Módulo 2. Introducción a Python

Unidad 2.1. El ecosistema Python

2.1.1 Librerías esenciales de Python

NumPy

NumPy, llamado así por **Num**erical **Py**thon, ha sido, por mucho tiempo, la librería clave para realizar todo el procesamiento numérico de Python. Proporciona las estructuras de datos, los algoritmos y las operaciones de unión necesarios para la mayoría de las aplicaciones científicas que involucran datos numéricos en Python.

NumPy se caracteriza, entre otras cuestiones, por lo siguiente:

- facilita el trabajo con *arrays* (vectores y matrices);
- facilita funciones para realizar cálculos de elementos u operaciones matemáticas entre *arrays*.
- provee herramientas para leer y escribir conjuntos de datos basados en *arrays* en el disco;
- posee operaciones de álgebra lineal y generación de números aleatorios;
- en el análisis de datos, funciona como una especie de contenedor para que los datos pasen entre algoritmos y bibliotecas. "Para datos numéricos, las matrices de NumPy son una forma mucho más eficiente de almacenar y manipular datos que cualquier otra de las estructuras de datos estándar incorporadas en Python" (López Briega, 2014, https://relopezbriega.github.io/blog/2014/05/28/python-librerias-esenciales-para-el-analisis-de-datos/).

Array: es "un tipo de dato estructurado muy utilizado en análisis de datos, en informática científica y en el área del aprendizaje automático (*machine learning*)" (Pherkad, 2019, https://python-para-impacientes.blogspot.com/2019/10/primeros-pasos-con-numpy.html).
"Los datos (elementos) que puede contener un *array* son cadenas de caracteres, números enteros de distintos tamaños, números reales y booleanos, entre otros" (Pherkad, 2019, https://python-para-impacientes.blogspot.com/2019/10/primeros-pasos-con-numpy.html).
El número de elementos que puede contener como máximo un *array* define su tamaño. En *arrays* de más de una dimensión, se determina por el producto del número máximo de elementos de cada eje o dimensión.

Pandas

Pandas proporciona estructuras y funciones de datos de alto nivel diseñadas para trabajar con datos estructurados o tabulares de manera simple y rápida. Desde su aparición en 2010, ha ayudado a que Python sea un entorno de análisis de datos potente y productivo. Los principales objetos que Pandas utiliza para trabajar son:

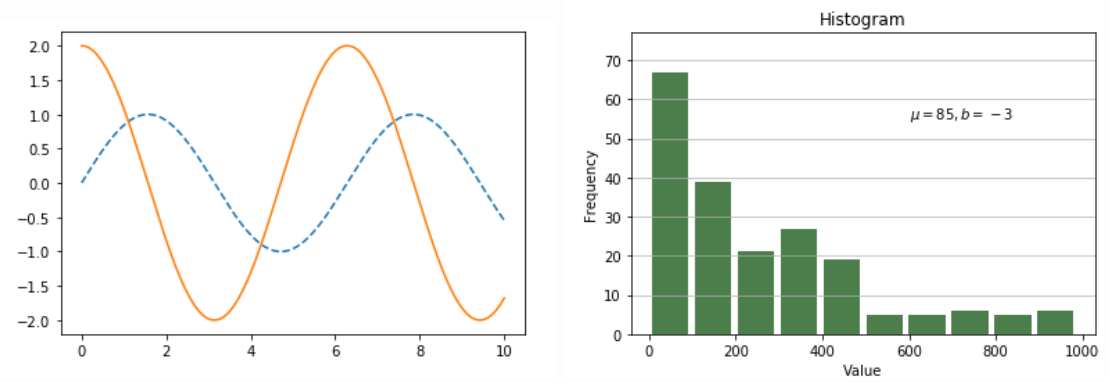
- el *dataframe*, una estructura de datos tabular orientada a columnas con etiquetas de fila y columna;
- y la serie, un objeto que se corresponde con una matriz unidimensional etiquetada.

Pandas combina el procesamiento de alto rendimiento de matrices (*arrays*) de NumPy con las capacidades flexibles de manipulación de datos que tienen las hojas de cálculo y bases de datos relacionales (como SQL). Proporciona funcionalidades de indexación para facilitar la transformación y fraccionado de tablas, realizar agregaciones y seleccionar subconjuntos de datos. Dado que la manipulación, preparación y limpieza de datos es un proceso tan importante dentro del análisis de datos y el *machine learning*, la biblioteca de Pandas es una de las más utilizadas (CloudPYME, 2015).

Matplotlib

Matplotlib es la biblioteca Python más popular para producir gráficos y otras visualizaciones de datos bidimensionales. Originalmente, fue creado por John D. Hunter y ahora es mantenida por un gran equipo de desarrolladores. Está diseñada para crear gráficos (*plots*) adecuados para la publicación del trabajo analítico. Si bien hay otras bibliotecas de visualización disponibles para los programadores de Python, Matplotlib es, sin duda, la más utilizada y, como tal, tiene una muy buena integración con el resto del ecosistema.

Figura 1. Ejemplos de gráficos de Matplotlib



Fuente: elaboración propia.

IPython y Jupyter

El proyecto IPython comenzó en 2001 como el proyecto paralelo del físico Fernando Pérez para hacer un mejor intérprete de comandos (*shell*) interactivo de Python. Si bien no proporciona herramientas analíticas de datos o computacionales por sí solo, IPython está diseñado desde cero para maximizar su productividad tanto en computación interactiva como

en desarrollo de *software*. Fomenta un flujo de trabajo de ejecución-exploración en lugar del típico flujo de edición, compilación y ejecución de muchos otros lenguajes de programación. También proporciona un fácil acceso al *shell* y al sistema de archivos del sistema operativo. Dado que gran parte de la codificación del análisis de datos implica exploración, prueba y error e iteración, IPython tiene una interfaz que ayuda a hacer el trabajo más rápido.

Años más tarde, en 2014, Fernando y el equipo de IPython anunciaron el proyecto Jupyter, una iniciativa más amplia para diseñar herramientas informáticas interactivas independientes del lenguaje. El *notebook* de IPython se convirtió en el *notebook* de Jupyter, con soporte para más de 40 lenguajes de programación. Desde entonces, IPython se puede utilizar como núcleo para usar Python a través de Jupyter Notebook.

Jupyter es un "cuaderno" de código interactivo basado en la web que ofrece soporte para docenas de lenguajes de programación. Es especialmente útil para la exploración y visualización de datos.

El *notebook* de Jupyter también permite crear contenido en Markdown y HTML, proporcionando un medio para crear documentos enriquecidos con código y texto. Otros lenguajes de programación también han implementado núcleos para Jupyter para poder usar lenguajes distintos de Python.

SciPy

SciPy es una colección de paquetes que abordan un gran número de problemas del ámbito de la computación científica y las matemáticas. Algunos de los paquetes que incluye son:

- **Scipy.integrate**: proporciona rutinas de integración numérica y resolución de ecuaciones diferenciales.
- **Scipy.linalg**: provee rutinas de álgebra lineal y matricial.
- **Scipy.optimize**: cuenta con optimizadores de funciones (mínimo de una función) y algoritmos de raíz.
- **Scipy.signal**: contiene herramientas de procesamiento de señales.
- **Scipy.sparse**: trabaja resolviendo problemas complejos con matrices y sistemas lineales dispersos.

