Entorno de Computación Estadística: Python

Máster de Estadística Aplicada

UNIVERSIDAD DE GRANADA

Javier Arnedo. arnedo@ugr.es.
Departamento de Estadística e Investigación Operativa.
Curso 2025-2026.

Table of contents

Prefacio: Aprendiendo a programar en la era de la IA

Antes de comenzar a aprender Python como una alternativa a R, quería incluir esta breve sección de discusión sobre la importancia de aprender a programar y entender lo que realmente se está haciendo al crear un programa para cualquier fin, pero especialmente para el análisis estadístico de datos.

Hoy en día, la inteligencia artificial (IA) y las herramientas automatizadas están transformando rápidamente la forma en que interactuamos con la tecnología. Desde asistentes virtuales hasta sistemas de recomendación, la IA está integrada en muchos aspectos de nuestra vida diaria. Sin embargo, a pesar de estos avances, aprender a programar, ya sea en un lenguaje de programación u otro, sigue siendo una habilidad extremadamente valiosa.

En febrero de 2025, Andrej Karpathy, ex director de IA en Tesla y OpenAI, popularizó el término "Vibe Coding" para describir la capacidad de los modelos grandes de lenguaje (LLM) como ChatGPT, Gemini, DeepSeek, o Grok, para generar código a partir de descripciones. Según Karpathy, el Vibe Coding representa una nueva forma de interactuar con los ordenadores, donde los usuarios pueden simplemente describir lo que quieren hacer en lenguaje natural, sin programar, y el modelo genera el código necesario para lograrlo. El término fue nombrado palabra del año por el diccionario Collins en 2025.

Aunque la capacidad de los modelos de lenguaje para generar código es impresionante, es importante comprender la manera en que este código se genera, lo cual como veremos, deriva necesariamente como consecuencia en que el código no siempre es correcto o eficiente. Los modelos de lenguaje no tienen una comprensión profunda de los conceptos de programación o la lógica detrás del código que generan. En su lugar, se basan en patrones estadísticos aprendidos a partir de grandes cantidades de datos para predecir qué "palabra" es más probable que siga a una determinada entrada. En concreto, los LLM actuales se basan en una arquitectura de Red

Neuronal Artificial denominada Transformer, que conceptualmente no deja de ser un modelo de Aprendizaje Estadístico como los introducidos en la sección 5.4 de estos materiales.

En su libro El mito de la Inteligencia Artificial, Erik Larson argumenta que, utilizando los modelos basados en las aproximaciones actuales, no seremos capaces de obtener una Inteligencia Artificial General que pueda razonar y comprender el mundo de la misma manera en la que lo hacemos los humanos. Según Larson, los modelos actuales son simplemente herramientas estadísticas avanzadas que pueden imitar ciertos aspectos del comportamiento humano, pero carecen de una comprensión profunda y verdadera inteligencia.

El Vibe Coding implica dejarse llevar por la intuición para guiar la generación de código, confiando en que el modelo producirá resultados útiles basados en patrones aprendidos. Sin embargo, esta aproximación puede generar código que contenga bugs, malentendidos u otros problemas. Con suerte, algunos de estos errores, como por ejemplo el enlazar un paquete que no existe, serán detectados por el interprete o el compilador y se podría consultar de nuevo con la IA para intentar refinar el resultado. Sin embargo, hay otro tipo de errores mucho más peligrosos que son los que surgen en un código que se ejecuta correctamente y devuelve unos resultados aparentemente razonables, pero incorrectos. Este tipo de errores son especialmente peligrosos en los programas encargados de analizar datos, porque pueden llevarnos a conclusiones completamente falsas que no se derivan de los datos.

Esto no quiere decir que se tenga que desterrar necesariamente la IA como herramienta a la hora de programar, pero si se quiere llegar a resultados en los que se confíe, en contra del Vibe Coding, los LLM se deberán utilizar como generadores de ideas, o como un asistente que escriba un borrador que posteriormente será revisado, probado y entendido en su totalidad por el programador. Por lo tanto, la comprensión profunda de los conceptos de programación y la lógica detrás del código sigue siendo esenciales para desarrollar soluciones efectivas, eficientes y sobre todo, correctas.

En este contexto, en esta asignatura se pretende ofrecer un aprendizaje real sobre la programación y por ello se desaconseja el uso de IA para la resolución de las tareas. Los ejercicios están planteados para que, mediante su resolución, se pongan en práctica y se afiancen los conceptos descritos en los materiales. Que estos ejercicios los resuelva una IA no ayudará al objetivo de aprender a programar, del mismo modo que tampoco se aprende a programar copiando los resultados de un compañero.

La Universidad de Granada, ha puesto a disposición un documento con recomendaciones sobre el uso ético y responsable de la inteligencia artificial en el ámbito académico: **https://ceprud.ugr.es/formacion-tic/inteligencia-artificial/recomendaciones-ia#contenido0**

En el siguiente enlace podéis encontrar, además, algunas recomendaciones para su uso a la hora de programar: **https://simonwillison.net/2025/Mar/11/using-llms-for-code/#llms-amplify-existing-expertise**

1 Introducción a Python como alternativa a R.

Python es un lenguaje de programación interpretado, de alto nivel y propósito general. Fue creado por Guido van Rossum y lanzado por primera vez en 1991. Python es conocido por su sintaxis simple y fácil de aprender. Es un lenguaje de programación muy versátil y se puede utilizar para crear aplicaciones de escritorio, aplicaciones web, aplicaciones móviles, juegos y mucho más.

1.1 Principales diferencias con R.

Python y R son dos de los lenguajes de programación más populares en el campo de la ciencia de datos y la estadística. Aunque ambos lenguajes son muy poderosos y versátiles, tienen algunas diferencias clave:

Objetivo: Python es un lenguaje de programación de propósito general y ampliamente utilizado en el desarrollo de software, aplicaciones web y aplicaciones móviles, mientras que R es un lenguaje de programación específicamente diseñado para el análisis de datos y la estadística. Por lo general, R se utiliza de una manera interactiva, mientras que Python se utiliza tanto de manera interactiva como para desarrollar aplicaciones y sistemas más complejos.

Sintaxis: La sintaxis de Python es simple y fácil de aprender, mientras que la sintaxis de R es más compleja y puede ser difícil de entender para los principiantes. Aunque R está diseñado para ejecutar análisis de datos básicos fácilmente y en cuestión de minutos, las cosas se complican con tareas complejas, y a los usuarios de R les lleva más tiempo dominar el lenguaje.

Paquetes: Python tiene una amplia gama de paquetes disponibles para el análisis de datos, incluidos NumPy, Pandas, Matplotlib y Scikit-learn. R también tiene una amplia gama de paquetes disponibles, incluidos dplyr, ggplot2 y caret, pero al ser un lenguaje orientado al análisis estadístico, su funcionalidad básica, sin utilizar paquetes específicos, es más amplia que la de Python.

Eficiencia: Partiendo de que ninguno de los dos lenguajes es especialmente eficiente cuando es comparado con lenguajes compilados y de más bajo nivel como C/C++, Python es generalmente más rápido que R, especialmente cuando se trata de realizar cálculos numéricos y manipulaciones de datos. Python suele ser más rápido en la mayoría de los cálculos numéricos, especialmente cuando se usa junto con paquetes optimizadas en C/C++ (como `NumPy` o `pandas`). R no siempre es tan rápido en cálculos intensivos debido a su naturaleza interpretativa y la falta de optimización en algunos de sus paquetes nativos. Sin embargo, al usar paquetes de alto rendimiento como `Rcpp`, R puede mejorar sustancialmente en rendimiento.

Escalabilidad: Python tiene una mejor integración con tecnologías de big data, como `Apache Spark`, `Hadoop`, y bases de datos NoSQL. Con paquetes como `Dask` y `PySpark`, Python permite manejar grandes conjuntos de datos de manera distribuida.

Aprendizaje Estadístico: Python es ampliamente preferido en el área de la minería de datos y el aprendizaje estadístico avanzado gracias a paquetes como `scikit-learn`, `TensorFlow` y `PyTorch`, que están optimizadas y son mantenidas por grandes comunidades. R también tiene herramientas para aprendizaje estadístico (`caret`, `mlr3`, `tidymodels`), pero no es tan eficiente ni cuenta con el mismo soporte, sobre todo en aprendizaje profundo.

Comunidad: Python tiene una comunidad de usuarios muy grande y activa, lo que significa que hay una gran cantidad de recursos disponibles para sus usuarios. R también tiene una comunidad razonablemente grande de usuarios, pero es generalmente más pequeña que la comunidad de Python. Python es el lenguaje más utilizado del mundo según el índice **TIOBE**, mientras que R es el 17º.

Dependiendo del problema que se quiera resolver, puede ser más conveniente utilizar Python o R. En general, Python es más adecuado para tareas de programación generales, aplicaciones web, aprendizaje automático y análisis de datos a gran escala, mientras que R es más adecuado para análisis estadístico, visualización de datos y análisis de datos a pequeña escala.

1.2 Entornos de desarrollo integrado (IDE) para Python.

Existen varios entornos de desarrollo que se pueden utilizar para programar en Python. Algunos de los entornos de desarrollo más populares son:

Específicos para programar en Python:

- Jupyter Notebook: Jupyter Notebook es un entorno de desarrollo interactivo que permite escribir y ejecutar código en un entorno basado en web. Jupyter Notebook es muy popular entre los científicos de datos y los analistas de datos debido a su capacidad para crear documentos interactivos que combinan código, texto y visualizaciones. Jupyter Notebook es un entorno web para programación interactiva, ideal para análisis de datos, visualización, investigación y enseñanza. Permite combinar en un solo documento código, texto, ecuaciones y gráficos, lo que facilita la creación de trabajos reproducibles y colaborativos. Es gratuito, pero menos adecuado para proyectos complejos o de gran escala.
- PyCharm: PyCharm es un IDE profesional perfecto para desarrollos complejos en Python. Ofrece herramientas avanzadas de depuración, pruebas y gestión de proyectos, con amplia personalización y soporte para plugins que lo hacen compatible con múltiples lenguajes y frameworks. Es ideal para proyectos a gran escala, aunque su rendimiento puede requerir más recursos. Está disponible en una edición gratuita (Community) y otra de pago (Professional) que incluye funciones adicionales.
- Spyder: Spyder es un IDE ligero diseñado para la ciencia de datos y la computación científica. Su interfaz, similar a MATLAB, integra un explorador de variables, una consola IPython, y soporte para paquetes como NumPy y pandas, lo cual facilita el análisis de datos y la creación de prototipos. Viene preinstalado en Anaconda, que es un sistema de distribución de software para ciencia de datos que incluye más de 200 paquetes.

Es gratuito y de código abierto, aunque tiene menos opciones de personalización y plugins en comparación con PyCharm.

De propósito general o multilenguaje:

- Visual Studio Code: Visual Studio Code es un editor de código ligero y altamente personalizable de Microsoft que se puede utilizar para programar en Python. Visual Studio Code proporciona una serie de extensiones que permiten a los usuarios personalizar su entorno de desarrollo y agregar funcionalidades adicionales.
- Google Colab: Google Colab es un entorno de desarrollo basado en la nube que permite escribir y ejecutar código en Python en un entorno basado en web. Google Colab es muy popular entre los científicos de datos y los analistas de datos debido a su capacidad para ejecutar código en la nube de forma gratuita.
- Rstudio: Rstudio es un entorno de desarrollo que fue diseñado para R, pero soporta otros lenguajes como Python. Útil sobre todo si ya se ha trabajado con R en Rstudio, ya que permite trabajar con ambos lenguajes en un mismo entorno.

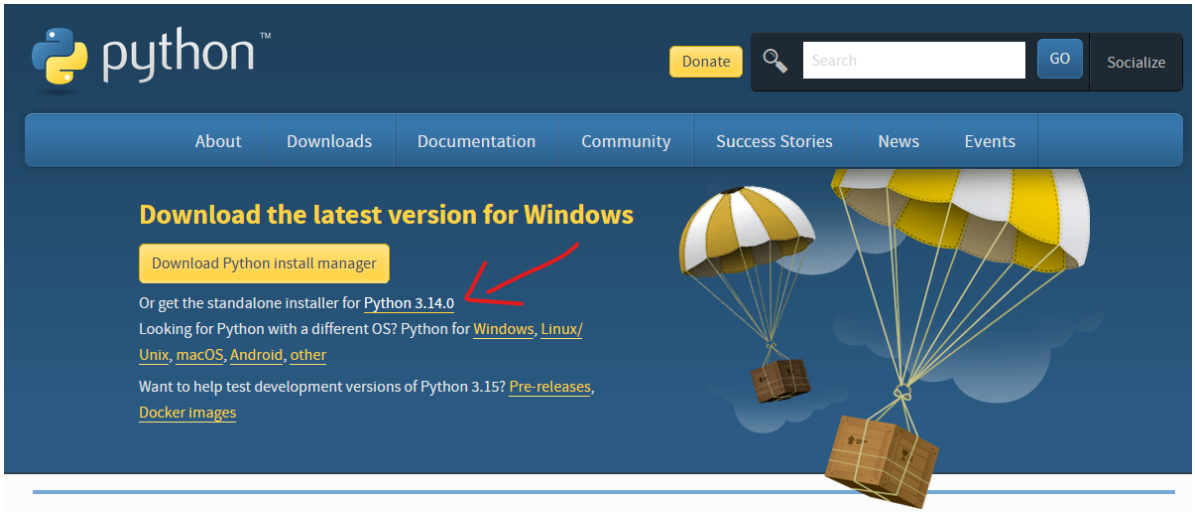
Hay muchos otros entornos de desarrollo disponibles para programar en Python, y la elección de un entorno de desarrollo dependerá de las necesidades y preferencias individuales de cada usuario. En nuestro caso, dado que hemos aprendido R en los temas anteriores de la asignatura y que Rstudio es un entorno de desarrollo muy popular en el campo de la estadística aplicada y el análisis de datos, podría ser una buena idea utilizar Rstudio para programar tanto en R como en Python sin tener que cambiar de entorno de desarrollo.

1.3 Descarga e instalación de Python.

Para poder utilizar python necesitaremos dos componentes, en primer lugar el propio lenguaje de programación Python y en segundo lugar un entorno de desarrollo.

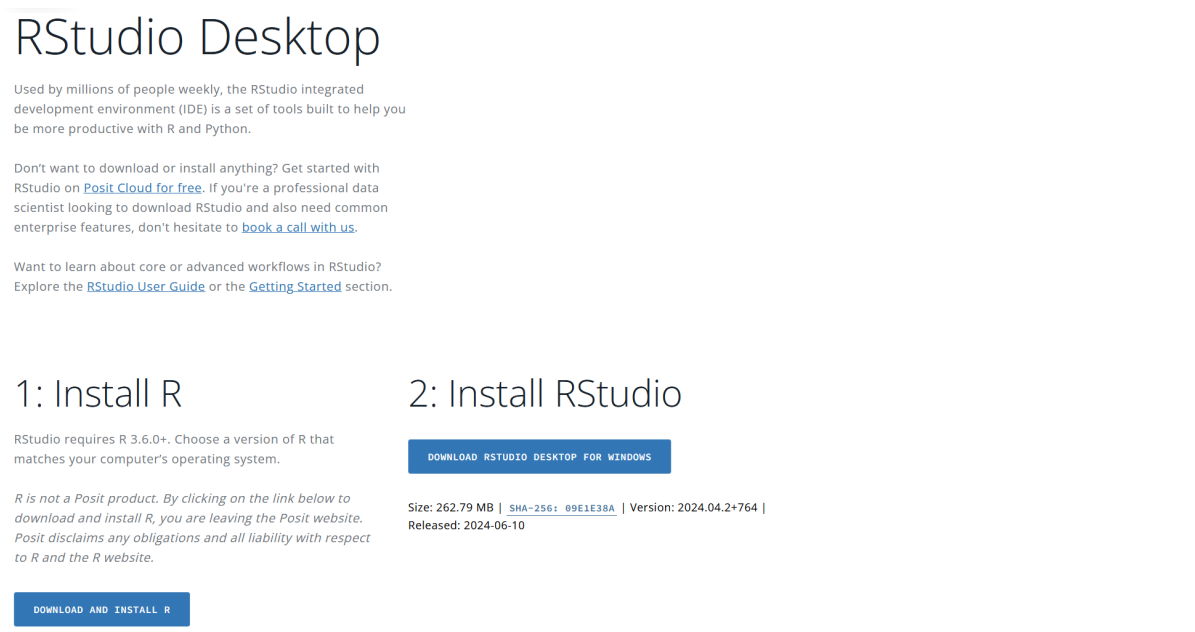
Para descargar e instalar Python, se puede visitar el sitio web oficial de Python en **https://www.python.org/downloads/**. En el sitio web oficial de Python, se pueden encontrar instrucciones detalladas sobre cómo descargar e instalar Python en diferentes sistemas operativos.

Por facilidad, y si es la primera vez que se instala Python se puede descargar el instalador único mediante el enlace que indica la última versión, en este momento Python 3.14.0. Si se quisiesen gestionar diferentes versiones se podría descargar el install manager, aunque de momento no es muy intuitivo.



Una vez instalado podemos abrir la aplicación Python instalada, la cual nos permitirá ejecutar instrucciones de Python de forma interactiva. También podemos abrir un script de Python con un editor de texto y ejecutarlo desde la consola del sistema. Pero esta forma de trabajar no es la más cómoda, por lo que es recomendable instalar un entorno de desarrollo.

Para descargar e instalar un entorno de desarrollo, se puede visitar el sitio web oficial del entorno de desarrollo deseado y seguir las instrucciones de instalación proporcionadas en el sitio web. Para instalar Rstudio, por ejemplo, se puede visitar el sitio web oficial de Rstudio en **https://posit.co/download/rstudio-desktop/**.



1.4 Funcionamiento de RStudio para Python

Una vez instalado, Rstudio nos permitirá ejecutar código tanto en R como en Python, además de crear documentos dinámicos que combinan código, texto y visualizaciones como vermos en el siguiente capítulo.

