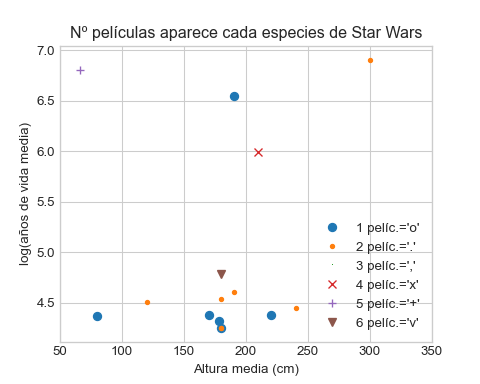


#### MARCAS, COLORES Y TAMAÑOS

En Matplotlib se pueden personalizar estos elementos, no obstante, resulta algo tedioso. El package Seaborn aporta aquí algunas facilidades, como se verá.

Veamos un ejemplo de esto.

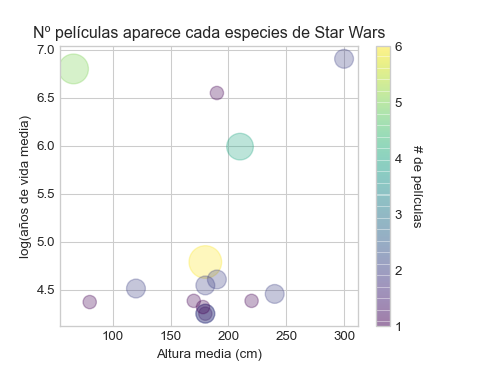
marcas = ['o', '.', ',', 'x', '+', 'v']  
for i in range(6):  
 plt.plot(altura\_media[num\_peliculas==i+1], np.log(vida\_media[num\_peliculas==i+1])  
 , marcas[i]  
 , label= str(i+1) + " pelíc.='{0}'".format(marcas[i]))  
plt.legend(numpoints=1, loc="lower right")  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("log(años de vida media)")  
plt.xlim(50, 350);  
plt.show();



Observa como en cada iteración, se etiqueta el plot con un label. De esta forma se van generando elementos que posteriormente se incluirán en la leyenda. El color, en este caso va cambiando en cada iteración por defecto.

Ahora un ejemplo donde se modifica la paleta de colores viridis y también el tamaño de los puntos. Se incluye una leyenda con la escala continua de colores.

plt.scatter(altura\_media, np.log(vida\_media)  
 , c=num\_peliculas, s=100\*num\_peliculas, alpha=0.3,  
 cmap='viridis');  
  
# leyenda del color  
cbar = plt.colorbar(); # Muestra la paleta  
cbar.ax.get\_yaxis().labelpad = 15 # da espacio al título  
cbar.ax.set\_ylabel('# de películas', rotation=270) # añade título  
  
plt.gcf().subplots\_adjust(bottom=0.15)  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("log(años de vida media)");  
plt.show();



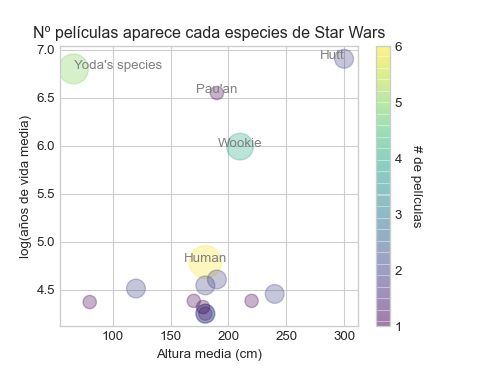
Otras paletas interesantes son: 'jet', 'RdBu' y 'cubehelix'.

#### TEXTOS

Para añadir textos a los gráficos, hay dos métodos.

El primero es plt.text() y está orientado simplemente a añadir texto.

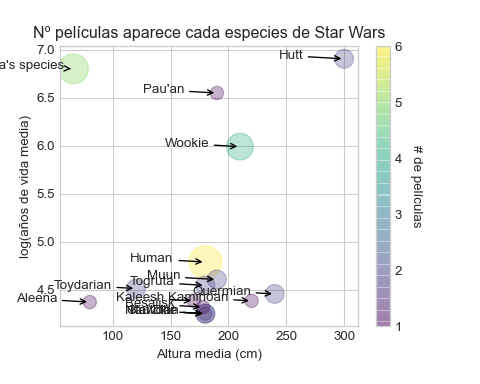
# Recuperamos el gráfico de especies  
plt.scatter(altura\_media, np.log(vida\_media)  
 , c=num\_peliculas, s=100\*num\_peliculas, alpha=0.3,  
 cmap='viridis')  
  
# leyenda del color  
cbar = plt.colorbar(); # Muestra la paleta  
cbar.ax.get\_yaxis().labelpad = 15 # da espacio al título  
cbar.ax.set\_ylabel('# de películas', rotation=270) # añade título  
  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("log(años de vida media)");  
  
# Add labels to the plot  
style = dict(size=10, color='grey')  
  
  
plt.text(300, np.log(1000), "Hutt", \*\*style, ha='right')  
plt.text(66, np.log(900), "Yoda's species", \*\*style, ha='left')  
plt.text(190, np.log(700), "Pau'an", \*\*style, ha='center')  
plt.text(210, np.log(400), "Wookie", \*\*style, ha='center')  
plt.text(180, np.log(120), "Human", \*\*style, ha='center');  
plt.show();



Es relevante observar que hay que añadir los textos de uno en uno, o bien, usando un loop

El siguiente método .annotate() permite personalizar la ubicación y flechas.

# Recuperamos el gráfico de especies  
plt.scatter(altura\_media, np.log(vida\_media)  
 , c=num\_peliculas, s=100\*num\_peliculas, alpha=0.3,  
 cmap='viridis')  
   
# leyenda del color  
cbar = plt.colorbar(); # Muestra la paleta  
cbar.ax.get\_yaxis().labelpad = 15 # da espacio al título  
cbar.ax.set\_ylabel('# de películas', rotation=270) # añade título  
  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("log(años de vida media)");  
  
# Add labels to the plot  
style = dict(size=10, color='green')  
  
for x1,y1,n1 in zip(altura\_media, np.log(vida\_media),nombre):  
 plt.annotate(n1,(x1, y1)  
 , arrowprops=dict(arrowstyle="->")  
 , textcoords="offset points" # desplaza el punto  
 , xytext=(-40,0) # distancia desplazamiento  
 , ha='center') # centrado  
plt.show();



No es muy estético, pero puede ser útil en determinados casos.

### LIBRERÍA SEABORN

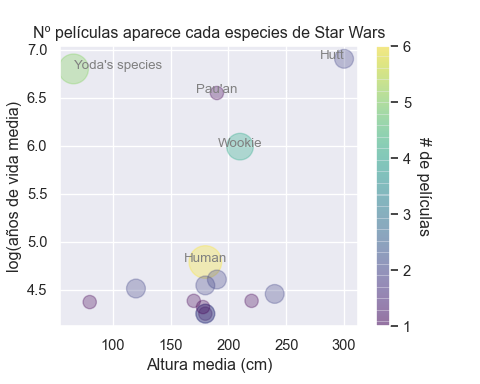
Automatiza la generación de gráficos con data frames aprovechando la asignación de nombres a sus filas y columnas.

Se carga la librería y configura el estilo por defecto.

import seaborn as sns  
sns.set()

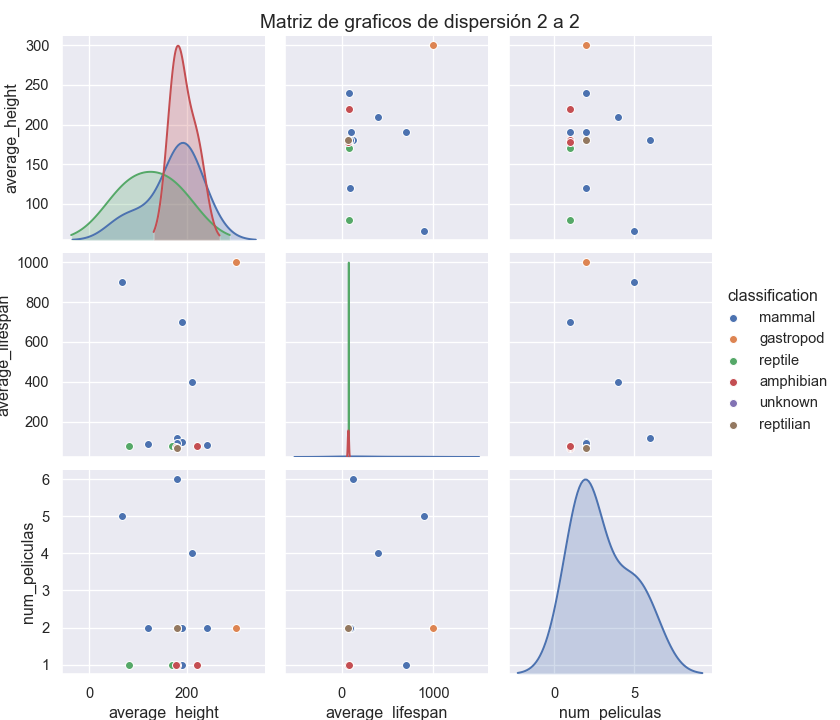
Vemos qué gráfico anterior cambia de estilo.

# Recuperamos el gráfico de especies  
plt.scatter(altura\_media, np.log(vida\_media)  
 , c=num\_peliculas, s=100\*num\_peliculas, alpha=0.3,  
 cmap='viridis')  
  
# leyenda del color  
cbar = plt.colorbar(); # Muestra la paleta  
cbar.ax.get\_yaxis().labelpad = 15 # da espacio al título  
cbar.ax.set\_ylabel('# de películas', rotation=270) # añade título  
  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("log(años de vida media)");  
  
# Add labels to the plot  
style = dict(size=10, color="grey")  
  
  
plt.text(300, np.log(1000), "Hutt", \*\*style, ha='right')  
plt.text(66, np.log(900), "Yoda's species", \*\*style, ha='left')  
plt.text(190, np.log(700), "Pau'an", \*\*style, ha='center')  
plt.text(210, np.log(400), "Wookie", \*\*style, ha='center')  
plt.text(180, np.log(120), "Human", \*\*style, ha='center');  
plt.show();



Una primera muestra de las posibilidades de Seaborn, para el análsis exploratorio, es el gráfico de dispersión 2 a 2.

import warnings  
warnings.filterwarnings('ignore') # filtramos warnings para evitar problemas con especies de un individuo.  
  
# Especies de mas de un individuo  
sns.pairplot(species\_df, hue="classification", height=2.5);  
  
plt.gcf().subplots\_adjust(top=0.95)  
plt.suptitle('Matriz de graficos de dispersión 2 a 2')  
  
plt.show();



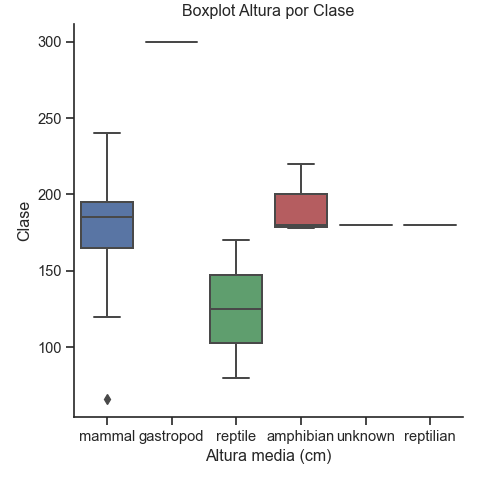
En diagonal se muestra un gráfico de densidad (parecido a un histograma).

En el resto del cuadrante, se muestra el gráfico de dispersión entre dos variables. También se añaden los colores con las especies.

Impresionante!

Veamos ahora la facilidad para analizar boxplots.

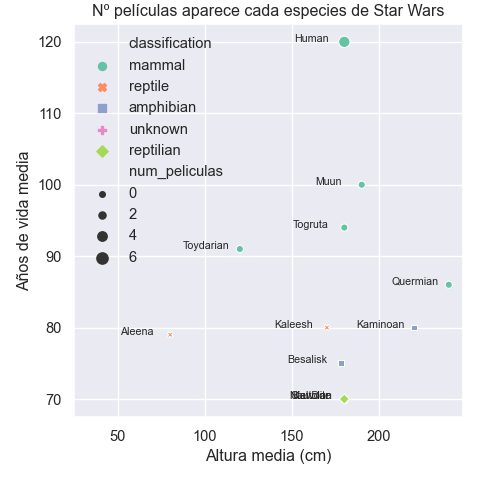
warnings.filterwarnings('default') # filtramos warnings para evitar problemas con especies de un individuo.  
  
with sns.axes\_style(style='ticks'):  
 g = sns.catplot("classification", "average\_height", data=species\_df, kind="box")  
  
plt.gcf().subplots\_adjust(top=0.95)  
plt.title("Boxplot Altura por Clase")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("Clase");  
  
plt.show();



Finalmente vemos el gráfico de dispersión, donde se ha personalizado el color y marca para cada especie. También el tamaño en función del número de películas.

Para hacer el gráfico más visual, sólo se muestran las especies con menos de 200 años de vida media esperada.

# Recuperamos el gráfico de especies  
sns.set(rc={'figure.figsize':(9,7)});  
cmap = sns.cubehelix\_palette(dark=.3, light=.8, as\_cmap=True);  
  
ax = sns.scatterplot(x='average\_height', y='average\_lifespan'  
 , hue='classification', size="num\_peliculas"   
 , style="classification", palette="Set2"  
 , data=species\_df[species\_df.average\_lifespan<200]);  
ax.set(xlim=(25, None));  
  
plt.gcf().subplots\_adjust(top=0.95)  
plt.title("Nº películas aparece cada especies de Star Wars")  
plt.xlabel("Altura media (cm)")  
plt.ylabel("Años de vida media");  
  
# Añadimos etiquetas  
for x1,y1,n1 in zip(altura\_media[species\_df.average\_lifespan<200]  
 ,vida\_media[species\_df.average\_lifespan<200]  
 ,nombre[species\_df.average\_lifespan<200]):  
 plt.annotate(n1,(x1, y1)  
 , size=8  
 , textcoords="offset points" # desplaza el punto  
 , xytext=(-25,0) # distancia desplazamiento  
 , ha='center') # centrado  
plt.show();



Entremos ahora a profundizar en la forma en que Python trata las colecciones de objetos.

## COLECCIONES DE OBJETOS

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_30_colec_obj.ipynb)

### DEFINCIÓN DE LOS OBJETOS

#### TIPO BÁSICOS

En Python, el tipo de dato se fija en la creación del objeto.

Los principales tipo básicos son:

|  |  |  |
| --- | --- | --- |
| Ejemplo | Constructores | Tipo de dato |
| x = “Hola Mundo” | x = str(“Hello World”) | str |
| x = 20 | x = int(20) | int |
| x = 20.5 | x = float(20.5) | float |
| x = True | x = bool(5) | bool |

Para saber el tipo de un objeto, utiliza type().

**Recuerda**: Para saber el tipo del objeto utiliza type().

a = 3 > 4  
print("El tipo de 'a' es:",type(a),"y vale",a)

## El tipo de 'a' es: <class 'bool'> y vale False

#### OPERACIONES CON BOOLEANOS

Se utiliza and, or and not para operar con True y False.

True or not True

## True

#### COMPARADORES BOOLEANOS

Los típicos: <, <=, >,>=

Y además.

3 == 4. # igualdad

## False

5 != 5 # desigualdad

## False

Comparación de objetos con is y is not

a = "Soy string"  
type(a) is str

## True

a = 45.  
type(a) is not int

## True

**Recuerda**: Para negar expresiones lógicas, se utiliza not.

not (True or not False)

## False

#### *CASTINGS*

Utiliza los constructores de cada tipo para realizar *casting*.

print('True es',int(True),'y False es',int(False))

## True es 1 y False es 0

### LISTAS

Sus elementos van entre corchetes y separados por comas: [ele1, elem2, ...].

Sus elementos pueden ser objetos de distinto tipo.

a = True  
b = "Texto"  
c = 3.1416  
d = [a, b, c]  
d

## [True, 'Texto', 3.1416]

#### LONGITUD

Con len() se obtiene el número de elementos.

len(d)

## 3

#### OPERACIONES CON LISTAS

Se puede concatenar listas con +.

[1,2,3] + [4,5,6] # suma listas

## [1, 2, 3, 4, 5, 6]

Incluso repetirlas con \*

[1,2,3] \* 2 # multiplicación por escalar

## [1, 2, 3, 1, 2, 3]

Buscar si la lista contiene (o no contiene) un elemento con in (resp. not in).

a = 'estoy'  
b = ['aquí','estoy']  
a in b

## True

c = ['aquí','no estoy']  
a not in c

## True

Mínimos y máximos.

a = [3,1,5]  
min(a)

## 1

#### OPERACIONES CON TEXTO

En Python el tipo de texto str se comporta como una lista.

a = 'tengo longitud 17'  
len(a)

## 17

Concatenar *strings* con +.

a = "El símbolo suma" + " concatena" + " el texto"  
a

## 'El símbolo suma concatena el texto'

El texto también se multiplica

a = "abc" \* 3  
a

## 'abcabcabc'

Hay que tener cuidado con sumar texto y otros tipos (int, float, bool).

a = "Texto más " + 24  
a

## ---------------------------------------------------------------------------  
## TypeError Traceback (most recent call last)  
## <ipython-input-20-478e9ae4b08d> in <module>()  
## ----> 1 a = "Texto más " + 24  
## 2 a  
  
## TypeError: must be str, not int

Para evitar el error, antes hay que hacer casting con el constructor str().

a = "Texto más " + str(24)  
a

## 'Texto más 24'

Un alternativa para concatenar texto contenido en una lista es ''.join().

a = ["Soy","texto","en","lista"]  
' '.join(a)

## 'Soy texto en lista'

De esta forma es fácil cambiar el carácter de separación.

