prog_datasci_2_python

October 5, 2022

1 Fundamentos de Programación

1.1 Unidad 2: Breve introducción en la programación en Python

---

1.1.1 Instrucciones de uso

Este documento es un *notebook* interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, os recomendamos que, en primer lugar, leáis las explicaciones y el código que os proporcionamos. De este modo, tendréis un primer contacto con los conceptos que exponemos. Ahora bien, **la lectura es solo el principio!** Una vez hayáis leído el contenido, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes que os permitan comprobar que habéis entendido la funcionalidad y explorar los detalles de implementación. En último lugar, os recomendamos también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

1.1.2 Introducción

En esta unidad presentaremos el lenguaje de programación Python y su sintaxis básica. Explicaremos qué son las variables, veremos cómo las podemos definir y utilizar en Python, e introduciremos los principales tipos simples (enteros, *floats*, complejos) y compuestos (cadenas de caracteres, listas, *tuples* y diccionarios).

A continuación, se incluye la tabla de contenidos, que podéis usar para navegar por el documento:

```
     <li><a href="#4.6.-Colab">4.6. Colab</a></li>
</ul>
<li><a href="#5.-Variables-y-tipos-de-variables">5. Variables y tipos de variables</a></li>
    <ul style="list-style-type:none">
   <li><a href="#5.1.-Declaración-de-variables">5.1. Declaración de variables</a></li>
    <li><a href="#5.2.-Palabras-reservadas">5.2. Palabras reservadas</a></li>
    <li><a href="#5.3.-Funciones-básicas">5.3. Funciones básicas</a></li>
        <ul style="list-style-type:none">
        <li><a href="#5.3.1-Función-print">5.3.1 Función print</a></li>
        <li><a href="#5.3.2-Función-input">5.3.2 Función input</a></li>
        </ul>
    <li><a href="#5.4.-Tipo-de-datos">5.4. Tipo de datos</a></li>
        <ul style="list-style-type:none">
     <li><a href="#5.4.1-Tipos-numéricos-y-cadenas-de-caracteres">5.4.1 Tipos numéricos y cadena
     <li><a href="#5.4.2-Listas,-tuples-y-diccionarios">5.4.2 Listas, tuples y diccionarios</a><
    </ul>
</ul>
<li><a href="#6.-Organización-del-código">6. Organización del código</a></li>
<li><a href="#7.-Operadores">7. Operadores</a></li>
<ul style="list-style-type:none">
   <li><a href="#7.1.-Operadores-aritméticos">7.1. Operadores aritméticos</a></li>
   <li><a href="#7.2.-Operadores-de-asignación">7.2. Operadores de asignación</a></li>
   <li><a href="#7.3.-Operadores-relacionales">7.3. Operadores relacionales</a></li>
    <li><a href="#7.4.-Operadores-lógicos">7.4. Operadores lógicos</a></li>
   <li><a href="#7.5.-Operadores-de-pertenencia">7.5. Operadores de pertenencia</a></li>
    <li><a href="#7.6.-Numpy">7.6. Numpy</a></li>
</ul>
 <li><a href="#8.-Recursos-para-el-programador-de-Python">8. Recursos para el programador de Pyth
<li><a href="#9.-Ejercicios-y-preguntas-teóricas">9. Ejercicios y preguntas teóricas</a></li>
 <ul style="list-style-type:none">
   <li><a href="#9.1-Instrucciones-importantes">9.1 Instrucciones importantes</a></li>
</ul>
<li><a href="#10.-Bibliografía">10. Bibliografía</a></li>
```

2 1. Qué es la programación?

En la Wikipedia encontramos la siguiente definición de *programación*:

«Programación de ordenadores o **programación informática** (a menudo abreviado *programación* o *codificación*) es el proceso de escribir, probar, depurar y mantener el código fuente de programas.

Este código fuente está escrito en un lenguaje de programación. El objetivo de la programación es crear un programa que muestra un determinado comportamiento deseado.»