Antes de empezar a trabajar con Python, necesitaremos instalar el paquete `reticulate`, que será el encargado de comunicar RStudio con Python:

```
#R
install.packages("reticulate")
```

Para ejecutar comandos de Python de forma interactiva, podemos hacerlo desde la consola de Rstudio. Por defecto, la consola que encontraremos en la parte inferior de la ventana será una consola de R, pero podemos cambiarla a una consola de Python escribiendo el siguiente comando en la consola de R:

```
#R
reticulate::repl_python()
```

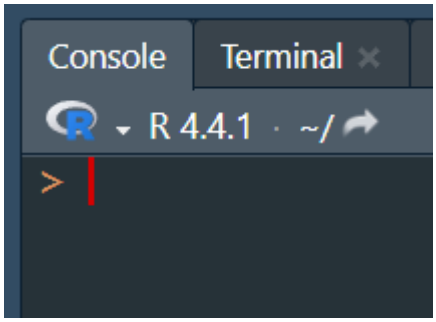
Una vez ejecutado este comando, la consola de R cambiará a una consola de Python que identificaremos por tres signos consecutivos ">" y podremos ejecutar comandos de Python de forma interactiva. Para regresar a la consola de R, podemos escribir alguno de los siguiente comandos en la consola de Python:

```
#Python
exit
```

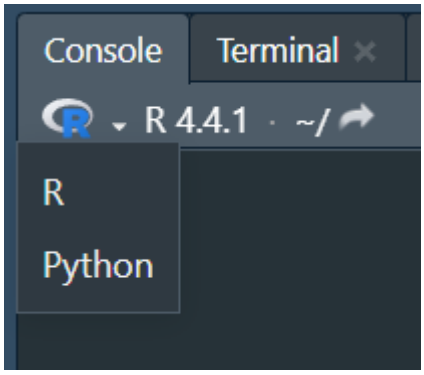
```
#Python
quit
```

El proceso de cambiar entre consolas de R y Python también puede realizarse mediante el icono del lenguaje de programación que estemos utilizando en ese momento, que encontraremos en la parte superior izquierda de la consola.

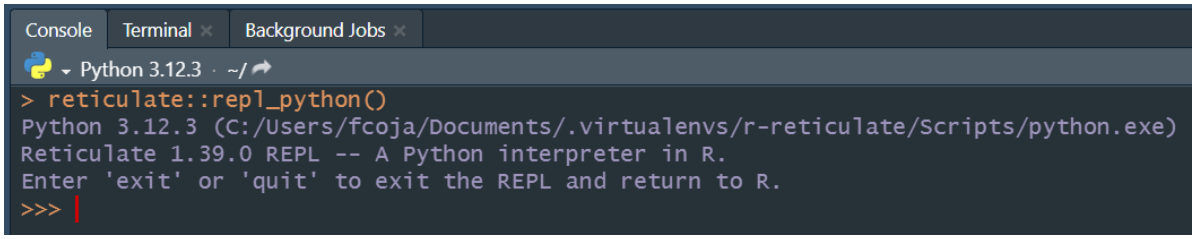
En este ejemplo, si estamos trabajando en R, podemos hacer click en su logo



Y se desplegará una lista de los lenguajes que se pueden utilizar.



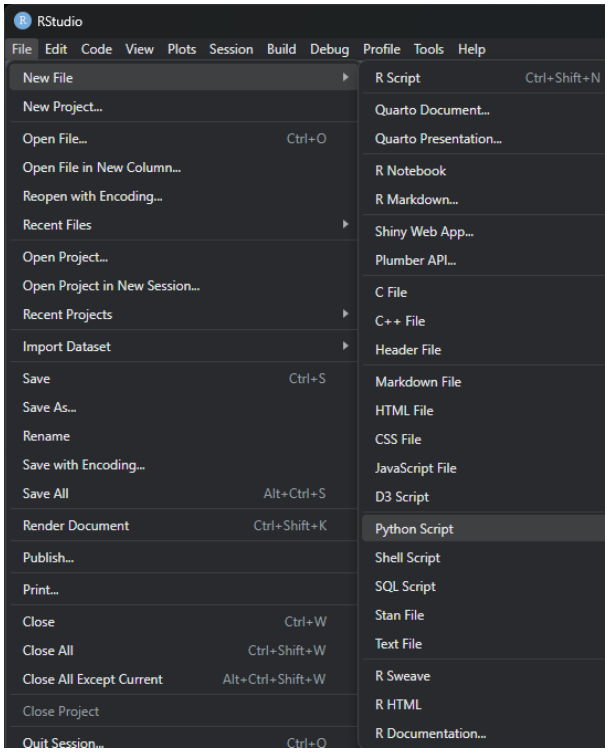
Al seleccionar Python, automáticamente se ejecutará el comando indicado anteriormente y cambiará a una consola de Python.



Para volver a R, podemos hacer click en el logo de Python y seleccionar R.

Para crear un script de Python, haremos click en el menú superior “File”, seleccionaremos “New File” y “Python Script”. Los scripts de Python tienen la extensión .py. Una vez creado el script, podremos ejecutarlo haciendo click en el botón “Run” 

que encontraremos en la parte superior derecha de la ventana al igual que en R.



La gran mayoría de las funcionalidades que proporciona RStudio al trabajar con R están presentes al trabajar con Python.

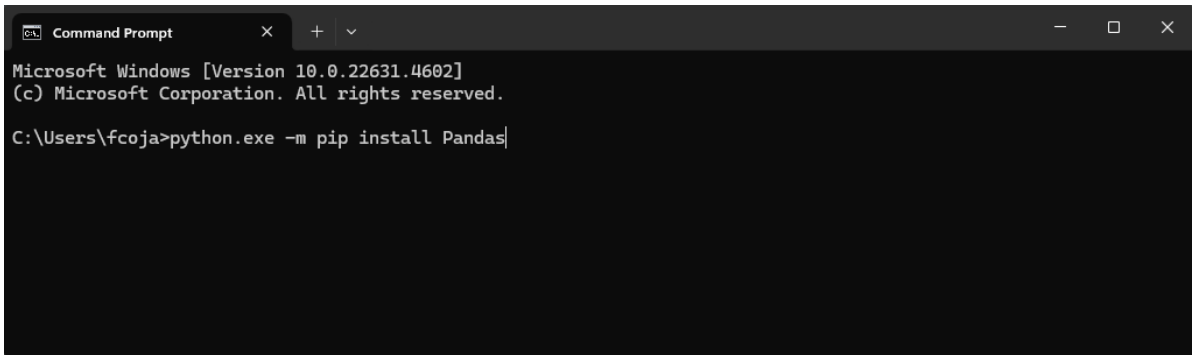
1.5 Paquetes en Python.

En Python, al igual que en R, los paquetes son bibliotecas de funciones y métodos que se utilizan para realizar tareas específicas. La funcionalidad base de Python es incluso más limitada que en R, por lo que es necesario instalar paquetes adicionales para realizar tareas básicas como el manejo de dataframes o la creación de gráficos.

Para instalar paquetes en Python, se puede utilizar el gestor de paquetes `pip` directamente o se pueden instalar mediante RStudio (más fácil en nuestro caso).

Para instalar un paquete con `pip` directamente, se puede utilizar el siguiente comando en la línea de comandos del sistema operativo, que en Windows podemos abrir mediante la aplicación `Command Prompt`:

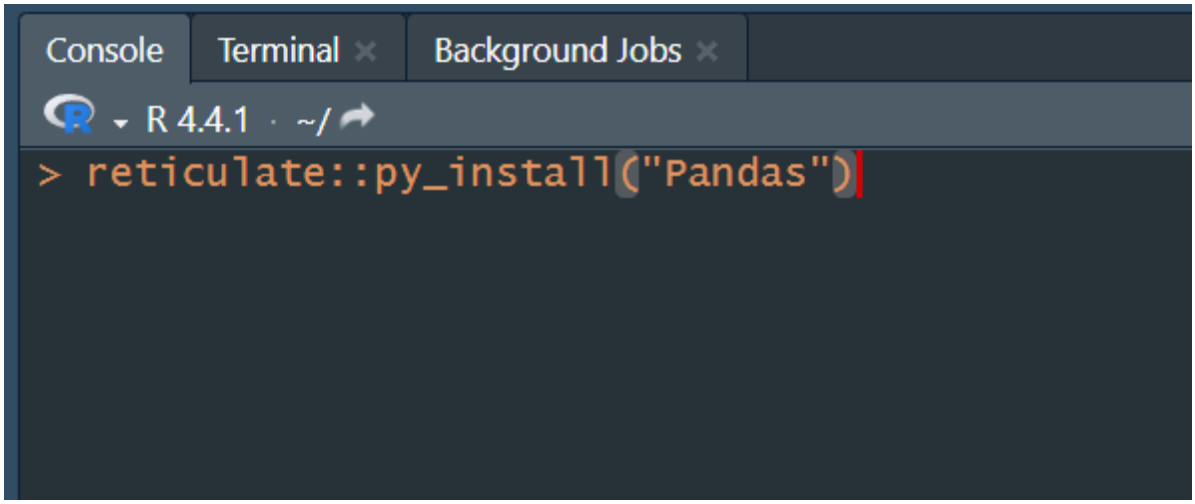
```
#Windows Command Prompt
python.exe -m pip install nombre_del_paquete
```



Alternativamente, y de manera más sencilla si estamos trabajando en RStudio, podemos instalar paquetes de Python utilizando la función `reticulate::py_install()` de la siguiente manera:

```
#R
reticulate::py_install("nombre_del_paquete")
```

Ojo, que aunque sea contraintutivo, esto debemos ejecutarlo **en la consola de R**, no en la de Python, porque no se pueden instalar paquetes internamente desde Python al igual que hacíamos en R mediante la función install.packages().



2 Creación de documentos dinámicos con Python.

Al igual que con R, en Python también es posible crear documentos dinámicos que combinan código, texto y visualizaciones. Para crear documentos dinámicos en Python, se pueden utilizar diferentes sistemas como Jupyter Notebooks o Quarto. A pesar de que el más utilizado al trabajar con Python exclusivamente es Jupyter, en esta asignatura vamos a centrarnos en Quarto, ya que nos permite trabajar con R y Python en un mismo documento de forma sencilla, además de tener una muy buena integración con Rstudio.

2.1 Introducción a Quarto.

Quarto es una versión de nueva generación y multilenguaje de R Markdown desarrollada por Posit (empresa desarrolladora de RStudio), que incluye docenas de nuevas funciones y capacidades, y al mismo tiempo puede renderizar la mayoría de los archivos Rmd con sintaxis RMarkdown existentes sin necesidad de modificaciones.

A diferencia de RMarkdown, Quarto es independiente de RStudio y puede usarse en cualquier editor de texto o entorno de desarrollo, como Visual Studio Code o JupyterLab. Esto lo hace más flexible para usuarios de diferentes lenguajes y plataformas.

Quarto incluye funciones de personalización avanzadas, como la configuración de temas globales para sitios web, soporte nativo para sintaxis de Markdown extendido (tablas, citas, listas avanzadas) y bibliotecas de plantillas, que facilitan el diseño visual y la consistencia en documentos grandes.

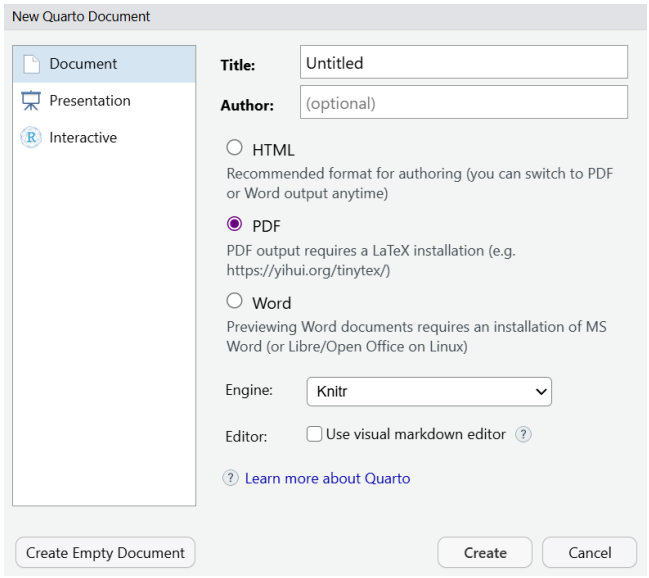
RMarkdown sigue siendo una gran opción para trabajos de análisis reproducible en R y la creación de documentos individuales. Quarto, en cambio, es una plataforma moderna y flexible que no solo mantiene las características de RMarkdown, sino que también amplía el alcance al permitir el uso de varios lenguajes, mayor personalización, interactividad y opciones de publicación profesional. Esto hace que Quarto sea ideal para proyectos de ciencia de datos, análisis multilenguaje, y publicaciones de gran escala en múltiples formatos.

2.2 Crear un documento Quarto

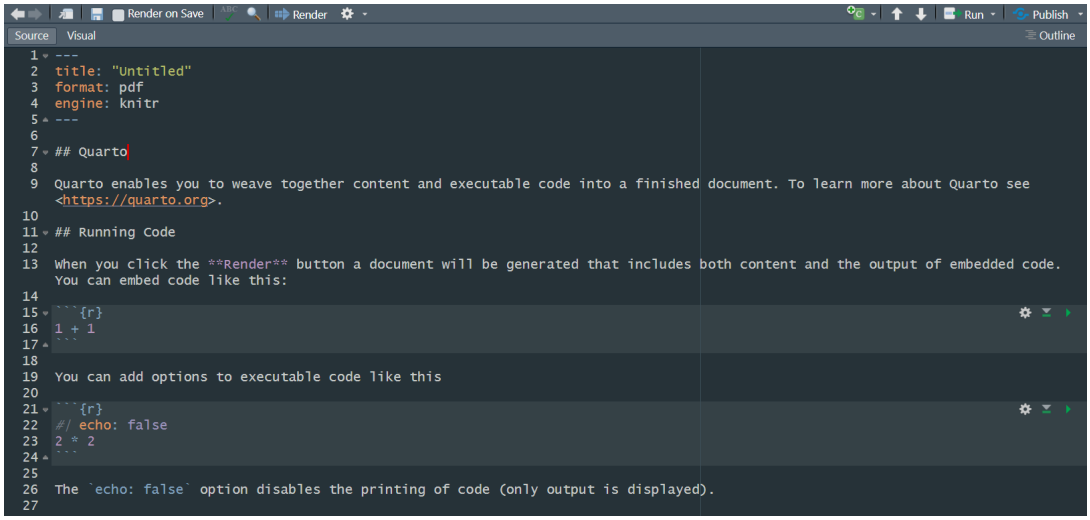
Para crear un documento Quarto en RStudio elegimos en el menú:

File > New File > Quarto Document

Lo que mostrará una caja de diálogo donde podemos especificar opciones básicas del documento:

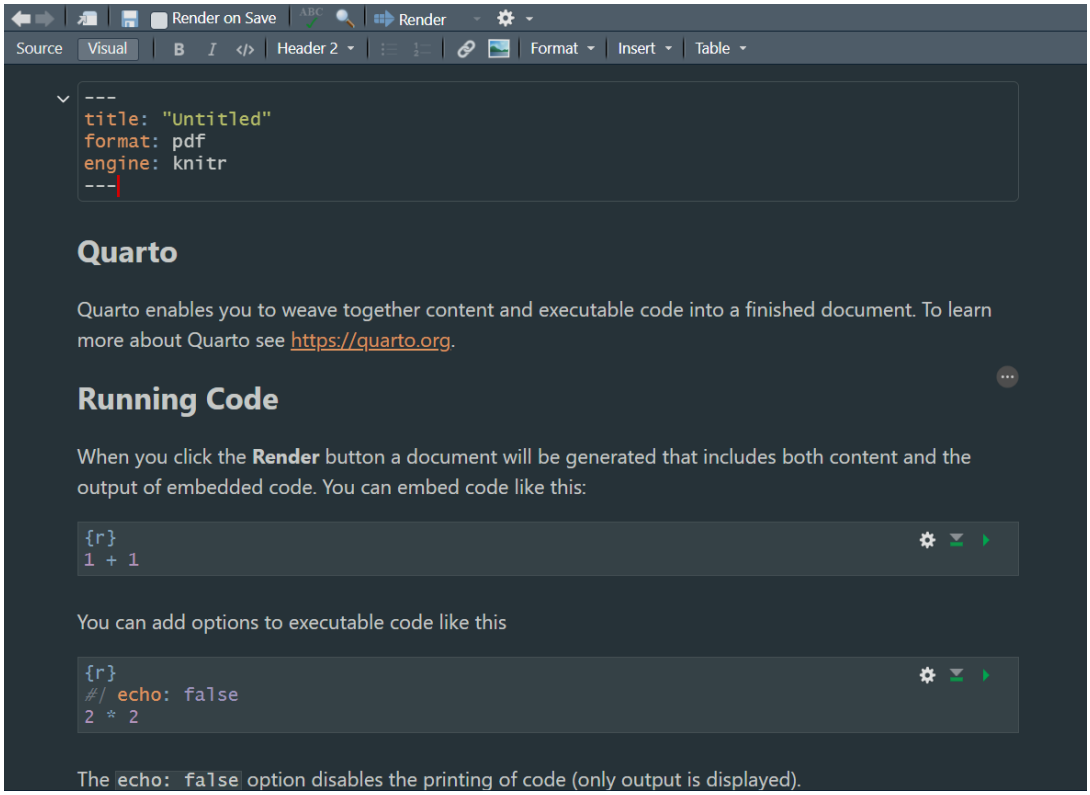


y, una vez definidas, podemos ver el documento creado en una pestaña de la ventana principal:



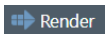
A diferencia de lo visto al crear documentos RMarkdown donde la extensión del documento era `Rmd`, en este caso la extensión del documento es `qmd`.

Al igual que al crear documentos RMarkdown, tenemos disponibles las pestañas `Source` y `Visual`, viendo directamente el código en `Source` o una versión más previsualizada en `Visual`.



Más allá de estas diferencias, la estructura que nos encontramos en el documento es muy similar a la estudiada en RMarkdown.

2.3 Compilar un documento Quarto

Para compilar un documento Quarto en RStudio podemos utilizar el boton Render 

de la barra de herramientas, o utilizar el acceso directo de teclado Ctrl+Shift+K.