'-'.join(a)

## 'Soy-texto-en-lista'

### TUPLAS

Sus elementos van entre paréntesis y separados por comas: (ele1, elem2, ...).

a = (1,2,3)  
a

## (1, 2, 3)

Permiten realizar varias asignaciones de objetos en una misma expresión.

a, b = 2, 6  
print(a,b)

## 2 6

Es habitual utilizar tuplas para devolver varios objetos en una misma función.

def desayuno(bacon, huevos):  
 return bacon, huevos  
  
bacon, huevos = desayuno(bacon = "muy hecho", huevos = 2)  
  
print("Quiero el bacon {} y {} huevos".format(bacon,huevos))

## Quiero el bacon muy hecho y 2 huevos

**Recuerda**: Son inmutables. Una vez fijado su valor, ya no tiene sentido modificarlo!

#### RANGOS

Son un caso especial de tuplas, definidas como secuencias.

a = range(3,6)  
a

## range(3, 6)

Para convertirlos a lista.

list(a)

## [3, 4, 5]

### DICCIONARIOS

Sus elementos van entre llaves, separados por comas con una clave: {'clave1': val1, 'clave2': val1,...}

Hay distintas formas de declararlos, todas equivalentes.

a = dict(uno=1, dos=2, tres=3)  
b = {'uno': 1, 'dos': 2, 'tres': 3}  
c = dict(zip(['uno', 'dos', 'tres'], [1, 2, 3]))  
d = dict([('dos', 2), ('uno', 1), ('tres', 3)])  
e = dict({'tres': 3, 'uno': 1, 'dos': 2})  
a == b == c == d == e

## True

**Sabías que**: Dentro de los elementos de un diccionario no hay una ordenación prefijada.

Se accede a sus elementos por la clave.

a["uno"]

## 1

Para consultar las claves.

a.keys()

## dict\_keys(['uno', 'dos', 'tres'])

Obtener la lista de pares (clave,valor) con items().

a.items()

## dict\_items([('uno', 1), ('dos', 2), ('tres', 3)])

#### FILTROS EN LISTAS

Utiliza índices enteros positivos (cero incluido).

a = [3,7,2,9,2,8,6]  
a[0]

## 3

**Recuerda**: Python empieza sus índices en 0, no 1.

Los negativos seleccionan empezando por la cola.

a[-2]

## 8

También se utilizan secuencias o *slices* definidos con :.

a[1:5]

## [7, 2, 9, 2]

Secuencias con paso distinto a 1 añadiendo un tercer elemento: ini:fin:paso.

a[1:5:2]

## [7, 9]

Puedes omitir el inicio o el fin para extender la secuencia.

a[:4]

## [3, 7, 2, 9]

O, los dos.

a[:]

## [3, 7, 2, 9, 2, 8, 6]

Los strings siguen comportándose como listas en este aspecto, ya que permiten aplicar filtros dentro del texto.

a = "no soy una lista"  
a[3:]

## 'soy una lista'

#### *LIST COMPREHENSION*

La comprensión de listas o *list comprehension* es una forma elegante de automatizar cálculos con listas.

Por ejemplo, si se quiere invertir la secuencia del 1 al 4.

a = range(1,5)  
b = [1 / x for x in a]  
b

## [1.0, 0.5, 0.3333333333333333, 0.25]

También admite condicionales.

a = [0,1,0,2,0,3,4,0]  
b = [1 / x for x in a if x != 0]  
b

## [1.0, 0.5, 0.3333333333333333, 0.25]

Los diccionarios, a su manera, admiten también este recurso.

a = {'uno': 1, 'dos': 2, 'tres': 3}  
b = {k: v\*v for k,v in a.items()}  
b

## {'uno': 1, 'dos': 4, 'tres': 9}

Continúa ahora con los objetos de tipo **numpy**.

## NUMPY

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_31_numpy.ipynb)

El package [numpy](https://numpy.org/) es la solución más popular dentro de Python para realizar computación científica.

Recoge las mejores prácticas introducidas en las Listas y organizadas para realizar cálculos de forma eficiente. Se estructuran como vectores o arrays de N, dimensiones de un mismo tipo de dato.

El primer paso, consiste en cargar la librería, habitualmente con *namespace* np.

import numpy as np

##### ACTIVIDAD GUIADA 2.3

Antes de entrar en materia vamos a presentar la actividad guiada de *numpy*.

Se trata de seleccionar la nave interestelar más rápida en MGLT (Megaluz / Hora), que pueda llevar a 4 tripulantes (como los protagonistas: Han Solo, Leia, Luke y Chewbacca). Además el coste de la nave debe ser com máximo 1M de créditos interestelares.

«Es la chatarra más veloz de la galaxia.»

―Lando Calrissian

Lando se refería al Halcón Milenario. ¿Será verdad?

Para la actividad contamos con los siguiente datos em formato pandas.

import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn; seaborn.set() # para el estilo de graficos  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
starships\_df = entidades\_df['starships'][["cost\_in\_credits","MGLT","crew"]].dropna()  
  
starships\_df.head()

## cost\_in\_credits MGLT crew  
## name   
## CR90 corvette 3.500000e+06 60.0 165.0  
## Star Destroyer 1.500000e+08 60.0 47060.0  
## Sentinel-class landing craft 2.400000e+05 70.0 5.0  
## Death Star 1.000000e+12 10.0 342953.0  
## Millennium Falcon 1.000000e+05 75.0 4.0

Transformamos la matriz numpy con las variables numéricas:

starship\_np = starships\_df.values  
starship\_np # las filas son las naves y las columnas: cost\_in\_credits, MGLT y crew

## array([[3.50000e+06, 6.00000e+01, 1.65000e+02],  
## [1.50000e+08, 6.00000e+01, 4.70600e+04],  
## [2.40000e+05, 7.00000e+01, 5.00000e+00],  
## [1.00000e+12, 1.00000e+01, 3.42953e+05],  
## [1.00000e+05, 7.50000e+01, 4.00000e+00],  
## [1.34999e+05, 8.00000e+01, 2.00000e+00],  
## [1.49999e+05, 1.00000e+02, 1.00000e+00],  
## [1.14335e+09, 4.00000e+01, 2.79144e+05],  
## [2.40000e+05, 5.00000e+01, 6.00000e+00],  
## [8.50000e+06, 4.00000e+01, 8.54000e+02],  
## [1.04000e+08, 6.00000e+01, 5.40000e+03],  
## [1.75000e+05, 1.20000e+02, 1.00000e+00],  
## [2.20000e+05, 9.10000e+01, 1.00000e+00],  
## [1.55000e+05, 1.00000e+02, 3.00000e+00]])

Nombres de las naves a array numpy.

starship\_names = starships\_df.index.values   
starship\_names

## array(['CR90 corvette', 'Star Destroyer', 'Sentinel-class landing craft',  
## 'Death Star', 'Millennium Falcon', 'Y-wing', 'X-wing', 'Executor',  
## 'Imperial shuttle', 'EF76 Nebulon-B escort frigate',  
## 'Calamari Cruiser', 'A-wing', 'B-wing', 'arc-170'], dtype=object)

#### TIPO DE DATOS EN NUMPY

Habitualmente se crean a partir de listas o tuplas con elementos homogéneos.

b\_np = np.array([True, False, False]) # booleano  
b\_np

## array([ True, False, False])

s\_np = np.array(['f', 'h', 'j']) # string  
s\_np

## array(['f', 'h', 'j'], dtype='<U1')

i\_np = np.array(range(3)) # entero  
i\_np

## array([0, 1, 2])

f\_np = np.array([2.4, 1.5, 3.0]) # coma flotante  
f\_np

## array([2.4, 1.5, 3. ])

Los 4 objetos comparten ser de tipo numpy.ndarray.

ej\_np = (b\_np, s\_np, i\_np, f\_np)  
[type(x) for x in ej\_np]

## [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>]

Sin embargo, cada uno tiene un tipo interno accesible con el método .dtype .

[x.dtype for x in ej\_np]

## [dtype('bool'), dtype('<U1'), dtype('int32'), dtype('float64')]

Observa que numpy define sus propios nombres para los tipos básicos donde añade el tamaño en bits:

* ‘U1’ es texto Unicode de 1 bit.
* ‘int64’, ‘float64’ numéricos de 64 bits.

Se puede definir este tipo en su construcción.

i\_np = np.array(range(3), dtype='int16')  
i\_np

## array([0, 1, 2], dtype=int16)

O convertir el numpy de un tipo interno a otro en un nuevo objeto con .astype()

s\_i\_np = i\_np.astype(str)  
s\_i\_np

## array(['0', '1', '2'], dtype='<U6')

El tipo de la matriz numérica de las naves es:

starship\_np.dtype

## dtype('float64')

El vector de nombres tiene dtype object. Este tipo se utiliza para numpy de texto.

starship\_names.dtype

## dtype('O')

Así se ve mejor.

print(starship\_names.dtype)

## object

#### FUNCIONES UNIVERSALES

En numpy las funciones universales o *ufunc*, son un conjunto de funciones vectorizan, es decir, que al aplicarlas sobre un objeto numpy, esta función, se aplica sobre cada uno de los elementos que lo componen.

**Recuerda**: Ya se comentó las funciones que vectorizan en el capitulo de funciones de R.

Típicamente, las funciones de este tipo son las funciones aritméticas.

np.array([1,2,3]) \* 2 # multiplicación por escalar

## array([2, 4, 6])

np.array([1,2,3]) + np.array([4,5,6]) # suma vectores

## array([5, 7, 9])

np.array([1,2,3]) \* np.array([4,5,6]) # producto elemento a elemento

## array([ 4, 10, 18])

NumPy ofrece una extensa colección de funciones matemáticas que se aplican de forma vectorial. Algunas de ellas son: abs,sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh y arctanh.

np.sqrt(np.array([1,2,3,4]))

## array([1. , 1.41421356, 1.73205081, 2. ])

#### SECUENCIAS EN NUMPY

La función propia de numpy es np.arange(ini,fin,paso)

np.arange(0,6,2)

## array([0, 2, 4])

Secuencias de ceros.

np.zeros(3)

## array([0., 0., 0.])

np.ones(3)

## array([1., 1., 1.])

Repetir todo el vector.

a = [1,2,3]  
b = np.tile(a, 4)  
b

## array([1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3])

Repetir cada elemento de un vector.

a = [1,2,3]  
b = np.repeat(a, 4)  
b

## array([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3])

#### DIMENSIONES EN NUMPY

np.random.seed(123) # Semilla para poder reproducir  
  
x1 = np.random.randint(10, size=6) # array 1-dimensional   
x2 = np.random.randint(10, size=(3, 4)) # array 2-dimensional  
x3 = np.random.randint(10, size=(3, 4, 5)) # array 3-dimensional

print("x3 número dimensiones: ", x3.ndim)

## x3 número dimensiones: 3

print("x3 shape:", x3.shape)

## x3 shape: (3, 4, 5)

print("x3 número de elmentos: ", x3.size)

## x3 número de elmentos: 60

print("X3 cantidad de bytes:", x3.nbytes, "bytes")

## X3 cantidad de bytes: 240 bytes

Nuestra matriz de naves interestelares.

print("Naves número dimensiones: ", starship\_np.ndim)

## Naves número dimensiones: 2

print("Nombres número dimensiones: ", starship\_names.ndim)

## Nombres número dimensiones: 1

print("Naves shape:", starship\_np.shape)

## Naves shape: (14, 3)

print("Nombres shape:", starship\_names.shape)

## Nombres shape: (14,)

print("Naves número de elmentos: ", starship\_np.size)

## Naves número de elmentos: 42

print("Nombres número de elmentos: ", starship\_names.size)

## Nombres número de elmentos: 14

print("Naves cantidad de bytes:", starship\_np.nbytes, "bytes")

## Naves cantidad de bytes: 336 bytes

print("Nombres cantidad de bytes:", starship\_names.nbytes, "bytes")

## Nombres cantidad de bytes: 112 bytes

### FILTROS EN NUMPY

Siguen las mismas reglas que en las listas, pero ahora con más dimensiones

x2

## array([[6, 1, 0, 1],  
## [9, 0, 0, 9],  
## [3, 4, 0, 0]])

Elemento de la matriz.

x2[0, 0]

## 6

#### FILTROS SLICING

Vector segunda columna.

x2[:,1]

## array([1, 0, 4])

Vector tercera fila.

x2[2,:]

## array([3, 4, 0, 0])

O, la misma fila, por omisión del segundo elemento.

x2[2]

## array([3, 4, 0, 0])

En nuestras naves, las columnas era: cost\_in\_credits, MGLT, crew.

Para acceder a la velocidad (MGLT).

starship\_np[:,1]

## array([ 60., 60., 70., 10., 75., 80., 100., 40., 50., 40., 60.,  
## 120., 91., 100.])

Para seleccionar la 5ª nave, se selecciona una fila 4.

starship\_np[4,:]

## array([1.0e+05, 7.5e+01, 4.0e+00])

Para ver su nombre.

starship\_names[4]

## 'Millennium Falcon'

#### INDEXACIÓN ELEGANTE

##### FILTROS POSICIONALES

Otra mejora respecto a las listas es la capacidad de indexar por una lista o array.

Para filtrar por un array unidimensional de posiciones.

x1

## array([2, 2, 6, 1, 3, 9])

ind = [0,2,5]  
x1[ind]

## array([2, 6, 9])

**Recuerda**: En el objeto list, esta operación no es posible. En su lugar se utilizan las list comprehension.

En 2 dimensiones, a diferencia de lo visto en R, se aparean por coordenadas filas y columnas, es lo que se llama Broadcasting. Lo verás más adelante.

x2

## array([[6, 1, 0, 1],  
## [9, 0, 0, 9],  
## [3, 4, 0, 0]])

fils = np.array([0,1])  
cols = np.array([0,3])  
  
x2[fils,cols]

## array([6, 9])

**Recuerda**: en arrays numpy los filtros por posición, se seleccionan apareando coordenadas fila y columna. Ve a la sección de Broadcasting, para saber más!

En R se obtendría el siguiente resultado.

x2[fils,:][:,cols]

## array([[6, 1],  
## [9, 9]])

##### FILTROS LÓGICOS

Otro mecanismo *fancy indexing* son los filtros lógicos.

Antes recordemos que para obtener el número de tripulantes (crew).

starship\_np[:,2]

## array([1.65000e+02, 4.70600e+04, 5.00000e+00, 3.42953e+05, 4.00000e+00,  
## 2.00000e+00, 1.00000e+00, 2.79144e+05, 6.00000e+00, 8.54000e+02,  
## 5.40000e+03, 1.00000e+00, 1.00000e+00, 3.00000e+00])

Para saber cuáles son las naves de 4 o más tripulantes.

starship\_np[:,2] >= 4

## array([ True, True, True, True, True, False, False, True, True,  
## True, True, False, False, False])

Si queremos rescatar sus nombres.

starship\_names[starship\_np[:,2] >= 4]

## array(['CR90 corvette', 'Star Destroyer', 'Sentinel-class landing craft',  
## 'Death Star', 'Millennium Falcon', 'Executor', 'Imperial shuttle',  
## 'EF76 Nebulon-B escort frigate', 'Calamari Cruiser'], dtype=object)

Continúan funcionando los operadores distinto !=, en in, no en not in.

'Millennium Falcon' in starship\_names[starship\_np[:,2] >= 4]

## True

Para reproducir el mismo comportamiento que el operador a %in% b de R se utiliza np.isin(a, b) .

a = np.array([1, 2, 3, 4])  
b = np.array([3, 4, 5])  
np.isin(a,b)

## array([False, False, True, True])

##### OPERACIONES BOOLEANAS

Se utilizan los &, | y ~ en vez de los and, or, not (respec.) utilizados en los tipos lógicos. Es necesario cerrar las expresiones entre paréntesis.