Figura 2. Creación de una matriz dispersa

```
In [2]: import numpy as np
        from scipy import sparse

        row_ind = np.array([0, 1, 1, 3, 4])
        col_ind = np.array([0, 2, 4, 3, 4])
        data = np.array([5, 6, 7, 8, 9], dtype=float)
        mat_coo = sparse.coo_matrix((data, (row_ind, col_ind)))
        print(mat_coo.toarray())

        [[5. 0. 0. 0. 0.]
         [0. 0. 6. 0. 7.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 8. 0.]
         [0. 0. 0. 0. 9.]]
```

Fuente: elaboración propia.

- **Scipy.stats**: contiene un gran número de distribuciones de probabilidad continuas y discretas estándar (funciones de densidad, funciones de distribución continua, logarítmicas, Poisson, etc.), pruebas estadísticas (test-t, prueba K-S) y estadísticas más descriptivas. Este es uno de los paquetes de SciPy más usados en ciencia de datos, sobre todo en la etapa de exploración de los datos.

Scikit-learn

Desde el inicio del proyecto en 2010, Scikit-learn se ha convertido en el principal kit de herramientas para el *machine learning* de uso general para programadores de Python. Esta librería incluye distintos submódulos para modelos, como se enumeran a continuación:

- **Clasificación**: SVM (*support vector machines*), *nearest neighbors* (vecinos cercanos), *random forest*, regresión logística, árboles de decisión, etcétera.
- **Regresión**: Lasso (*least absolute shrinkage and selection operator*), regresión cresta, regresión bayesiana, polinomial, etcétera.
- ***Clustering***: *k-means*, *spectral clustering*, etcétera.
- **Reducción de dimensionalidad**: PCA (*principal component analysis*), selección de características, factorización matricial, etcétera.
- **Selección de modelo**: *Grid search*, *cross validation*, métricas, etcétera.
- **Preprocesamiento**: extracción de características y normalización.

Se puede decir Scikit-learn ha sido la librería fundamental que permitió que Python sea el lenguaje de programación de ciencia de datos más expandido en la actualidad junto con R, el cual posee otras ventajas, pero requiere de un *background* mucho más orientado a las matemáticas y a la estadística.

Statsmodels

En comparación con Scikit-learn, Statsmodels contiene algoritmos para estadísticas clásicas y econometría. Este incluye paquetes como los siguientes:

- **Modelos de regresión**: regresión lineal, modelos lineales generalizados, modelos lineales robustos, modelos lineales de efectos mixtos, etcétera.

- **Análisis de varianza** (ANOVA).
- **Análisis de series de tiempo**: AR, ARMA, ARIMA, VAR y otros modelos.
- **Métodos no paramétricos**: estimación de la densidad del núcleo, regresión del núcleo.
- **Visualización de resultados de modelos estadísticos**.

Statsmodels está más centrado en inferencia estadística y proporciona estimaciones basadas en la incertidumbre; Scikit-learn, por el contrario, está más centrado en la predicción, dentro del campo del aprendizaje supervisado.

2.1.2 Anaconda

Para comenzar a trabajar con Python y poder utilizar el *notebook* de Jupyter, vamos a utilizar una *suite* bastante completa que se llama *Anaconda*, la cual nos va a facilitar el trabajo de instalar el ambiente y la mayoría de las librerías. Trabajar con el *notebook* de Jupyter nos va a ayudar a ir resolviendo paso a paso cada uno de los códigos que vayamos aprendiendo para trabajar con los datos y, más adelante, para comenzar a entrenar algunos códigos de *machine learning*. En ellos también se pueden crear visualizaciones de gráficos y escribir comentarios en medio del código para poder hacer notas respecto de lo que vayamos aprendiendo, de manera que después sea más fácil entender lo que hicimos.

Entre las características más destacables de Anaconda, encontramos las siguientes:

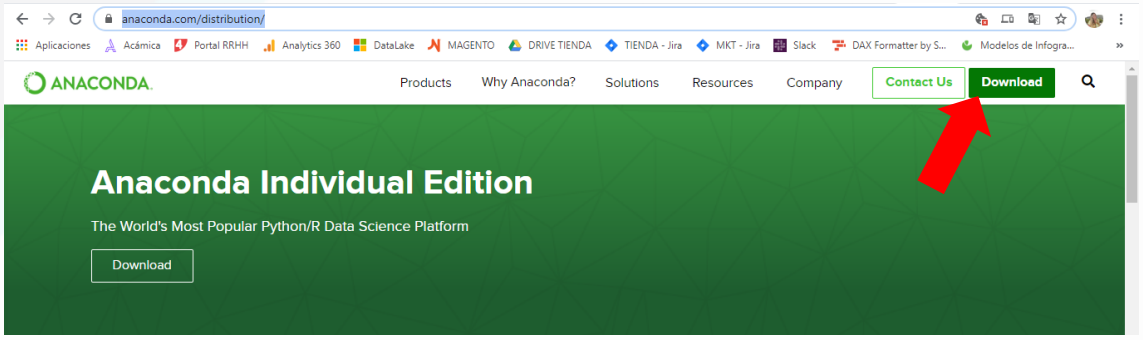
- es gratis y de código abierto, por lo que en internet se puede consultar fácilmente la documentación y muchos foros de la comunidad que permiten resolver dudas;
- es una *suite* multiplataforma que se puede instalar en cualquier sistema operativo;
- permite instalar y administrar los paquetes de Python de una manera muy sencilla;
- se pueden desarrollar modelos de *data science* utilizando diversos entornos de desarrollo de acuerdo con el gusto del consumidor, como Jupyter, JupyterLab, Spyder y Rstudio;
- cuenta con la posibilidad de trabajar con todos los paquetes de datos enumerados en las páginas anteriores;
- cuenta con Anaconda Navigator, que es una interfaz gráfica de usuario (GUI) que permite acceder de manera sencilla a todos los ambientes con los que cuenta;
- resuelve el problema de las dependencias de paquetes y el manejo de las compatibilidades de las distintas versiones;
- la ejecución de las tareas es más veloz que en otras *suites;*
- los proyectos son portables. Esto permite compartir proyectos con otras personas y ejecutarlos en distintas plataformas.

Sabiendo todo esto, vamos a comenzar instalando Anaconda en nuestra computadora. Para hacerlo, vamos a seguir los siguientes pasos:

1. Descargar Anaconda.

"Nos dirigimos a la *home* de Anaconda: https://www.anaconda.com/ e iremos a la sección de Download (descargas)" (Na8, 2018, https://www.aprendemachinelearning.com/instalar-ambiente-de-desarrollo-python-anaconda-para-aprendizaje-automatico/).