Para renderizar en formato PDF se necesita tener instalado LaTeX en el sistema. Si no está instalado, una opción es utilizar el paquete `tinytex` en R y ejecutar la función `install_tinytex()` para instalar una versión mínima de LaTeX.

```
#R
library(tinytex)
install_tinytex()
```

Si solo tenemos código R, o R y Python simultáneamente en el documento, el motor de renderizado que utilizará Quarto, será por defecto Knitr. Pero si solo utilizamos el lenguaje Python en el documento, por defecto Quarto utilizará Jupyter como motor de renderizado, y para ello debemos tenerlo instalado previamente. Para evitar problemas se puede forzar a que el motor sea siempre knitr, independientemente de tener código r o python, mediante el comando en el encabezado "engine: knitr".

```
---
title: "Solo python"
format: pdf
engine: knitr
---
```

Internamente el proceso de renderizado de un documento Quarto funciona de manera muy similar a la estudiada para documentos RMarkdown. Cuando renderizamos un documento de Quarto, primero *knitr* ejecuta todos los bloques de código y crea un nuevo documento markdown (.md) que incluye el código y su salida. Luego, el archivo markdown generado es procesado por *pandoc*, que crea el formato final. El botón Render agrupa estas acciones y las ejecuta en el orden correcto.

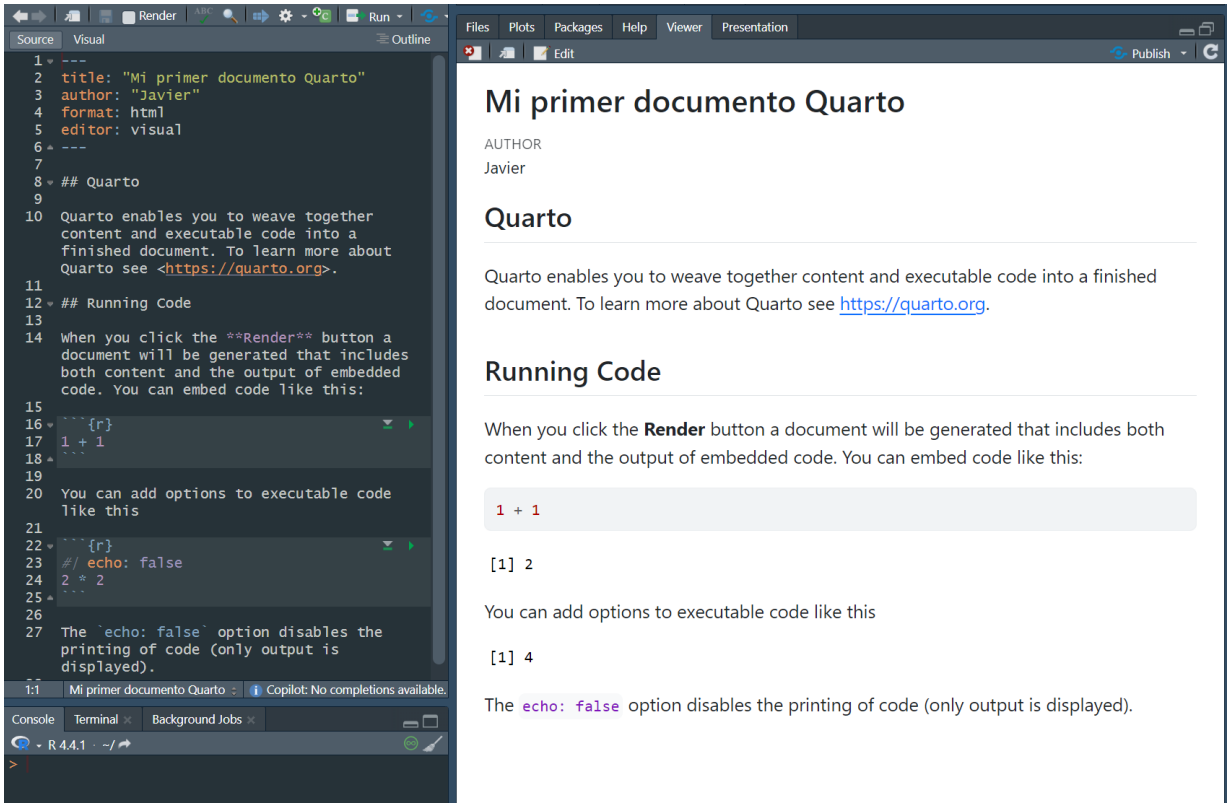
Al renderizar, Quarto genera un nuevo archivo que contiene el texto seleccionado, el código y los resultados del archivo `qmd`. El nuevo archivo puede ser un documento en formato HTML, PDF, MS Word, una presentación, sitio web, libro, documento interactivo u otro formato.

Al igual que con RMarkdown, podemos crear el documento utilizando comandos de R. En concreto debemos utilizar la función `quarto_render` del paquete `quarto`.

```
#R
library(quarto)
quarto_render("documento.qmd")
```

Aunque en este caso, cuando pulsamos en el boton Render, lo que se ejecuta por defecto es la función `quarto_preview`, que además de compilar el documento lo muestra mediante una previsualización del resultado en la pestaña Viewer.

```
#R
quarto_preview("documento.qmd")
```



2.4 Editar un documento Quarto

Para editar un documento Quarto, podemos hacerlo directamente en la pestaña Visual, donde podemos añadir texto, código y visualizaciones de forma interactiva. También podemos añadir bloques de código en la pestaña Source, donde podemos escribir código en su sintaxis.

Las principales diferencias con RMarkdown son:

2.4.1 Lenguajes de programación.

En Quarto, podemos utilizar varios lenguajes de programación en un mismo documento, como R, Python o Julia. Para ello, simplemente especificamos el lenguaje deseado entre corchetes al comienzo del chunk.

````
```{r}
library(ggplot2)

ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point()
```
````

````
```{python}
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

data = load_iris()
plt.figure()
plt.scatter(data.data[:, 0], data.data[:, 1], c=data.target)
plt.show()
```
````

2.4.2 Encabezado YAML

En RMarkdown los encabezados YAML son más simples y específicos, con opciones que funcionan principalmente para R. Aquí hay un ejemplo básico de encabezado en RMarkdown:

```
---
title: "Prueba"
author: "Javier Arnedo"
date: "2025-11-10"
output: pdf_document
---
```

En RMarkdown, la opción output puede aceptar valores como html_document, pdf_document, y word_document, entre otros.

En Quarto, el encabezado YAML es más flexible y tiene opciones adicionales, ya que permite múltiples formatos de salida en el mismo documento y configuraciones avanzadas. Por ejemplo:

```yaml
---
title: "Mi documento Quarto"
author: "Autor"
format:
  html:
    theme: flatly
  pdf:
    toc: true
  docx: default
---
```

Quarto utiliza format en lugar de output, y permite especificar múltiples formatos de salida en el mismo documento, facilitando la creación de versiones diferentes, cada una con sus propias opciones de formato, en un solo archivo fuente.

2.4.3 Opciones en los `chunks`

En Quarto, además de poder utilizar las opciones de los chunks de RMarkdown, podemos utilizar opciones específicas de Quarto. También tiene una sintaxis alternativa para definir estas opciones, donde las opciones se definen dentro del chunk con los caracteres `#|` precediéndolas. Estas opciones también pueden hacerse globales incluyéndolas en el encabezado YAML.

````
```{r}
#| label: fig-my-label
#| comment: ">"
# R code
```
````

2.4.4 Creación de Contenido Avanzado

RMarkdown requiere configuraciones específicas y paquetes adicionales para algunas funcionalidades avanzadas. Mientras que Quarto incluye soporte nativo para crear tablas de contenido, listas numeradas automáticas, notas al pie, etc., sin necesidad de configuraciones adicionales.

2.4.5 Interactividad

En RMarkdown, la interactividad se suele realizar a través de Shiny o htmlwidgets, y depende de paquetes específicos de R. Quarto, por su parte, permite agregar interactividad directamente en documentos HTML con soporte para Observable JavaScript y otras bibliotecas interactivas, lo que permite una integración más rica sin depender exclusivamente de Shiny.

2.4.6 En resumen

Característica	RMarkdown	Quarto
Soporte para múltiples lenguajes	R, Python (limitado)	R, Python, Julia
Formato de salida	Menos formatos, más paquetes	Más formatos en un sistema único
Flexibilidad en el formato de salida	Opciones limitadas	Configuraciones avanzadas y específicas
Personalización del estilo	Opciones de formato básicas	Soporte para plantillas, temas y bibliografía
Orientación	Enfoque en R y RStudio	Enfoque interdisciplinario para múltiples lenguajes
Interactividad	Dependiente de Shiny y htmlwidgets	Soporte nativo para Observable JavaScript y otros paquetes

La documentación del lenguaje, guias de inicio y enlaces para la descarga de Quarto para otros sistemas externos a RStudio se pueden encontrar en su página oficial: **https://quarto.org/**.

3 Sintaxis de Python para usuarios de R.

R y Python son lenguajes de programación bastante similares, sobre todo al nivel introductorio que los estamos trabajando en esta asignatura, pero a pesar de ello, al trabajar con cada uno de ellos debemos prestar atención a las diferencias de sintaxis que existen entre ambos. Más allá de las diferencias de sintaxis, también existen diferencias en la forma de trabajar con los datos, en los paquetes disponibles y en la forma de programar. Las funciones, ya sean estándar o de paquetes, pueden tener nombres diferentes, argumentos diferentes, o devolver resultados de forma diferente.

A continuación se muestran las principales diferencias de sintaxis entre R y Python:

3.1 Asignación de variables.

En R, las variables se asignan utilizando el símbolo `<-` o el símbolo `=`, mientras que en Python, las variables se asignan utilizando exclusivamente el símbolo `=`.

```
# R
x <- 5
y = 10
x
y
```

```
> [1] 5
> [1] 10
```

```
# Python
x = 5
y = 10
x
y
```

```
>>> 5
>>> 10
```

3.2 Nombres de variables.

En R, los nombres de las variables pueden contener letras, números, guiones bajos y puntos, mientras que en Python, los nombres de las variables pueden contener letras, números y guiones bajos, pero no pueden contener puntos.

```r
# R
x.1 <- 5
```

```python
# Python
x_1 = 5
```

3.3 Valor nulo

En R, el valor nulo se representa con `NULL`, mientras que en Python, el valor nulo se representa con `None`.

```r
# R
x <- NULL
```

```python
# Python
x = None
```

3.4 Tipos de variables

En R, los tipos de las variables suelen ser más específicos que en Python, sobre todos los orientados al análisis de datos como por ejemplo los factores o las fechas. En cambio, en Python los tipos y las estructuras de datos suelen ser más generales y flexibles.

En R un factor es un tipo de variable que se utiliza para representar datos categóricos, mientras que en Python, los datos categóricos tenemos que definirlos como un objeto personalizado, por ejemplo, utilizando la función Categorical del paquete Pandas. Ocurre lo mismo con las fechas, que en R podemos utilizar por defecto, pero en Python se suelen utilizar paquetes específicos.

Los tipos de variables en R y Python son similares, pero no idénticos. Por ejemplo, en R, los valores lógicos se representan como TRUE y FALSE, mientras que en Python, los valores lógicos se representan como True y False.

En R, podemos conocer el tipo de una variable utilizando la función class(), mientras que en Python, podemos conocer el tipo de una variable utilizando la función type().

La siguiente tabla muestra una comparación de los tipos de variables en R y Python:

Tipo de Datos R	Tipos de datos Python	Description
character	str	Cadenas de caracteres
numeric	float	Datos numéricos
integer	int	Datos numéricos que son enteros
logical	bool	Valores lógicos

Tipo de Datos R	Tipos de datos Python	Description
factor	objecto específico	Datos categóricos
Date	objecto específico	Fechas/tiempos

```r
# R
class("Hola mundo")
class(1.5)
class(as.integer(1))
class(TRUE)
class(factor(c("A", "B", "C")))
class(as.Date("2022-01-01"))
```

```
> [1] "character"
> [1] "numeric"
> [1] "integer"
> [1] "logical"
> [1] "factor"
> [1] "Date"
```

```python
# Python
import pandas as pd
type("Hola mundo")
type(1.5)
type(1)
type(True)
type(pd.Categorical(["A", "B", "C"]))
type(pd.to_datetime("2022-01-01"))
```

```
>>> <class 'str'>
>>> <class 'float'>
>>> <class 'int'>
>>> <class 'bool'>
>>> <class 'pandas.core.arrays.categorical.Categorical'>
>>> <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

3.5 Vectores y listas

En R, los vectores se crean utilizando la función `c()`, mientras que en Python, los vectores no existen, se utilizan listas mediante corchetes [ ]. Por lo tanto, en R, los vectores tienen

que contener el mísmo tipo de datos, mientras que en Python, al ser listas, pueden contener diferentes tipos de datos (¡incluso otras listas!).

```
# R
x <- c(1, 2, 3, 4, 5)
x
```

```
> [1] 1 2 3 4 5
```

```
# Python
x = [1, 'a', 3, 'b', 5]
x
y = [[1, 2], [3, 4], [5, 6]]
y
```

```
>>> [1, 'a', 3, 'b', 5]
>>> [[1, 2], [3, 4], [5, 6]]
```

3.6 Asignación de múltiples variables

A diferencia de R, Python permite asignar múltiples variables en una sola línea:

```
x, y, z = 1, 5, 10
x
y
z
```

```
>>> 1
>>> 5
>>> 10
```

También se puede asignar un mismo valor a varias variables:

```
x = y = z = 7
x
y
z
```

```
>>> 7
>>> 7
>>> 7
```

O desempaquetar automáticamente una lista en varias variables:

```
numeros = [2, 4, 6]
x, y, z = numeros
x
y
z
```

```
>>> 2
>>> 4
>>> 6
```

3.7 Indexación

En R, los índices comienzan en 1, mientras que en Python, los índices comienzan en 0. De la misma forma que en R, en Python podemos acceder a los elementos de una lista utilizando corchetes y utilizando rangos `from:to`, solo que en este caso el valor `to` no estará incluido en el rango. Para indicar que queremos acceder a todos los elementos de una lista, en Python utilizamos `:`, en lugar de dejar el campo vacío como en R.

```
# R
x <- c(1, 2, 3, 4, 5)
x[1]
x[1:3] # Índice 3 incluido
x[]
```

```
> [1] 1
> [1] 1 2 3
> [1] 1 2 3 4 5
```

```
# Python
x = [1, 2, 3, 4, 5]
x[0]
x[0:3] # Índice 3 no incluido
x[:]
```

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

Además en Python no podemos acceder a elementos que no existen en una lista para asignarlos, como en R, donde, si intentamos acceder a un elemento que no existe, se creará un nuevo elemento en esa posición.

```r
# R
x <- c(1, 2, 3)
x[4] <- 4
x
```

```
> [1] 1 2 3 4
```

```python
# Python
x = [1, 2, 3]
x[3] = 4
```

error: list assignment index out of range

Para cambiar dinámicamente una lista podemos utilizar la función append(valor) o insert(posición, valor).

```python
# Python
x = [1, 2, 3]
x.append(4) # Añade un elemento al final
x.insert(1, 5) # Añade un elemento en la posición 1
x
```

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

Este tipo de funciones, llamadas métodos, son propias del objeto lista, así que las utilizamos como una parte de este objeto como veremos más tarde en la sección de acceso a objetos.

El acceso a elementos de una lista anidada (donde tenemos elementos que son listas a su vez) se realiza añadiendo corchetes sucesivos para cada nivel de anidación.

```python
# Python
x = [[1, 2], [3, 4], [5, 6]]
x[1][0] #Primer corchete para la lista x y segundo para la lista interna
```

```
>>> 3
```

3.8 Matrices y arrays

En R, las matrices se crean utilizando la función `matrix()`, mientras que en Python, las matrices se crean utilizando la función `np.array()` del paquete NumPy, ya que no tenemos esta estructura de datos definida por defecto en Python base.

```r
# R
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
x
```

```
>      [,1] [,2] [,3]
> [1,]    1    3    5
> [2,]    2    4    6
```

```python
# Python
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
x
```

```
>>> array([[1, 2, 3],
>>>        [4, 5, 6]])
```

3.9 Diccionarios

Un diccionario es una estructura de datos que permite almacenar pares clave-valor. Son estructuras de datos muy útiles para almacenar información que se puede acceder rápidamente mediante una clave. En R, podríamos crear diccionarios utilizando la función `list()`, mientras que en Python, los diccionarios se crean utilizando llaves {} y separando las claves y los valores con dos puntos :. Para acceder a un elemento de un diccionario, en R utilizamos el símbolo $, mientras que en Python utilizamos corchetes [ ].

```r
# R
x <- list(a = 1, b = 2, c = 3)
#acceso al elemento 'a'
x$a
```

```
> [1] 1
```

```python
# Python
x = {'a': 1, 'b': 2, 'c': 3}
#acceso al elemento 'a'
x['a']
```

```
>>> 1
```

3.10 Paquetes

En R, los paquetes se cargan utilizando la función `library()`, mientras que en Python, los paquetes se cargan utilizando la palabra clave `import`. Además podemos asignarle un alias mediante la palabra clave `as`.

```r
# R
library(ggplot2)
```

```python
# Python
import pandas as pd
```

Los paquetes pueden tener, separados por puntos, submódulos que se pueden importar de forma independiente.

```python
# Python
import numpy as np #Importa todas las funciones del paquete numpy
import numpy.random as npr #Importa solo el submódulo random de numpy
```

Además, se puede utilizar la palabra clave `from` para importar funciones específicas de un paquete.

```python
# Python
from numpy.random import randint

randint(1, 10, 5) #Genera 5 números aleatorios entre 1 y 10
```

```
>>> array([4, 1, 4, 3, 7], dtype=int32)
```

Como hemos aprendido en el capítulo anterior, para instalar paquetes en Python, se puede utilizar el gestor de paquetes `pip` o la función `reticulate::py_install()` de R, mientras que en R, se puede utilizar la función `install.packages()` directamente.

3.11 Dataframes

En R, los dataframes se pueden crear utilizando directamente la función `data.frame()`, mientras que en Python, los dataframes se crean utilizando la función `pd.DataFrame()` del paquete Pandas.

```r
# R
df <- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))
df
```

```
>   x y
> 1 1 4
> 2 2 5
> 3 3 6
```

```python
# Python
import pandas as pd
df = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})
df
```

```
>>>    x  y
>>> 0  1  4
>>> 1  2  5
>>> 2  3  6
```

3.12 Aceso a objetos

En R, se usa el signo $ para referirse a una parte concreta y diferenciada de un objeto. En Python, se usa el signo . para tal efecto.

```r
# R
datos <- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))
datos$x
```

```
> [1] 1 2 3
```

```python
# Python
import pandas as pd
datos = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})
datos.x
```

```
>>> 0    1
>>> 1    2
>>> 2    3
>>> Name: x, dtype: int64
```

En Python, los objetos pueden tener métodos asociados, que son funciones especiales que se pueden aplicar a ese objeto. Por ejemplo, en Python, los dataframes tienen un método `head()` que muestra las primeras filas de ese dataframe. Mientras que en R, aunque puede haber diferentes versiones de una función para diferentes tipos de datos se ejecutan como funciones genéricas.