En el contexto de la ciencia de datos, los ordenadores son máquinas que nos permiten tratar, procesar y analizar cantidades ingentes de datos. Este tratamiento de la información se tiene que hacer mediante un tipo de conjunto de reglas: el **algoritmo**. Podemos entender los algoritmos como plantillas o «recetas» que se especifican a la máquina mediante algún lenguaje de programación y que nos permiten tratar la información de manera inherente a nuestras necesidades. Siguiendo con la metáfora de la receta, imaginad que queremos preparar un plato de cocina y que tenemos todos los ingredientes. Nuestra receta culinaria especificará a cada paso lo que el cocinero tiene que hacer. Por ejemplo, para hacer una tortilla:

1. Romper los huevos y batirlos.
2. Poner una paella al fuego.
3. Cuando la temperatura sea bastante alta, añadir los huevos batidos.
4. Esperar que cuaje bastante.
5. Dar la vuelta.
6. Repetir el paso 4 por la otra cara.
7. Poner nuestra tortilla al plato.

Los problemas que resolveremos mediante un ordenador normalmente tendrán un contexto más numérico, pero la idea de receta será la misma: un conjunto de instrucciones que se ejecutan desde el principio hasta el final y que llevan a cabo una acción. Así pues, el algoritmo o «receta» para calcular el cuadrado de un número cualquiera sería de la manera siguiente:

1. Representar en la pantalla el mensaje «Bienvenido a la calculadora de cuadrados:».
2. Pedir al usuario que introduzca un número mediante el teclado.
3. Guardar este número.
4. Tomar el número guardado como base para multiplicarlo por sí mismo.
5. Representar en pantalla el resultado del cálculo del paso 4.
6. Salir del programa.

Especificar a un ordenador las instrucciones que tienen que ejecutarse en un programa es una abstracción que nos permiten los lenguajes de programación. En nuestro caso, usaremos Python para escribir y ejecutar nuestros programas y así solucionar problemas de ciencia de datos.

3 2. Breve historia del lenguaje de programación Python

El lenguaje nació gracias a Guido van Rossum mientras trabajaba en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica) en los Países Bajos en 1989 durante las vacaciones de Navidad, a pesar de que no fue hasta febrero de 1991 cuando fue publicada su primera versión a USENET.

Guido van Rossum es, a pesar del largo recorrido del lenguaje desde entonces, una figura muy relevante en el grupo de usuarios y desarrolladores de Python y es conocido cómo BDFL (del inglés *benevolent dictator for life*, dictador vitalicio benevolente); sus sugerencias en las discusiones en que interviene se tienen mucho en cuenta respecto a la hoja de ruta del futuro del lenguaje. Actualmente, la *Python Software Foundation* (PSF) es la organización encargada de velar por el futuro y la diseminación del lenguaje. Como nota divertida, hay que destacar que la relación

entre el nombre del lenguaje y la serie de televisión Monty Python's Flying Circus no es una mera coincidencia.

Hay dos fechas importantes que han marcado la historia del lenguaje: el 16 de octubre de 2000, momento que la versión 2.0 fue publicada, y el 3 de diciembre de 2008, cuando se publicó la versión 3.0 del lenguaje.

El lenguaje de programación Python hace especial énfasis en su capacidad para que otros programadores lo puedan leer. Esta sencillez aparente a la hora de leer código Python le ha permitido disfrutar de bastante popularidad en círculos menos enfocados en el desarrollo puro de software, como puede ser el campo de la ciencia de datos, en que confluyen personas expertas en diferentes disciplinas (matemáticas, estadística, etc.). Hay documentos de mejora del lenguaje conocidos por sus siglas **PEP-Número** (del inglés *Python Enhancement Proposals*) y es, precisamente, en uno de estos documentos, el PEP-20, en que se presenta la filosofía del lenguaje y esta marcada tendencia por la simplicidad, legibilidad y belleza del código Python:

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

Muchos de estos principios en este momento podrán parecer extraños al alumnado, pero a medida que se profundiza en el uso del lenguaje muchos resultarán habituales en la hora de escribir y leer código Python.