Figura 3. Descargar Anaconda

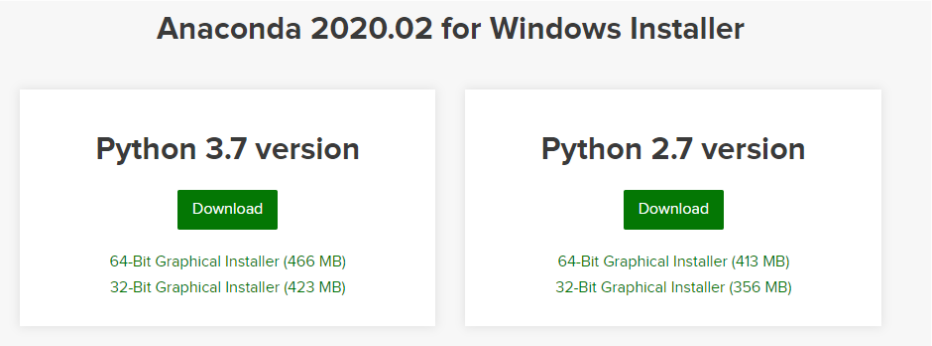


Fuente: captura de pantalla con modificaciones propias de la sección de descarga de Anaconda (s. f., https://www.anaconda.com/distribution/).

"Elegimos nuestra plataforma: Windows, Mac o Linux" (Na8, 2018, https://www.aprendemachinelearning.com/instalar-ambiente-de-desarrollo-python-anaconda-para-aprendizaje-automatico/).

Entonces se va a cargar una ventana en la que tendremos la opción de elegir entre dos versiones de Python. Allí haremos clic en la versión de Python 3.7 (y no la de 2.7) y seleccionaremos el instalador gráfico de 32 o 64 bit, de acuerdo con el sistema operativo con el que contemos (Na8, 2018).

Figura 4. Descarga del instalador de Anaconda



Con esto guardaremos en nuestro disco duro unos 477MB (según el sistema operativo) y obtendremos un archivo con un nombre similar a Anaconda3-2020.02-Windows-x86_64.exe.

2. Instalar Anaconda.

Luego de tener descargado el ejecutable, vamos a proceder a instalar la aplicación en nuestro sistema (se requiere tener permisos de administrador si se opta por la instalación para todos los usuarios).

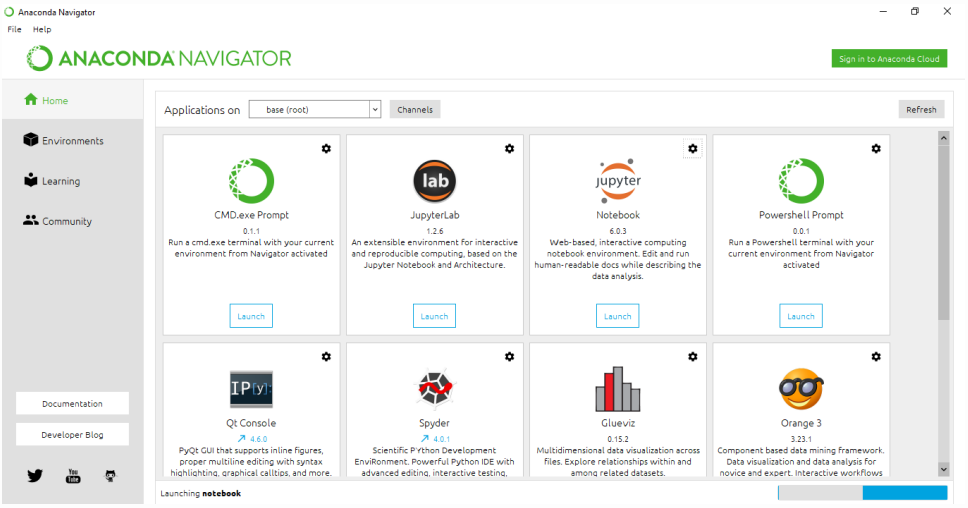
Posteriormente, buscaremos y ejecutaremos el archivo .exe que descargamos y se abrirá un *wizard* de instalación. Seguiremos todos los pasos que se vayan indicando. Podremos seleccionar una instalación solo para nuestro usuario o para todos y luego deberemos elegir la ruta del disco en donde instalaremos el programa.

Al instalarse, el tamaño total podrá superar los 5 GB (gigabytes) en el disco, por lo cual no estará de más que antes nos aseguremos de contar con esa capacidad de almacenamiento y que nos quede la suficiente cantidad para seguir trabajando después.

3. Iniciar Anaconda.

Luego de la instalación, es necesario reiniciar el equipo para que surtan los cambios y se pueda visualizar el programa en el panel de Inicio. Una vez verificado esto, seleccionamos Anaconda Navigator en el desplegable del menú Inicio y esperaremos unos minutos hasta que cargue.

Figura 5. Anaconda Navigator



4. Actualización de paquetes.

Vamos a comenzar actualizando la librería Scikit-learn, que es con la que más se suele trabajar, como dijimos antes, dentro del campo del *machine learning*. Para ello, iremos a Inicio > Anaconda y seleccionaremos: Anaconda Powershell Prompt. En la consola que se abrirá, vamos a escribir:

```
>>> conda update scikit-learn
```

Luego confirmamos indicando 'Y'.

2.1.3 Instalación o actualización de paquetes de Python

En algún momento durante el curso o posteriormente, puede surgir la necesidad de instalar algún paquete de Python adicional que no venga preinstalado en la *suite* de Anaconda. En general, todos tienen la misma lógica para su instalación y actualización; solo basta con cambiar el nombre del paquete y ejecutar en el *prompt*.

La instalación se puede ejecutar con alguno de los siguientes comandos:

```
>>> conda install nombre_paquete
```

En el caso de que este no funcione, se puede probar con el administrador de paquetes pip y ejecutar el siguiente:

```
>>> pip install nombre_paquete
```

También es posible actualizar paquetes utilizando el comando *conda update*:

```
>>> conda update nombre_paquete
```

O, en su defecto: `pip install --upgrade nombre_paquete`

Unidad 2.2. Primeros pasos en la programación en Python

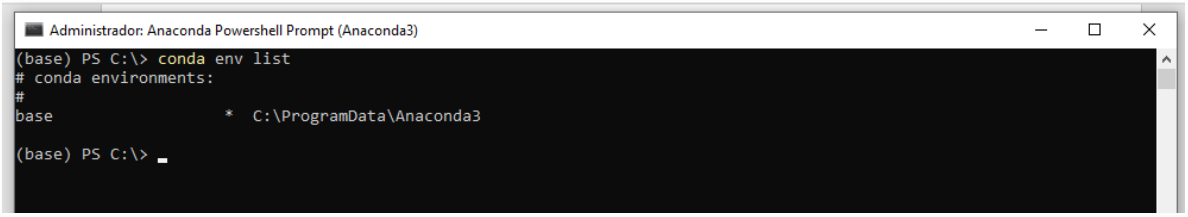
2.2.1 Creación de ambientes de trabajo en Anaconda

Anaconda tiene una funcionalidad esencial: podemos tener separados, en diferentes ambientes de trabajo, las distintas bibliotecas para los diversos trabajos que estemos realizando en Python. Si en un proyecto en particular vamos a usar ciertas bibliotecas con ciertas versiones de estas y con una versión de Python particular, podemos crear un entorno de trabajo con estas características y tenerlo individualizado bajo el nombre del proyecto que estemos realizando. Esto también es útil cuando, en una misma máquina, se conectan varios usuarios que llevan distintos proyectos. Una buena opción es crear un ambiente distinto para cada uno de ellos.