```r
# R
head(datos)
```

```python
# Python
datos.head()
```

Este era el caso de las funciones que comentamos anteriormente, append() e insert(), que son métodos de la clase lista.

3.13 Indentación

En R, los bloques de código se delimitan utilizando llaves `{}`, mientras que en Python, se utiliza la indentación (tabuladores).

```r
# R
x=10
if (x > 0) { #Inicio del bloque
  print("Línea 1:")
  print("x es mayor que 0")
} #Fin del bloque

y=15
if (y > 0) { #Inicio del bloque
  print("Línea 1:")
  print("y es mayor que 0")
} #Fin del bloque
```

```
> [1] "Línea 1:"
> [1] "x es mayor que 0"
> [1] "Línea 1:"
> [1] "y es mayor que 0"
```

```
# Python
x = 10
if x > 0: #Inicio del bloque
    print("Línea 1:")
    print("x es mayor que 0")

y = 15 #Sabemos que terminó el bloque porque no está indentado
if y > 0: #Inicio del bloque
    print("Línea 1:")
    print("y es mayor que 0")
```

```
>>> Línea 1:
>>> x es mayor que 0
>>> Línea 1:
>>> y es mayor que 0
```

3.14 Bucles y condicionales

En R, al igual que en Python, los bucles y condicionales se definen utilizando las palabras clave `for`, `while`, `if`, `else`… pero la sintaxis puede variar ligeramente entre ambos lenguajes.

La estructura del for en Python es:

```
for variable in lista:
    #Código a ejecutar
```

Un ejemplo que imprimiría los números 1, 7 , 18 y 23 sería:

```
for i in [1,7,18,23]:
    print(i)
```

Podemos apoyarnos en la función range para definir secuencias de números, por ejemplo, para imprimir los números del 1 al 5:

```
for i in range(1,6):
    print(i)
```

La función range tiene la particularidad de que el último número no está incluido en la secuencia. Su estructura puede ser:

```
range(inicio, fin) #Fin no incluido
```

```
range(1,4) #Representa 1,2,3
```

o bien, añadiendo el salto si queremos que sea diferente de 1.

```
range(inicio, fin, salto) #Fin no incluido
```

```
range(1,10,2) #Representa 1,3,5,7,9
```

Ejemplo de bucle for en R y Python:

```
# R
for (i in 1:5) {
  print(i)
}
```

```
> [1] 1
> [1] 2
> [1] 3
> [1] 4
> [1] 5
```

```
# Python
for i in range(1, 6):
    print(i)
```

```
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

Análogamente, la función while en Python tiene la siguiente estructura:

```
while condición:
    #Código a ejecutar
```

Ejemplo de bucle while en R y Python:

```r
# R
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
```

```
> [1] 1
> [1] 2
> [1] 3
> [1] 4
> [1] 5
```

```python
# Python
i = 1
while i <= 5:
    print(i)
    i += 1
```

```
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

La estructura condicional if en Python es:

```python
if condición:
    #Código a ejecutar si la condición es verdadera
else:
    #Código a ejecutar si la condición es falsa
```

Ejemplo de condicional if en R y Python:

```r
# R
x <- 5
if (x > 0) {
  print("x es mayor que 0")
} else {
  print("x es menor o igual que 0")
}
```

```
> [1] "x es mayor que 0"
```

```python
# Python
x = 5
if x > 0:
    print("x es mayor que 0")
else:
    print("x es menor o igual que 0")
```

```
>>> x es mayor que 0
```

3.15 Operadores lógicos

En R, los operadores lógicos son `&&`, `||` y `!`, mientras que en Python, los operadores lógicos son `and`, `or` y `not`.

```r
# R
x = 5
y = 10
if (x > 0 && y > 0) {
  print("x y y son mayores que 0")
}
```

```
> [1] "x y y son mayores que 0"
```

```python
# Python
x = 5
y = 10
if x > 0 and y > 0:
    print("x y y son mayores que 0")
```

```
>>> x y y son mayores que 0
```

3.16 Funciones

En R, las funciones se definen utilizando la palabra clave `function`, mientras que en Python, las funciones se definen utilizando la palabra clave `def`.

```r
# R
suma <- function(x, y) {
  return(x + y)
}

suma(2, 3)
```

```
> [1] 5
```

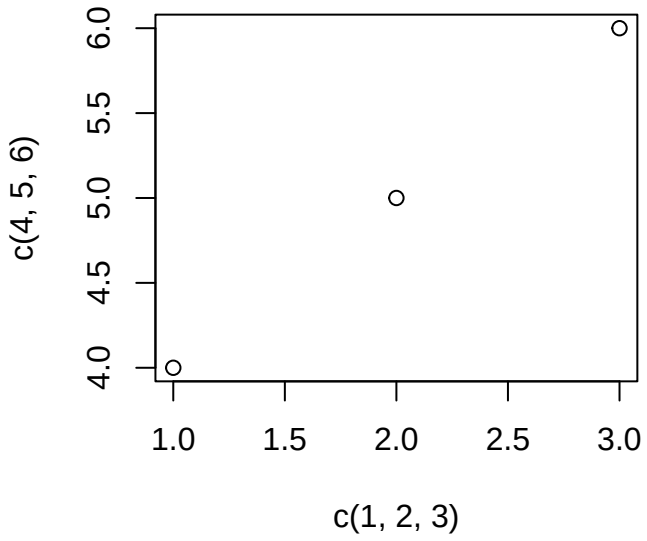
```python
# Python
def suma(x, y):
    return x + y

suma(2,3)
```

```
> 5
```

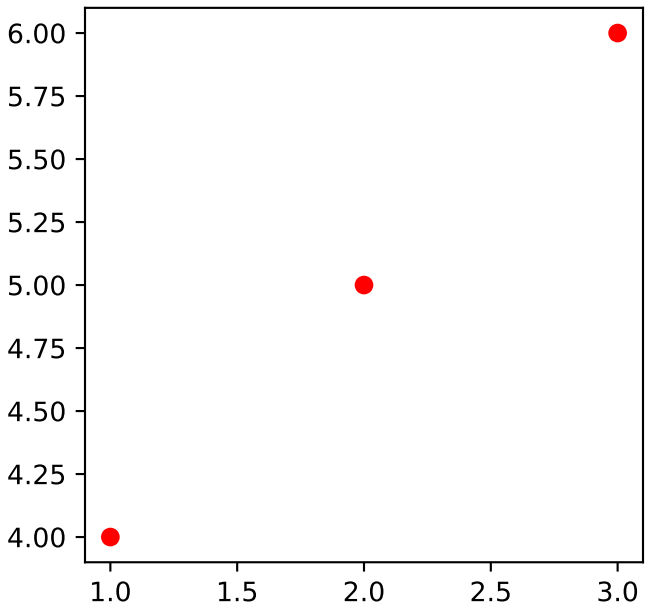
3.17 Gráficos

En R, los gráficos se pueden crear utilizando directamente la función `plot()` (aunque existan paquetes específicos como `ggplot`), mientras que en Python, los gráficos se crean utilizando paquetes externos como la función `plt.plot()` del paquete Matplotlib.

```r
# R
plot(c(1, 2, 3), c(4, 5, 6))
```



```python
# Python
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6], 'ro')
plt.show()
```



3.18 Autoimpresión y función print()

R tiene una funcionalidad de "autoimpresión" en la consola interactiva. Si escribes una variable o una expresión directamente, R automáticamente muestra su valor sin necesidad de print().

Esto es similar a cómo funciona Python en la consola interactiva. Sin embargo, en un script de Python ejecutado fuera de RStudio, como un programa y no como una interacción, necesitarías usar print() para mostrar el valor de una variable o una expresión.

Ejemplo en R:

```r
x <- 10
x  # Imprime automáticamente 10 sin usar print()
```

```
x = 10
x  # Imprime solo en el entorno interactivo
print(x)  # Imprime en cualquier entorno
```

Por esta razón, en los ejemplos de Python de este documento a partir de este punto, se utilizará print() para mostrar los resultados, aunque en la consola interactiva de Python, no sería necesario.

3.19 Importación y exportación de ficheros

En R, se pueden importar y exportar ficheros utilizando las funciones básicas `read.csv()`, `write.csv()`, `read.table()`, `write.table()`… mientras que en Python, necesitamos el paquete Pandas para poder importar y exportar ficheros utilizando las funciones `pd.read_csv()`, `pd.to_csv()`, `pd.read_table()`, `pd.to_table()`…

```
# R
datos <- read.csv("datos.csv")
write.csv(datos, "datos.csv")
```

```
# Python
import pandas as pd
datos = pd.read_csv("datos.csv")
datos.to_csv("datos.csv")
```

3.20 Otras diferencias

Estas son solo algunas de las diferencias de sintaxis entre R y Python. A medida que se trabaja más con ambos lenguajes, se pueden encontrar más diferencias y similitudes entre ellos. En el siguiente enlace se puede encontrar una guia de conversión de código de R a Python:

https://www.mit.edu/~amidi/teaching/data-science-tools/conversion-guide/r-python-data-manipulation/

La sintaxis completa Python puede consultarse en su documentación oficial, en el siguiente enlace:

https://docs.python.org/3/

Por último, en el siguiente enlace se puede encontrar una extensa guía de Python en español: **https://ellibrodepython.com/**

4 Principales paquetes para el análisis de datos.

Python es un lenguaje de programación muy popular en el campo del análisis de datos, por lo que hay muchos paquetes que se utilizan comúnmente en este ámbito. Algunos de los paquetes más populares son:

- NumPy: NumPy es un paquete que se utiliza para realizar cálculos numéricos en Python. NumPy proporciona una serie de funciones y métodos que facilitan la manipulación de vectores, matrices y matrices multidimensionales. NumPy es la base de muchos otros paquetes de análisis de datos en Python, como Pandas y Scikit-learn.
- Pandas: Pandas es un paquete que se utiliza para el análisis de datos y manipulación de datos. Proporciona una serie de estructuras de datos y funciones que facilitan el preprocesamiento y transformación de datos. Recientemente se están desarrollando nuevos paquetes que permiten trabajar con datos de forma más eficiente, como `Polars`.
- Matplotlib: Matplotlib es un paquete que se utiliza para crear gráficos y visualizaciones de datos. Otros paquetes de visualización de datos populares en Python son Seaborn y Plotly.
- Scikit-learn: Scikit-learn es paquete que se utiliza para el aprendizaje estadístico y la minería de datos. Scikit-learn proporciona una serie de algoritmos de aprendizaje automático que se pueden utilizar para entrenar modelos como regresión lineal, regresión logística, árboles de decisión, bosques aleatorios, máquinas de vectores de soporte, k-means, etc.
- TensorFlow: TensorFlow es un paquete que se utiliza para el aprendizaje profundo y la inteligencia artificial. TensorFlow proporciona una serie de funciones y métodos que facilitan la creación y entrenamiento de modelos de aprendizaje profundo como redes neuronales convolucionales, redes neuronales recurrentes, redes neuronales generativas adversarias, etc. Otros paquetes de aprendizaje profundo populares en Python son PyTorch y Keras.
- Statsmodels: Statsmodels es un paquete que se utiliza para el análisis estadístico en Python. Statsmodels facilita la realización de pruebas estadísticas, ajuste de modelos, análisis de series temporales, etc.
- Scipy: Scipy es un paquete que se utiliza para realizar cálculos científicos en Python. Scipy proporciona una serie de funciones que permiten la aplicación de diferentes modelos estadísticos, contrastes de hipótesis, integración numérica, optimización, interpolación, transformada de Fourier, álgebra lineal, etc.

A continuación veremos a ver algunos de ellos en más detalle:

4.1 NumPy.

NumPy es un paquete de Python que se utiliza para realizar cálculos numéricos en Python. Algunas de las funciones y métodos más comunes de NumPy son:

- Creación de vectores y matrices: NumPy proporciona una serie de funciones para crear matrices, como `np.array()`, `np.zeros()`, `np.ones()`, `np.random.rand()`, `np.random.randn()`, `np.arange()`, `np.linspace()`, `np.eye()`, `np.diag()`, `np.empty()`, `np.full()`, `np.tile()`, `np.repeat()`, ...
- Operaciones matemáticas: NumPy proporciona una serie de funciones para realizar operaciones matemáticas en matrices, como `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()`, `np.power()`, `np.sqrt()`, `np.exp()`, `np.log()`, `np.sin()`, `np.cos()`, `np.tan()`, `np.arcsin()`, `np.arccos()`, `np.arctan()`, `np.dot()`, `np.cross()`, `np.inner()`, `np.outer()`, `np.linalg.norm()`, ...
- Manipulación de matrices: NumPy proporciona una serie de funciones para manipular matrices, como `np.reshape()`, `np.transpose()`, `np.hstack()`, `np.vstack()`, `np.concatenate()`, `np.split()`, `np.flip()`, `np.roll()`, `np.rot90()`, `np.flipud()`, `np.fliplr()`, `np.rollaxis()`, `np.swapaxes()`, `np.moveaxis()`, `np.squeeze()`, `np.expand_dims()`, `np.tile()`, `np.repeat()`, ...
- Estadísticas: NumPy proporciona una serie de funciones para realizar cálculos estadísticos en matrices, como `np.mean()`, `np.median()`, `np.std()`, `np.var()`, `np.min()`, `np.max()`, `np.sum()`, `np.prod()`, `np.percentile()`, `np.corrcoef()`, `np.cov()`, `np.histogram()`, `np.bincount()`, ...
- Álgebra lineal: NumPy proporciona una serie de funciones para realizar operaciones de álgebra lineal en matrices, como `np.linalg.inv()`, `np.linalg.det()`, `np.linalg.eig()`, `np.linalg.svd()`, `np.linalg.qr()`, `np.linalg.cholesky()`, `np.linalg.solve()`, `np.linalg.lstsq()`, ...

Para utilizar NumPy en Python, primero es necesario importar el paquete NumPy utilizando la palabra clave `import`. Por ejemplo, para importar NumPy y asignarle un alias `np`, se puede utilizar el siguiente comando:

```
import numpy as np
```

Una vez importado NumPy, se pueden utilizar mediante el alias `np`. Por ejemplo, para crear una matriz de ceros de tamaño 3x3, se puede utilizar la función `np.zeros()` de la siguiente manera:

```
import numpy as np

matriz = np.zeros((3, 3))
print(matriz)
```

La principal diferencia entre los arrays y las listas de Python es que los arrays de NumPy son más eficientes en términos de memoria y rendimiento, ya que están optimizados para realizar cálculos numéricos. Además, los arrays de NumPy pueden tener múltiples dimensiones, mientras que las listas de Python base solo pueden tener una dimensión. Los arrays de NumPy son exclusivamente numéricos, mientras que las listas de Python pueden contener cualquier tipo de datos.

Podemos crear arrays de una dimensión, dos dimensiones, tres dimensiones, etc. y realizar operaciones matemáticas con ellos.

```
import numpy as np

# Crear un array de una dimensión
array1d = np.array([1, 2, 3, 4, 5])
print(array1d)
```

```
>>> [1 2 3 4 5]
```

```
# Crear un array de dos dimensiones
array2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array2d)
```

```
>>> [[1 2 3]
>>>  [4 5 6]
>>>  [7 8 9]]
```

```
# Crear un array de tres dimensiones
array3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
print(array3d)
```

```
>>> [[[ 1  2]
>>>   [ 3  4]]
>>>
>>>  [[ 5  6]
>>>   [ 7  8]]
```

```
>>>
>>>  [[ 9 10]
>>>   [11 12]]]
```

```
# Realizar operaciones matemáticas con arrays
array1d = np.array([1, 2, 3, 4, 5])
array2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(np.dot(array1d, array1d)) # Producto escalar
```

```
>>> 55
```

```
print(np.dot(array2d, array2d)) # Producto matricial
```

```
>>> [[ 30  36  42]
>>>  [ 66  81  96]
>>>  [102 126 150]]
```

```
print(np.linalg.det(array2d)) # Determinante de una matriz
```

```
>>> 0.0
```

```
evals, evects = np.linalg.eig(array2d) # Autovalores y autovectores de una matriz
print(evals)
```

```
>>> [ 1.61168440e+01 -1.11684397e+00 -1.30367773e-15]
```

```
print(evects)
```

```
>>> [[-0.23197069 -0.78583024  0.40824829]
>>>  [-0.52532209 -0.08675134 -0.81649658]
>>>  [-0.8186735   0.61232756  0.40824829]]
```

```
print(np.mean(array1d)) # Media de un array
```

```
>>> 3.0
```

```
print(np.mean(array2d, axis=0)) # Media por columnas de una matriz
```

```
>>> [4. 5. 6.]
```

Algo a destacar en este paquete es que los valores faltantes suelen tratarse conjuntamente con los valores de otros típos no numéricos mediante el tipo `np.nan` (Not A Number), que sería similar a los NA que podemos encontrar en R. Esto posibilita su tratamiento especial en las funciones y métodos de este paquete de orientación numérica.

Podemos crear un array de NumPy incluyendo estos valores faltantes y seguirá siendo de tipo numérico:

```
a = np.array([2, np.nan, 4, 1, np.nan])
a
a.dtype
```

```
>>> array([ 2., nan,  4.,  1., nan])
>>> dtype('float64')
```

Lo cual no ocurre si utilizamos valores `None`:

```
b = np.array([2, None, 4, 1, None])
b.dtype
```

```
>>> dtype('O')
```

Para más información sobre NumPy, se puede consultar la documentación oficial de NumPy en el siguiente enlace: https://numpy.org/doc/.

4.2 Pandas.

Pandas es un paquete de Python que se utiliza para el análisis de datos. Pandas proporciona una serie de estructuras de datos, métodos y funciones que facilitan la manipulación y el análisis de datos. Algunas de las más comunes son:

- Series: Una serie es una estructura de datos unidimensional que se utiliza para representar datos de una sola variable. Una serie se puede crear utilizando la función `pd.Series()`.

- Dataframes: Un dataframe es una estructura de datos bidimensional que se utiliza para representar datos de múltiples variables. Un dataframe se puede crear utilizando la función `pd.DataFrame()`.
- Lectura y escritura de datos: Pandas proporciona una serie de funciones para leer y escribir datos en diferentes formatos, como CSV, Excel, SQL, JSON, HTML, …
- Indexación y selección: Pandas proporciona una serie de métodos para indexar y seleccionar datos en un dataframe, como `loc[]`, `iloc[]`, `at[]`, `iat[]`, `[]`, `head()`, `tail()`, `sample()`, `query()`, `filter()`, `where()`, `mask()`, `drop()`, `dropna()`, `fillna()`, `replace()`, …
- Agrupación y agregación: Pandas proporciona una serie de métodos para agrupar y agregar datos en un dataframe, como `groupby()`, `agg()`, `apply()`, `transform()`, `pivot_table()`, `melt()`, `stack()`, `unstack()`, …
- Unión y concatenación: Pandas proporciona una serie de funciones para unir y concatenar datos en un dataframe, como `pd.merge()`, `pd.join()`, `pd.concat()`, `pd.append()`, …
- Visualización de datos: Pandas proporciona una serie de métodos para visualizar datos en un dataframe, como `plot()`, `hist()`, `boxplot()`, `scatter()`, `bar()`, `pie()`, `line()`, `area()`, `hexbin()`, `kde()`, `density()`, …

Para utilizar Pandas en Python, primero es necesario importar el paquete Pandas utilizando la palabra clave `import`. Por ejemplo, para importar Pandas y asignarle un alias `pd`, se puede utilizar el siguiente comando:

```
import pandas as pd
```

Una vez importado Pandas, se puede utilizar mediante el alias `pd`. Por ejemplo, para crear un dataframe a partir de un diccionario, se puede utilizar la función `pd.DataFrame()` de la siguiente manera:

```
import pandas as pd

datos = {'nombre': ['Juan', 'María', 'Pedro', 'Ana'],
         'edad': [25, 30, 35, 40],
         'ciudad': ['Granada', 'Sevilla', 'Málaga', 'Córdoba']}

df = pd.DataFrame(datos)
print(df)
```

```
>>>   nombre  edad   ciudad
>>> 0   Juan    25  Granada
>>> 1  María    30  Sevilla
>>> 2  Pedro    35   Málaga
>>> 3    Ana    40  Córdoba
```

Podemos realizar operaciones básicas con los dataframes, como seleccionar columnas, filtrar filas, añadir columnas, eliminar columnas, etc.