Por ejemplo:

( A > 1 ) | ( B < 5)

Respecto a nuestras naves, las que tienen 4 o más tripulantes y de cuestan menos de 1M de créditos.

starship\_names\_sel = starship\_names[(starship\_np[:,2] >= 4) & (starship\_np[:,0] <10\*\*6)]  
starship\_names\_sel

## array(['Sentinel-class landing craft', 'Millennium Falcon',  
## 'Imperial shuttle'], dtype=object)

**Recuerda**: las 2 expresiones van entre paréntesis.

Ahora filtramos la matriz de datos de estas naves.

starship\_np\_sel = starship\_np[(starship\_np[:,2] >= 4) & (starship\_np[:,0] <10\*\*6), :]  
starship\_np\_sel

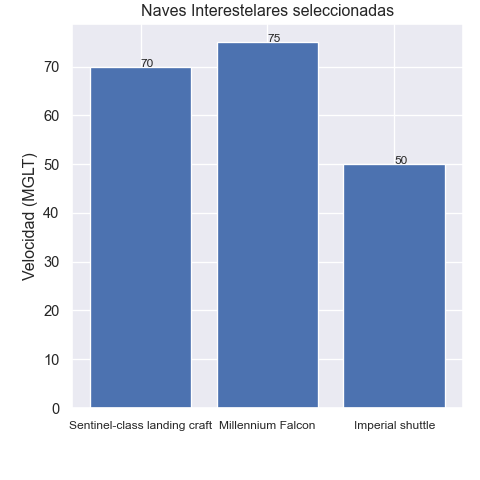
## array([[2.4e+05, 7.0e+01, 5.0e+00],  
## [1.0e+05, 7.5e+01, 4.0e+00],  
## [2.4e+05, 5.0e+01, 6.0e+00]])

Gráficamente.

starship\_names\_sel

## array(['Sentinel-class landing craft', 'Millennium Falcon',  
## 'Imperial shuttle'], dtype=object)

ind = np.arange(len(starship\_names\_sel)) # posiciones  
  
plt.bar(ind, starship\_np\_sel[:,1] );  
  
for i in ind:  
 plt.text(ind[i],starship\_np\_sel[i,1], int(starship\_np\_sel[i,1]),size=9)  
  
plt.gcf().subplots\_adjust(left=0.15,bottom=0.15)  
plt.ylabel('Velocidad (MGLT)')  
plt.title('Naves Interestelares seleccionadas')  
p = plt.xticks(ind, starship\_names\_sel,fontsize=9);  
plt.show();



Parece que el Halcón Milenario con 75 MGLT, es la nave que andábamos buscando!

### MODIFICAR ARRAYS

Para modificar un valor de un array.

x2

## array([[6, 1, 0, 1],  
## [9, 0, 0, 9],  
## [3, 4, 0, 0]])

x2[1,1] = 20  
x2

## array([[ 6, 1, 0, 1],  
## [ 9, 20, 0, 9],  
## [ 3, 4, 0, 0]])

Mucha precaución con el tipo interno del numpy ya que es fijo. No se va a modificar.

x2[1,1] = 20.765  
x2

## array([[ 6, 1, 0, 1],  
## [ 9, 20, 0, 9],  
## [ 3, 4, 0, 0]])

**Recuerda**: Si intentas insertar un valor en coma flotante sobre un numpy con tipo entero, éste se va a cortar.

Se puede modificar N valores con un vector de N posiciones.

x2[1,:] = x2[0,:] \* 10   
x2

## array([[ 6, 1, 0, 1],  
## [60, 10, 0, 10],  
## [ 3, 4, 0, 0]])

También, se pueden modificar varios a la vez, con un único valor.

x2[[1,2],[2,2]] = 55  
x2

## array([[ 6, 1, 0, 1],  
## [60, 10, 55, 10],  
## [ 3, 4, 55, 0]])

#### VISTAS NO COPIADAS

Cuando se asigna un array numpy a otro objeto, ya sea filtrado con [,] o no, éste retorna una vista, no una copia. Es muy importante tenerlo en cuenta, cuando se modifica el objeto.

x2

## array([[ 6, 1, 0, 1],  
## [60, 10, 55, 10],  
## [ 3, 4, 55, 0]])

x2\_1 = x2

x2\_1[1,0] = 11

x2\_1

## array([[ 6, 1, 0, 1],  
## [11, 10, 55, 10],  
## [ 3, 4, 55, 0]])

**Importante**: Los cambios en una vista no copiada, afectan a todas las copias!!!

#### COPIAS DE ARRAYS

Para copiar un array a otro objeto utiliza copy().

x2\_2 = x2.copy()

x2\_2[0,0] = 50

x2

## array([[ 6, 1, 0, 1],  
## [11, 10, 55, 10],  
## [ 3, 4, 55, 0]])

Ahora, el objeto original, no cambia.

#### CAMBIOS DE *SHAPE*

Con .reshape() se puede modificar las dimensiones del array.

x1

## array([2, 2, 6, 1, 3, 9])

x1.reshape(2,3)

## array([[2, 2, 6],  
## [1, 3, 9]])

#### CONCATENAR ARRAYS

Para concatenar 2 arrays unidimensionales.

x = np.array([1, 2, 3])  
y = np.array([3, 2, 1])  
np.concatenate([x, y])

## array([1, 2, 3, 3, 2, 1])

grid = np.arange(6).reshape(2,3)  
  
# concatenar filas, (axis = 0)  
np.concatenate([grid, grid])

## array([[0, 1, 2],  
## [3, 4, 5],  
## [0, 1, 2],  
## [3, 4, 5]])

# concatenar columnas, (axis = 1)  
np.concatenate([grid, grid], axis=1)

## array([[0, 1, 2, 0, 1, 2],  
## [3, 4, 5, 3, 4, 5]])

Lo mismo con vstack y hstack, pero con arrays de distinta dimensión.

x = np.array([1, 2, 3])  
grid = np.array([[9, 8, 7],  
 [6, 5, 4]])  
  
# apilado vertical  
np.vstack([x, grid])

## array([[1, 2, 3],  
## [9, 8, 7],  
## [6, 5, 4]])

# apilado horizontal  
y = np.array([[99],  
 [99]])  
np.hstack([grid, y])

## array([[ 9, 8, 7, 99],  
## [ 6, 5, 4, 99]])

#### BROADCASTING DE ARRAYS

El Broadcasting es una operación de alineamiento que sigue de forma estricta las siguientes reglas de interacción entre 2 arrays:

* Regla 1: Si dos arrays difieren en el número de dimensiones, el shape (dimensiones) del que tenga menor dimensión se aumenta con unos por la izquierda.

[3,1,6] x [2,6] ----> [3,1,6] x [1,2,6]  
[3,5,4] x [3,5] ----> [3,5,4] x [1,3,5]

* Regla 2: Si dos arrays tienen shape distinto en alguna dimensión, el array con shape igual a 1 se estira (copia) para que concuerde con el de la misma dimensión del otro array.

[3,1,6] x [1,2,6] ----> [3,2,6] x [3,2,6]  
[3,5,4] x [1,3,5] ----> [3,5,4] x [3,3,5]

* Regla 3: Si alguna dimensión tiene tamaño distinto y no hay ninguna dimensión 1, se lanza un error.

[3,2,6] x [3,2,6] ----> Concuerdan  
[3,5,4] x [3,3,5] ----> Error !!!

En la práctica, esto significa facilitar algunas operaciones.

a = np.ones((2,2))  
a

## array([[1., 1.],  
## [1., 1.]])

b = np.tile(0.5, 2)  
b

## array([0.5, 0.5])

a - b

## array([[0.5, 0.5],  
## [0.5, 0.5]])

#### CONVERSIÓN A LISTAS

Los numpy se convierten a lista con .tolist().

a = np.array([2,3,7,3])  
b = a.tolist()  
b

## [2, 3, 7, 3]

type(b)

## <class 'list'>

#### VALORES FALTANTES

Por definición, un NaN es un número en coma flotante, distinto a cualquier otro número.

np.nan != np.nan

## True

a = np.arange(6).reshape(3,2) \* 1.0  
  
a[[1,2],[1] ] = np.nan  
  
a

## array([[ 0., 1.],  
## [ 2., nan],  
## [ 4., nan]])

**Observa**: como se ha aplicado broadcast al asignar np.nan.

np.isnan(a)

## array([[False, False],  
## [False, True],  
## [False, True]])

Numpy es la base de la computación con arrays. Veamos ahora qué aportan los objetos **pandas**.

## PANDAS

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_32_pandas.ipynb)

Abreviación de Panel Data, los objetos pandas dan soporte al análisi de datos con variables de distinta tipología: categóricas, binarias, numéricas…

Respecto a los numpy, permiten indexar las filas y columnas. Este tipo de indexación, facilita, por ejemplo, realizar análisis de series temporales, no necesariamente con una frecuencia fija.

Como siempre el primer paso es cargar la librería con *namespace* pd

import pandas as pd

##### ACTIVIDAD GUIADA 2.4

Esta vez, se trata de seleccionar el mejor plantea posible para ubicar la academia Jedi. Los parámetros de la búsqueda son:

* Días largos para entrenar mucho
* Mucha agua para poder refrescarse
* Poca densidad de población para no ser molestado
* Buen clima (temperado o tropical)

«Si existe un auténtico centro del universo, ahora estás en el planeta más alejado de él»

―Luke Skywalker sobre Tatooine el planeta desértico dónde nacieron Anakin y Luke Skywalker.

El primer paso consiste en cargar los datos de los planetas.

import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set() # para el estilo de graficos  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
planets\_df = entidades\_df['planets'][["rotation\_period","diameter","climate"  
 ,"temperate\_tropical","gravity"  
 ,"surface\_water","population"]].dropna()  
planets\_df = planets\_df[planets\_df.diameter>0]  
  
  
planets\_df.head()

## rotation\_period diameter ... surface\_water population  
## name ...   
## Tatooine 23.0 10465.0 ... 1.0 2.000000e+05  
## Alderaan 24.0 12500.0 ... 40.0 2.000000e+09  
## Yavin IV 24.0 10200.0 ... 8.0 1.000000e+03  
## Bespin 12.0 118000.0 ... 0.0 6.000000e+06  
## Endor 18.0 4900.0 ... 8.0 3.000000e+07  
##   
## [5 rows x 7 columns]

### PANDAS SERIES

Se definen como un array uni-dimensional indexado.

Se puede crear desde una lista.

a = pd.Series([1.2, 0.4, 2.3])  
a

## 0 1.2  
## 1 0.4  
## 2 2.3  
## dtype: float64

Como se ve, une valores e índice. Se accede a los valores en formato numpy con .values

a.values

## array([1.2, 0.4, 2.3])

Los índices tienen su propio tipo y se accede con .index

a.index

## RangeIndex(start=0, stop=3, step=1)

Mismas funcionalidades *slicing*, funciones, *fancy indexing* que los arrays numpy.

a[1:3] \* 2

## 1 0.8  
## 2 4.6  
## dtype: float64

Se puede declarar el índice de forma explícita.

a = pd.Series([1.2, 0.4, 2.3], index=['a','b','c'])  
a

## a 1.2  
## b 0.4  
## c 2.3  
## dtype: float64

O bien, a partir de un diccionario.

a = pd.Series({'a': 1.2, 'b': 0.4, 'c': 2.3})  
a

## a 1.2  
## b 0.4  
## c 2.3  
## dtype: float64

El índice añade la capacidad de filtrar a partir de sus valores.

a[['b','c']]

## b 0.4  
## c 2.3  
## dtype: float64

Incluso tiene los métodos de un diccionario.

a.keys()

## Index(['a', 'b', 'c'], dtype='object')

list(a.items())

## [('a', 1.2), ('b', 0.4), ('c', 2.3)]

#### ÍNDEXADORES: LOC, ILOC

Se utilizan para modificar el comportamiento de los filtros.

Su función es evitar colisiones entre índices y posiciones. Por ejemplo, la siguiente serie tiene índices enteros.

a = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])  
a

## 1 a  
## 3 b  
## 5 c  
## dtype: object

Si se accede al valor del índice 1.

a[1]

## 'a'

En cambio, si se pide los valores de las posiciones 1 a 3 con slicing.

a[1:3]

## 3 b  
## 5 c  
## dtype: object

Para evitar estas confusiones se introduce .loc que permite realizar un acceso a los valores siempre basados en el índice.

a.loc[1]

## 'a'

a.loc[1:3]

## 1 a  
## 3 b  
## dtype: object

Por contra .iloc tiene la finalidad contraria, realizar el acceso por posición.

a.iloc[1]

## 'b'

a.iloc[1:3]

## 3 b  
## 5 c  
## dtype: object

### OBJETOS PANDAS DATAFRAME

Se pueden ver como una generalización de los numpy o incluso de los diccionarios.

Para poderlo visualizar, partimos de dos series, una serie del diámetro y otra de la población de los planetas.

diameter\_dict = {'Tatooine': 10465.0,'Alderaan': 12500.0  
 ,'Bespin': 118000.0,'Endor': 4900.0,'Naboo': 12120.0}  
population\_dict = {'Tatooine': 2.000000e+05,'Alderaan': 2.000000e+09  
 ,'Bespin': 6.000000e+06,'Endor': 3.000000e+07,'Naboo': 4.500000e+09}

population = pd.Series(population\_dict)  
population

## Tatooine 2.000000e+05  
## Alderaan 2.000000e+09  
## Bespin 6.000000e+06  
## Endor 3.000000e+07  
## Naboo 4.500000e+09  
## dtype: float64

diameter = pd.Series(diameter\_dict)  
diameter

## Tatooine 10465.0  
## Alderaan 12500.0  
## Bespin 118000.0  
## Endor 4900.0  
## Naboo 12120.0  
## dtype: float64

Ahora construimos un DataFrame con el método pd.DataFrame().

planetas = pd.DataFrame({'population': population,  
 'diameter': diameter})  
planetas

## population diameter  
## Tatooine 2.000000e+05 10465.0  
## Alderaan 2.000000e+09 12500.0  
## Bespin 6.000000e+06 118000.0  
## Endor 3.000000e+07 4900.0  
## Naboo 4.500000e+09 12120.0

Al igual que las series, se accede a su índice fila con .index.

planetas.index

## Index(['Tatooine', 'Alderaan', 'Bespin', 'Endor', 'Naboo'], dtype='object')

También se accede al índice de las columnas con .columns.

planetas.columns

## Index(['population', 'diameter'], dtype='object')

Se pueden construir a partir de numpy, especificando el nombre de las columnas.

a = np.ones((3,2))  
a

## array([[1., 1.],  
## [1., 1.],  
## [1., 1.]])

b = pd.DataFrame(a, columns=["col1","col2"])  
b

## col1 col2  
## 0 1.0 1.0  
## 1 1.0 1.0  
## 2 1.0 1.0

Se puede recuperar el numpy con .values.

b.values

## array([[1., 1.],  
## [1., 1.],  
## [1., 1.]])

Se accede a los tipos internos con .dtypes.

planets\_df.dtypes

## rotation\_period float64  
## diameter float64  
## climate object  
## temperate\_tropical int32  
## gravity float64  
## surface\_water float64  
## population float64  
## dtype: object

#### GESTIÓN DE COLUMNAS EN DATAFRAMES

Se accede a las series que forman un DataFrame utilizando la anotación de diccionario.

planets\_df["population"]

## name  
## Tatooine 2.000000e+05  
## Alderaan 2.000000e+09  
## Yavin IV 1.000000e+03  
## Bespin 6.000000e+06  
## Endor 3.000000e+07  
## Naboo 4.500000e+09  
## Kamino 1.000000e+09  
## Geonosis 1.000000e+11  
## Utapau 9.500000e+07  
## Mustafar 2.000000e+04  
## Kashyyyk 4.500000e+07  
## Corellia 3.000000e+09  
## Rodia 1.300000e+09  
## Ord Mantell 4.000000e+09  
## Mon Cala 2.700000e+10  
## Chandrila 1.200000e+09  
## Sullust 1.850000e+10  
## Ryloth 1.500000e+09  
## Glee Anselm 5.000000e+08  
## Muunilinst 5.000000e+09  
## Name: population, dtype: float64

También, como si fuera un atributo.

planets\_df.diameter

## name  
## Tatooine 10465.0  
## Alderaan 12500.0  
## Yavin IV 10200.0  
## Bespin 118000.0  
## Endor 4900.0  
## Naboo 12120.0  
## Kamino 19720.0  
## Geonosis 11370.0  
## Utapau 12900.0  
## Mustafar 4200.0  
## Kashyyyk 12765.0  
## Corellia 11000.0  
## Rodia 7549.0  
## Ord Mantell 14050.0  
## Mon Cala 11030.0  
## Chandrila 13500.0  
## Sullust 12780.0  
## Ryloth 10600.0  
## Glee Anselm 15600.0  
## Muunilinst 13800.0  
## Name: diameter, dtype: float64

Se pueden crear nuevas columnas o series. Por ejemplo, calculamos:

* El área en Km² del planeta (suponiendo que son esféricos): .
* La densidad poblacional en: habitantes / Km².

import math  
planets\_df["area"] = 4 \* math.pi \* (planets\_df["diameter"]/2)\*\*2  
planets\_df["density"] = planets\_df["population"] / planets\_df["area"]

Para seleccionar varias columnas se utilizan listas. Observa que ahora, se mantiene la clase DataFrame.

planets\_df[["population","diameter","area","density"]].head()

## population diameter area density  
## name   
## Tatooine 2.000000e+05 10465.0 3.440554e+08 0.000581  
## Alderaan 2.000000e+09 12500.0 4.908739e+08 4.074367  
## Yavin IV 1.000000e+03 10200.0 3.268513e+08 0.000003  
## Bespin 6.000000e+06 118000.0 4.374354e+10 0.000137  
## Endor 3.000000e+07 4900.0 7.542964e+07 0.397722

Modificar el nombre de una columna con .rename().

planets\_df.rename(columns={"area": "Area", "density": "Densidad"}, inplace=True)  
planets\_df.head()