Por defecto, Anaconda se instala con un ambiente genérico llamado BASE. Podemos chequear esto yendo a Anaconda Powershell Prompt (a partir de ahora, lo llamaremos simplemente *prompt* o consola) y ejecutando el siguiente comando:

```
>>> conda env list
```

Figura 6. Base



Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver, la consulta arroja el nombre Base y la ubicación en el disco de este ambiente por defecto.

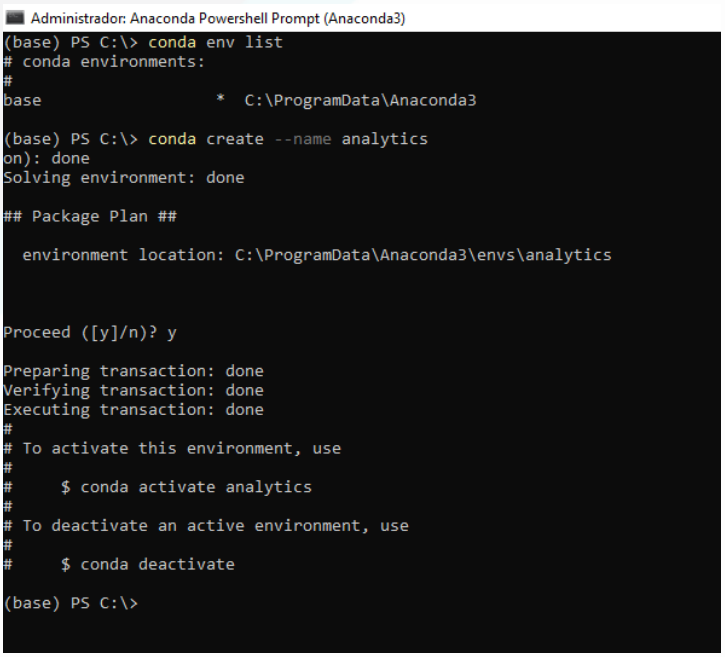
Ahora procederemos a crear un ambiente en donde trabajaremos a lo largo del módulo que se llame *analytics* (el alumno puede elegir el nombre de su preferencia).

Vamos a la consola y escribimos:

```
>>> conda create –name analytics
```

Luego confirmamos la ubicación del ambiente y la ejecución del comando escribiendo *Y*.
Una vez hecho esto, ya tenemos creado el ambiente del trabajo en Anaconda con el que vamos a desempeñarnos.

Figura 7. Ambiente de trabajo



Fuente: elaboración propia a base del *software* Anaconda (2018).

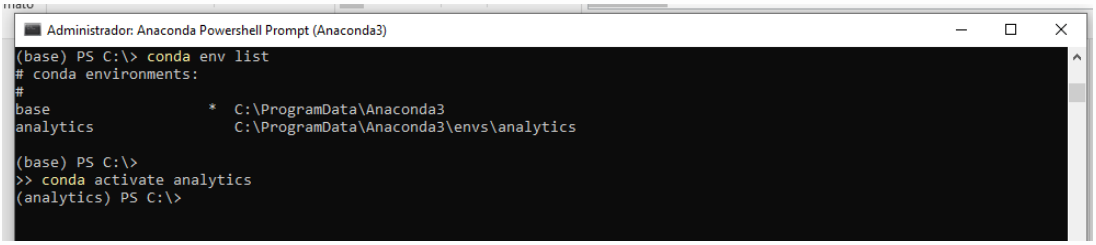
Luego vamos a volver a chequear los ambientes creados utilizando **conda env list** y veremos que figuran los dos ambientes: base y analytics, pero base figura con un asterisco (*) adelante que indica que el entorno se encuentra activo.

Ahora lo que tenemos que hacer es activar nuestro ambiente de trabajo. Para eso vamos a ejecutar nuevamente en la consola el siguiente comando:

```
>>> conda activate analytics
```

Y vamos a observar que ahora entre paréntesis, delante de la línea donde se escribe el código, ya no dice *base*, si no *analytics* (o el nombre que se eligió). Esto indica que el entorno ya está activo y listo para trabajar.

Figura 8. Entorno preparado para ser utilizado



Fuente: elaboración propia a base del *software* Anaconda (2018).

Al final del trabajo, sobre todo si se trabaja con múltiples proyectos o con varios usuarios conectados a la misma aplicación de Anaconda, el comando para desactivar el entorno es:

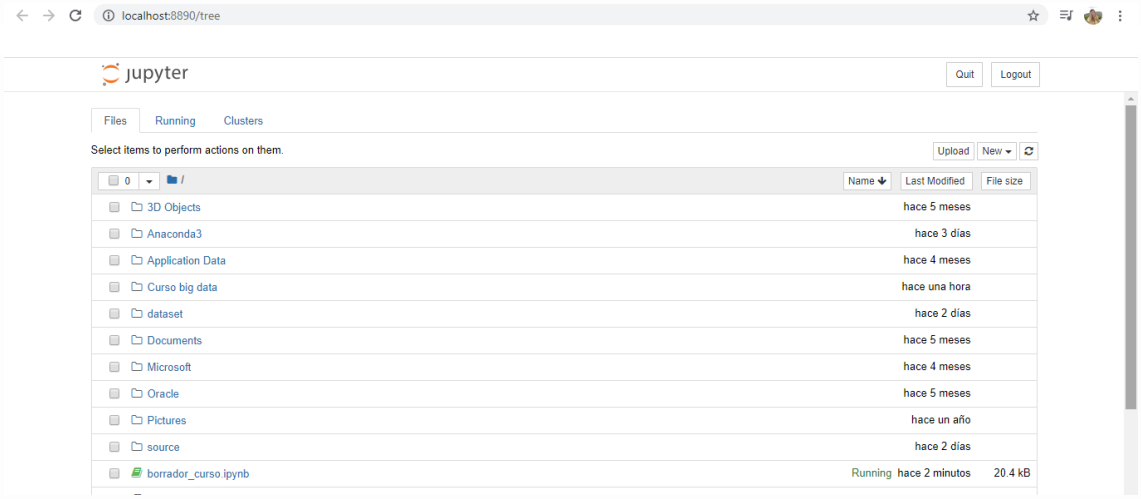
```
>>> conda deactivate
```

Y por defecto vuelve al entorno base nuevamente.

2.2.2 Iniciando en Jupyter

Para comenzar a trabajar, vamos a ir a Anaconda Navigator, seleccionaremos el logo de Jupyter y haremos clic en Launch, o bien, desde el menú Inicio dentro de Anaconda3, seleccionaremos Jupyter. Entonces se abrirá una ventana en nuestro navegador como la siguiente:

Figura 9. Jupyter

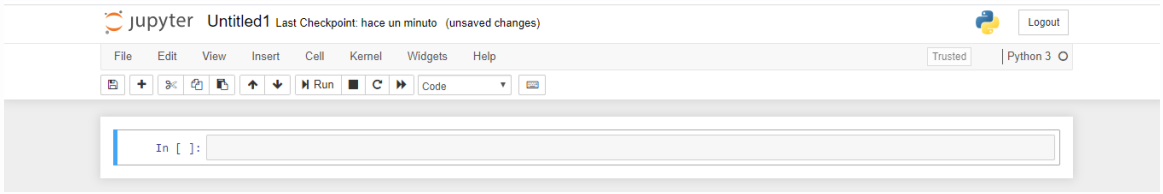


Fuente: elaboración propia a base del *software* Anaconda (2018).