```
import pandas as pd

datos = {'nombre': ['Juan', 'María', 'Pedro', 'Ana'],
         'edad': [25, 30, 35, 40],
         'ciudad': ['Granada', 'Sevilla', 'Málaga', 'Córdoba']}

df = pd.DataFrame(datos)

print(df)
```

```
>>>   nombre  edad   ciudad
>>> 0   Juan    25  Granada
>>> 1  María    30  Sevilla
>>> 2  Pedro    35   Málaga
>>> 3    Ana    40  Córdoba
```

```
print(df['nombre']) # Seleccionar una columna
```

```
>>> 0     Juan
>>> 1    María
>>> 2    Pedro
>>> 3      Ana
>>> Name: nombre, dtype: object
```

```
print(df[['nombre', 'ciudad']]) # Seleccionar varias columnas
```

```
>>>   nombre   ciudad
>>> 0   Juan  Granada
>>> 1  María  Sevilla
>>> 2  Pedro   Málaga
>>> 3    Ana  Córdoba
```

```
print(df.loc[0]) # Seleccionar una fila
```

```
>>> nombre       Juan
>>> edad           25
>>> ciudad    Granada
>>> Name: 0, dtype: object
```

```
print(df.loc[0:2]) # Seleccionar varias filas
```

```
>>>   nombre  edad   ciudad
>>> 0   Juan    25  Granada
>>> 1  María    30  Sevilla
>>> 2  Pedro    35   Málaga
```

```
print(df[df['edad'] > 30]) # Filtrar filas
```

```
>>>   nombre  edad   ciudad
>>> 2  Pedro    35   Málaga
>>> 3    Ana    40  Córdoba
```

```
df['nueva_columna'] = df['edad'] * 2 # Añadir una columna
print(df)
```

```
>>>   nombre  edad   ciudad  nueva_columna
>>> 0   Juan    25  Granada             50
>>> 1  María    30  Sevilla             60
>>> 2  Pedro    35   Málaga             70
>>> 3    Ana    40  Córdoba             80
```

```
df=df.drop('nueva_columna', axis=1) # Eliminar una columna
print(df)
```

```
>>>   nombre  edad   ciudad
>>> 0   Juan    25  Granada
>>> 1  María    30  Sevilla
>>> 2  Pedro    35   Málaga
>>> 3    Ana    40  Córdoba
```

Además, Pandas proporciona una serie de funciones para realizar operaciones estadísticas en un dataframe, como calcular la media, la mediana, la desviación estándar, el mínimo, el máximo, la suma, el producto, el percentil, la correlación, la covarianza, el histograma, el conteo, ... como veremos en el siguiente capítulo.

Para más información sobre Pandas, se puede consultar la documentación oficial de Pandas en el siguiente enlace: https://pandas.pydata.org/docs/.

Un buen comienzo para aprender Pandas es la guía de aprendizaje de Pandas en 10 minutos que se puede encontrar en el siguiente enlace: https://pandas.pydata.org/docs/user_guide/10min.html.

Al igual que se puede encontrar en esta guía rápida vamos a ver a continuación un resumen de las funcionalidades más importantes de Pandas.

*Nota: En estos ejemplos no vamos a utilizar la función print() por simplicidad y por conservar la estructura de la guía original. En un script de Python ejecutado fuera de RStudio, como un programa y no como una interacción, se necesitaría usar print() para mostrar el valor de una variable o una expresión como hemos visto anteriormente.

En primer lugar importaremos el paquete Pandas y NumPy, que utilizaremos en los ejemplos.

```
import pandas as pd
import numpy as np
```

4.2.1 Creación de objetos

Podemos crear una Series pasando una lista de valores. Pandas asignará automáticamente un índice entero a la Serie.

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
s
```

```
>>> 0    1.0
>>> 1    3.0
>>> 2    5.0
>>> 3    NaN
>>> 4    6.0
>>> 5    8.0
>>> dtype: float64
```

4.2.2 Creación de un DataFrame

Podemos crear un DataFrame pasando una estructura de NumPy con un índice de fechas y columnas etiquetadas:

```
dates = pd.date_range("20130101", periods=6) #Crea 6 fechas consecutivas
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
df
```

```
>>>                    A         B         C         D
>>> 2013-01-01 -0.372123  0.303535  0.383237  0.136115
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203
```

También podemos crear un DataFrame pasando un diccionario de objetos, donde las claves son las etiquetas de las columnas y los valores son los datos de las columnas.

```
df2 = pd.DataFrame({ "A": 1.0, "B": pd.Timestamp("20130102"),
                     "C": pd.Series(1, index=list(range(4)), dtype="float32"),
                     "D": np.array([3] * 4, dtype="int32"),
                     "E": pd.Categorical(["test", "train", "test", "train"]),
                     "F": "foo", })
df2
```

```
>>>      A          B    C  D      E    F
>>> 0  1.0 2013-01-02  1.0  3   test  foo
>>> 1  1.0 2013-01-02  1.0  3  train  foo
>>> 2  1.0 2013-01-02  1.0  3   test  foo
>>> 3  1.0 2013-01-02  1.0  3  train  foo
```

Las columnas del DataFrame resultante tienen diferentes tipos de datos:

```
df2.dtypes
```

```
>>> A          float64
>>> B    datetime64[s]
>>> C          float32
```

```
>>> D            int32
>>> E         category
>>> F           object
>>> dtype: object
```

4.2.3 Visualización de datos

Podemos utilizar .head() y .tail() para ver las primeras y últimas filas del DataFrame respectivamente:

```
df.head() # Muestra los 6 primeros valores por defecto.
df.tail(3) # Podemos indicar el número en concreto que queremos mostrar.
```

```
>>>                    A         B         C         D
>>> 2013-01-01 -0.372123  0.303535  0.383237  0.136115
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681
>>>                    A         B         C         D
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203
```

Visualización de índices, columnas o los datos de un DataFrame:

```
df.index
```

```
>>> DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
>>>                '2013-01-05', '2013-01-06'],
>>>               dtype='datetime64[ns]', freq='D')
```

```
df.columns
```

```
>>> Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
df.values
```

```
>>> array([[-0.37212287,  0.30353452,  0.38323725,  0.13611495],
>>>        [ 0.48168621, -1.38876162,  0.08689103, -0.54728383],
>>>        [ 0.10853342,  1.43093991,  0.60282816,  1.62802581],
>>>        [-1.04492505, -0.20445581,  0.48502377,  0.44602683],
>>>        [-0.51326245, -0.55307354, -1.06432741,  0.10168053],
>>>        [ 0.12951894, -0.64276619,  0.99640293, -0.14820254]])
```

También podemos obtener una representación NumPy de los datos subyacentes en un DataFrame:

```
df.to_numpy()
```

```
>>> array([[-0.37212287,  0.30353452,  0.38323725,  0.13611495],
>>>        [ 0.48168621, -1.38876162,  0.08689103, -0.54728383],
>>>        [ 0.10853342,  1.43093991,  0.60282816,  1.62802581],
>>>        [-1.04492505, -0.20445581,  0.48502377,  0.44602683],
>>>        [-0.51326245, -0.55307354, -1.06432741,  0.10168053],
>>>        [ 0.12951894, -0.64276619,  0.99640293, -0.14820254]])
```

Ojo, los arrays de NumPy tienen un tipo de datos único, mientras que los DataFrames de pandas pueden tener diferentes tipos de datos por columna.

Además los arrays de NumPy no tienen etiquetas de columna (índices), mientras que los DataFrames de Pandas sí.

Un resumen estadístico rápido de tus datos:

```
df.describe()
```

```
>>>               A         B         C         D
>>> count  6.000000  6.000000  6.000000  6.000000
>>> mean  -0.201762 -0.175764  0.248343  0.269394
>>> std    0.549565  0.963729  0.708327  0.743548
>>> min   -1.044925 -1.388762 -1.064327 -0.547284
>>> 25%   -0.477978 -0.620343  0.160978 -0.085732
>>> 50%   -0.131795 -0.378765  0.434131  0.118898
>>> 75%    0.124273  0.176537  0.573377  0.368549
>>> max    0.481686  1.430940  0.996403  1.628026
```

Transposición de datos:

```
df.T
```

```
>>>    2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
>>> A   -0.372123    0.481686    0.108533   -1.044925   -0.513262    0.129519
>>> B    0.303535   -1.388762    1.430940   -0.204456   -0.553074   -0.642766
>>> C    0.383237    0.086891    0.602828    0.485024   -1.064327    0.996403
>>> D    0.136115   -0.547284    1.628026    0.446027    0.101681   -0.148203
```

Ordenación de las filas por índice:

```
df.sort_index(ascending=False)
```

```
>>>                    A         B         C         D
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284
>>> 2013-01-01 -0.372123  0.303535  0.383237  0.136115
```

Ordenación de las columnas por nombre:

```
df.sort_index(axis=1, ascending=False)
```

```
>>>                    D         C         B         A
>>> 2013-01-01  0.136115  0.383237  0.303535 -0.372123
>>> 2013-01-02 -0.547284  0.086891 -1.388762  0.481686
>>> 2013-01-03  1.628026  0.602828  1.430940  0.108533
>>> 2013-01-04  0.446027  0.485024 -0.204456 -1.044925
>>> 2013-01-05  0.101681 -1.064327 -0.553074 -0.513262
>>> 2013-01-06 -0.148203  0.996403 -0.642766  0.129519
```

Ordenación por valores:

```
df.sort_values(by="B")
```

```
>>>                    A         B         C         D
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027
>>> 2013-01-01 -0.372123  0.303535  0.383237  0.136115
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
```

4.2.4 Selección de datos

Es recomendable utilizar los métodos de acceso a datos optimizados de pandas, como DataFrame.at(), DataFrame.iat(), DataFrame.loc() y DataFrame.iloc() para un código de producción eficiente, aunque los métodos de acceso a datos estándar de Python también funcionan.

Seleccionar una fila según una etiqueta en concreto:

```
df.loc[dates[0]]
```

```
>>> A   -0.372123
>>> B    0.303535
>>> C    0.383237
>>> D    0.136115
>>> Name: 2013-01-01 00:00:00, dtype: float64
```

Seleccionar todas las filas (:) con columnas específicas:

```
df.loc[:, ["A", "B"]]
```

```
>>>                    A         B
>>> 2013-01-01 -0.372123  0.303535
>>> 2013-01-02  0.481686 -1.388762
>>> 2013-01-03  0.108533  1.430940
>>> 2013-01-04 -1.044925 -0.204456
>>> 2013-01-05 -0.513262 -0.553074
>>> 2013-01-06  0.129519 -0.642766
```

Mostrar un rango de etiquetas de filas y columnas:

```
df.loc["20130102":"20130104", ["A", "B"]] #Ambas están incluidas
```

```
>>>                    A         B
>>> 2013-01-02  0.481686 -1.388762
>>> 2013-01-03  0.108533  1.430940
>>> 2013-01-04 -1.044925 -0.204456
```

Seleccionar una fila y una etiqueta en concreto devuelve un solo valor:

```
df.loc[dates[0], "A"]
```

```
>>> np.float64(-0.37212287482996076)
```

Si solo queremos un escalar en concreto podriamos utilizar también:

```
df.at[dates[0], "A"]
```

```
>>> np.float64(-0.37212287482996076)
```

Seleccionar una fila específica por su posición:

```
df.iloc[3]
```

```
>>> A   -1.044925
>>> B   -0.204456
>>> C    0.485024
>>> D    0.446027
>>> Name: 2013-01-04 00:00:00, dtype: float64
```

Seleccionar un rango de filas y columnas por posición:

```
df.iloc[3:5, 0:2] #Incluye el primer valor pero no el último
```

```
>>>                    A         B
>>> 2013-01-04 -1.044925 -0.204456
>>> 2013-01-05 -0.513262 -0.553074
```

Seleccionar filas y columnas específicas mediante listas de índices de posición:

```
df.iloc[[1, 2, 4],[0, 2]]
```

```
>>>                    A         C
>>> 2013-01-02  0.481686  0.086891
>>> 2013-01-03  0.108533  0.602828
>>> 2013-01-05 -0.513262 -1.064327
```

Seleccionar un valor en concreto por posición:

```
df.iat[1, 1]
```

```
>>> np.float64(-1.3887616173268726)
```

Seleccionar filas donde df.A es mayor que 0:

```
df[df["A"] > 0]
```

```
>>>                    A         B         C         D
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203
```

Seleccionar valores de un DataFrame donde se cumple una condición:

```
df[df > 0]
```

```
>>>                    A         B         C         D
>>> 2013-01-01       NaN  0.303535  0.383237  0.136115
>>> 2013-01-02  0.481686       NaN  0.086891       NaN
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026
>>> 2013-01-04       NaN       NaN  0.485024  0.446027
>>> 2013-01-05       NaN       NaN       NaN  0.101681
>>> 2013-01-06  0.129519       NaN  0.996403       NaN
```

Usa el método isin() para filtrar:

```
df2 = df.copy()
df2["E"] = ["one", "one", "two", "three", "four", "three"]
df2[df2["E"].isin(["two", "four"])]
```

```
>>>                    A         B         C         D     E
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026   two
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681  four
```

4.2.5 Establecimiento de valores

Establecer una nueva columna alineando los datos automáticamente mediante su índice. En el siguiente ejemplo, como el rango de fechas empieza en el día 2, la serie se encaja a partir de este segundo día, atendiendo al índice de fechas que teníamos en el Data Frame. El primer día de nuestro DataFrame tiene por lo tanto un valor `NaN` en esa posición y como el sexto día de la serie, es decir el día 2013-01-07, no está en el índice, se obvia.

```
s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20130102", periods=6))
df["F"] = s1
df
```

```
>>>                    A         B         C         D    F
>>> 2013-01-01 -0.372123  0.303535  0.383237  0.136115  NaN
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203  5.0
```

Establecer valores por posición:

```
df.at[dates[0], "A"] = 0
df.iat[0, 1] = 0
df
```

```
>>>                    A         B         C         D    F
>>> 2013-01-01  0.000000  0.000000  0.383237  0.136115  NaN
>>> 2013-01-02  0.481686 -1.388762  0.086891 -0.547284  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  1.628026  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  0.446027  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  0.101681  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403 -0.148203  5.0
```

Establecer mediante un array de NumPy:

```
df.loc[:, "D"] = np.array([5] * len(df))
df
```

```
>>>                    A         B         C    D    F
>>> 2013-01-01  0.000000  0.000000  0.383237  5.0  NaN
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
```

4.2.6 Copia de datos

¡Importante! si utilizamos la asignación `df2 = df`, cualquier modificación en `df2` también modificará `df`. Para evitar esto, podemos utilizar el método `copy()`.