## rotation\_period diameter ... Area Densidad  
## name ...   
## Tatooine 23.0 10465.0 ... 3.440554e+08 0.000581  
## Alderaan 24.0 12500.0 ... 4.908739e+08 4.074367  
## Yavin IV 24.0 10200.0 ... 3.268513e+08 0.000003  
## Bespin 12.0 118000.0 ... 4.374354e+10 0.000137  
## Endor 18.0 4900.0 ... 7.542964e+07 0.397722  
##   
## [5 rows x 9 columns]

Eliminar las columnas con .drop().

planets\_df.drop(columns="Area", inplace=True, errors="ignore")  
planets\_df.head()

## rotation\_period diameter ... population Densidad  
## name ...   
## Tatooine 23.0 10465.0 ... 2.000000e+05 0.000581  
## Alderaan 24.0 12500.0 ... 2.000000e+09 4.074367  
## Yavin IV 24.0 10200.0 ... 1.000000e+03 0.000003  
## Bespin 12.0 118000.0 ... 6.000000e+06 0.000137  
## Endor 18.0 4900.0 ... 3.000000e+07 0.397722  
##   
## [5 rows x 8 columns]

### FILTROS EN DATAFRAMES

Funciona la misma lógica de selección que numpys y series pandas, pero ahora con dos dimensiones.

planets\_df.loc[planets\_df.Densidad > 50, ['population', 'Densidad']]

## population Densidad  
## name   
## Geonosis 1.000000e+11 246.223329  
## Mon Cala 2.700000e+10 70.641981

Como curiosidad, el slicing y filtros lógicos, se realiza por filas.

planets\_df[planets\_df.Densidad > 50]

## rotation\_period diameter ... population Densidad  
## name ...   
## Geonosis 30.0 11370.0 ... 1.000000e+11 246.223329  
## Mon Cala 21.0 11030.0 ... 2.700000e+10 70.641981  
##   
## [2 rows x 8 columns]

Otra curiosidad. Ahora los filtros por posición, se comportan como en R, a diferencia de numpy.

a = planets\_df.iloc[[1,7,14],[1,2,4]]  
a

## diameter climate gravity  
## name   
## Alderaan 12500.0 temperate 1.0  
## Geonosis 11370.0 temperate, arid 0.9  
## Mon Cala 11030.0 temperate 1.0

En cambio, en numpy, se apareaban las coordenadas 2 a 2.

planets\_df.values[[1,4,5],[1,2,4]]

## array([12500.0, 'temperate', 1.0], dtype=object)

Continúa realizándose la asignación de vistas sin copia.

a = pd.DataFrame(np.ones((3,2)), columns=["col1","col2"])  
b = a.iloc[0:2,0:1]  
b

## col1  
## 0 1.0  
## 1 1.0

b.iloc[1,0] = 20  
a

## col1 col2  
## 0 1.0 1.0  
## 1 20.0 1.0  
## 2 1.0 1.0

**Recuerda**: Cuando sea necesario realiza la copia del objeto con .copy(), de esta forma eviatarás que se modifiquen todas las copias.

a = pd.DataFrame(np.ones((3,2)), columns=["col1","col2"])  
b = a.iloc[0:2,0:1].copy()  
b.iloc[1,0] = 20  
a

## col1 col2  
## 0 1.0 1.0  
## 1 1.0 1.0  
## 2 1.0 1.0

Ya hemos visto como ver las primeras 5 filas con .head(). También es posible ver las últimas filas con .tail().

planets\_df.tail(2)

## rotation\_period diameter ... population Densidad  
## name ...   
## Glee Anselm 33.0 15600.0 ... 5.000000e+08 0.653990  
## Muunilinst 28.0 13800.0 ... 5.000000e+09 8.357222  
##   
## [2 rows x 8 columns]

#### FUNCIONES UNIVERSALES SOBRE PANDAS

Las funciones universales (ufunc) que se han visto en el capítulo anterior, se pueden aplicar de igual modo sobre objetos pandas.

a = np.sqrt(planets\_df[["rotation\_period","diameter"]].tail(2))  
a

## rotation\_period diameter  
## name   
## Glee Anselm 5.744563 124.899960  
## Muunilinst 5.291503 117.473401

Vemos como en este caso se ha aplicado la función de numpy, raiz cuadrada, sobre un DataFrame y esta se ha aplicado sobre cada elmento respetando los índices fila y columna de la tabla.

#### ALINEAMIENTO DE LOS ÍNDICES

Cuando se realizan operaciones aritméticas, entre dos pandas, los índices fila y columna se alinean automáticamente.

a = planets\_df.iloc[[1,3,4],[4,7]]  
a

## gravity Densidad  
## name   
## Alderaan 1.00 4.074367  
## Bespin 1.50 0.000137  
## Endor 0.85 0.397722

b = planets\_df.iloc[[1,4,7],[4,7]]  
b

## gravity Densidad  
## name   
## Alderaan 1.00 4.074367  
## Endor 0.85 0.397722  
## Geonosis 0.90 246.223329

a + b

## gravity Densidad  
## name   
## Alderaan 2.0 8.148733  
## Bespin NaN NaN  
## Endor 1.7 0.795443  
## Geonosis NaN NaN

Cuando se resta un vector fila a un Data Frame, siguiendo las reglas del Broadcasting, esta operación se aplica a todas las filas.

a

## gravity Densidad  
## name   
## Alderaan 1.00 4.074367  
## Bespin 1.50 0.000137  
## Endor 0.85 0.397722

c = a.iloc[0]  
c

## gravity 1.000000  
## Densidad 4.074367  
## Name: Alderaan, dtype: float64

a - c

## gravity Densidad  
## name   
## Alderaan 0.00 0.000000  
## Bespin 0.50 -4.074229  
## Endor -0.15 -3.676645

#### VALORES FALTANTES

None y np.nan en Pandas tienen mucha similitud.

print(planets\_df.columns)

## Index(['rotation\_period', 'diameter', 'climate', 'temperate\_tropical',  
## 'gravity', 'surface\_water', 'population', 'Densidad'],  
## dtype='object')

planetsNA = planets\_df[["gravity", "climate", "temperate\_tropical"]].head().copy()  
planetsNA.head()

## gravity climate temperate\_tropical  
## name   
## Tatooine 1.00 arid 0  
## Alderaan 1.00 temperate 1  
## Yavin IV 1.00 temperate, tropical 1  
## Bespin 1.50 temperate 1  
## Endor 0.85 temperate 1

print("planetsNA tiene shape: ", planetsNA.shape)

## planetsNA tiene shape: (5, 3)

planetsNA.dtypes # vemos los tipos internos

## gravity float64  
## climate object  
## temperate\_tropical int32  
## dtype: object

Ahora inyectamos None y NaN.

planetsNA.loc[1:4,"gravity"] = None  
planetsNA.loc[0:2,"temperate\_tropical"] = np.nan  
planetsNA.loc[0:2,"climate"] = np.nan  
planetsNA.loc[2:4,"climate"] = None  
planetsNA

## gravity climate temperate\_tropical  
## name   
## Tatooine 1.00 NaN NaN  
## Alderaan NaN NaN NaN  
## Yavin IV NaN None 1.0  
## Bespin NaN None 1.0  
## Endor 0.85 temperate 1.0

planetsNA.dtypes

## gravity float64  
## climate object  
## temperate\_tropical float64  
## dtype: object

Observa que:

* gravity que era de tipo float, al insertar None, éste se ha convertido en np.nan.
* climate con tipo object, diferencia NaN y None.
* rotation\_period que inicialmente era de tipo entero (int), al asignar np.nan, hace el casting automáticamente a float

Cuatro métodos interesantes:

* isnull() / notnull(): Marca con True/False los valores faltantes.
* dropna(): Elimina filas y columnas con algún valor faltante.
* fillna(): Devuelve una copia de los datos con valores faltantes rellenados.

planetsNA.isnull() # np.nan y None se comportan igual.

## gravity climate temperate\_tropical  
## name   
## Tatooine False True True  
## Alderaan True True True  
## Yavin IV True True False  
## Bespin True True False  
## Endor False False False

planetsNA.dropna() # np.nan y None se comportan igual.

## gravity climate temperate\_tropical  
## name   
## Endor 0.85 temperate 1.0

planetsNA.fillna(0) # una forma de rellenar missings.

## gravity climate temperate\_tropical  
## name   
## Tatooine 1.00 0 0.0  
## Alderaan 0.00 0 0.0  
## Yavin IV 0.00 0 1.0  
## Bespin 0.00 0 1.0  
## Endor 0.85 temperate 1.0

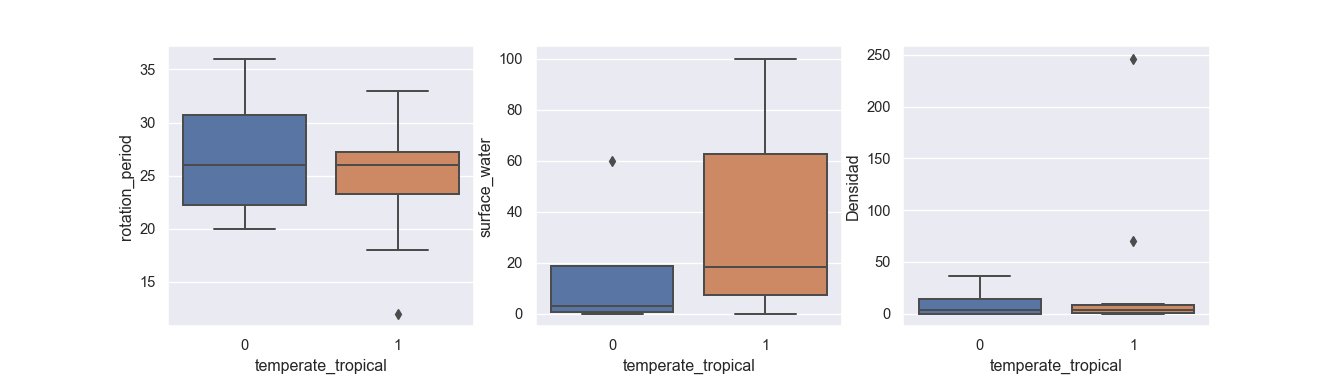
##### ACTIVIDAD GUIADA 2.4

Ahora tenemos todas las herramientas para resolver la actividad.

* Días largos para entrenar mucho
* Mucha agua para poder refrescarse
* Poca densidad de población para no ser molestado
* Buen clima (temperado o tropical)

Veamos su distribución con Boxplot.

sns.set(rc={'figure.figsize':(14,4)})  
fig, axs = plt.subplots(ncols=3)  
plt.gcf().subplots\_adjust(bottom=0.15)  
sns.boxplot(x='temperate\_tropical', y='rotation\_period', data=planets\_df, ax=axs[0])  
sns.boxplot(x='temperate\_tropical', y='surface\_water', data=planets\_df, ax=axs[1])  
sns.boxplot(x='temperate\_tropical', y='Densidad',data=planets\_df, ax=axs[2]);  
plt.show();



* Rotaciones por encima de 24, están por encima de la mediana (eje central boxplot).
* % de superficie del agua por encima de 10%, evita planetas no idóneos..
* Densidad por debajo de 50 personas por Km² es más que suficiente parar filtrar planetas demasiado poblados.

planets\_df\_sel = planets\_df[(planets\_df.Densidad<50)   
 & (planets\_df.rotation\_period>24)   
 & (planets\_df.surface\_water>10)]  
print(planets\_df\_sel.shape)

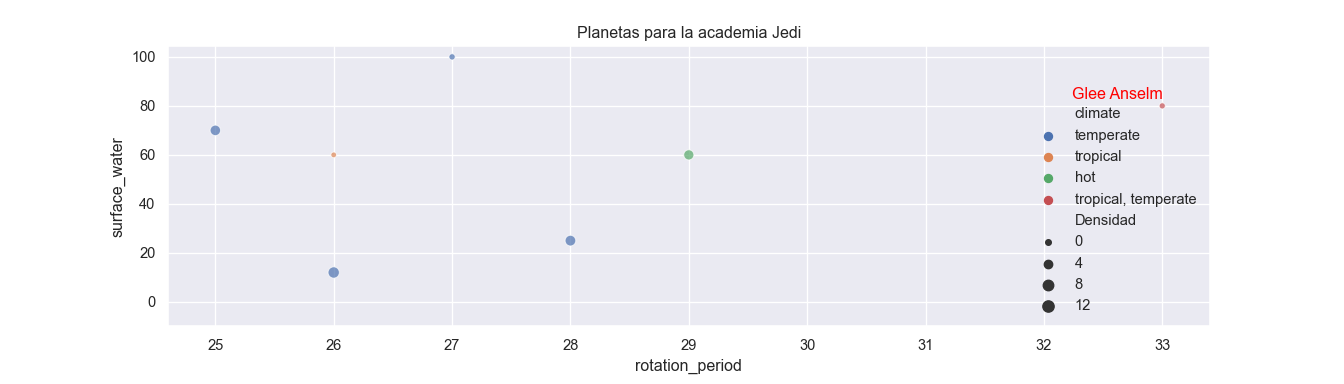
## (7, 8)

planets\_df\_sel

## rotation\_period diameter ... population Densidad  
## name ...   
## Naboo 26.0 12120.0 ... 4.500000e+09 9.751185  
## Kamino 27.0 19720.0 ... 1.000000e+09 0.818533  
## Kashyyyk 26.0 12765.0 ... 4.500000e+07 0.087907  
## Corellia 25.0 11000.0 ... 3.000000e+09 7.891981  
## Rodia 29.0 7549.0 ... 1.300000e+09 7.261304  
## Glee Anselm 33.0 15600.0 ... 5.000000e+08 0.653990  
## Muunilinst 28.0 13800.0 ... 5.000000e+09 8.357222  
##   
## [7 rows x 8 columns]

Quedan 7 planetas. Veamos ahora un gráfico de dispersión con un resumen de las principales variables.

sns.set(rc={'figure.figsize':(10,6)});  
g = sns.scatterplot(x="rotation\_period", y = "surface\_water", data=planets\_df\_sel  
 , alpha =0.7  
 , hue= "climate"  
 , size="Densidad")  
g.set\_title('Planetas para la academia Jedi');  
plt.legend(loc='lower right')  
limit = g.set(ylim=(-10, None));  
  
# buscamos el planeta que hay que etiquetar  
  
planets\_df\_sel\_final = planets\_df\_sel[(planets\_df\_sel.rotation\_period>30)&(planets\_df\_sel.surface\_water>60) ]  
  
# insertamos la etiqueta  
p = plt.annotate(planets\_df\_sel\_final.index[0],  
 (planets\_df\_sel\_final.rotation\_period[0],planets\_df\_sel\_final.surface\_water[0])  
 , textcoords="offset points" # desplaza el punto  
 , xytext=(0,5) # distancia desplazamiento  
 , ha='right'  
 , color="red") # centrado)  
plt.show();



Parece que ya tenemos destino perfecto para la academia Jedi! Se llama **Glee Anselm!!!**

Veamos sus datos.

planets\_df\_sel\_final

## rotation\_period diameter ... population Densidad  
## name ...   
## Glee Anselm 33.0 15600.0 ... 500000000.0 0.65399  
##   
## [1 rows x 8 columns]

## CONTROL DE FLUJO

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_40_contr_flujo.ipynb)

Las herramientas de control de flujo permiten automatizar tareas. En este capítulo se trabajarán las funciones condicionales y bucles.

##### DATOS DE EJEMPLO

Este capítulo, utilizara ejemplos de vehículos de Star Wars.

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set() # para el estilo de graficos  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
vehicles\_df = entidades\_df['vehicles'][["cost\_in\_credits","length","max\_atmosphering\_speed"  
 ,"crew","cargo\_capacity","pilots","films"]].dropna()  
  
vehicles\_df.head()

## cost\_in\_credits ... films  
## name ...   
## Sand Crawler 150000.0 ... [http://swapi.dev/api/films/1/, http://swapi.d...  
## T-16 skyhopper 14500.0 ... [http://swapi.dev/api/films/1/]  
## X-34 landspeeder 10550.0 ... [http://swapi.dev/api/films/1/]  
## Storm IV Twin-Pod cloud car 75000.0 ... [http://swapi.dev/api/films/2/]  
## Sail barge 285000.0 ... [http://swapi.dev/api/films/3/]  
##   
## [5 rows x 7 columns]

### FUNCIONES

Hagamos un breve repaso sobre lo que se ha tratado hasta el momento respecto a las funciones en Python.

Se ha introducido el concepto de función en Python en el capítulo de [Elementos básicos de Python](modulo1_tema4_Py_10_elem_bas.ipynb).

A continuación, en el capítulo de [Listas, tuplas y diccionarios](modulo1_tema4_Py_30_colec_obj.ipynb), se ha explicado la posibilidad de devolver varios elementos a la vez en forma de tuplas.

En [numpy](modulo1_tema4_Py_31_numpy.ipynb) se han explicado las funciones universales (ufunc), como una grupo de funciones que vectorizan al aplicarse sobre los elementos de los array numpy.

Finalmente, en [pandas](modulo1_tema4_Py_32_pandas.ipynb), se ha comentado la posibilidad de aplicar las funciones universales de numpy a objetos pandas (Series y DataFrame) y como estas funciones respetan los índices fila y columna. También se ha comentado el alineamiento de los índices cuando se realizan operaciones aritméticas entre dos objetos pandas.