En ella se podrán visualizar los archivos y carpetas que se encuentren en la misma ruta en que esté instalado Anaconda. Esto es importante porque, de aquí en más, esta ruta será el punto de partida sobre el que se comenzarán a escribir las distintas rutas de acceso a los distintos archivos que queramos buscar en nuestro ordenador o guardar.

Abriremos un nuevo *notebook* en blanco para comenzar a trabajar. Para ello, seleccionaremos arriba a la derecha New > Python 3. Entonces se abrirá el *notebook* de Jupyter en otra ventana del navegador.

Figura 10. *Notebook* de Jupyter



Fuente: elaboración propia a base del *software* Anaconda (2018).

Jupyter Notebook es más amigable y sencillo para trabajar que el *shell* de IPython porque es como un cuaderno de notas. En el desplegable **Code** debajo de la barra de herramientas, se puede seleccionar **Markdown**, que hace que cambie la funcionalidad de cada celda de código que se abra debajo (el rectángulo donde dice "In ()") y sirva para escribir texto de soporte al código que estemos armando: teoría, ayuda memoria, datos extra, consignas o lo que se quiera. Allí, dependiendo de si está seleccionado Code o Markdown, se puede ir

intercalando código con texto. Cada nueva celda que se abra (se agrega una nueva con el segundo recuadro de la barra que tiene un signo +), puede adquirir una u otra funcionalidad. Para escribir texto dentro de celdas de código, simplemente se puede agregar con un # delante de la siguiente forma:

Figura 11. Escribir texto dentro de celdas de código

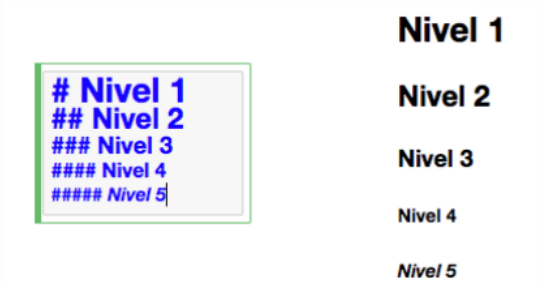
```
In [39]: # row indices
         row_ind = np.array([0, 1, 1, 3, 4])
         # column indices
         col_ind = np.array([0, 2, 4, 3, 4])
         # data to be stored in COO sparse matrix
         data = np.array([1, 2, 3, 4, 5], dtype=float)
         data

Out[39]: array([1., 2., 3., 4., 5.])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

El ejecutor va a ignorar estas líneas y no va a arrojar error confundiéndolas con código.

En el caso de seleccionar Markdown, si utilizamos # o doble ##, el texto se convertirá a formato de título con tamañas más grandes de letra (cada # extra que se agregue será un nivel de jerarquía de título menor y, en consecuencia, más chico); si encerramos el texto entre asteriscos (** texto **), podemos resaltarlo con negrita. De esta forma, si creamos un proyecto de *machine learning*, por ejemplo, que lleve varias etapas y líneas de código, se pueden ir agregando títulos y comentarios y colocando el código para cada etapa. Esto es una buena práctica para que quede prolijo y presentable.

Figura 12. Proyecto con varias etapas



Fuente: Martín, 2018, https://www.adictosaltrabajo.com/2018/01/18/primeros-pasos-con-jupyter-notebook/

Práctica:

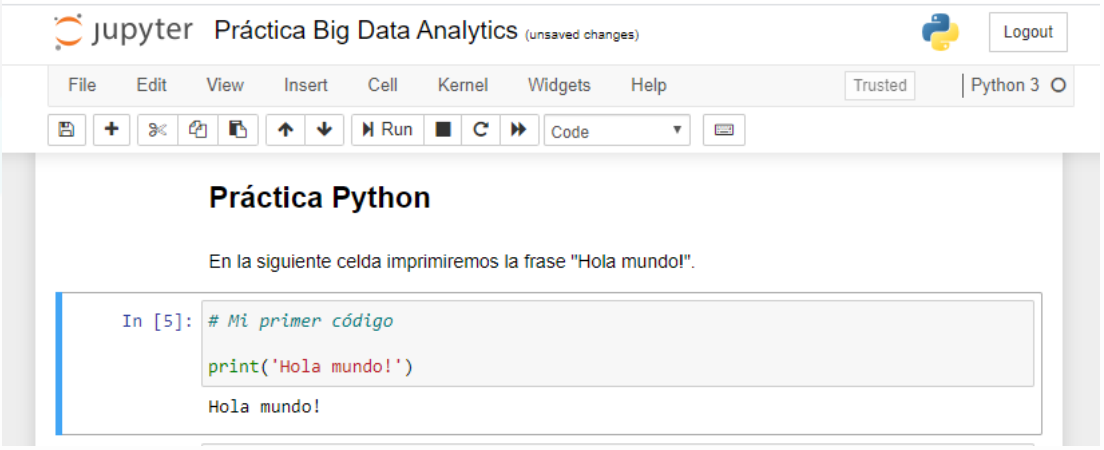
A modo de práctica, se le recomienda al alumno comenzar un *notebook* con un título y una descripción (celdas Markdown) y ejecutar la primera línea de código en Python en una celda de Code.

El primer comando básico que ejecuta un programador cuando arranca en un lenguaje es imprimir el texto: “Hola mundo!”.

Nota: Luego de escribir el código o el texto, se ejecuta utilizando el método abreviado del teclado Ctrl + Enter.

El resultado de su práctica debería quedar algo similar a la siguiente pantalla:

Figura 13. Resultado de práctica



Fuente: elaboración propia a base del *software* Anaconda (2018).

2.2.3 Los comandos mágicos

Los comandos mágicos son comandos extra propios de Jupyter (no pueden ejecutarse en IPython) que básicamente le hacen la vida más fácil al programador. Vamos a enumerar algunos de ellos, ya que serán de ayuda para comprender lo que se desarrollará sobre Python durante el desarrollo del curso y luego servirán para armar *scripts* de programación propios.

El signo ?

El uso del signo de interrogación (?) antes o después de una variable mostrará información general sobre el objeto. Por ejemplo, creamos una lista simple y la llamamos "a". Luego en otra utilizamos el signo de interrogación y ejecutamos (Ctrl + Enter) para ver su información.

Figura 14. Signo de interrogación

```
In [30]: a=[1, 2, 3]

In [32]: a?
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

El sistema automáticamente abrirá por debajo una ventana con información básica sobre la variable que consultamos:

Figura 15. Información básica sobre la variable consultada

```
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Esto es especialmente para no perderse buscándola entre todas las líneas de código, es posible sacarse la duda de esta manera rápida y sencilla. Este comando también sirve con funciones.

Figura 16. Comando con funciones

```
In [27]: import pandas as pd

In [33]: pd.read_csv?
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En este ejemplo utilizamos el (?) para entender de qué se trata el comando de Pandas **read_csv.** En la ventana automáticamente se nos va a mostrar toda la documentación oficial que hay sobre este comando con todos sus parámetros.