Finalmente, es interesante presentar al alumnado el término *pythonic*, puesto que aparece en muchas búsquedas en relación con la sintaxis de Python. Los lenguajes de programación hacen uso habitualmente de estructuras que sirven para expresar tareas o partes de código en un lenguaje común que no está considerado directamente en la sintaxis del lenguaje. Este concepto (en inglés *idiom*) está muy extendido en la comunidad de Python y muchas veces se traduce en mensajes en foros del estilo *«which is the most pythonic way to do..?»* ('cuál es la manera más pythónica de hacer..?'). Se trata de un concepto avanzado, pero que en Python es especialmente relevante, dado que algunos de estos conceptos han sido implementados en el lenguaje de manera más eficiente y pueden hacer que un código u otro se ejecute de manera más rápida si es más pythónica. Este concepto será

ampliado más adelante.

4 3. Versiones e intérpretes disponibles

Actualmente coexisten dos versiones o ramas de desarrollo: las familias 2.7 y 3.6 (en el momento de redacción de este documento, las últimas versiones son la 2.7.18 y la 3.10.1, respectivamente).

Python es un lenguaje de programación muy vivo y en evolución continua, al que se van añadiendo y corrigiendo funcionalidades muy a menudo. La rama de desarrollo 2.7.x es una versión muy madura y estable, pero fue con la aparición de la versión 3.0 cuando hubo un salto importante en la sintaxis del lenguaje. Por ejemplo, la misma expresión *print* en la rama 2.7.x pasó a tener forma de función print () en la rama 3.x. Estos cambios no se limitan a correcciones estéticas, sino que se han introducido para mejorar el rendimiento del lenguaje, la organización del código y de las librerías en el ámbito interno, etc. En resumen, la rama 3.x es el futuro de Python, mientras que la rama 2.7.x representaría el pasado del lenguaje.

En enero de 2020 finalizó el apoyo de la rama 2.x de Python, y, actualmente, la familia que se usa es la 3.x. Por este motivo, en este curso usamos Python 3.x. Aun así, si alguna vez tenéis que trabajar con código antiguo desarrollado por Python 2.x, veréis que no tendréis problemas, puesto que la esencia del lenguaje continúa siendo la misma.

En cuanto a los intérpretes, el más conocido es la implementación en C llamada CPython, que es el intérprete del lenguaje Python desarrollado por la comunidad del proyecto de código libre y soportado y dirigido por la Python Software Foundation. A pesar de que CPython es la implementación estándar y a día de hoy soporta más plataformas (cuarenta y dos en el momento de redacción de este documento), no es ni mucho menos la única. Otras implementaciones famosas son Jython, escrito en Java y que funciona en la máquina virtual de Java; PyPy, un intérprete también escrito en C, pero que pone especial énfasis en el rendimiento y a tratar la concurrencia y el paralelismo de una manera diferente; IronPython, una implementación del lenguaje para la plataforma .NET, etc.

5 4. Sintaxis básica del lenguaje de programación Python

5.1 4.1. Intérprete frente a script

Hay tres formas básicas de ejecutar código Python:

1. Ejecutar el intérprete de Python desde una terminal.
2. Escribir código Python en un fichero (típicamente con extensión PY, y que es conocido habitualmente con el nombre de *script*) y ejecutarlo usando la orden:

```
$ python script_name.py
```

3. Escribir código Python mediante *Jupyter Notebook*.

La primera forma es útil para probar órdenes del lenguaje Python de forma rápida o simplemente para usar el intérprete como calculadora. La segunda, por el contrario, nos permitirá ejecutar una y otra vez el código contenido en nuestro fichero de *script* sin tener que escribir en el intérprete

el mismo código una y otra vez. La tercera opción es la que usaremos en esta asignatura y lo explicaremos con más detalle en el apartado 4.5.

5.2 4.2. Sobre los tabuladores y los espacios en blanco

En Python, los bloques de código se indican mediante espacios en blanco. Más adelante hablaremos de los diferentes tipos de bloques de código, pero a modo de introducción haremos uso del ejemplo siguiente:

```
1 def suma(a, b):
2     return a + b
3
4 print(suma(2, 3))
```

En las líneas 1 y 2, definimos mediante la palabra clave **def** una función que recibe como argumentos dos variables de nombre `a` y `b` y que en la línea 2 devuelve la suma de estas dos variables.

En la línea 4, llamamos a esta función suma con los valores 2 y 3, y mediante la orden **print** escribimos el resultado que nos aparece por la terminal.