En el capítulo de gestión de datos, se tratará 2 conceptos adicionales sobre las funciones.

Por un lado, se verá como vectorizar cualquier tipo de función, sobre los elementos de los objetos pandas, a través de la función apply.

**Recuerda**: En el tema de R las funciones apply ya se definieron con la misma finalidad.

El segundo concepto, van a ser las funciones de agregación (suma, media, cuantiles…). Éstas se definirán primero sobre objetos numpy y posteriormente sobre pandas.

La combinación de ambos conceptos, funciones vectorizadas y funciones de agregación sobre objetos pandas, van a terminar de rellenar tu caja de herramientas para poder iniciarte en data science.

### CONDICIONALES

Hagamos también aquí un breve repaso de los aspectos que ya se han comentado respecto a estructuras condicionales.

En el capítulo de los [Elementos básicos de Python](modulo1_tema4_Py_10_elem_bas.ipynb) se ha introducido la estructura básica *if-else*.

En el capítulo [Listas, tuplas y diccionarios](modulo1_tema4_Py_30_colec_obj.ipynb) se ha utilizado la escritura if-else en las *list comprehension*.

[x for x in range(5) if x%2==0] # lista de pares menores a 5

## [0, 2, 4]

Los capítulos de [Numpy](modulo1_tema4_Py_31_numpy.ipynb) y [Pandas](modulo1_tema4_Py_32_pandas.ipynb) también han servido para tratar los filtros, como un forma de operaciones condicionales sobre vectores y data frames.

Una nueva forma de realizar transformaciones condicionales con arrays numpy y series pandas es np.where().

Por ejemplo, veamos la siguiente variable *crew* que nos informa del número de tripulantes que puede llevar como máximo un vehículo.

vehicles\_df[['crew']].head()

## crew  
## name   
## Sand Crawler 46  
## T-16 skyhopper 1  
## X-34 landspeeder 1  
## Storm IV Twin-Pod cloud car 2  
## Sail barge 26

Para transformar esta variable en una nueva que indique si un vehículo puede llevar 1, 2 o 3 o más tripulantes.

vehicles\_df['crew\_r'] = np.where(vehicles\_df.crew<3, vehicles\_df.crew, 3)  
vehicles\_df[['crew','crew\_r']].head()

## crew crew\_r  
## name   
## Sand Crawler 46 3  
## T-16 skyhopper 1 1  
## X-34 landspeeder 1 1  
## Storm IV Twin-Pod cloud car 2 2  
## Sail barge 26 3

**Observa**: esta función de numpy aplica elemento a elemento. Observa también que se ha aplicado a una serie de un DataFrame, obteniendo una nueva serie.

### BUCLES

Los bucles for, se han tratado de forma básica en las *list comprehension* del capítulo de [Listas, tuplas y diccionarios](modulo1_tema4_Py_30_colec_obj.ipynb).

Una forma de definir bucles con for es a través de listas y diccionarios.

crew\_list = [1,2,3]  
for n\_crew in crew\_list:  
   
 n\_vehi = vehicles\_df[vehicles\_df.crew\_r == n\_crew].shape[0]  
   
 if n\_vehi == 1:  
 txt\_vehicles = 'vehículo'  
 else:  
 txt\_vehicles = 'vehiculos'  
   
 txt\_crew = 'tripulante'  
 if n\_crew != 1:  
 txt\_crew = txt\_crew + 's'  
   
 txt\_n\_crew = str(n\_crew)  
 if n\_crew==3:  
 txt\_n\_crew = txt\_n\_crew + ' o más'  
   
 print("Hay {} {} con {} {}".format(n\_vehi, txt\_vehicles, txt\_n\_crew, txt\_crew))

## Hay 8 vehiculos con 1 tripulante  
## Hay 3 vehiculos con 2 tripulantes  
## Hay 6 vehiculos con 3 o más tripulantes

Este bucle recorre la lista n\_crew y nos indica el número de vehículos con esa cantidad de tripulantes.

Cuando se recorre un diccionario con un for, por defecto, éste lo hace por sus claves.

vehiculos\_tripulantes = {'un': 8, 'dos': 3, '3 o más': 6}  
  
for x in vehiculos\_tripulantes:  
 print(x)

## un  
## dos  
## 3 o más

Se puede iterar sobre la tupla (clave,valor) con .items()

for txt\_n\_crew, n\_vehi in vehiculos\_tripulantes.items():  
 print("Hay",n\_vehi,"vehículo/s con",txt\_n\_crew, "tripulante/s")

## Hay 8 vehículo/s con un tripulante/s  
## Hay 3 vehículo/s con dos tripulante/s  
## Hay 6 vehículo/s con 3 o más tripulante/s

Los bucles con numpy, tienen una lógica simétrica a las listas.

Para poder recorrer 2 colecciones de objetos (listas, diccionarios, numpy, series…) con el mismo número de elementos de forma correlativa se utiliza la función zip().

a = [1,2,3]  
b = [8,3,6]  
  
for x,y in zip(a,b):  
 print("Hay",y,"vehículo/s con",[x if x<3 else '3 o más'][0], "tripulante/s")

## Hay 8 vehículo/s con 1 tripulante/s  
## Hay 3 vehículo/s con 2 tripulante/s  
## Hay 6 vehículo/s con 3 o más tripulante/s

Para iterar **por columnas** de un DataFrame se utiliza .iteritems(). En cada iteración se obtiene la tupla(nombre columna, valor serie).

for nombreCol,variable in vehicles\_df.iteritems():  
 print(nombreCol,", tiene tipo:",variable.dtype)

## cost\_in\_credits , tiene tipo: float64  
## length , tiene tipo: float64  
## max\_atmosphering\_speed , tiene tipo: float64  
## crew , tiene tipo: int64  
## cargo\_capacity , tiene tipo: float64  
## pilots , tiene tipo: object  
## films , tiene tipo: object  
## crew\_r , tiene tipo: int64

De la misma forma, pero ahora iterando un DataFrame **por filas** con .iterrows().

for nombre,fila in vehicles\_df.iloc[0:2,0:2].iterrows():  
 print(nombre,", cuesta:",fila.cost\_in\_credits," y mide:",fila.length , "metros")

## Sand Crawler , cuesta: 150000.0 y mide: 36.8 metros  
## T-16 skyhopper , cuesta: 14500.0 y mide: 10.4 metros

## GESTIÓN DE DATOS

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_50_gest_dat.ipynb)

A continuación, se presentan las funciones para la lectura/escritura de datos, cruce y construcción de tablas resumen.

Al final del capítulo se presenta la forma de manejar datos temporales.

##### ACTIVIDAD GUIADA 2.5

Se trata de analizar los personajes de la serie:

«Preferiría ser un monstruo que cree en algo, que sacrificaría todo para mejorar la galaxia, que ser alguien que se quede al margen y mire como si no tuviera repercusión en ellos.»

―Princesa Leia Organa

Esta actividad consiste en cruzar datos de personajes y planetas para construir descriptivos resumen de los datos de personajes.

El primer paso consiste en cargar los datos de los personajes y planetas.

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set() # para el estilo de graficos  
  
entidades = ['planets','starships','vehicles','people','species']  
entidades\_df = {x: pd.read\_pickle('www/' + x + '\_df.pkl') for x in entidades}  
  
# planetas  
planets\_df = entidades\_df['planets'][["climate","temperate\_tropical","population","url"]].dropna()  
  
# Datos principales  
people\_df = entidades\_df['people'][["height","mass","eye\_color","birth\_year","gender","homeworld"]].dropna()  
  
people\_df.head()

## height mass ... gender homeworld  
## name ...   
## Luke Skywalker 172.0 77.0 ... male http://swapi.dev/api/planets/1/  
## C-3PO 167.0 75.0 ... none http://swapi.dev/api/planets/1/  
## R2-D2 96.0 32.0 ... none http://swapi.dev/api/planets/8/  
## Darth Vader 202.0 136.0 ... male http://swapi.dev/api/planets/1/  
## Leia Organa 150.0 49.0 ... female http://swapi.dev/api/planets/2/  
##   
## [5 rows x 6 columns]

planets\_df.head()

## climate ... url  
## name ...   
## Tatooine arid ... http://swapi.dev/api/planets/1/  
## Alderaan temperate ... http://swapi.dev/api/planets/2/  
## Yavin IV temperate, tropical ... http://swapi.dev/api/planets/3/  
## Bespin temperate ... http://swapi.dev/api/planets/6/  
## Endor temperate ... http://swapi.dev/api/planets/7/  
##   
## [5 rows x 4 columns]

### IMPORTAR Y EXPORTAR DATOS

La forma más simple de importar datos estructurados (en forma de matriz de filas y columnas), es a través de los DataFrames. El motivo es simple, estos objetos permiten almacenar datos de distinto tipo en un único objeto o tabla de datos.

#### LECTURA TEXTO CON SEPARADOR

Para leer el siguiente fichero de texto:

# mostramos las 5 primeras filas  
n=5  
with open('www/mtcars.csv') as f:  
 muestra\_texto = ""  
 for i in range(5):  
 muestra\_texto +=f.readline()  
print(muestra\_texto)

## "mpg","cyl","disp","hp","drat","wt","qsec","vs","am","gear","carb"  
## 21,6,160,110,3.9,2.62,16.46,0,1,4,4  
## 21,6,160,110,3.9,2.875,17.02,0,1,4,4  
## 22.8,4,108,93,3.85,2.32,18.61,1,1,4,1  
## 21.4,6,258,110,3.08,3.215,19.44,1,0,3,1

Dado que se trata de un fichero con un carácter separado, la función genérica para este tipo de ficheros es pd.read\_table().

mtcars = pd.read\_table("www/mtcars.csv",sep=',', decimal=".")  
mtcars.head()

## mpg cyl disp hp drat wt qsec vs am gear carb  
## 0 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4  
## 1 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4  
## 2 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1  
## 3 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1  
## 4 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

Se puede personalizar la carga con los siguiente parámetros:

* decimal: El separador decimal.
* sep: El separador de columnas

También es común el uso de enconding="latin\_1" cuando el fichero se ha creado con Windows.

Dada la estructura del fichero, con pd.read\_csv() la carga es más simple.

mtcars = pd.read\_csv("www/mtcars.csv")

A parte de .head(), es una buena práctica verificar su carga con .shape(), y describe().

mtcars.shape

## (32, 11)

mtcars.describe()

## mpg cyl disp ... am gear carb  
## count 32.000000 32.000000 32.000000 ... 32.000000 32.000000 32.0000  
## mean 20.090625 6.187500 230.721875 ... 0.406250 3.687500 2.8125  
## std 6.026948 1.785922 123.938694 ... 0.498991 0.737804 1.6152  
## min 10.400000 4.000000 71.100000 ... 0.000000 3.000000 1.0000  
## 25% 15.425000 4.000000 120.825000 ... 0.000000 3.000000 2.0000  
## 50% 19.200000 6.000000 196.300000 ... 0.000000 4.000000 2.0000  
## 75% 22.800000 8.000000 326.000000 ... 1.000000 4.000000 4.0000  
## max 33.900000 8.000000 472.000000 ... 1.000000 5.000000 8.0000  
##   
## [8 rows x 11 columns]

#### ESCRITURA TEXTO CON SEPARADOR

Para la escritura, puedes utilizar sus equivalentes: .to\_csv(). Cabe destacar algunas pequeñas variaciones.

mtcars.to\_csv("www/mtcars2.csv", decimal=",", sep=";", index=False, encoding="latin\_1")

* index: Campo lógico True, False. Por defecto, inserta el número de fila.

Observa como el nuevo csv ahora tiene el formato de csv europeo con codificación windows.

# mostramos las 5 primeras filas  
n=5  
with open('www/mtcars2.csv') as f:  
 muestra\_texto = ""  
 for i in range(5):  
 muestra\_texto +=f.readline()  
print(muestra\_texto)

## mpg;cyl;disp;hp;drat;wt;qsec;vs;am;gear;carb  
## 21,0;6;160,0;110;3,9;2,62;16,46;0;1;4;4  
## 21,0;6;160,0;110;3,9;2,875;17,02;0;1;4;4  
## 22,8;4;108,0;93;3,85;2,32;18,61;1;1;4;1  
## 21,4;6;258,0;110;3,08;3,215;19,44;1;0;3;1

Para leer texto con ancho fijo revisa la función [pd.read\_fwf()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_fwf.html).

#### LECTURA Y ESCRITURA EN FORMATO PICKLE

Los objetos Python se pueden hacer persistentes en disco con el formato pickle.

**Sabías que**: pickle significa pepinillo o vinagreta. Es decir, que el formato pickle significa que estamos poniendo los objetos Python en conserva.

Para guardar un DataFrame en formato pickle utiliza el método .to\_pickle("nombre.pkl").

mtcars.to\_pickle("www/mtcars.pkl")

!dir www/mtcars\*  
  
## www/mtcars2.csv www/mtcars.csv www/mtcars.pkl

Para cargar un fichero pickle (sacarlo de la conserva), se utiliza pd.read\_pickle().

mtcars\_pkl = pd.read\_pickle("www/mtcars.pkl")  
mtcars\_pkl.shape

## (32, 11)

### CRUCE ENTRE TABLAS

Para definir un cruce en pandas, es muy importante ser conscientes de la presencia de los índices fila y columna de las tablas origen. En este sentido se puede realizar el cruce por 3 sistemas distintos:

* Por índices
* Por ordenación
* Por campos clave

Veamos los dos últimos, ya que el cruce por índices es la opción por defecto en todas las funciones que se van a mostrar.

#### CRUCES POR ORDENACIÓN SIN ÍNDICE COMPARTIDO

Una forma de cruzar 2 DataFrames es a partir de la ordenación compartida, ya sea de las filas o las columnas.

##### ORDENACIÓN DE TABLAS

Para ordenar una serie, se utiliza el método .sort.values().

a = people\_df.birth\_year.sort\_values(ascending=False)  
a.head()

## name  
## Yoda 896.0  
## Jabba Desilijic Tiure 600.0  
## Chewbacca 200.0  
## C-3PO 112.0  
## Dooku 102.0  
## Name: birth\_year, dtype: float64

* ascending: Campo lógico. Por defecto ordena de forma ascendente.

En DataFrames, no es muy distinto, pero ahora hay que indicar el campo de ordenación en el parámetro by.

people\_df\_Ord = people\_df.sort\_values(by=["gender","height"], ascending=[True, False])  
people\_df\_Ord.head()

## height mass ... gender homeworld  
## name ...   
## Padmé Amidala 185.0 45.0 ... female http://swapi.dev/api/planets/8/  
## Ayla Secura 178.0 55.0 ... female http://swapi.dev/api/planets/37/  
## Luminara Unduli 170.0 56.2 ... female http://swapi.dev/api/planets/51/  
## Barriss Offee 166.0 50.0 ... female http://swapi.dev/api/planets/51/  
## Beru Whitesun lars 165.0 75.0 ... female http://swapi.dev/api/planets/1/  
##   
## [5 rows x 6 columns]

Observa como ascending permite escoger una ordenación distinta para cada campo de ordenación.

De esta forma sabemos que *Padmé Amidala*, con 185 cm, era la mujer más alta de la serie.

##### UNIÓN POR COLUMNAS SIN ÍNDICE COMPARTIDO

Para unir las columnas de dos DataFrames que comparten ordenación utiliza pd.concat().

Para ver un ejemplo, antes vamos a preparar 2 tablas de ejemplo. Para la primera, seleccionamos las 4 primeras columnas y reseteamos el índice con .reset\_index().

people\_df1 = people\_df.iloc[:,:4].reset\_index()  
people\_df1.head()

## name height mass eye\_color birth\_year  
## 0 Luke Skywalker 172.0 77.0 blue 19.0  
## 1 C-3PO 167.0 75.0 yellow 112.0  
## 2 R2-D2 96.0 32.0 red 33.0  
## 3 Darth Vader 202.0 136.0 yellow 41.9  
## 4 Leia Organa 150.0 49.0 brown 19.0

**Importante**: Al resetar el índice con .reset\_index(), éste se ha convertido en un nuevo campo **name** de la tabla.

Ahora la segunda tabla con el resto de campos.

people\_df2 = people\_df.iloc[:,4:]  
people\_df2.head()

## gender homeworld  
## name   
## Luke Skywalker male http://swapi.dev/api/planets/1/  
## C-3PO none http://swapi.dev/api/planets/1/  
## R2-D2 none http://swapi.dev/api/planets/8/  
## Darth Vader male http://swapi.dev/api/planets/1/  
## Leia Organa female http://swapi.dev/api/planets/2/

Las dos tablas people\_df1 y people\_df2, comparten ordenación, pero no un mismo índice. Para realizar el cruce ignore\_index=True evita que la función utilice los índices.

a = pd.concat([people\_df1, people\_df2], ignore\_index=True)  
a.head()

## name height mass eye\_color birth\_year gender homeworld  
## 0 Luke Skywalker 172.0 77.0 blue 19.0 NaN NaN  
## 1 C-3PO 167.0 75.0 yellow 112.0 NaN NaN  
## 2 R2-D2 96.0 32.0 red 33.0 NaN NaN  
## 3 Darth Vader 202.0 136.0 yellow 41.9 NaN NaN  
## 4 Leia Organa 150.0 49.0 brown 19.0 NaN NaN

###### UNIÓN POR FILAS

Para unir dos DataFrames con la misma ordenación de sus columnas, se puede utilizar pd.concat, aunque es más directo con el método .append().

a1 = people\_df.iloc[0:2,:2]  
a2 = people\_df.iloc[3:5,:2]  
a = a1.append(a2)  
a1

## height mass  
## name   
## Luke Skywalker 172.0 77.0  
## C-3PO 167.0 75.0

a2

## height mass  
## name   
## Darth Vader 202.0 136.0  
## Leia Organa 150.0 49.0

a

## height mass  
## name   
## Luke Skywalker 172.0 77.0  
## C-3PO 167.0 75.0  
## Darth Vader 202.0 136.0  
## Leia Organa 150.0 49.0

**Importante**: en caso de no compartir los mismos nombres de columnas, se puede utilizar igualmente el parámetro ignore\_index=True.