Figura 17. Parámetros del comando de Pandas read_csv

```
Signature:
pd.read_csv(
 filepath_or_buffer: Union[str, pathlib.Path, IO[~AnyStr]],
 sep=',',
 delimiter=None,
 header='infer',
 names=None,
 index_col=None,
                    [ … ]
)
Docstring:
Read a comma-separated values (csv) file into DataFrame.
Also supports optionally iterating or breaking of the file
into chunks.
Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_g
uide/io.html>`_.
```

Fuente: Pandas, s. f., https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

El comando %run

El comando %run puede ejecutar cualquier archivo de programa Python dentro del entorno de su sesión IPython. Suponga que tiene el siguiente *script* simple almacenado en ipython_code.py:

Figura 18. Script en ipython_code.py

```
def f(x, y, z):
    return (-x + y)**2 / z*0.29

a = 6
b = 12
c = 10.44

result = f(a, b, c)
result
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Se puede ejecutar dicho código desde Jupyter simplemente indicando el nombre del archivo con el comando %run.

```
>>> %run ipython_script_test.py
```

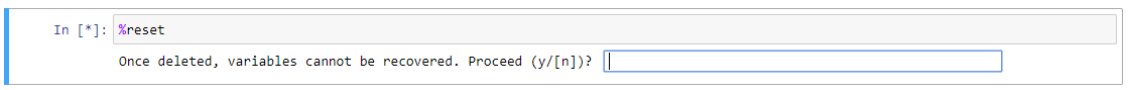
El comando %time o %timeit

Los comandos mágicos %time o %timeit son los que nos permiten saber cuánto tiempo tarda en ejecutarse un código en Python. Ambos comandos posibilitan medir el tiempo que tarda la ejecución de la tarea escrita inmediatamente después en la misma línea de código. El comando %time devuelve el tiempo de CPU (*central processing unit*) y el de reloj; por otro lado, el comando %timeit sirve para medir el tiempo medio de ejecución. Este último es más lento porque ejecuta varias veces para medir el tiempo medio de ejecución, pero por esta razón es el más preciso (Rodríguez, 2019).

El comando %reset

El comando %reset es útil cuando venimos trabajando declarando varios objetos y queremos volver a ejecutar —por algún cambio que queremos realizar en el código— todo lo que habíamos programado en nuestro *notebook*. De esta manera, podemos saber si, cuando llamamos estos objetos nuevamente, el resultado que arroja corresponde a la ejecución nueva y no a la anterior. Luego de ejecutar aparecerá una ventana donde se nos pedirá una confirmación del procedimiento para más seguridad. Hay que tipear "Y" para avanzar.

Figura 19. Comando %reset



Fuente: elaboración propia a base del *software* Anaconda (2018).

2.2.4. Generalidades del lenguaje Python

Antes de comenzar a programar para empezar con el trabajo puramente analítico de los datos, es importante tener en cuenta algunas generalidades y conceptos esenciales de programación en Python y comprender la mecánica del lenguaje.

Semántica del lenguaje

El diseño del lenguaje Python se distingue por su énfasis en la legibilidad, simplicidad y explicitud. Por esta razón, usa espacios en blanco (tabulaciones o espacios) para estructurar el código en lugar de usar llaves, como en muchos otros lenguajes. Considere un bucle **for** de un algoritmo de clasificación con una sentencia condicional adentro:

Figura 20. Bucle for de un algoritmo de clasificación con una sentencia condicional adentro

```
In [52]: continentes = ['Asia', 'Europa', 'America', 'Oceania', 'África', 'Córdoba']
         ciudades = ['Córdoba', 'Carlos Paz', 'Alta Gracia', 'Rio Cuarto', 'Villa María']

         contador = 0

         for i in continentes:

             if continentes[contador] in ciudades:

                 print ('Este elemento no es un continente: ' +  continentes[contador] )

             else:

                 print ('Este elemento es un continente: ' +  continentes[contador] )

             contador = contador + 1

         Este elemento es un continente: Asia
         Este elemento es un continente: Europa
         Este elemento es un continente: America
         Este elemento es un continente: Oceania
         Este elemento es un continente: África
         Este elemento no es un continente: Córdoba
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Los dos puntos indican el inicio de un bloque de código indentado, con lo cual el código que venga después —que está incluido dentro del anterior— debe indentarse un poco más hacia la derecha para marcar la jerarquía en la ejecución del código.

En el ejemplo, el condicional **if** es una ejecución que se debe realizar en cada *loop*, con lo cual se corre un nivel a la derecha. Las sentencias **print** están incluidas dentro del condicional, dependen de su verificación para imprimirse, con lo cual se baja otro nivel de indentación.

En las siguientes imágenes, vamos a demostrar que un cambio en los niveles de indentación puede modificar el resultado del código.

Figura 21. Cambios en los niveles de indentación

```
In [60]: continentes = ['Asia', 'Europa', 'America', 'Oceania', 'África', 'Córdoba']
         ciudades = ['Córdoba', 'Carlos Paz', 'Alta Gracia', 'Rio Cuarto', 'Villa María']

         contador = 0

         for i in continentes:

             if continentes[contador] in ciudades:

                 print ('Este elemento no es un continente: ' +  continentes[contador] )

             else:

                 print ('Este elemento es un continente: ' +  continentes[contador] )

         contador = contador + 1

         Este elemento es un continente: Asia
         Este elemento es un continente: Asia
         Este elemento es un continente: Asia
         Este elemento es un continente: Asia
         Este elemento es un continente: Asia
         Este elemento es un continente: Asia
```

```
In [61]: continentes = ['Asia', 'Europa', 'America', 'Oceania', 'África', 'Córdoba']
         ciudades = ['Córdoba', 'Carlos Paz', 'Alta Gracia', 'Rio Cuarto', 'Villa María']

         contador = 0

         for i in continentes:

         if continentes[contador] in ciudades:

         print ('Este elemento no es un continente: ' +  continentes[contador] )

         else:

         print ('Este elemento es un continente: ' +  continentes[contador] )

         contador = contador + 1

           File "<ipython-input-61-63e5e7e3cfbe>", line 8
             if continentes[contador] in ciudades:
             ^
         IndentationError: expected an indented block
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Igualdad de variables y pasajes de argumentos

Al asignar una variable (o nombre) en Python, se está creando una referencia al objeto en el lado derecho del signo igual (=). Esto significa que, una vez hecha la referencia, toda operación que afecte a la primera variable impactará en los parámetros de la segunda y viceversa.

Figura 22. Variables

```
In [66]: a= [1,2,3,4]

In [68]: b=a

In [69]: b
Out[69]: [1, 2, 3, 4]

In [71]: a.append(10)

In [72]: b
Out[72]: [1, 2, 3, 4, 10]
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Atributos y métodos

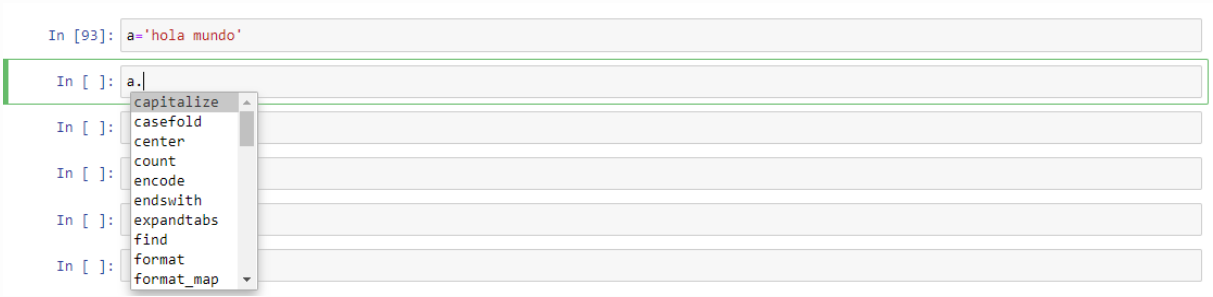
Los objetos en Python generalmente tienen atributos (otros objetos de Python almacenados "dentro" del objeto) y métodos (funciones asociadas al objeto que puede tener acceso a los datos almacenados dentro del objeto). Supongamos el siguiente ejemplo.