Copia de un DataFrame:

```
df2 = df.copy()
df2
```

```
>>>                    A         B         C    D    F
>>> 2013-01-01  0.000000  0.000000  0.383237  5.0  NaN
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
```

Modificación en una copia:

```
df2 = df.copy()
df2[df2 > 0] = -df2
df2
```

```
>>>                    A         B         C    D    F
>>> 2013-01-01  0.000000  0.000000 -0.383237 -5.0  NaN
>>> 2013-01-02 -0.481686 -1.388762 -0.086891 -5.0 -1.0
>>> 2013-01-03 -0.108533 -1.430940 -0.602828 -5.0 -2.0
>>> 2013-01-04 -1.044925 -0.204456 -0.485024 -5.0 -3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327 -5.0 -4.0
>>> 2013-01-06 -0.129519 -0.642766 -0.996403 -5.0 -5.0
```

4.2.7 Manipulación de datos faltantes

Eliminar filas con datos faltantes.

```
df.dropna()
```

```
>>>                    A         B         C    D    F
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
```

Rellenar datos faltantes:

```
df.fillna(value=5)
```

```
>>>                    A         B         C    D    F
>>> 2013-01-01  0.000000  0.000000  0.383237  5.0  5.0
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
```

Obtener valores lógicos que indiquen los valores faltantes:

```
pd.isna(df)
```

```
>>>                 A      B      C      D      F
>>> 2013-01-01  False  False  False  False   True
>>> 2013-01-02  False  False  False  False  False
>>> 2013-01-03  False  False  False  False  False
>>> 2013-01-04  False  False  False  False  False
>>> 2013-01-05  False  False  False  False  False
>>> 2013-01-06  False  False  False  False  False
```

Ojo, que en este tipo de operaciones no se modifica el DataFrame a no ser que se vuelva a asignar:

```
df # Tras las operaciones anteriores, df sigue igual

df=df.dropna() # Al reasignarlo lo modificamos
df # Ahora ya no existe la fila con valores faltantes en df
```

```
>>>                    A         B         C    D    F
>>> 2013-01-01  0.000000  0.000000  0.383237  5.0  NaN
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
>>>                    A         B         C    D    F
>>> 2013-01-02  0.481686 -1.388762  0.086891  5.0  1.0
>>> 2013-01-03  0.108533  1.430940  0.602828  5.0  2.0
>>> 2013-01-04 -1.044925 -0.204456  0.485024  5.0  3.0
>>> 2013-01-05 -0.513262 -0.553074 -1.064327  5.0  4.0
>>> 2013-01-06  0.129519 -0.642766  0.996403  5.0  5.0
```

4.2.8 Aplicación de funciones

Aplicar funciones a los datos:

```
# suma de cada columna
df.apply(np.sum)
```

```
>>> A    -0.838449
>>> B    -1.358117
>>> C     1.106818
>>> D    25.000000
>>> F    15.000000
>>> dtype: float64
```

Aplicar funciones a los datos en el otro eje:

```
# suma de cada fila
df.apply(np.sum, axis=1)
```

```
>>> 2013-01-02     5.179816
>>> 2013-01-03     9.142301
>>> 2013-01-04     7.235643
>>> 2013-01-05     6.869337
>>> 2013-01-06    10.483156
>>> Freq: D, dtype: float64
```

4.2.9 Operaciones con cadenas de caracteres

Series tiene un atributo str que permite operaciones vectorizadas en cadenas de caracteres, como en el siguiente ejemplo que transforma todas las cadenas de una lista a minúsculas:

```
s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])
s.str.lower()
```

```
>>> 0       a
>>> 1       b
>>> 2       c
>>> 3    aaba
>>> 4    baca
>>> 5     NaN
>>> 6    caba
>>> 7     dog
>>> 8     cat
>>> dtype: object
```

4.2.10 Concatenación

Concatenar objetos de Pandas:

```
df = pd.DataFrame(np.random.randn(10, 4))
df
pieces = [df[:3], df[3:7], df[7:]]
pd.concat(pieces)
```

```
>>>           0         1         2         3
>>> 0 -0.037845  0.730344  1.468646  0.526169
>>> 1  0.173226 -0.816924 -0.933567 -2.387303
>>> 2 -1.172623  1.467394  1.179479  0.214656
>>> 3 -0.228447  2.198458  1.398075  0.533423
```

```
>>> 4  1.153888  0.286850  0.798731 -0.365671
>>> 5 -0.726036  0.213158 -2.172033  0.364350
>>> 6  1.345347  0.891159 -0.254082 -0.575979
>>> 7  1.366230 -0.729820  0.532666  0.264053
>>> 8 -0.378257  0.658867 -0.027134  1.297477
>>> 9  0.483181  0.865827  0.957777 -0.990842
>>>           0         1         2         3
>>> 0 -0.037845  0.730344  1.468646  0.526169
>>> 1  0.173226 -0.816924 -0.933567 -2.387303
>>> 2 -1.172623  1.467394  1.179479  0.214656
>>> 3 -0.228447  2.198458  1.398075  0.533423
>>> 4  1.153888  0.286850  0.798731 -0.365671
>>> 5 -0.726036  0.213158 -2.172033  0.364350
>>> 6  1.345347  0.891159 -0.254082 -0.575979
>>> 7  1.366230 -0.729820  0.532666  0.264053
>>> 8 -0.378257  0.658867 -0.027134  1.297477
>>> 9  0.483181  0.865827  0.957777 -0.990842
```

4.2.11 Mezcla

Mezclas del estilo de SQL join, donde podemos especificar una columna clave para unir dos DataFrames. Por ejemplo:

```
left = pd.DataFrame({ "key": ["foo", "bar"], "lval": [1, 2] })
left
```

```
>>>    key  lval
>>> 0  foo     1
>>> 1  bar     2
```

```
right = pd.DataFrame({ "key": ["foo", "bar"], "rval": [4, 5] })
right
```

```
>>>    key  rval
>>> 0  foo     4
>>> 1  bar     5
```

```
pd.merge(left, right, on="key")
```

```
>>>    key  lval  rval
>>> 0  foo     1     4
>>> 1  bar     2     5
```

Si las claves estuvieran repetidas se duplicarían las entradas en la mezcla para cada combinación:

```
left = pd.DataFrame({ "key": ["foo", "foo"], "lval": [1, 2] })
left
```

```
>>>    key  lval
>>> 0  foo     1
>>> 1  foo     2
```

```
right = pd.DataFrame({ "key": ["foo", "foo"], "rval": [4, 5] })
right
```

```
>>>    key  rval
>>> 0  foo     4
>>> 1  foo     5
```

```
pd.merge(left, right, on="key")
```

```
>>>    key  lval  rval
>>> 0  foo     1     4
>>> 1  foo     1     5
>>> 2  foo     2     4
>>> 3  foo     2     5
```

4.2.12 Agrupamiento

El proceso de "group by" incluye los pasos de dividir, aplicar una función y combinar los resultados.

```
df = pd.DataFrame({ "A": ["foo", "bar", "foo", "bar", "foo", "bar",
  "foo", "foo"],
  "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
  "C": np.random.randn(8),
  "D": np.random.randn(8) })

df
```

```
>>>      A      B         C         D
>>> 0  foo    one -0.708833  0.910001
>>> 1  bar    one -1.032807  2.114205
>>> 2  foo    two  1.423682 -1.493102
>>> 3  bar  three -0.270420  0.396490
>>> 4  foo    two -0.103991  0.562336
>>> 5  bar    two -1.315388  0.526037
>>> 6  foo    one -2.317629  0.395497
>>> 7  foo  three  0.002963  0.496027
```

```
df.groupby("A")[["C", "D"]].sum()
```

```
>>>             C         D
>>> A
>>> bar -2.618615  3.036732
>>> foo -1.703807  0.870758
```

Agrupar por múltiples columnas forma un índice jerárquico, al que luego se pueden aplicar a funciones de agregación:

```
df.groupby(["A", "B"]).sum()
```

```
>>>                   C         D
>>> A   B
>>> bar one   -1.032807  2.114205
>>>     three -0.270420  0.396490
>>>     two   -1.315388  0.526037
>>> foo one   -3.026462  1.305497
>>>     three  0.002963  0.496027
>>>     two    1.319692 -0.930766
```

4.2.13 Datos categóricos

Pandas inculye el tipo de datos categóricos, que pueden tomar un número limitado de valores. Serían el equivalente a los factores de R.

```
df = pd.DataFrame(
    {"id": [1, 2, 3, 4, 5, 6],
    "raw_grade": ["a", "b", "b", "a", "a", "e"]}
)
```

Convertir la columna “raw_grade” a un tipo de datos categóricos:

```
df["grade"] = df["raw_grade"].astype("category")
df["grade"]
```

```
>>> 0    a
>>> 1    b
>>> 2    b
>>> 3    a
>>> 4    a
>>> 5    e
>>> Name: grade, dtype: category
>>> Categories (3, object): ['a', 'b', 'e']
```

Renombrar las categorías a un nombre más significativo:

```
new_categories = ["very good", "good", "very bad"]
df["grade"] = df["grade"].cat.rename_categories(new_categories)
```

Reordenar las categorías y añadir las que faltan:

```
df["grade"] = df["grade"].cat.set_categories(
    ["very bad", "bad", "medium", "good", "very good"]
)
```

La ordenación es por el orden de las categorías, no por el orden lexicográfico:

```
df.sort_values(by="grade")
```

```
>>>    id raw_grade      grade
>>> 5   6         e   very bad
>>> 1   2         b       good
>>> 2   3         b       good
>>> 0   1         a  very good
>>> 3   4         a  very good
>>> 4   5         a  very good
```

Agrupar por una columna categórica con `observed=false` también muestra las categorías vacías:

```
df.groupby("grade", observed=False).size()
```

```
>>> grade
>>> very bad     1
>>> bad          0
>>> medium       0
>>> good         2
>>> very good    3
>>> dtype: int64
```

4.2.14 Exportación e importación de datos

Escribe un DataFrame a un archivo CSV:

```
  df.to_csv("archivo.csv")
```

Lee un DataFrame desde un archivo CSV:

```
  pd.read_csv("archivo.csv")
```

Excel

Escribe un DataFrame a un archivo Excel:

```
  df.to_excel("archivo.xlsx", sheet_name="Hoja1")
```

Lee un DataFrame desde un archivo Excel:

```
pd.read_excel("archivo.xlsx", "Hoja1", index_col=None, na_values=["NA"])
```

4.3 Matplotlib.

Matplotlib es un paquete de Python que se utiliza para crear gráficos y visualizaciones de datos. Algunos de los tipos de gráficos que se pueden crear con Matplotlib son:

- Gráficos de líneas: Los gráficos de líneas se utilizan para representar datos en función de una variable continua. Se pueden crear utilizando la función `plt.plot()`.
- Gráficos de barras: Los gráficos de barras se utilizan para representar datos en función de una variable categórica. Se pueden crear utilizando la función `plt.bar()`.
- Histogramas: Los histogramas se utilizan para representar la distribución de datos en función de una variable continua. Se pueden crear utilizando la función `plt.hist()`. Para añadir una línea de densidad de probabilidad, lo más cómodo sería utilizar otro paquete de graficación llamado Seaborn, que se basa en Matplotlib y tiene una función llamada `sns.distplot()` que realiza esta tarea.
- Gráficos de dispersión: Los gráficos de dispersión se utilizan para representar datos en función de dos variables continuas. Se pueden crear utilizando la función `plt.scatter()`.
- Gráficos de caja: Los gráficos de caja se utilizan para representar datos en función de una variable categórica y una variable continua. Se pueden crear utilizando la función `plt.boxplot()`.
- Gráficos de violín: Los gráficos de violín se utilizan para representar datos en función de una variable categórica y una variable continua. Son similares a los diagramas de caja, pero muestran la distribución de los datos de forma más detallada. Grafica una estimación de la densidad de probabilidad de los datos de forma vertical. Se pueden crear utilizando la función `plt.violinplot()`.
- Gráficos de sectores: Los gráficos de sectores se utilizan para representar datos en función de una variable categórica. Se pueden crear utilizando la función `plt.pie()`.

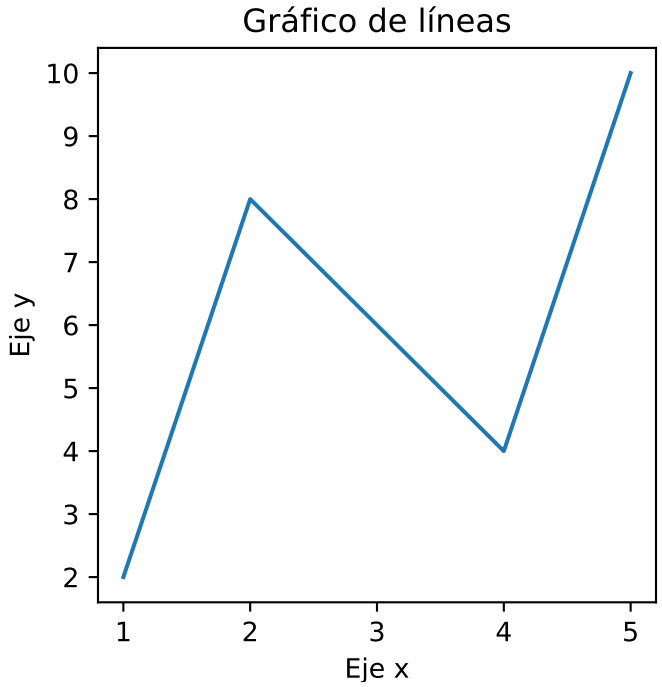
Para utilizar Matplotlib en Python, primero es necesario importar el paquete Matplotlib utilizando la palabra clave `import`. Por ejemplo, para importar Matplotlib y asignarle un alias `plt`, se puede utilizar el siguiente comando:

```
import matplotlib.pyplot as plt
```

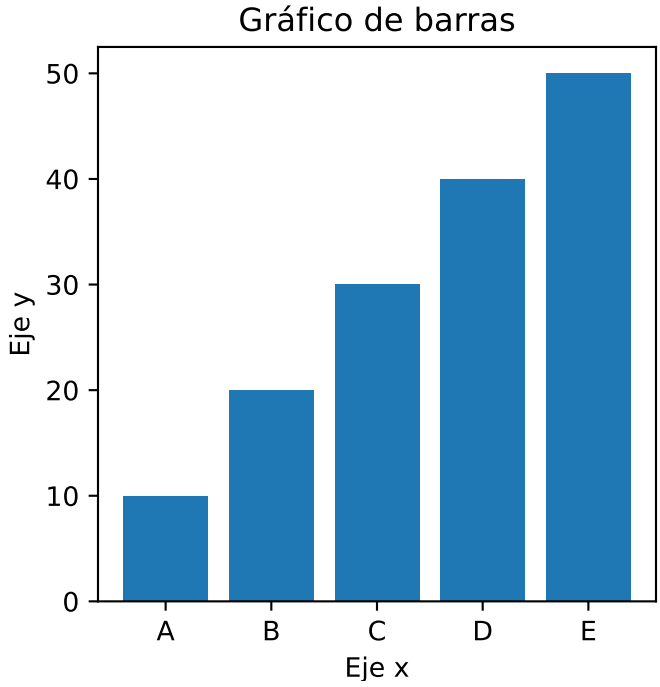
Una vez importado Matplotlib, se puede utilizar mediante el alias `plt`.

Ejemplos de gráficos:

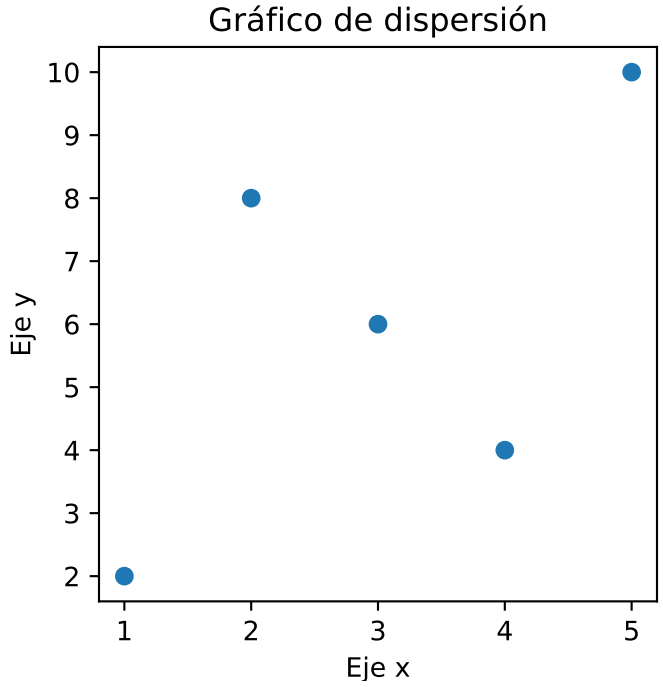
```
import matplotlib.pyplot as plt

# Gráfico de líneas

x = [1, 2, 3, 4, 5]
y = [2, 8, 6, 4, 10]

plt.plot(x, y)
plt.xlabel('Eje x')
plt.ylabel('Eje y')
plt.title('Gráfico de líneas')
plt.show()
```



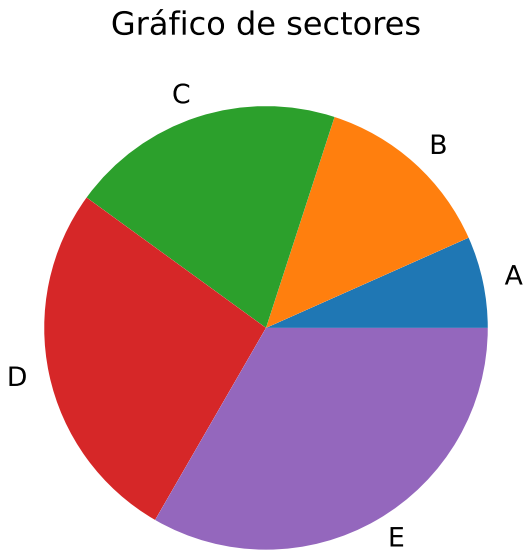
```
# Gráfico de barras

x = ['A', 'B', 'C', 'D', 'E']
y = [10, 20, 30, 40, 50]

plt.bar(x, y)
plt.xlabel('Eje x')
plt.ylabel('Eje y')
plt.title('Gráfico de barras')
plt.show()
```



```
# Gráfico de dispersión

x = [1, 2, 3, 4, 5]
y = [2, 8, 6, 4, 10]

plt.scatter(x, y)
plt.xlabel('Eje x')
plt.ylabel('Eje y')
plt.title('Gráfico de dispersión')
plt.show()
```

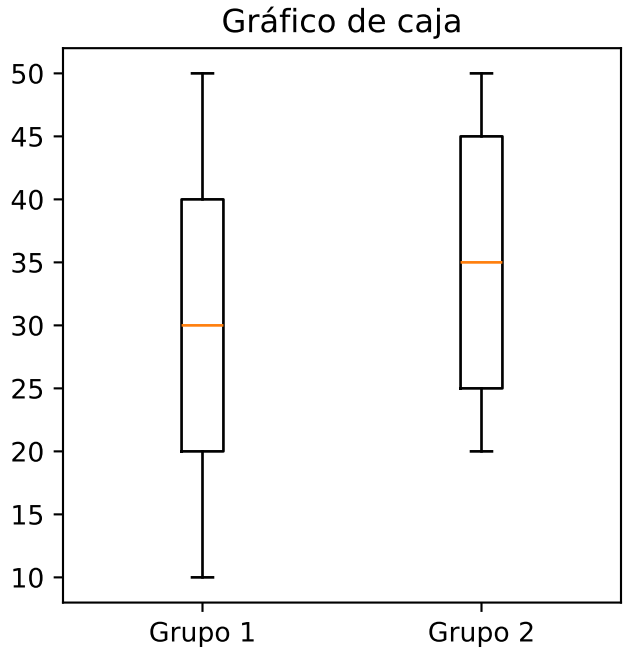


```
# Gráfico de sectores

x = [10, 20, 30, 40, 50]
labels = ['A', 'B', 'C', 'D', 'E']

plt.pie(x, labels=labels);
plt.title('Gráfico de sectores')
plt.show()
```