#### CRUCE POR CAMPOS CLAVE

Antes de realizar un cruce por campos clave es importante saber si éste tiene valores duplicados.

##### VECTOR DE CLAVES

Para obtener las claves únicas de una serie o DataFrame utiliza .unique().

a = people\_df.homeworld.unique()  
a

## array(['http://swapi.dev/api/planets/1/',  
## 'http://swapi.dev/api/planets/8/',  
## 'http://swapi.dev/api/planets/2/',  
## 'http://swapi.dev/api/planets/20/',  
## 'http://swapi.dev/api/planets/14/',  
## 'http://swapi.dev/api/planets/22/',  
## 'http://swapi.dev/api/planets/23/',  
## 'http://swapi.dev/api/planets/24/',  
## 'http://swapi.dev/api/planets/28/',  
## 'http://swapi.dev/api/planets/10/',  
## 'http://swapi.dev/api/planets/29/',  
## 'http://swapi.dev/api/planets/30/',  
## 'http://swapi.dev/api/planets/6/',  
## 'http://swapi.dev/api/planets/31/',  
## 'http://swapi.dev/api/planets/7/',  
## 'http://swapi.dev/api/planets/36/',  
## 'http://swapi.dev/api/planets/37/',  
## 'http://swapi.dev/api/planets/42/',  
## 'http://swapi.dev/api/planets/43/',  
## 'http://swapi.dev/api/planets/49/',  
## 'http://swapi.dev/api/planets/51/',  
## 'http://swapi.dev/api/planets/52/',  
## 'http://swapi.dev/api/planets/53/'], dtype=object)

Para saber qué filas de un DataFrame están duplicadas, utiliza .duplicated().

people\_df\_dup = people\_df.iloc[[1,1,2,3,3,4,5,5,6],:]  
people\_df\_dup.duplicated()

## name  
## C-3PO False  
## C-3PO True  
## R2-D2 False  
## Darth Vader False  
## Darth Vader True  
## Leia Organa False  
## Owen Lars False  
## Owen Lars True  
## Beru Whitesun lars False  
## dtype: bool

**Observa**: la función .duplicated() marca como False la primera copia y como True la segunda copia.

Para eliminar los duplicados, se utiliza .drop\_duplicates(). Con keep=False se elimina todas filas con duplicados.

people\_df\_dup.drop\_duplicates(keep=False)

## height mass ... gender homeworld  
## name ...   
## R2-D2 96.0 32.0 ... none http://swapi.dev/api/planets/8/  
## Leia Organa 150.0 49.0 ... female http://swapi.dev/api/planets/2/  
## Beru Whitesun lars 165.0 75.0 ... female http://swapi.dev/api/planets/1/  
##   
## [3 rows x 6 columns]

##### FUSIÓN CON CLAVES

Para cruzar 2 tablas con claves, utiliza pd.merge().

###### INNER JOIN

Recordemos que inner join consiste en construir una tabla que tenga claves coincidentes en las tablas.

Para ver un ejemplo, vamos a seleccionar, por un lado planetas con un clima malo, es decir, que no sea temperado o tropical.

planets\_clima\_df = planets\_df[planets\_df.temperate\_tropical==0]  
planets\_clima\_df

## climate ... url  
## name ...   
## Tatooine arid ... http://swapi.dev/api/planets/1/  
## Mustafar hot ... http://swapi.dev/api/planets/13/  
## Mygeeto frigid ... http://swapi.dev/api/planets/16/  
## Felucia hot, humid ... http://swapi.dev/api/planets/17/  
## Saleucami hot ... http://swapi.dev/api/planets/19/  
## Eriadu polluted ... http://swapi.dev/api/planets/21/  
## Rodia hot ... http://swapi.dev/api/planets/23/  
## Trandosha arid ... http://swapi.dev/api/planets/29/  
## Socorro arid ... http://swapi.dev/api/planets/30/  
## Sullust superheated ... http://swapi.dev/api/planets/33/  
## Tund unknown ... http://swapi.dev/api/planets/41/  
## Ojom frigid ... http://swapi.dev/api/planets/55/  
##   
## [12 rows x 4 columns]

Por otro lado, seleccionamos personajes con color de ojos azul.

people\_eyes\_df = people\_df[people\_df.eye\_color=="blue"]  
people\_eyes\_df

## height mass ... gender homeworld  
## name ...   
## Luke Skywalker 172.0 77.0 ... male http://swapi.dev/api/planets/1/  
## Owen Lars 178.0 120.0 ... male http://swapi.dev/api/planets/1/  
## Beru Whitesun lars 165.0 75.0 ... female http://swapi.dev/api/planets/1/  
## Anakin Skywalker 188.0 84.0 ... male http://swapi.dev/api/planets/1/  
## Chewbacca 228.0 112.0 ... male http://swapi.dev/api/planets/14/  
## Lobot 175.0 79.0 ... male http://swapi.dev/api/planets/6/  
## Qui-Gon Jinn 193.0 89.0 ... male http://swapi.dev/api/planets/28/  
## Luminara Unduli 170.0 56.2 ... female http://swapi.dev/api/planets/51/  
## Barriss Offee 166.0 50.0 ... female http://swapi.dev/api/planets/51/  
##   
## [9 rows x 6 columns]

El campo coincidente clave para el cruce es url que equivale al campo homeworld de people\_df.

a\_inner = pd.merge(people\_eyes\_df, planets\_clima\_df, left\_on=["homeworld"], right\_on=["url"])  
a\_inner.head()

## height mass ... population url  
## 0 172.0 77.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 1 178.0 120.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 2 165.0 75.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 3 188.0 84.0 ... 200000.0 http://swapi.dev/api/planets/1/  
##   
## [4 rows x 10 columns]

Vemos que hay cuatro personajes de ojos azules que nacieron en planetas con mal clima. El problema es que con el cruce se han perdido los nombres de ambas tablas contenidos en los índices. Rescatemos antes del cruce con la función reset\_index().

people\_eyes\_df = people\_eyes\_df.reset\_index()  
planets\_clima\_df = planets\_clima\_df.reset\_index()  
  
a\_inner = pd.merge(people\_eyes\_df, planets\_clima\_df, left\_on=["homeworld"], right\_on=["url"])  
a\_inner.head()

## name\_x height ... population url  
## 0 Luke Skywalker 172.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 1 Owen Lars 178.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 2 Beru Whitesun lars 165.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 3 Anakin Skywalker 188.0 ... 200000.0 http://swapi.dev/api/planets/1/  
##   
## [4 rows x 12 columns]

**Observa**: el campo name estaba duplicado en ambas tablas sin ser campo clave. pd.merge() ha incluido ambos en la tabla final añadiendo el sufijo \_x para tabla la izquierda (people\_df) y \_y tabla derecha (planets\_df).

El cruce se ha hecho de forma implícita con la opción how="inner".

###### OUTER JOIN

Para conservar **todos los registros de las tablas originales**, tanto de la tabla izquierda como derecha, coincida o no, se utiliza el parámetro how = "outer".

a\_outer = pd.merge(people\_eyes\_df, planets\_clima\_df, how = "outer", left\_on=["homeworld"], right\_on=["url"])  
a\_outer

## name\_x height ... population url  
## 0 Luke Skywalker 172.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 1 Owen Lars 178.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 2 Beru Whitesun lars 165.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 3 Anakin Skywalker 188.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 4 Chewbacca 228.0 ... NaN NaN  
## 5 Lobot 175.0 ... NaN NaN  
## 6 Qui-Gon Jinn 193.0 ... NaN NaN  
## 7 Luminara Unduli 170.0 ... NaN NaN  
## 8 Barriss Offee 166.0 ... NaN NaN  
## 9 NaN NaN ... 2.000000e+04 http://swapi.dev/api/planets/13/  
## 10 NaN NaN ... 1.900000e+07 http://swapi.dev/api/planets/16/  
## 11 NaN NaN ... 8.500000e+06 http://swapi.dev/api/planets/17/  
## 12 NaN NaN ... 1.400000e+09 http://swapi.dev/api/planets/19/  
## 13 NaN NaN ... 2.200000e+10 http://swapi.dev/api/planets/21/  
## 14 NaN NaN ... 1.300000e+09 http://swapi.dev/api/planets/23/  
## 15 NaN NaN ... 4.200000e+07 http://swapi.dev/api/planets/29/  
## 16 NaN NaN ... 3.000000e+08 http://swapi.dev/api/planets/30/  
## 17 NaN NaN ... 1.850000e+10 http://swapi.dev/api/planets/33/  
## 18 NaN NaN ... 0.000000e+00 http://swapi.dev/api/planets/41/  
## 19 NaN NaN ... 5.000000e+08 http://swapi.dev/api/planets/55/  
##   
## [20 rows x 12 columns]

Observa ahora, no ha descartado ningún registro de las tablas origen. Además, ha rellenado con NaN los campos que no cruzan. Este comportamiento en SQL es conocido como *FULL JOIN* o *OUTTER JOIN* .

###### LEFT JOIN

Para imponer que conserve **todos los valores origen de la tabla izquierda** y que descarte los de la taba derecha que no sean compartidos utiliza how="left".

a\_left = pd.merge(people\_eyes\_df, planets\_clima\_df, how="left", left\_on=["homeworld"], right\_on=["url"])  
a\_left

## name\_x height ... population url  
## 0 Luke Skywalker 172.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 1 Owen Lars 178.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 2 Beru Whitesun lars 165.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 3 Anakin Skywalker 188.0 ... 200000.0 http://swapi.dev/api/planets/1/  
## 4 Chewbacca 228.0 ... NaN NaN  
## 5 Lobot 175.0 ... NaN NaN  
## 6 Qui-Gon Jinn 193.0 ... NaN NaN  
## 7 Luminara Unduli 170.0 ... NaN NaN  
## 8 Barriss Offee 166.0 ... NaN NaN  
##   
## [9 rows x 12 columns]

Ahora ha conservado todos los personajes.

###### RIGHT JOIN

Para preservar **todos los valores de la tabla derecha** utiliza how="right".

a\_right = pd.merge(people\_eyes\_df, planets\_clima\_df, how="right", left\_on=["homeworld"], right\_on=["url"])  
a\_right

## name\_x height ... population url  
## 0 Luke Skywalker 172.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 1 Owen Lars 178.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 2 Beru Whitesun lars 165.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 3 Anakin Skywalker 188.0 ... 2.000000e+05 http://swapi.dev/api/planets/1/  
## 4 NaN NaN ... 2.000000e+04 http://swapi.dev/api/planets/13/  
## 5 NaN NaN ... 1.900000e+07 http://swapi.dev/api/planets/16/  
## 6 NaN NaN ... 8.500000e+06 http://swapi.dev/api/planets/17/  
## 7 NaN NaN ... 1.400000e+09 http://swapi.dev/api/planets/19/  
## 8 NaN NaN ... 2.200000e+10 http://swapi.dev/api/planets/21/  
## 9 NaN NaN ... 1.300000e+09 http://swapi.dev/api/planets/23/  
## 10 NaN NaN ... 4.200000e+07 http://swapi.dev/api/planets/29/  
## 11 NaN NaN ... 3.000000e+08 http://swapi.dev/api/planets/30/  
## 12 NaN NaN ... 1.850000e+10 http://swapi.dev/api/planets/33/  
## 13 NaN NaN ... 0.000000e+00 http://swapi.dev/api/planets/41/  
## 14 NaN NaN ... 5.000000e+08 http://swapi.dev/api/planets/55/  
##   
## [15 rows x 12 columns]

Ahora ha conservado todos los planetas con mal clima.

###### DEFINCIÓN DE LAS CLAVES

Cuando las claves son campos comunes no es necesario definir los campos left\_on y right\_on.

Por otro lado, si uno de los campos clave está contenido en el índice del DataFrame, se puede utilizar el parámetro left\_index=True o right\_index=True o ambos.

### RESUMENES DE AGREGADOS

Las librerías numpy y pandas implementan una extensa colección de funciones de resumen.

#### DESCRIBE

Para tener una primera impresión sobre las estadísticas de un DataFrame se utiliza .describe().

Antes incluyamos a people\_df algunas variables nuevas de distinto tipo y además algunos missings.

people\_dfSumm = people\_df.copy()  
  
people\_dfSumm.loc[people\_dfSumm.index[[0,2,4,8,20]],'height'] = np.nan  
people\_dfSumm.loc[people\_dfSumm.index[[0,2,4,8,20]],'eye\_color'] = np.nan  
people\_dfSumm["Alto"] = people\_dfSumm.height>188  
people\_dfSumm["Fecha\_hoy"] = np.datetime64('2020-09-06')  
  
people\_dfSumm.head()

## height mass ... Alto Fecha\_hoy  
## name ...   
## Luke Skywalker NaN 77.0 ... False 2020-09-06  
## C-3PO 167.0 75.0 ... False 2020-09-06  
## R2-D2 NaN 32.0 ... False 2020-09-06  
## Darth Vader 202.0 136.0 ... True 2020-09-06  
## Leia Organa NaN 49.0 ... False 2020-09-06  
##   
## [5 rows x 8 columns]

people\_df.describe()

## height mass birth\_year  
## count 36.000000 36.000000 36.000000  
## mean 173.638889 112.233333 91.091667  
## std 30.941560 215.205026 169.124347  
## min 66.000000 17.000000 8.000000  
## 25% 170.000000 72.000000 31.375000  
## 50% 179.000000 79.000000 46.500000  
## 75% 188.000000 84.000000 67.500000  
## max 228.000000 1358.000000 896.000000

Por defecto, calcula los siguientes estadísticos básicos sobre variables numéricas:

* count: Número de valores informados (distintos de NaN).
* mean: Media.
* str: Desviación típica.
* min: Mínimo.
* 25%: Cuantil 25%.
* 50%: Cuantil 50% o mediana.
* 75%: Cuantil 75%.
* max: Máximo.

Para incluirlas todas se utiliza include='all'.

people\_dfSumm.describe(include='all')

## height mass ... Alto Fecha\_hoy  
## count 31.000000 36.000000 ... 36 36  
## unique NaN NaN ... 2 1  
## top NaN NaN ... False 2020-09-06 00:00:00  
## freq NaN NaN ... 28 36  
## first NaN NaN ... NaN 2020-09-06 00:00:00  
## last NaN NaN ... NaN 2020-09-06 00:00:00  
## mean 176.580645 112.233333 ... NaN NaN  
## std 29.759899 215.205026 ... NaN NaN  
## min 66.000000 17.000000 ... NaN NaN  
## 25% 171.500000 72.000000 ... NaN NaN  
## 50% 180.000000 79.000000 ... NaN NaN  
## 75% 189.000000 84.000000 ... NaN NaN  
## max 228.000000 1358.000000 ... NaN NaN  
##   
## [13 rows x 8 columns]

Ahora, para las variables no numéricas, informa del count y además:

* unique: Número de valores únicos.
* top: Valor más frecuente.
* freq: Frecuencia observada del valor top.
* first y last: Para las datetime se informa de la primera y última fecha.

#### ESTADÍSTICOS BASICOS

Para calcular los estadísticos mostrados en la función .describe() con funciones numpy:

def resumen\_numericas\_numpy(x):  
 return {  
 "count":np.sum(~np.isnan(x))  
 ,"mean":np.nanmean(x)  
 ,"std":np.nanstd(x)  
 ,"min":np.min(x)  
 ,"quantile 25, 50, 75":np.nanquantile(x, [0.25,.5,.75])  
 ,"max":np.max(x)}  
   
resumen\_numericas\_numpy(people\_dfSumm.height)

## {'count': 31, 'mean': 176.58064516129033, 'std': 29.275966355278175, 'min': 66.0, 'quantile 25, 50, 75': array([171.5, 180. , 189. ]), 'max': 228.0}

**Importante**: Las funciones equivalentes np.mean, np.std o np.quantile devuelven NaN cuando el array contienen algún valor NaN.