Figura 23. Atributos y métodos

Fuente: elaboración propia a base del *software* Anaconda (2018).

Apretamos Tab luego de escribir el nombre del objeto seguido de punto (.):

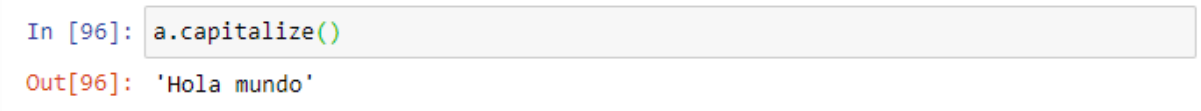
Figura 24. Presionar Tab luego de escribir el nombre del objeto con punto



Fuente: elaboración propia a base del *software* Anaconda (2018).

A continuación, se listan todos los métodos o funciones asociados al objeto. Seleccionamos Capitalize, por ejemplo, y ejecutamos.

Figura 25. Selección de Capitalize



Fuente: elaboración propia a base del *software* Anaconda (2018).

Este método asigna mayúscula al comienzo de una cadena de texto. Se lo invita al alumno a probar el resto de los métodos e investigar con el comando mágico '?' la documentación de estos métodos para aprender a utilizarlos. A continuación, se dejan algunos ejemplos:

Figura 26. Ejemplos

```
In [242]: z=[1,2,3,4,'hola']
          z.extend(['x','y'])
          z
Out[242]: [1, 2, 3, 4, 'hola', 'x', 'y']

In [239]: a=[1,2,3]
          b=['a','b','c','d','e']
          a.insert(1,b)
          a
Out[239]: [1, ['a', 'b', 'c', 'd', 'e'], 2, 3]
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Operaciones binarias y comparaciones

Muchas de las operaciones matemáticas y comparaciones entre objetos suelen ser como cabría esperar de acuerdo con lo que ocurre en otros lenguajes de programación. Como ejemplo, miremos los siguientes casos de matemática y lógica básica.

Figura 27. Casos de matemática y lógica básica

```
In [1]: 5-7
Out[1]: -2

In [2]: 10+12.2
Out[2]: 22.2

In [3]: 2>=5
Out[3]: False
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Para verificar si dos referencias se refieren al mismo objeto o no, se puede usar la palabra clave *is/is not*:

Figura 28. Is/is not

```
In [7]: a= [1, 2, 3]
        b = a
        c = list(a)

In [8]: a is b
Out[8]: True

In [10]: a is c
Out[10]: False

In [11]: a == c
Out[11]: True
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como la función **list** siempre crea una nueva lista de Python (es decir, una copia de la original), podemos estar seguros de que c es diferente de a. Comparar con **is** no es lo mismo que el operador **==**, como se ve en el ejemplo anterior.

En la siguiente tabla, se listan las operaciones binarias más usadas.

Tabla 1. Operaciones binarias más usadas

Operation	Description
A + B	Suma de A + B.
A - B	Sustracción de B en A.
A * B	Multiplicación de A por B.
A/ B	División de A sobre B.
A // B	División de A sobre B que devuelve un entero. Se redondea para abajo.
A** B	A elevado a la B (exponente).
A== B	Devuelve True si A es igual a B.
A != B	Devuelve True si A es distinto de B.
A <= B, A < B	Devuelve True si A es menor o menor o igual a B.
A > B, A >= B	Devuelve True si A es mayor o mayor o igual a B.
A is B	Devuelve True si A y B hacen referencia al mismo objeto de Python.
A is not B	Devuelve True si A y B hacen no referencia al mismo objeto de Python.
A % B	Devuelve el resto de la operación de división de A sobre B. Si es 0, se puede decir que A es múltiplo de B.

Fuente: elaboración propia.

Objetos mutables e inmutables

La mayoría de los objetos en Python, como listas, diccionarios, NumPy *arrays* y la mayoría de los tipos definidos por el usuario, son mutables. Esto significa que el objeto o los valores que contienen pueden modificarse.

Figura 29. Objetos mutables

```
In [20]: Lista = ['hola,', ['como, estas'], '?']

In [23]: Lista[1]=('todo', 'bien')

In [24]: Lista
Out[24]: ['hola,', ('todo', 'bien'), '?']
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En el ejemplo se crea una lista embebida, en donde la lista [‘como’, ‘estas’] es el argumento número 1 (recuerde que siempre se cuentan los argumentos partiendo desde el 0). En la siguiente operación, reemplazamos este primer argumento por (‘todo’, ‘bien’) y volvemos a llamar al objeto lista para ver cómo queda. Efectivamente, surtió el cambio en la lista original.

Pero hay otros, como las cadenas de texto o las tuplas, que son inmutables, es decir, no pueden modificarse.

Figura 30. Objetos inmutables

```
In [29]: Tupla = ('Chau,', 'nos', 'vemos')
         Tupla

Out[29]: ('Chau,', 'nos', 'vemos')

In [30]: Tupla[2]=('volveremos a ver')

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-86454bcd0618> in <module>
----> 1 Tupla[2]=('volveremos a ver')

TypeError: 'tuple' object does not support item assignment
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Tipos escalares

Python, junto con su biblioteca estándar, tiene un pequeño conjunto de tipos integrados para manejar datos numéricos, cadenas, valores booleanos (*true or false*) y fechas y horas. Estos tipos de “valor único” a veces se denominan *tipos escalares* y nos referimos a ellos como *escalares*. En la siguiente tabla, listamos los tipos escalares que encontramos en Python.

Tabla 2. Tipos escalares en Python

Type	Description
None	The Python “null” value
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number
bool	A True or False value
int	Arbitrary precision signed integer

Fuente: McKinney, 2017.

Fecha y hora

El módulo incorporado de fecha y hora de Python proporciona distintos tipos de estas. Importemos el módulo y veamos un ejemplo de creación de un objeto fecha-hora y de operaciones con él.

Figura 31. Objeto fecha-hora

```
In [1]: from datetime import datetime, date, time

In [6]: dt= datetime(2020, 4, 23, 11, 20, 22)
        dt
Out[6]: datetime.datetime(2020, 4, 23, 11, 20, 22)

In [7]: dt.day
Out[7]: 23

In [8]: dt.minute
Out[8]: 20

In [9]: dt.date()
Out[9]: datetime.date(2020, 4, 23)

In [10]: dt.time()
Out[10]: datetime.time(11, 20, 22)
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

El método *strftime* formatea una fecha y hora como una cadena de texto para que su visualización sea más sencilla.