Como se puede observar a la línea 2, la palabra clave **return** está a cuatro espacios de distancia del comienzo de la línea anterior (línea 1). Esto es así porque estamos definiendo un bloque de código (en otros lenguajes como C o Java, se usan para la definición de bloques de código los caracteres especiales «{» y «}»).

Hay dos posibilidades para definir la indentación de código (*indentación* es un anglicismo, y en castellano tendríamos que usar el término *sangrado*, a pesar de que *indentación* está ampliamente reconocido por la comunidad informática): mediante espacios o tabulaciones. Podremos usar de manera indistinta una forma u otra, pero siempre teniendo en cuenta que nunca tendrán que mezclarse en un mismo fichero de código. Esto es así porque las tabulaciones no tienen un carácter que las distinga y no suelen presentarse de forma idónea en muchos editores de texto. Muchas veces, al presionar la tecla «Tab» en una aplicación, este carácter se convierte automáticamente en un número fijo de espacios en blanco. Por lo tanto, **será recomendable configurar el editor que elegimos para escribir código Python de forma que el tabulador se traduzca en cuatro espacios en blanco**.

En el documento PEP-8, que es la guía de estilo que siguen la mayoría de programadores Python, se detalla este aspecto.

5.3 4.3. Comentarios

Los comentarios, fragmentos de texto que no se interpretan, son una parte fundamental de cualquier programa, puesto que facilitan la lectura y permiten añadir aclaraciones necesarias para la correcta comprensión del código.

Si no se añaden comentarios al código, puede ser que el usuario no recuerde algunas de las decisiones que tomó cuando lo escribió. Por otro lado, si se lee código escrito por otra persona, los comentarios permiten seguir mejor los razonamientos detrás de este código.

Python, se pueden incluir comentarios en el código de una sola línea (precedidos por el símbolo **#**) o bien de múltiples líneas (que se indican con """ o bien '''). Los comentarios de múltiples líneas se acostumbran a usar para documentar funciones, métodos o clases en Python.

[1]:
```
# Esto es un comentario. Cuando ejecutamos esta celda
# el comentario se ignora.
```

[2]:
```
def suma(a,b):
    """ Esta función devuelve
    la suma de los argumentos a y b """
    return a + b
```

5.4 4.4. Guía de estilo

En el apartado anterior se ha tratado la guía de estilo oficial del lenguaje Python, que está disponible en linea como PEP-8. Como resumen, estos son los aspectos básicos que se tienen que seguir:

1. Las funciones y los nombres de palabras se escribirán en minúscula y los nombres largos que contengan varias palabras se unirán mediante el carácter «_».
2. Los nombres de clases tendrán su letra inicial en mayúscula para diferenciarlas de variables, funciones y objetos.
3. Las constantes irán en mayúsculas y separadas por el carácter «_» si contienen varias palabras.
4. Como ya se ha comentado en el apartado anterior, no se usarán tabuladores, sino que siempre usaremos cuatro espacios en blanco.
5. Los comentarios tienen que ser frases completas y tienen que estar actualizados (¡es importante que correspondan al código que se está comentando!).

Es muy importante seguir la guía de estilo cuando se programa Python (¡recordadlo cuando preparéis el código para las entregas de la asignatura!).

5.5 4.5. IPython y Notebook

IPython fue originalmente desarrollado como un conjunto de herramientas para facilitar el desarrollo y el procesamiento de datos Python a usuarios del mundo científico. La «i» del nombre proviene de *interactivo*, puesto que dispone de herramientas que interactúan con el usuario, como son el autocompletado de código, una ayuda extensa, etc.

El **Jupyter Notebook**, a su vez, es otra herramienta muy útil para la exploración de código, para representar resultados de manera gráfica, o por el prototipaje rápido, etc., todo esto mediante la interacción vía navegador web (Notebook arranca de manera local un servidor web). Las versiones anteriores de Jupyter Notebook se conocían como IPython Notebook.