Ahora con funciones equivalentes en pandas:

def resumen\_numericas\_pandas(x):  
 return {  
 "count":x.count()  
 ,"mean":x.mean()  
 ,"std":x.std()  
 ,"min":x.min()  
 ,"quantile 25, 50, 75":x.quantile([0.25,.5,.75])  
 ,"max":x.max()}  
   
resumen\_numericas\_pandas(people\_dfSumm.height)

## {'count': 31, 'mean': 176.58064516129033, 'std': 29.759899410166454, 'min': 66.0, 'quantile 25, 50, 75': 0.25 171.5  
## 0.50 180.0  
## 0.75 189.0  
## Name: height, dtype: float64, 'max': 228.0}

Observa que la opción por defecto en pandas, es eliminar los NaN.

def resumen\_no\_numericas\_numpy(x):  
 value, counts = np.unique(x[~pd.isnull(x)], return\_counts=True)  
 return {  
 "count":np.sum(~pd.isnull(x))  
 ,"unique": len(np.unique(x[~pd.isnull(x)]))  
 ,"top": value[np.argmax(counts)]  
 ,"freq":np.max(counts)}  
  
resumen\_no\_numericas\_numpy(people\_dfSumm.eye\_color)

## {'count': 31, 'unique': 7, 'top': 'brown', 'freq': 9}

**Importante**: En variables no numéricas se ha filtrado los valores None o NaN con pd.isnull().

Ahora con funciones pandas.

def resumen\_no\_numericas\_pandas(x):  
 value, counts = np.unique(x[~pd.isnull(x)], return\_counts=True)  
 return {  
 "count":x.count()  
 ,"unique": x.nunique()  
 ,"top": list(x.mode())[0]  
 ,"freq":(x==list(x.mode())[0]).sum()}  
  
resumen\_no\_numericas\_pandas(people\_dfSumm.eye\_color)

## {'count': 31, 'unique': 7, 'top': 'brown', 'freq': 9}

#### TABLAS DE FRECUENCIAS

##### FRECUENCIAS DE UNA VARIABLE

Para obtener las frecuencias de una serie pandas se utiliza .value\_counts().

people\_dfSumm.eye\_color.value\_counts()

## brown 9  
## blue 8  
## yellow 5  
## orange 3  
## hazel 2  
## red 2  
## black 2  
## Name: eye\_color, dtype: int64

Por defecto, ordena de más a menos frecuente.

Se puede ordenarlo posteriormente por el índice con .sort\_index().

people\_dfSumm.eye\_color.value\_counts().sort\_index()

## black 2  
## blue 8  
## brown 9  
## hazel 2  
## orange 3  
## red 2  
## yellow 5  
## Name: eye\_color, dtype: int64

##### TABLAS CRUZADAS

Para obtener las frecuencias de dos variables utiliza pd.crosstab().

pd.crosstab(people\_dfSumm.eye\_color,people\_dfSumm.gender)

## gender female hermaphrodite male none  
## eye\_color   
## black 0 0 2 0  
## blue 3 0 5 0  
## brown 1 0 8 0  
## hazel 1 0 1 0  
## orange 0 1 2 0  
## red 0 0 1 1  
## yellow 0 0 4 1

##### TRAMEAR VARIABLES

En pandas series se puede tramear una variable con la función pd.cut()

people\_dfSumm["Altura\_Cat"] = pd.cut(people\_dfSumm.height,[-np.infty,171,189,np.infty], right=False)  
  
people\_dfSumm[["height","Altura\_Cat"]].head(8)

## height Altura\_Cat  
## name   
## Luke Skywalker NaN NaN  
## C-3PO 167.0 [-inf, 171.0)  
## R2-D2 NaN NaN  
## Darth Vader 202.0 [189.0, inf)  
## Leia Organa NaN NaN  
## Owen Lars 178.0 [171.0, 189.0)  
## Beru Whitesun lars 165.0 [-inf, 171.0)  
## Biggs Darklighter 183.0 [171.0, 189.0)

Observa como pd.cut() ha trameado en intervalos la variable height. Con right=False se le pide que el intervalo sea cerrado a la izquierda y abierto a la derecha.

Las frecuencias de cada categoría.

people\_dfSumm["Altura\_Cat"].value\_counts().sort\_index()

## [-inf, 171.0) 8  
## [171.0, 189.0) 15  
## [189.0, inf) 8  
## Name: Altura\_Cat, dtype: int64

**Importante**: pd.cut() crea una variable de tipo pd.Categorical. Este tipo de datos contienen .categories() y .codes(), parecidos a los factores vistos en R.

people\_dfSumm["Altura\_Cat"].dtype

## CategoricalDtype(categories=[[-inf, 171.0), [171.0, 189.0), [189.0, inf)],  
## ordered=True)

Para recuperar las categorías, utiliza .cat.categories

people\_dfSumm["Altura\_Cat"].cat.categories

## IntervalIndex([[-inf, 171.0), [171.0, 189.0), [189.0, inf)],  
## closed='left',  
## dtype='interval[float64]')

Los códigos internos de los valores .cat.codes.

people\_dfSumm["Altura\_Cat"].cat.codes.head()

## name  
## Luke Skywalker -1  
## C-3PO 0  
## R2-D2 -1  
## Darth Vader 2  
## Leia Organa -1  
## dtype: int8

Para modificar las etiquetas puedes modificar directamente los índices de las categorías.

people\_dfSumm["Altura\_Cat"].cat.categories = ["Bajo/a","Mediano/a","Alto/a"]  
  
people\_dfSumm[["height","Altura\_Cat"]].head(8)

## height Altura\_Cat  
## name   
## Luke Skywalker NaN NaN  
## C-3PO 167.0 Bajo/a  
## R2-D2 NaN NaN  
## Darth Vader 202.0 Alto/a  
## Leia Organa NaN NaN  
## Owen Lars 178.0 Mediano/a  
## Beru Whitesun lars 165.0 Bajo/a  
## Biggs Darklighter 183.0 Mediano/a

##### ACTIVIDAD GUIADA 2.5

Veamos ahora, un resumen de la altura de los personajes por el tipo de clima de su planeta.

Los pasos a realizar son:

1. Cruzamos las tablas de personajes y planetas
2. Construimos la variable trameada de altura en puntos de corte 171 cm y 189 cm.
3. Cruzamos la altura con el clima del planeta.

personajes\_df = pd.merge(people\_df.reset\_index(),   
 planets\_df.reset\_index(), left\_on=["homeworld"], right\_on=["url"])  
personajes\_df.index= personajes\_df.name\_x # indexamos por nombre del personaje  
personajes\_df.head()

## name\_x ... url  
## name\_x ...   
## Luke Skywalker Luke Skywalker ... http://swapi.dev/api/planets/1/  
## C-3PO C-3PO ... http://swapi.dev/api/planets/1/  
## Darth Vader Darth Vader ... http://swapi.dev/api/planets/1/  
## Owen Lars Owen Lars ... http://swapi.dev/api/planets/1/  
## Beru Whitesun lars Beru Whitesun lars ... http://swapi.dev/api/planets/1/  
##   
## [5 rows x 12 columns]

personajes\_df["Altura\_Cat"] = pd.cut(personajes\_df.height,[-np.infty,171,189,np.infty], right=False)  
personajes\_df.Altura\_Cat.cat.categories = ["Bajo/a","Mediano/a","Alto/a"]  
  
personajes\_df.Altura\_Cat.value\_counts().sort\_index()

## Bajo/a 7  
## Mediano/a 15  
## Alto/a 5  
## Name: Altura\_Cat, dtype: int64

summ\_altura\_clima = pd.crosstab(personajes\_df.Altura\_Cat,personajes\_df.temperate\_tropical)  
  
summ\_altura\_clima

## temperate\_tropical 0 1  
## Altura\_Cat   
## Bajo/a 2 5  
## Mediano/a 6 9  
## Alto/a 2 3

Para saber el % de personajes de cada estatura que vive en clima temparado-tropical o no, utiliza normalize="index". De esta forma cada fila va a sumar 1 (es decir, 100%).

summ\_altura\_clima = pd.crosstab(personajes\_df.Altura\_Cat,personajes\_df.temperate\_tropical, normalize="index")  
  
summ\_altura\_clima.sort\_index(axis=1, ascending=False)

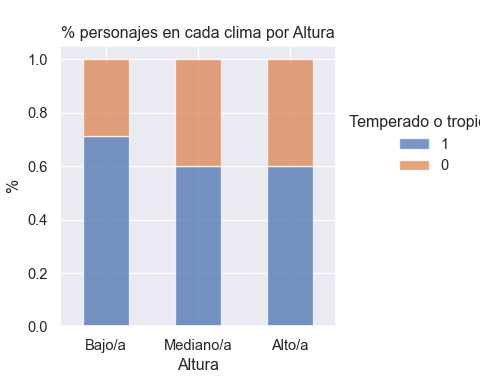
## temperate\_tropical 1 0  
## Altura\_Cat   
## Bajo/a 0.714286 0.285714  
## Mediano/a 0.600000 0.400000  
## Alto/a 0.600000 0.400000

Gráficamente.

summ\_altura\_clima = summ\_altura\_clima.sort\_index(axis=1, ascending=False) # cambiamos orden columnas para el gráfico  
summ\_altura\_clima

## temperate\_tropical 1 0  
## Altura\_Cat   
## Bajo/a 0.714286 0.285714  
## Mediano/a 0.600000 0.400000  
## Alto/a 0.600000 0.400000

g = summ\_altura\_clima.plot.bar(stacked=True,include\_bool=True, alpha=0.75, rot=0)  
g.legend(bbox\_to\_anchor=(1, 0.8),title="Temperado o tropical")  
plt.gcf().subplots\_adjust(bottom=0.15,right=0.7)  
plt.title("% personajes en cada clima por Altura")  
plt.ylabel("%")  
plt.xlabel("Altura");  
plt.show();



Se observa que un **70% de los personajes bajos** han nacido en climas temperado o tropicales, frente al **60% de los personajes altos**.

### AGREGADOS POR SUBGRUPOS

Además de frecuencias, es necesario calcular más estadísticos relevantes. Por ejemplo, la mediana de la edad (birth\_year), el peso (mass) mínimo de los personajes, o la media de habitantes (population) de sus planetas por tipo de clima y tramo de altura (alto, medio bajo) de los personajes.

#### AGRUPACIÓN O GROUPBY

Para poder dar una respuesta al problema planteado, antes es necesario introducir el concepto de Agrupación o GroupBy.

La solución planteada en pandas es la de definir una partición de la tabla mediante .grouby(). Una vez definida la petición se puede calcular estadísticos de resumen.

summ1\_altura\_clima = personajes\_df.groupby(["temperate\_tropical"  
 ,"Altura\_Cat"])[["mass","birth\_year","population"]].median()  
  
summ1\_altura\_clima

## mass birth\_year population  
## temperate\_tropical Altura\_Cat   
## 0 Bajo/a 75.0 79.50 2.000000e+05  
## Mediano/a 81.5 36.45 2.000000e+05  
## Alto/a 124.5 47.45 2.110000e+07  
## 1 Bajo/a 49.0 21.00 3.000000e+09  
## Mediano/a 80.0 46.00 1.500000e+09  
## Alto/a 82.0 92.00 4.500000e+08

Observa cómo se ha aplicado la función mean sobre cada una de las variables para cada combinación de clima de los planetas y altura de los personas.

También aparece un elmento nuevo, es la capacidad de los DataFrames de tener un índice compuesto por más de una variable, en este caso temperate\_tropical y Altura\_Cat.

Una forma de sortear los índices múltiples, es incorporarlos como variables con .reset\_index().

summ1\_altura\_clima = summ1\_altura\_clima.reset\_index()  
  
summ1\_altura\_clima

## temperate\_tropical Altura\_Cat mass birth\_year population  
## 0 0 Bajo/a 75.0 79.50 2.000000e+05  
## 1 0 Mediano/a 81.5 36.45 2.000000e+05  
## 2 0 Alto/a 124.5 47.45 2.110000e+07  
## 3 1 Bajo/a 49.0 21.00 3.000000e+09  
## 4 1 Mediano/a 80.0 46.00 1.500000e+09  
## 5 1 Alto/a 82.0 92.00 4.500000e+08

Ahora los 2 índices se han convertido a columnas de la tabla.

#### AGGREGATE

Para aplicar distintas funciones de agregación a distintas variables, se utiliza .aggregate() o abreviadamente .agg() de forma combinada con .groupby().

Un primer pase, es aplicar varias funciones a todas las variables seleccionadas del data frame.

summ2\_altura\_clima = personajes\_df.groupby(  
 ["temperate\_tropical","Altura\_Cat"])[  
 ["mass","birth\_year","population"]].agg(["min","median","mean"])  
summ2\_altura\_clima

## mass ... population   
## min median ... median mean  
## temperate\_tropical Altura\_Cat ...   
## 0 Bajo/a 75.0 75.0 ... 2.000000e+05 2.000000e+05  
## Mediano/a 74.0 81.5 ... 2.000000e+05 2.668000e+08  
## Alto/a 113.0 124.5 ... 2.110000e+07 2.110000e+07  
## 1 Bajo/a 20.0 49.0 ... 3.000000e+09 2.806000e+09  
## Mediano/a 45.0 80.0 ... 1.500000e+09 4.889634e+09  
## Alto/a 66.0 82.0 ... 4.500000e+08 1.665000e+09  
##   
## [6 rows x 9 columns]

Observa que ahora hay múltiples índices columna.

summ2\_altura\_clima.columns

## MultiIndex([( 'mass', 'min'),  
## ( 'mass', 'median'),  
## ( 'mass', 'mean'),  
## ('birth\_year', 'min'),  
## ('birth\_year', 'median'),  
## ('birth\_year', 'mean'),  
## ('population', 'min'),  
## ('population', 'median'),  
## ('population', 'mean')],  
## )

Una forma de combinar los dos niveles del índice en un solo nombre del tipo Variable\_Estadistico, es mediante la función de vectorización de funciones .map() (ver más información en: [map](https://www.w3schools.com/python/ref_func_map.asp)). La función para combinar los 2 niveles es: "\_".join.

summ2\_altura\_clima.columns = summ2\_altura\_clima.columns.map("\_".join) # combina nombres columna  
summ2\_altura\_clima = summ2\_altura\_clima.reset\_index() # pasamos los indices fila a columnas.  
  
summ2\_altura\_clima

## temperate\_tropical Altura\_Cat ... population\_median population\_mean  
## 0 0 Bajo/a ... 2.000000e+05 2.000000e+05  
## 1 0 Mediano/a ... 2.000000e+05 2.668000e+08  
## 2 0 Alto/a ... 2.110000e+07 2.110000e+07  
## 3 1 Bajo/a ... 3.000000e+09 2.806000e+09  
## 4 1 Mediano/a ... 1.500000e+09 4.889634e+09  
## 5 1 Alto/a ... 4.500000e+08 1.665000e+09  
##   
## [6 rows x 11 columns]

Para resolver la cuestión planteada, se necesita poder decidir, qué estadsítico se quiere aplicar en cada variable.

Se puede definir qué estadístico se va a aplicar sobre cada variable, mediante un diccionario.

summ3\_altura\_clima = personajes\_df.groupby(  
 ["temperate\_tropical","Altura\_Cat"]).agg({'name\_x': 'count', 'birth\_year': 'median'  
 ,'mass':'min','population':'mean'})  
  
summ3\_altura\_clima = summ3\_altura\_clima.reset\_index()  
summ3\_altura\_clima

## temperate\_tropical Altura\_Cat name\_x birth\_year mass population  
## 0 0 Bajo/a 2 79.50 75.0 2.000000e+05  
## 1 0 Mediano/a 6 36.45 74.0 2.668000e+08  
## 2 0 Alto/a 2 47.45 113.0 2.110000e+07  
## 3 1 Bajo/a 5 21.00 20.0 2.806000e+09  
## 4 1 Mediano/a 9 46.00 45.0 4.889634e+09  
## 5 1 Alto/a 3 92.00 66.0 1.665000e+09

Observa, por ejemplo, que hay 2 personajes de estatura baja que nacieron en un planeta de clima NO temperado o tropical, con mediana de edad 79.5 años BBY, peso mínimo 75 Kg y que viven en planetas con una media de 200.000 habitantes.

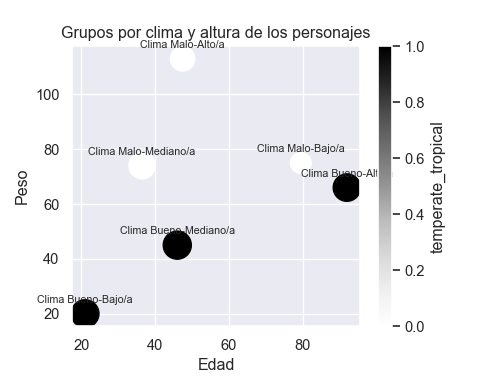
Para poder identificar cada fila con una etiqueta, vamos a combinar las variables de temperado\_tropical con la altura del personaje.

summ3\_altura\_clima["clima\_altura"] = pd.Series(  
 np.where(summ3\_altura\_clima["temperate\_tropical"]==0,"Clima Malo","Clima Bueno")  
 ) + "-" + summ3\_altura\_clima["Altura\_Cat"].astype(str)  
  
summ3\_altura\_clima

## temperate\_tropical Altura\_Cat ... population clima\_altura  
## 0 0 Bajo/a ... 2.000000e+05 Clima Malo-Bajo/a  
## 1 0 Mediano/a ... 2.668000e+08 Clima Malo-Mediano/a  
## 2 0 Alto/a ... 2.110000e+07 Clima Malo-Alto/a  
## 3 1 Bajo/a ... 2.806000e+09 Clima Bueno-Bajo/a  
## 4 1 Mediano/a ... 4.889634e+09 Clima Bueno-Mediano/a  
## 5 1 Alto/a ... 1.665000e+09 Clima Bueno-Alto/a  
##   
## [6 rows x 7 columns]

Para representar este Pandas de forma gráfica, se puede utilizar directamente .plot.scatter().

summ3\_altura\_clima.plot.scatter(x="birth\_year",y="mass",s=20\*np.log(summ3\_altura\_clima.population),c="temperate\_tropical");  
plt.gcf().subplots\_adjust(bottom=0.15, left=0.15)  
plt.title("Grupos por clima y altura de los personajes");  
plt.xlabel("Edad");  
plt.ylabel("Peso");  
for index,x in summ3\_altura\_clima.iterrows():  
 plt.text(x["birth\_year"],x["mass"]+4,x["clima\_altura"],horizontalalignment='center',size=8)  
plt.show();



Se observa que, los 3 grupos de clima Malo (bolas blancas) tienen personajes de mayor peso.