Figura 32. Método strftime

```
In [14]: dt.strftime('%d/%m/%y %H:%M')
Out[14]: '23/04/20 11:20'
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Flujo de control

Python tiene varias *keywords* integradas para lógica condicional, bucles y otros conceptos de flujo de control estándar que se encuentran también en otros lenguajes de programación.

- **Flujo condicional: if, elif y else.**

La declaración **if** es uno de los tipos de declaración de flujo más conocidas. Comprueba una condición que, si es verdadera, se evalúa el código en el bloque siguiente.

Figura 33. Declaración if

```
In [17]: x=-2
         if x < 0:
             print('Es un numero negativo')
         Es un numero negativo
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Una declaración **if** puede ser seguida opcionalmente por uno o más bloques **elif** y un bloque final que agrupe todas las condiciones no declaradas: **else** (es decir, si todas las condiciones son falsas).

Figura 34. Declaración if más bloques elif y bloque else

```
In [179]: x=4

          if x < 0:
              print(f"El número {x} es un número negativo")
          elif x == 0:
              print(f"El número {x} es nulo")
          elif 5 > x > 0:
              print(f"El número {x} es positivo menor a 5")
          else:
              print(f"El número {x} es positivo mayor a 5")

          El número 4 es positivo menor a 5
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En este código, además, utilizamos una función nueva, que es la de llamar a la variable dentro de un *print()* de cadena de texto. En este caso, nótese que incorporamos la *f* delante de las comillas (dentro del *print*), lo cual habilita a que, cuando se esté ejecutando el código, no se tome la llave con la variable *x* como un texto literal, sino que sea traducida al valor que asume la variable *x* dentro de la cadena de texto que se está imprimiendo.

- **Bucles: for, loops.**

Los bucles **for** son para iterar sobre una lista o tupla, o un iterador. La sintaxis estándar para un bucle for es:

```
>>> for valor in lista:
 # realiza una operación con cada uno de esos valores.
```

Puede avanzar un bucle for a la siguiente iteración, omitiendo el resto del bloque, utilizando una sentencia condicional y la palabra clave *continue*.
En el ejemplo a continuación, se crea un bucle for y se omiten los valores None.

Figura 35. Bucle for omitiendo los valores None

```
In [35]: tupla_loop = (10, 20, None, 30, None, 0.5)

         total = 0

         for i in tupla_loop:
             if i is None:
                 continue
             total += i
             print(i)

         10
         20
         30
         0.5
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Un for loop se puede frenar por completo con la palabra clave *break*. En el siguiente código, vemos cómo se suman elementos de la lista hasta alcanzar el valor *5*.

Figura 36. For loop

```
In [39]: secuencia = [1, 2, 0, 4, 6, 5, 2, 1]
         total_hasta_5 = 0
         for value in secuencia:
             if value == 5:
                 break
             total_hasta_5 += value
             print(value)

1
2
0
4
6
```

```
In [48]: for i in range(3):
             for j in range(4):
                 if j > i:
                     break
                 print((i, j))

(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

- **Bucles while.**

Un ciclo while especifica una condición y un bloque de código que se ejecutará hasta que la condición se evalúe como falsa o el ciclo finalice explícitamente con *break*.

Figura 37. Ciclo while

```
In [95]: x = 256
         total = 0
         while x > 0:
             if total > 500:
                 break
             total = total + x # es lo mismo que decir: "total += x"
             x = x // 2
             print('el valor resultante de x es', x, 'del valor total:', total)

el valor resultante de x es 128 del valor total: 256
el valor resultante de x es 64 del valor total: 384
el valor resultante de x es 32 del valor total: 448
el valor resultante de x es 16 del valor total: 480
el valor resultante de x es 8 del valor total: 496
el valor resultante de x es 4 del valor total: 504
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En este ejemplo, vemos cómo la condición es que ***x*** asuma valores positivos, pero sujeto a la condición de que *total* no supere el valor 500 (porque se ejecuta la función *break* llegado a ese valor).

El valor *total*, que es el contador de la función, en cada loop asume el valor de *total* de la ejecución anterior y se le suma el valor de ***x*** de la ejecución actual.

Sigamos la lógica. ***Total*** arranca el loop asumiendo el valor cero y ***x*** vale 256, con lo cual, al final de la ejecución, ***x*** asume el nuevo valor que se extrae de dividir $x/2 = 128$. En la segunda ejecución del loop, total vale 256 + 128 =3 84, mientras que *x* ahora vale $128/2 = 64$. Y el bucle sigue así hasta que el valor de *total* alcanza 500 y se ejerce el comando *break*, que hace que se deje de ejecutar el código.

- **Rangos.**

La función de rango devuelve un iterador que produce una secuencia de enteros espaciados de manera uniforme. Tiene la opción de poder asignar un inicio, un final y un intervalo.

Figura 38. Función de rango

```
In [96]: range(4)
Out[96]: range(0, 4)

In [97]: list(range(4))
Out[97]: [0, 1, 2, 3]

In [110]: list(range(0,20,2))
Out[110]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [111]: list(range(10,-5,-2))
Out[111]: [10, 8, 6, 4, 2, 0, -2, -4]
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver, el final definido para el rango no se incluye en Python dentro del *output* de la operación.

Referencias

Anaconda. (s. f.). [Captura de pantalla con modificaciones propias de la sección de descarga de Anaconda]. Recuperado de https://www.anaconda.com/distribution/

Anaconda. (2018). Anaconda (Versión 3.7) [*Software* de computación]. Austin, US.

CloudPYME. (2015). *Procesamiento de datos con Python entornos y aplicaciones*. Recuperado de https://issuu.com/cloudpyme2/docs/procesamiento_de_datos_con_python-_

López Briega, R. E. (2014). Python - Librerías esenciales para el análisis de datos. Recuperado de https://relopezbriega.github.io/blog/2014/05/28/python-librerias-esenciales-para-el-analisis-de-datos/

Martín, G. (2018). [Imagen sin título sobre niveles de un proyecto]. Recuperado de https://www.adictosaltrabajo.com/2018/01/18/primeros-pasos-con-jupyter-notebook/

McKinney, W. (2017). *Python for Data Analysis*. Massachusetts, US: O'Reilly Media.

Na8 (Nombre de usuario). (2018). Instalar ambiente de Desarrollo Python Anaconda para Aprendizaje Automático. Recuperado de https://www.aprendemachinelearning.com/instalar-ambiente-de-desarrollo-python-anaconda-para-aprendizaje-automatico/

Pandas. (s. f.). Pandas.read_csv. Recuperado de https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Pherkad (Nombre de usuario). (2019). Primeros pasos con Numpy. Recuperado de https://python-para-impacientes.blogspot.com/2019/10/primeros-pasos-con-numpy.html

Rodríguez, D. (2019). Seis comandos mágicos de Jupyter Notebooks. Recuperado de https://www.analyticslane.com/2019/04/12/seis-comandos-magicos-de-jupyter-notebooks/