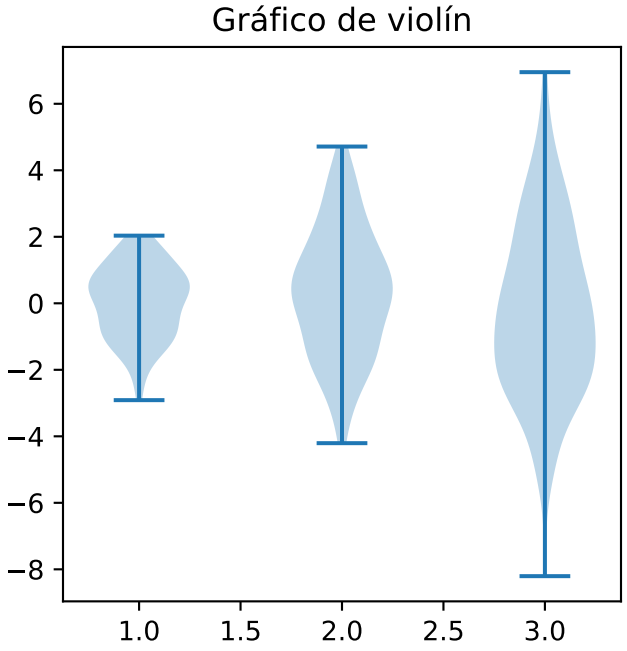


```
# Gráfico de caja

datos = [[10, 20, 30, 40, 50], [20, 25, 35, 45, 50]]
plt.boxplot(datos, labels= ["Grupo 1", "Grupo 2"]);
plt.title('Gráfico de caja')
plt.show()
```

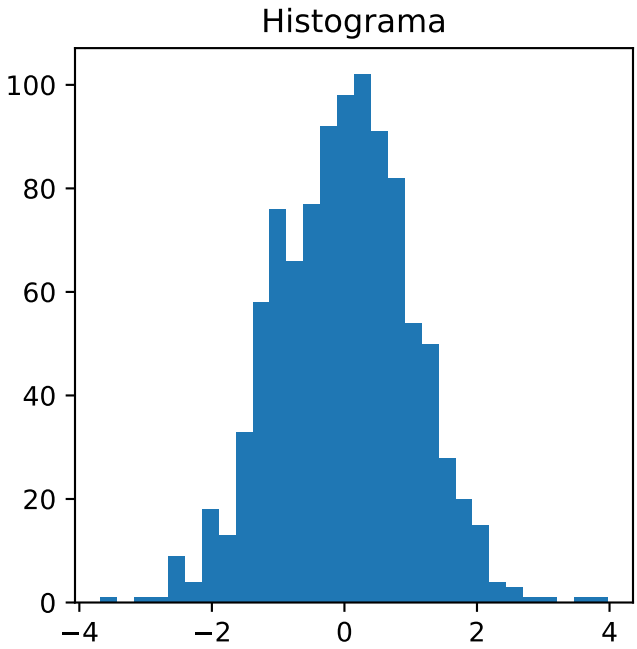
```
>>> <string>:1: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been re
```



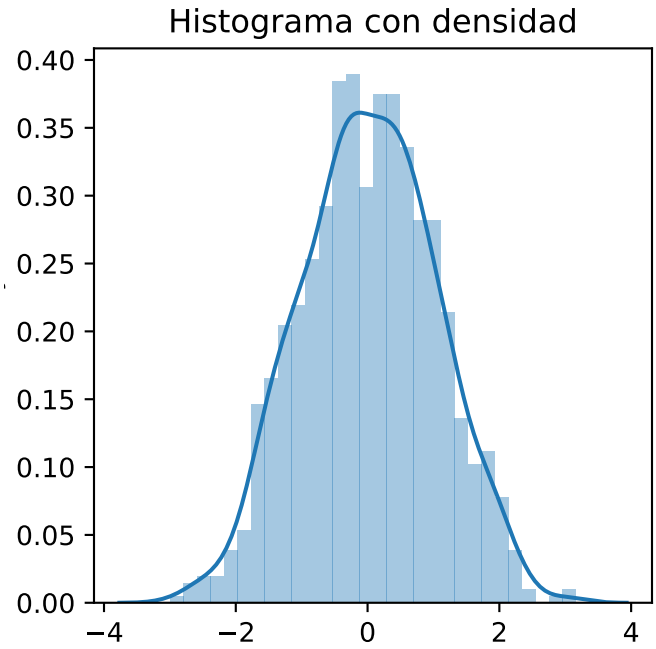
```
# Gráfico de violín

# Aprovechamos para mostrar cómo se puede realizar un bucle en una sola línea,
# la siguiente línea de código nos construye tres listas de 100 números aleatorios
# con media 0 y desviación estándar 1, 2 y 3 respectivamente.

datos = [np.random.normal(0, std, 100) for std in range(1, 4)]
plt.violinplot(datos);
plt.title('Gráfico de violín')
plt.show()
```



```
# Histograma

datos = np.random.randn(1000)
plt.hist(datos, bins=30)
plt.title('Histograma')
plt.show()
```



```
# Histograma con densidad de probabilidad

import seaborn as sns

datos = np.random.randn(1000)
sns.distplot(datos, bins=30, kde=True)
plt.title('Histograma con densidad')
plt.show()
```



Para más información sobre Matplotlib, se puede consultar la documentación oficial de Matplotlib en el siguiente enlace: https://matplotlib.org/stable/contents.html.

Un buen comienzo para aprender Matplotlib es el tutorial de Matplotlib que se puede encontrar en el siguiente enlace: https://matplotlib.org/stable/tutorials/index.html.

4.4 SciKit-learn.

Scikit-learn es un paquete de Python que se utiliza para el aprendizaje automático/estadístico y para minería de datos en general. Veremos algunos ejemplos de cómo utilizar Scikit-learn para entrenar modelos de aprendizaje en Python en el siguiente capítulo.

Por lo general el nombre de instalación de un paquete coincide con su nombre de uso, pero en este caso para instalar el paquete se utiliza el nombre "scikit-learn", y para importarlo se utiliza "sklearn".

Para utilizar Scikit-learn en Python, primero es necesario importar el paquete Scikit-learn utilizando la palabra clave `import`. Por ejemplo, para importar Scikit-learn y asignarle un alias `sk`, se puede utilizar el siguiente comando:

```
import sklearn as sk
```

Una vez importado Scikit-learn, se puede acceder a las funciones utilizando el alias `sk`. Por ejemplo, para crear un modelo de regresión lineal a partir de un conjunto de datos, se puede utilizar la función `LinearRegression()` de la siguiente manera:

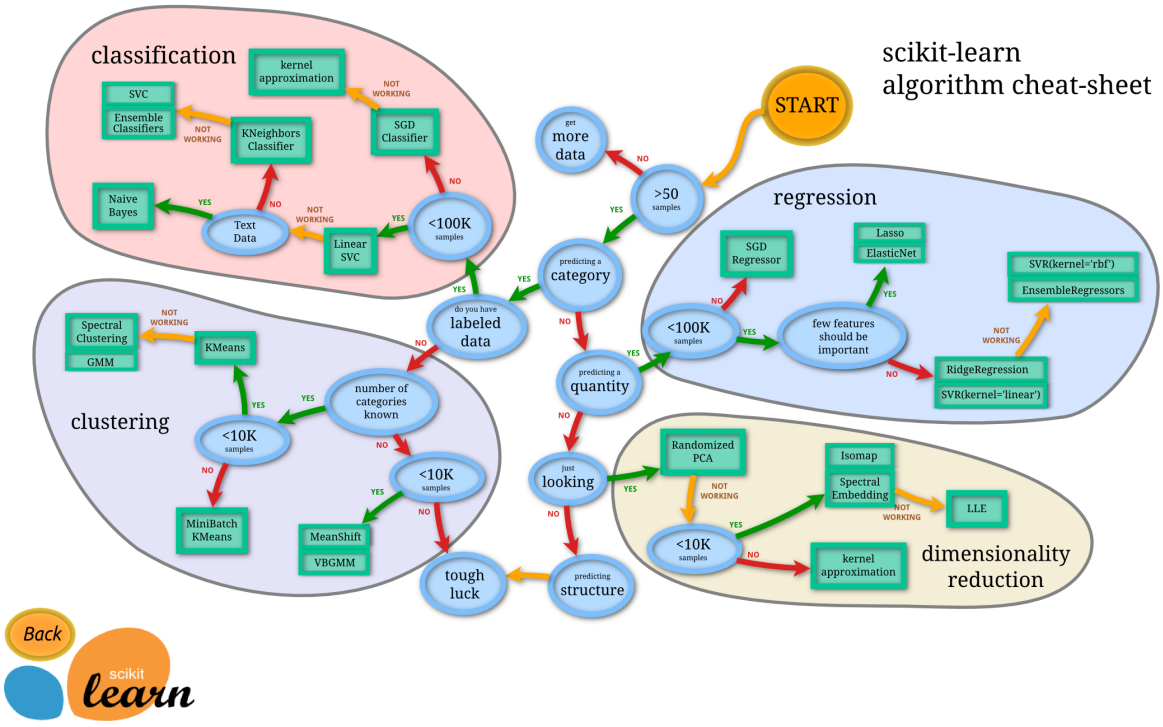
Ejemplo de regresión lineal con Scikit-learn:

```
import numpy as np
import sklearn.linear_model as lm #importamos solo el subpaquete

# Datos de ejemplo
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

# Crear el modelo de regresión lineal
modelo = lm.LinearRegression()

# Entrenar el modelo
fit = modelo.fit(X, y)

# Predecir valores
predicciones = fit.predict(X)

print(predicciones)
```

```
>>> [ 2.  4.  6.  8. 10.]
```

Scikit-learn contiene modelos para clasificación, regresión, clustering y reducción de dimensionalidad. Además, proporciona funciones para dividir los datos en conjuntos de entrenamiento y prueba, para evaluar los modelos, para ajustar los hiperparámetros, para realizar validación cruzada, para realizar selección de modelos, para realizar preprocesamiento de datos, para realizar selección de características...

En la siguiente imagen se muestra un mapa de aprendizaje automático que muestran diferentes algoritmos de minería de datos, agrupados en clasificación, regresión, reducción de la dimensionalidad y agrupamiento, y cómo se relacionan entre sí. Se puede encontrar de manera interactiva en el siguiente enlace: https://scikit-learn.org/stable/machine_learning_map.html.



Para más información sobre Scikit-learn, se puede consultar la documentación oficial de Scikit-learn en el siguiente enlace: https://scikit-learn.org/stable/documentation.html.

Un buen comienzo para aprender Scikit-learn es el tutorial básico de Scikit-learn que se puede encontrar en el siguiente enlace: https://scikit-learn.org/stable/tutorial/basic/tutorial.html.

5 Análisis estadísticos elementales de datos con Python.

En Python, al igual que en R, es posible realizar análisis estadísticos elementales de datos, como cálculos de estadísticas descriptivas, contrastes de hipótesis, modelos de regresión, análisis de varianza, aprendizaje estadístico, etc. A diferencia de R, en Python no existe un paquete base que proporcione todas las funciones y métodos necesarios para realizar análisis estadísticos, por lo que es necesario utilizar diferentes paquetes para realizarlas. Apoyándonos en los paquetes NumPy, Pandas, Matplotlib y Scikit-learn estudiados en el capítulo anterior, podemos realizar análisis estadísticos elementales de datos en Python.

5.1 Estadística descriptiva.

Para realizar cálculos de estadística descriptiva en Python, se pueden utilizar los paquetes NumPy o Pandas. Algunas de las funciones y métodos más comunes son:

- Media: La media es el valor promedio de un conjunto de datos. La media se puede calcular utilizando la función `np.mean()` de NumPy o el método `mean()` de Pandas.
- Mediana: La mediana es el valor central de un conjunto de datos. La mediana se puede calcular utilizando la función `np.median()` de NumPy o el método `median()` de Pandas.
- Moda: La moda es el valor que aparece con mayor frecuencia en un conjunto de datos. La moda se puede calcular utilizando la función de NumPy `np.argmax(np.bincount())` o el método `mode()` de Pandas.
- Desviación estándar: La desviación estándar es una medida de dispersión que indica cuánto varían los valores de un conjunto de datos con respecto a la media. La desviación estándar se puede calcular utilizando la función `np.std()` de NumPy o el método `std()` de Pandas.
- Varianza: La varianza es una medida de dispersión que indica cuánto varían los valores de un conjunto de datos con respecto a la media. La varianza se puede calcular utilizando la función `np.var()` de NumPy o el método `var()` de Pandas.
- Cuantiles: Los cuantiles son valores que dividen un conjunto de datos en partes iguales. Los cuantiles se pueden calcular utilizando la función `np.percentile()` de NumPy o el método `quantile()` de Pandas.
- Correlación: La correlación es una medida de la relación entre dos variables. La correlación se puede calcular utilizando la función `np.corrcoef()` de NumPy o el método `corr()` de Pandas.
- Covarianza: La covarianza es una medida de la relación entre dos variables. La covarianza se puede calcular utilizando la función `np.cov()` de NumPy o el método `cov()` de Pandas.

- Resumen general: Podemos hacer un resumen de los estadísticos más comunes utilizando el método `describe()` de Pandas.
- Tabla de frecuencias: Podemos hacer una tabla de frecuencias de una variable categórica utilizando el método `value_counts()` de Pandas. Si se quiere que los valores queden ordenados por el índice deberíamos añadir el método `sort_index()`.

Ejemplo de cálculos de estadísticas descriptivas con Pandas:

```
import pandas as pd

data = {
    'Edad': [23, 25, 31, 35, 26, 29, 42, 30, 34, 28],
    'Salario': [50000, 54000, 61000, 67000, 52000,
                58000, 75000, 59000, 64000, 57000],
    'Experiencia': [1, 3, 8, 10, 3, 5, 15, 6, 9, 4]
}

df = pd.DataFrame(data)

print("Tabla de datos:")
print(df)

print("\nTabla de frecuencias de la experiencia:")
print(df['Experiencia'].value_counts().sort_index())

print("\nEstadísticas descriptivas:")
print(df.describe())  # Resumen estadístico básico

print("\nMedia de cada columna:")
print(df.mean())

print("\nMediana de cada columna:")
print(df.median())

print("\nVarianza de cada columna:")
print(df.var())

print("\nCorrelación entre columnas:")
print(df.corr())

print("\nCovarianza entre columnas:")
print(df.cov())
```

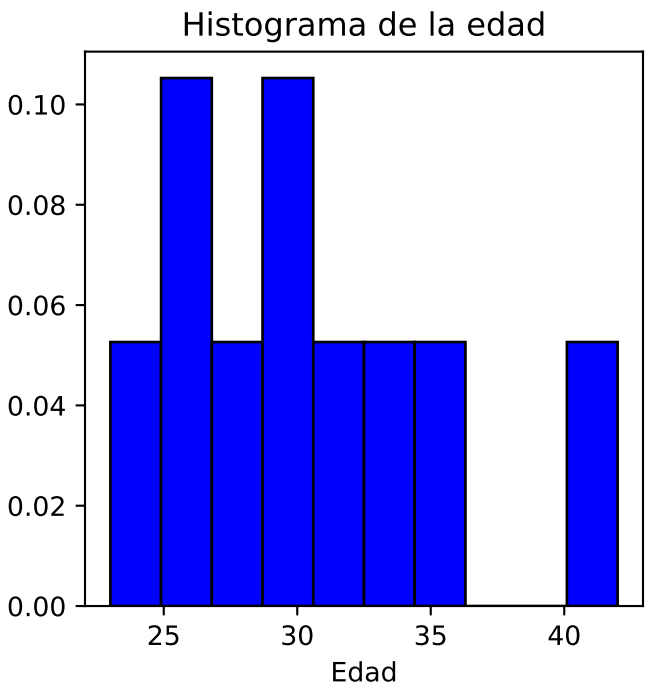
```
print("\nPercentiles de cada columna:")
print(df.quantile([0.25, 0.5, 0.75]))
```

```
>>> Tabla de datos:
>>>    Edad  Salario  Experiencia
>>> 0    23    50000            1
>>> 1    25    54000            3
>>> 2    31    61000            8
>>> 3    35    67000           10
>>> 4    26    52000            3
>>> 5    29    58000            5
>>> 6    42    75000           15
>>> 7    30    59000            6
>>> 8    34    64000            9
>>> 9    28    57000            4
>>>
>>> Tabla de frecuencias de la experiencia:
>>> Experiencia
>>> 1     1
>>> 3     2
>>> 4     1
>>> 5     1
>>> 6     1
>>> 8     1
>>> 9     1
>>> 10    1
>>> 15    1
>>> Name: count, dtype: int64
>>>
>>> Estadísticas descriptivas:
>>>             Edad       Salario  Experiencia
>>> count  10.000000     10.000000    10.000000
>>> mean   30.300000  59700.000000     6.400000
>>> std     5.578729   7484.057129     4.168666
>>> min    23.000000  50000.000000     1.000000
>>> 25%    26.500000  54750.000000     3.250000
>>> 50%    29.500000  58500.000000     5.500000
>>> 75%    33.250000  63250.000000     8.750000
>>> max    42.000000  75000.000000    15.000000
>>>
>>> Media de cada columna:
```

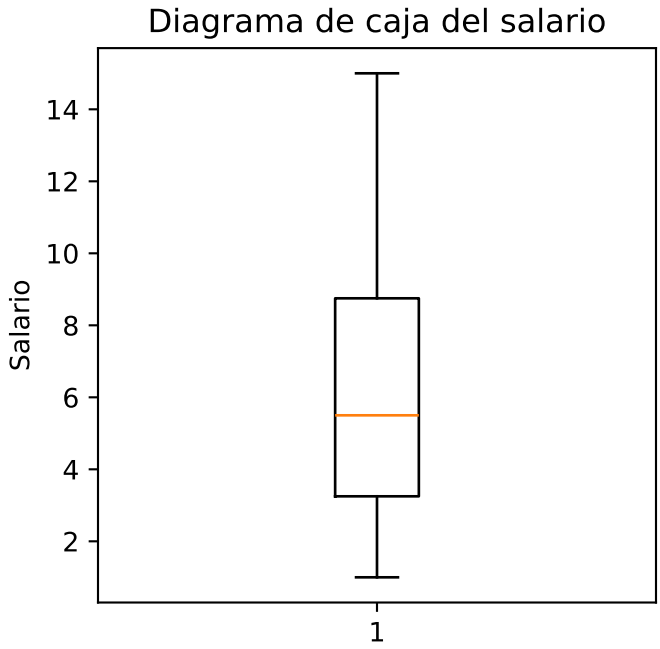
```
>>> Edad              30.3
>>> Salario        59700.0
>>> Experiencia        6.4
>>> dtype: float64
>>>
>>> Mediana de cada columna:
>>> Edad              29.5
>>> Salario        58500.0
>>> Experiencia        5.5
>>> dtype: float64
>>>
>>> Varianza de cada columna:
>>> Edad           3.112222e+01
>>> Salario        5.601111e+07
>>> Experiencia    1.737778e+01
>>> dtype: float64
>>>
>>> Correlación entre columnas:
>>>                   Edad   Salario  Experiencia
>>> Edad          1.000000  0.992379     0.992820
>>> Salario       0.992379  1.000000     0.990787
>>> Experiencia   0.992820  0.990787     1.000000
>>>
>>> Covarianza entre columnas:
>>>                       Edad       Salario   Experiencia
>>> Edad             31.122222  4.143333e+04     23.088889
>>> Salario       41433.333333  5.601111e+07  30911.111111
>>> Experiencia      23.088889  3.091111e+04     17.377778
>>>
>>> Percentiles de cada columna:
>>>         Edad  Salario  Experiencia
>>> 0.25  26.50  54750.0         3.25
>>> 0.50  29.50  58500.0         5.50
>>> 0.75  33.25  63250.0         8.75
```