#### FILTRADO

A parte de agregar, .groupby() se puede utilizar de forma combinada con la función .filter() para seleccionar grupos de la tabla original que cumplan ciertas condiciones.

Por ejemplo, seleccionamos el grupo de persoanjes con un mismo clima y tramo de altura dónde la mediana de edad de este grupo sea superior a 90 años.

seleccion1 = personajes\_df.groupby(  
 ["temperate\_tropical","Altura\_Cat"]).filter(lambda fila: fila['birth\_year'].median() > 90)  
  
seleccion1

## name\_x ... Altura\_Cat  
## name\_x ...   
## Jar Jar Binks Jar Jar Binks ... Alto/a  
## Chewbacca Chewbacca ... Alto/a  
## Ki-Adi-Mundi Ki-Adi-Mundi ... Alto/a  
##   
## [3 rows x 13 columns]

Vemos como el grupo formado por “Jar Jar Binks”, “Chewbacca”, “Ki-Adi-Mundi” forman parte del mismo grupo de clima y altura y tienen una edad mediana superior a 90 años.

#### TRANSFORMACIONES

Otra utilidad de .groupby() combinado don .transform() es poder transformar los datos originales con agregados, de forma simple.

Comparamos la edad de cada personaje, respecto a la mediana de su grupo de clima y tramo de altura.

compara1 = personajes\_df.groupby(  
 ["temperate\_tropical","Altura\_Cat"]).birth\_year.transform(lambda x: np.abs(x - x.median()))  
  
compara1.sort\_values(ascending=False).head()

## name\_x  
## Jabba Desilijic Tiure 554.0  
## Chewbacca 108.0  
## Palpatine 61.0  
## Jar Jar Binks 40.0  
## C-3PO 32.5  
## Name: birth\_year, dtype: float64

Observa como “Jabba Desilijic Tiure” tiene una diferencia de 554 años sobre la media de su grupo.

#### APLICACIÓN DE FUNCIONES

Finalmente, para ganar aun mayor flexibilidad para transformar datos, .groupby() se combina con .apply() para poder aplicar transformaciones de forma selectiva en ciertas columnas del DataFrame.

def centrado(x):  
 x['Edad\_c'] = x['birth\_year']-x['birth\_year'].median()  
 return x  
  
personajes\_df2 = personajes\_df.groupby(["temperate\_tropical","Altura\_Cat"]).apply(centrado)  
  
personajes\_df2.head()

## name\_x height ... Altura\_Cat Edad\_c  
## name\_x ...   
## Luke Skywalker Luke Skywalker 172.0 ... Mediano/a -17.45  
## C-3PO C-3PO 167.0 ... Bajo/a 32.50  
## Darth Vader Darth Vader 202.0 ... Alto/a -5.55  
## Owen Lars Owen Lars 178.0 ... Mediano/a 15.55  
## Beru Whitesun lars Beru Whitesun lars 165.0 ... Bajo/a -32.50  
##   
## [5 rows x 14 columns]

Observa que la nueva variable de Edad\_c, permite saber que Luke está 19 años por debajo de la mediana de su grupo de clima (malo) y tramo de altura (mediano).

### TABLAS PIVOTE

Las tablas pivotantes .pivot\_table() son una solución intermedia entre los pd.crosstabs() orientados obtener tablas cruzadas (filas x columnas) y las agregaciones realizadas con .groupby().

Es decir, aplicar distintas funciones de agregación, sobre cruces de filas y columnas predefinidos.

personajes\_df.pivot\_table("birth\_year", index='Altura\_Cat', columns='temperate\_tropical')

## temperate\_tropical 0 1  
## Altura\_Cat   
## Bajo/a 79.500000 32.600000  
## Mediano/a 35.316667 106.500000  
## Alto/a 47.450000 114.666667

Esta tabla nos informa de la media de la edad BBY (‘birth\_year’) sobre el cruce de las categorías de altura y clima.

Pivot table, permite crear multiple índice en las filas y columnas.

personajes\_df.pivot\_table("birth\_year", index=['gender','Altura\_Cat'], columns='temperate\_tropical')

## temperate\_tropical 0 1  
## gender Altura\_Cat   
## female Bajo/a 47.000000 19.000000  
## Mediano/a NaN 47.000000  
## hermaphrodite Mediano/a NaN 600.000000  
## male Bajo/a NaN 37.000000  
## Mediano/a 35.316667 44.083333  
## Alto/a 47.450000 114.666667  
## none Bajo/a 112.000000 33.000000

También seleccionar que función de agregación se requiere en cada cruce definido.

personalizarDict = {'name\_x': 'count', 'birth\_year': 'median'  
 ,'mass':'min','population':'mean'}  
personajes\_df.pivot\_table(index='Altura\_Cat', columns='temperate\_tropical'  
 , aggfunc=personalizarDict)

## birth\_year mass ... name\_x population   
## temperate\_tropical 0 1 0 ... 1 0 1  
## Altura\_Cat ...   
## Bajo/a 79.50 21.0 75.0 ... 5 200000.0 2.806000e+09  
## Mediano/a 36.45 46.0 74.0 ... 9 266800000.0 4.889634e+09  
## Alto/a 47.45 92.0 113.0 ... 3 21100000.0 1.665000e+09  
##   
## [3 rows x 8 columns]

Observa cómo se ha obtenido la misma tabla que en el ejercicio con .agg(), pero ahora, con otra orientación de filas y columnas.

### DATETIME: GESTIÓN DE FECHAS Y HORARIOS

La librería para gestión de fechas y horarios en Python es datetime. Además, también se utiliza la librería de funcionalidades ampliadas dateutil.

from datetime import datetime  
from dateutil import parser

#### CREAR FECHAS HORAS

Para crear fechas.

fecha = datetime(year=2020, month=9, day=7) # con datetime

#### OBTENER COMPONENTES

Para obtener los meses y días de la semana en castellano.

import locale  
  
locale.setlocale(locale.LC\_TIME, 'es\_ES.UTF-8')

## 'es\_ES.UTF-8'

fecha.strftime('%A')

## 'lunes'

Más información sobre otros componentes en la [sección strftime](https://docs.python.org/3/library/datetime.html" \l "strftime-and-strptime-behavior), o bien, en la documentación de [datetime](https://docs.python.org/3/library/datetime.html) de Python.

#### DATETIME Y NUMPY

Se pueden definir vectores de tipo de datos datetime64.

fecha = np.array('2020-09-07', dtype=np.datetime64)  
fecha

## array('2020-09-07', dtype='datetime64[D]')

Ahora se pueden realizar operaciones vectorizadas de forma óptima en numpy.

fecha + np.arange(12)

## array(['2020-09-07', '2020-09-08', '2020-09-09', '2020-09-10',  
## '2020-09-11', '2020-09-12', '2020-09-13', '2020-09-14',  
## '2020-09-15', '2020-09-16', '2020-09-17', '2020-09-18'],  
## dtype='datetime64[D]')

También declarar datetime con fecha y hora.

np.datetime64('2020-09-07 12:00')

## numpy.datetime64('2020-09-07T12:00')

#### DATETIME Y PANDAS

Para definir fechas.

fecha = pd.to\_datetime("2020-9-7")  
fecha

## Timestamp('2020-09-07 00:00:00')

Convertir.

fecha.strftime("%A, %d de %m de %Y")

## 'lunes, 07 de 09 de 2020'

Vectorizar operaciones.

fecha + pd.to\_timedelta(np.arange(12), 'D')

## DatetimeIndex(['2020-09-07', '2020-09-08', '2020-09-09', '2020-09-10',  
## '2020-09-11', '2020-09-12', '2020-09-13', '2020-09-14',  
## '2020-09-15', '2020-09-16', '2020-09-17', '2020-09-18'],  
## dtype='datetime64[ns]', freq=None)

##### INDEXAR PANDAS CON DATETIME

Combinar ambas funcionalidades aporta mucha utilidad.

index = pd.DatetimeIndex(['2019-08-07', '2019-09-07',  
 '2020-08-07', '2020-09-07'])  
fecha = pd.Series([0, 1, 2, 3], index=index)  
fecha

## 2019-08-07 0  
## 2019-09-07 1  
## 2020-08-07 2  
## 2020-09-07 3  
## dtype: int64

Se puede seleccionar por rangos.

fecha['2019-08-07':'2020-08-07']

## 2019-08-07 0  
## 2019-09-07 1  
## 2020-08-07 2  
## dtype: int64

Incluso sólo por año.

fecha['2020']

## 2020-08-07 2  
## 2020-09-07 3  
## dtype: int64

#### SERIES TEMPORALES PANDAS

Para definir una serie pandas de datetime.

fecha = pd.to\_datetime(['2019-08-07', '2019-09-07',  
 '2020-08-07', '2020-09-07'])  
fecha

## DatetimeIndex(['2019-08-07', '2019-09-07', '2020-08-07', '2020-09-07'], dtype='datetime64[ns]', freq=None)

Convertir a frecuencia diaria.

fecha.to\_period('D')

## PeriodIndex(['2019-08-07', '2019-09-07', '2020-08-07', '2020-09-07'], dtype='period[D]', freq='D')

Restar fechas.

fecha - fecha[0]

## TimedeltaIndex(['0 days', '31 days', '366 days', '397 days'], dtype='timedelta64[ns]', freq=None)

#### SECUENCIAS DE FECHAS

Para definir una secuencia.

pd.date\_range('2020-09-03', '2020-09-07')

## DatetimeIndex(['2020-09-03', '2020-09-04', '2020-09-05', '2020-09-06',  
## '2020-09-07'],  
## dtype='datetime64[ns]', freq='D')

O bien, especificando la longitud de la secuencia y la periodicidad.

pd.period\_range('2020-07', periods=8, freq='M')

## PeriodIndex(['2020-07', '2020-08', '2020-09', '2020-10', '2020-11', '2020-12',  
## '2021-01', '2021-02'],  
## dtype='period[M]', freq='M')

#### FRECUENCIAS

Para generar frecuencias utiliza pd.timedelta\_range().

pd.timedelta\_range(0, periods=9, freq="2H30T")

## TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',  
## '12:30:00', '15:00:00', '17:30:00', '20:00:00'],  
## dtype='timedelta64[ns]', freq='150T')

Puedes ampliar conocimientos en la sección de [“Time Series/Date”](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html) de la documentación de Pandas.

# ANEXO: README DE PYTHON

[**Abre en Colab**](https://colab.research.google.com/github/griu/mbdds_fc20/blob/master/Python/modulo1_tema4_Py_60_anexo.ipynb)

## PREPARACIÓN DEL ENTORNO COLAB

Des de [Colab](http://colab.research.google.com/), hay que clonar el repositorio cada vez que inicias un nuevo libro. En los libros se incluye el código necesario para ello.

## PREPARACIÓN ENTORNO LOCAL-JUPYTER (OPCIONAL)

#### CLONAR REPOSITORIO

En local puedes utilizar el mismo proyecto que has clonado en el [README DE R](../R/README.md).

Para actualizarlo de nuevo, des de consola.

cd mbdds\_fc20  
git pull  
cd Python

#### CREAR EN LOCAL UN NUEVO ENVIRONMENT DE ANACONDA

Abrimos una línea de comandos (con *Anaconda 3.0* ya disponible).

* Windows: Escribimos Anaconda en el menú Inicio y aparecerá la consola MS-DOS de Anaconda.
* Linux: Abrimos Terminal

conda deactivate  
conda create -n mbdds\_rpy20 python=3.6.9  
conda activate mbdds\_rpy20

Verifica que se ha creado y está activo.

conda info --envs

#### INSTALA LAS LIBRERIAS DE PYTHON

cd mbdds\_fc20/Python  
conda activate mbdds\_rpy20  
python -m pip install -r requirementsColab.txt

#### PUBLICAR EL KERNEL

Para acceder al nuevo environment desde Jupyter necesitas publicar el kernel.

python -m ipykernel install --user --name mbdds\_rpy20 --display-name "mbdds\_rpy20"

Puede tardar unos minutos en publicarse.

#### LANZAR ENTORNO JUPYTER NOTEBOOK

Para acceder al servidor Jupyter.

conda activate mbdds\_rpy20  
jupyter notebook

Debería abrirse un navegador con acceso a Jupyter desde donde podrás acceder a los notebooks. Habitualmente el servidor Jupyter se abre en <http://localhost:8888/> .

## IMPORTAR DATOS DE STARWARS SWAPI

Datos procesados a partir de [https://swapi.dev/documentation#root](https://swapi.dev/documentation" \l "root).

import requests  
import pandas as pd  
import numpy as np

# exec(open('get\_entity.py').read())  
def get\_entity(entity = None, url\_base = 'http://swapi.dev/api/'):  
# https://swapi.dev/documentation#root  
   
 page\_num = 1  
 results\_acum = []  
 entity\_list = ["films","people","planets","species","starships","vehicles"]  
   
 if(entity not in entity\_list):  
 page\_num = None  
 print('Call a valid entity: ' + ','.join(entity\_list))  
   
 while(page\_num is not None):  
 url = "".join([url\_base, entity, '/?page=', str(page\_num)])  
 rjson = requests.get(url).json()  
 results\_acum = results\_acum + rjson['results']  
 if(rjson['next'] is not None):  
 page\_num = page\_num + 1  
 else:  
 page\_num = None  
   
 if (len(results\_acum) > 0):  
 results\_df = pd.DataFrame(results\_acum)  
 else:  
 results\_df = entity\_list  
   
 return(results\_df)  
  
def str2num(df,num\_list):  
 num\_list = [lab for lab in num\_list if lab in df.columns]  
 num\_list = [lab for lab in num\_list if df[lab].dtype==object]  
 for x in num\_list:  
 if (x=="crew"):  
 df.loc[df[x] == "30-165",x] = "165"  
 df[x] = pd.to\_numeric(df[x].replace('[^0-9\.-]', '', regex=True), errors='coerce')  
 return(df)

res = get\_entity()

res

films\_df = get\_entity("films")  
people\_df = get\_entity("people")  
planets\_df = get\_entity("planets")  
species\_df = get\_entity("species")  
starships\_df = get\_entity("starships")  
vehicles\_df = get\_entity("vehicles")  
  
people\_df.index = people\_df.name  
planets\_df.index = planets\_df.name  
species\_df.index = species\_df.name  
starships\_df.index = starships\_df.name  
vehicles\_df.index = vehicles\_df.name  
  
people\_df.drop("name",inplace=True, axis=1)  
planets\_df.drop("name",inplace=True, axis=1)  
species\_df.drop("name",inplace=True, axis=1)  
starships\_df.drop("name",inplace=True, axis=1)  
vehicles\_df.drop("name",inplace=True, axis=1)

people\_num = ["height","mass","birth\_year"]  
planets\_num = ["diameter","rotation\_period","orbital\_period","population","surface\_water", "gravity"]  
species\_num = ["average\_height","average\_lifespan"]  
starships\_num = ["passengers","length","crew","cost\_in\_credits","cargo\_capacity"  
 ,"hyperdrive\_rating","MGLT","max\_atmosphering\_speed"]  
vehicles\_num = ["cargo\_capacity","cost\_in\_credits","crew","length","max\_atmosphering\_speed","passengers"]

people\_df.loc[(people\_df["gender"].isin(['n/a', 'none'])) | people\_df["gender"].isna(),"gender"] = 'none'  
planets\_df["temperate\_tropical"] = planets\_df.climate.apply(lambda x: ("temperate" in x) or ("tropical" in x)).astype(int)

people\_df2 = str2num(people\_df,people\_num)  
planets\_df2 = str2num(planets\_df,planets\_num)  
species\_df2 = str2num(species\_df,species\_num)  
starships\_df2 = str2num(starships\_df,starships\_num)  
vehicles\_df2 = str2num(vehicles\_df,vehicles\_num)

planets\_df2.loc["Bespin","gravity"] = 1.5

people\_df2.to\_pickle("www/people\_df.pkl")  
planets\_df2.to\_pickle("www/planets\_df.pkl")  
species\_df2.to\_pickle("www/species\_df.pkl")  
starships\_df2.to\_pickle("www/starships\_df.pkl")  
vehicles\_df2.to\_pickle("www/vehicles\_df.pkl")

vehicles\_df2.describe()