Las medidas numéricas podemos complementarlas con medidas gráficas, como histogramas, diagramas de caja, diagramas de dispersión, etc. Para ello, podemos utilizar el paquete Matplotlib como hemos visto en el capítulo anterior. Por ejemplo, para crear un histograma de la edad, un diagrama de caja del salario y un diagrama de dispersión de la experiencia y el salario, se puede utilizar el siguiente código:

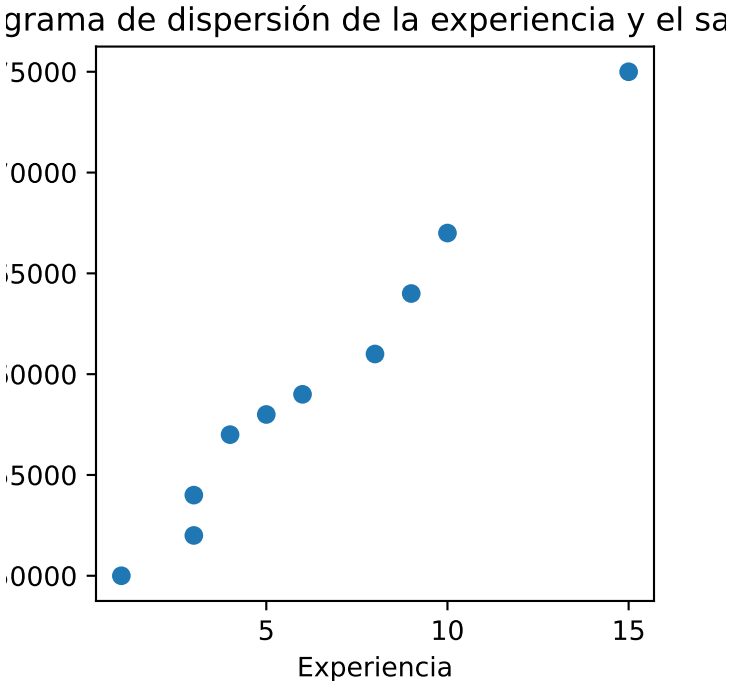
```
import matplotlib.pyplot as plt

# Histograma de la edad incuyendo la función de densidad
# estimada mediante un kernel gaussiano

plt.hist(df['Edad'], color='blue', edgecolor='black', density=True)
plt.title('Histograma de la edad')
plt.xlabel('Edad')
plt.ylabel('Frecuencia')
plt.show()
```



```
# Diagrama de caja del salario

plt.boxplot(df['Experiencia']);
plt.title('Diagrama de caja del salario')
plt.ylabel('Salario')
plt.show()
```



```
# Diagrama de dispersión de la experiencia y el salario

plt.scatter(df['Experiencia'], df['Salario'])
plt.title('Diagrama de dispersión de la experiencia y el salario')
plt.xlabel('Experiencia')
plt.ylabel('Salario')
plt.show()
```



5.2 Contrastes de hipótesis.

En Python, al igual que en R, es posible realizar contrastes de hipótesis para determinar si estadísticamente podemos decantarnos por una hipótesis nula o alternativa. Para realizar pruebas de hipótesis en Python, se pueden utilizar las funciones del paquete SciPy, aunque también se pueden utilizar otros paquetes como StatsModels, Pingouin, etc. Algunas de las pruebas de hipótesis más comunes que se pueden realizar en Python son:

- Prueba t de una muestra: La prueba t de una muestra se utiliza para determinar si la media de una muestra es significativamente diferente de un valor conocido. La prueba t de una muestra se puede realizar utilizando la función `ttest_1samp()` de SciPy.
- Prueba t de dos muestras: La prueba t de dos muestras se utiliza para determinar si hay una diferencia significativa entre las medias de dos muestras independientes. La prueba t de dos muestras se puede realizar utilizando la función `ttest_ind()` de SciPy.
- Prueba t de muestras emparejadas: La prueba t de muestras emparejadas se utiliza para determinar si hay una diferencia significativa entre las medias de dos muestras emparejadas. La prueba t de muestras emparejadas se puede realizar utilizando la función `ttest_rel()` de SciPy.
- Prueba ANOVA: La prueba ANOVA se utiliza para determinar si hay una diferencia significativa entre las medias de tres o más grupos independientes. La prueba ANOVA se puede realizar utilizando la función `f_oneway()` de SciPy.
- Prueba de chi-cuadrado: La prueba de chi-cuadrado se utiliza para determinar si hay una relación significativa entre dos variables categóricas. La prueba de chi-cuadrado se puede realizar utilizando la función `chi2_contingency()` de SciPy.
- Prueba de Kolmogorov-Smirnov: La prueba de Kolmogorov-Smirnov se utiliza para determinar si dos muestras provienen de la misma distribución. La prueba de Kolmogorov-Smirnov se puede realizar utilizando la función `ks_2samp()` de SciPy.
- Prueba de Shapiro-Wilk: La prueba de Shapiro-Wilk se utiliza para determinar si una muestra proviene de una distribución normal. La prueba de Shapiro-Wilk se puede realizar utilizando la función `shapiro()` de SciPy.
- Prueba de Levene: La prueba de Levene se utiliza para determinar si las varianzas de dos o más grupos son iguales. La prueba de Levene se puede realizar utilizando la función `levene()` de SciPy.

Ejemplo de pruebas de hipótesis con SciPy:

```
import numpy as np
import scipy.stats as st

# Datos de ejemplo

grupo1 = np.array([23, 25, 31, 35, 26, 29, 42, 30, 34, 28])
grupo2 = np.array([25, 27, 33, 37, 28, 31, 44, 32, 36, 30])

# Prueba t de dos muestras

t_stat, p_value = st.ttest_ind(grupo1, grupo2)
print("Prueba t de dos muestras:")
print("Estadístico t:", t_stat)
print("Valor p:", p_value)

if p_value < 0.05:
    print("Hay una diferencia significativa entre los grupos.")
else:
    print("No hay una diferencia significativa entre los grupos.")

# Prueba de chi-cuadrado

tabla = np.array([[10, 20, 30], [15, 25, 35]])

chi2, p_value, dof, expected = st.chi2_contingency(tabla)

print("\nPrueba de chi-cuadrado:")
print("Estadístico chi-cuadrado:", chi2)
print("Valor p:", p_value)

if p_value < 0.05:
    print("Hay una relación significativa entre las variables.")
else:
    print("No hay una relación significativa entre las variables.")

# Prueba de Shapiro-Wilk

stat, p_value = st.shapiro(grupo1)

print("\nPrueba de Shapiro-Wilk:")
print("Estadístico de prueba:", stat)
print("Valor p:", p_value)
```

```
if p_value < 0.05:
    print("La muestra no proviene de una distribución normal.")
else:
    print("La muestra proviene de una distribución normal.")
```

```
>>> Prueba t de dos muestras:
>>> Estadístico t: -0.8016405884111758
>>> Valor p: 0.4332168119869395
>>> No hay una diferencia significativa entre los grupos.
>>>
>>> Prueba de chi-cuadrado:
>>> Estadístico chi-cuadrado: 0.27692307692307694
>>> Valor p: 0.870696738961232
>>> No hay una relación significativa entre las variables.
>>>
>>> Prueba de Shapiro-Wilk:
>>> Estadístico de prueba: 0.9501297629479081
>>> Valor p: 0.6700359857350207
>>> La muestra proviene de una distribución normal.
```

5.3 Modelos de regresión.

En Python, al igual que en R, es posible ajustar modelos de regresión para predecir una variable continua en función de una o más variables independientes. Para ajustar modelos de regresión en Python, se pueden utilizar varios paquetes, como por ejemplo SciKit-learn. Hay que tener en cuenta, que al ser Python un lenguaje de programación generalista, no tiene tantas funciones estadísticas como R, por lo que es posible que necesitemos utilizar varios paquetes para realizar un análisis completo. Además a la hora de definir el modelo o los datos no se suelen utilizar fórmulas si no que se utilizan directamente en diferentes parámetros de la función los arrays de NumPy o DataFrames de Pandas.

En R:

```
modelo <- lm(y ~ x1 + x2, data = datos)
```

En Python:

```
X=datos[['x1','x2']]
y=datos['y']
```

```
modelo <- lm.LinearRegression()
fit = modelo.fit(X, y)
```

Algunos de los modelos de regresión más comunes que se pueden ajustar en Python son:

Regresión Lineal Simple

```
import pandas as pd
import sklearn.linear_model as lm

# Datos de ejemplo en formato Pandas

X = pd.DataFrame([1, 2, 3, 4, 5])
y = pd.Series([2, 4, 6, 8, 10])

# Crear el modelo de regresión lineal

modelo = lm.LinearRegression()

# Entrenar el modelo

fit = modelo.fit(X, y)

# Predecir valores

predicciones = fit.predict(X)
print(predicciones)

# Coeficientes

print(fit.coef_)
print(fit.intercept_)
```

```
>>> [ 2.  4.  6.  8. 10.]
>>> [2.]
>>> 0.0
```

scikit-learn no proporciona p-valores para los coeficientes de regresión, por lo que si queremos realizar un contraste de hipótesis para determinar si los coeficientes son significativamente diferentes de cero, podemos utilizar las funciones de Scypy para calcular el estadístico t y el valor p utilizando las fórmulas necesarias.

O quizás nos sea más interesante en este caso utilizar el paquete StatsModels para realizar la regresión, ya que ésta sí nos proporciona los p-valores:

```
import statsmodels.api as sm

X = sm.add_constant(X)
modelo = sm.OLS(y, X).fit()
print(modelo.summary())
```

```
>>> C:\Users\fcoja\DOCUME~1\VIRTUA~1\R-RETI~1\Lib\site-packages\statsmodels\stats\stattools.p
>>>   warn("omni_normtest is not valid with less than 8 observations; %i "
>>>                             OLS Regression Results
>>> ==============================================================================
>>> Dep. Variable:                      y   R-squared:                       1.000
>>> Model:                            OLS   Adj. R-squared:                  1.000
>>> Method:                 Least Squares   F-statistic:                 5.582e+30
>>> Date:                Mon, 10 Nov 2025   Prob (F-statistic):           1.67e-46
>>> Time:                        12:03:23   Log-Likelihood:                161.95
>>> No. Observations:                   5   AIC:                            -319.9
>>> Df Residuals:                       3   BIC:                            -320.7
>>> Df Model:                           1
>>> Covariance Type:            nonrobust
>>> ==============================================================================
>>>                  coef    std err          t      P>|t|      [0.025      0.975]
>>> ------------------------------------------------------------------------------
>>> const      -3.109e-15   2.81e-15     -1.107      0.349    -1.2e-14    5.83e-15
>>> 0              2.0000   8.46e-16   2.36e+15      0.000       2.000       2.000
>>> ==============================================================================
>>> Omnibus:                        nan   Durbin-Watson:                   0.018
>>> Prob(Omnibus):                  nan   Jarque-Bera (JB):                0.770
>>> Skew:                         0.844   Prob(JB):                        0.680
>>> Kurtosis:                     2.078   Cond. No.                         8.37
>>> ==============================================================================
>>>
>>> Notes:
>>> [1] Standard Errors assume that the covariance matrix of the errors is correctly specifie
```

Regresión Lineal Múltiple

```
import pandas as pd
import sklearn.linear_model as lm
```

```
# Datos de ejemplo

X = pd.DataFrame({'X1': [1, 2, 3, 4, 5], 'X2': [2, 4, 6, 8, 10]})
y = pd.Series([3, 6, 9, 12, 15])

# Crear el modelo de regresión lineal

modelo = lm.LinearRegression()

# Entrenar el modelo

fit = modelo.fit(X, y)

# Predecir valores

predicciones = fit.predict(X)
print(predicciones)
```

```
>>> [ 3.  6.  9. 12. 15.]
```

Regresión Logística

```
import pandas as pd
import sklearn.linear_model as lm

# Datos de ejemplo

X = pd.DataFrame({'X1': [1, 2, 3, 4, 5], 'X2': [2, 4, 6, 8, 10]})
y = pd.Series([0, 0, 0, 1, 1])

# Crear el modelo de regresión logística

modelo = lm.LogisticRegression()

# Entrenar el modelo

fit = modelo.fit(X, y)

# Predecir valores

predicciones = fit.predict(X)
```

```
print(predicciones)
```

```
>>> [0 0 0 1 1]
```

5.4 Aprendizaje estadístico.

Como hemos visto en la sección anterior, es posible utilizar Python para realizar análisis estadísticos básicos, pero no es su fuerte, ya que al no estar diseñado específicamente con este fin, hay muchas funcionalidades que pueden ser menos intuitivas o directas que en R. Sin embargo, donde sí destaca Python es en las técnicas más computacionales del aprendizaje estadístico, ya que cuenta con una serie de paquetes muy potentes y versátiles que permiten realizar análisis predictivos y de clasificación de una forma muy eficiente.

Los bosques aleatorios son un algoritmo de clasificación que se utiliza para predecir una variable categórica en función de una o más variables independientes. Son modelos que solucionan el mismo problema que modelos más simples como la Regresión Logística o el Análisis Discriminante Lineal, pero que suelen tener un mejor rendimiento en la práctica con conjuntos de datos complejos, ya que por ejemplo no tienen problemas con la multicolinealidad, no requieren de una distribución normal de los datos, no requieren de una relación lineal entre las variables independientes y la variable dependiente, etc. Los bosques aleatorios se pueden crear utilizando la función `RandomForestClassifier()` de SciKit-learn.

Su implementación en Python sería análoga a la de las regresiones que hemos visto anteriormente:

```
import pandas as pd
import sklearn.ensemble as en

# Datos de ejemplo

X = pd.DataFrame({'X1': [1, 2, 3, 4, 5], 'X2': [2, 4, 6, 8, 10]})
y = pd.Series([0, 0, 0, 1, 1])

# Crear el modelo de bosque aleatorio

modelo = en.RandomForestClassifier()

# Entrenar el modelo

fit = modelo.fit(X, y)
```

```
# Predecir valores

predicciones = fit.predict(X)
print(predicciones)
```

```
>>> [0 0 0 1 1]
```

Que la implementación siga la misma estructura es de gran utilidad, ya que facilita el uso de diferentes algorimos, por muy complicados que sean.

Otro ejemplo complejo es el de Máquinas de Vectores Soporte:

```
import pandas as pd
import sklearn.svm as sv

# Datos de ejemplo

X = pd.DataFrame({'X1': [1, 2, 3, 4, 5], 'X2': [2, 4, 6, 8, 10]})
y = pd.Series([0, 0, 0, 1, 1])

# Crear el modelo de máquinas de vectores soporte

modelo = sv.SVC()

# Entrenar el modelo

fit = modelo.fit(X, y)

# Predecir valores

predicciones = fit.predict(X)
print(predicciones)
```

```
>>> [0 0 0 1 1]
```

El libro “Introduction to Statistical Learning” de James, Witten, Hastie y Tibshirani es un buen punto de partida para aquellos que quieran aprender más sobre el aprendizaje estadístico. Su versión online está disponible en el siguiente enlace: https://www.statlearning.com/. Tanto con ejemplos en Python como en R.

También hay muchos cursos en línea y tutoriales que pueden ayudar a los principiantes a aprender los conceptos básicos del aprendizaje estadístico y a aplicarlos en Python. Algunos de los cursos más populares son:

- "Machine Learning" de Andrew Ng en Coursera: https://www.coursera.org/learn/machine-learning.
- "Machine Learning" de Google Developers: https://developers.google.com/machine-learning/crash-course.
- "Introduction to Machine Learning" de Udacity: https://www.udacity.com/course/intro-to-machine-learning--ud120.
- "Machine Learning A-Z" de Udemy: https://www.udemy.com/course/machinelearning/.

6 Conclusiones.

Como hemos visto a lo largo de los capítulos anteriores, Python es un lenguaje de programación muy versátil y potente que se puede utilizar para una amplia variedad de tareas de análisis de datos, visualizaciones, estadísticas y aprendizaje.

Este documento ha pretendido ser una guía de iniciación, que permita a los usuarios de R familiarizarse con Python y sus principales paquetes para el análisis de datos. Aunque Python no tiene tantas funciones estadísticas como R, sus características lo convierten en una excelente alternativa, sobre todo para tareas más computacionales y de minería de datos masivos.

En el mundo empresarial, Python es sin lugar a dudas cada vez más utilizado, ya que su facilidad de uso, su amplia comunidad de desarrolladores y su gran cantidad de paquetes disponibles lo convierten en una herramienta muy valiosa para el análisis de datos y la toma de decisiones basada en datos. Su integración en sistemas de producción, su capacidad para trabajar con grandes volúmenes de datos y su capacidad para realizar análisis predictivos y de clasificación lo convierten en una herramienta muy potente para cualquier empresa que quiera aprovechar al máximo sus datos.

Para aquellos que quieran profundizar en Python, existen multitud de recursos en línea, como tutoriales, cursos, libros, foros, etc., que les permitirán adquirir los conocimientos necesarios para utilizar Python de forma eficiente en sus proyectos. En la guía docente de la asignatura y al final de cada sección se han incluido enlaces a la documentación oficial de los paquetes y a otros recursos que pueden ser de utilidad para aquellos que quieran aprender más.