Un notebook se organiza en celdas que siguen un orden secuencial. Hay 3 tipos de celdas, pero se usan dos principalmente: **texto** (denominado *markdown*) y **código**. Ambos tipos de celdas se pueden ejecutar, pero tienen funcionamientos diferentes. Una celda de tipo código se emplea para ejecutar el bloque de código de Python que esté dentro esta celda. Por otro lado, una celda *markdown* nos permite dar formato en el texto (por ejemplo, se pueden poner elementos en **negrita**

o *cursiva*) o añadir imágenes al notebook. El tercer tipo de celda se denomina *Raw NBConvert* y se usa para poner código que no se quiera ejecutar, puesto que estas celdas no son evaluadas por el notebook. El tipo de la celda no es fijo. Una vez creada, se puede cambiar el tipo de una celda (por ejemplo, puedes convertir una celda de código a texto, y viceversa).

Así pues, los notebooks nos proporcionan un ambiente muy dinámico, puesto que podemos escribir y evaluar nuestro código fácilmente. También nos permite combinar bloques de texto y bloques de código, cosa que lo convierte en una herramienta muy adecuada en el ámbito formativo.

5.6 4.6. Colab

Actualmente, el desarrollo de código también se puede hacer mediante plataformas en la nube. La computación en la nube (*cloud computing*) permite tener acceso remoto a programas, almacenar archivos y procesar grandes volúmenes de datos, sin tener las limitaciones de una máquina local. Una plataforma muy conocida y utilizada es **Google Colab**, un entorno que nos permite escribir y ejecutar código Python mediante notebooks completamente desde la nube. Su uso ha ido en aumento los últimos años, puesto que permite configurar determinadas prestaciones del equipo que son difíciles de tener a escala local (por ejemplo, se puede activar la GPU si se necesita un alto poder de computación). Por otro lado, como se ejecuta desde la nube, también permite compartir fácilmente los notebooks cuando se trabaja en equipo.

En esta asignatura usaremos Google Colab, puesto que nos proporciona un entorno común y colaborativo entre todos los estudiantes.

6 5. Variables y tipos de variables

Podemos entender una **variable** como un contenedor en el que podemos poner nuestros datos a fin de guardarlos y tratarlos más adelante. En Python, las variables no tienen tipos, es decir, no tenemos que indicar si la variable será numérica, un carácter, una cadena de caracteres o una lista, por ejemplo. Además, las variables pueden ser declaradas e inicializadas en cualquier momento, a diferencia de otros lenguajes de programación.

6.1 5.1. Declaración de variables

Para **declarar** una variable, usamos la expresión *nombre_de_variable = valor*. Se recomienda repasar el documento PEP-8 para indicar nombres de variables correctas, pero, *grosso modo*, evitaremos usar mayúscula en la inicial, separaremos las diferentes palabras con el carácter «_» y no utilizaremos acentos ni caracteres específicos de nuestra codificación como el símbolo del «€» o la «ñ», por ejemplo.

Veamos unos cuantos ejemplos de declaraciones de variables y cómo tenemos que usarlas:

[3]:
```
# Declaramos una variable de nombre 'variable_numerica'
# que contiene el valor entero 12.
variable_numerica = 12
```

```
# Declaramos una variable de nombre monstruo
# que contiene el valor 'Godzilla'.
monstruo = 'Godzilla'

# Declaramos una variable de nombre 'planetas'
# que es una lista de cadenas de caracteres.
planetas = ['Mercurio', 'Venus', 'Tierra', 'Marte']
```

[4]:
```
mi_edad = 25
mi_edad_en_5 = mi_edad + 5
# 'Imprimimos' el valor calculado que será, efectivamente, 30
print(mi_edad_en_5)
```

```
30
```

6.2 5.2. Palabras reservadas

En Python, hay un conjunto de palabras, llamadas **palabras reservadas**, que no podemos usar para declarar variables, puesto que tienen un significado especial y su nombre está reservado para ciertas tareas internas del lenguaje de programación.

Algunas de estas palabras forman parte de la sintaxis básica de Python (por ejemplo: *True*, *False*, *def*, *return*, *for*, *if*). Otras forman parte de funcionalidades más avanzadas (async, await).

Para consultar cuáles son las palabras reservadas en Python, se puede ejecutar el código siguiente:

[5]:
```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

6.3 5.3. Funciones básicas

Una parte clave en el diseño es que el código sea dinámico e interactivo para el usuario. Para conseguirlo, hay funciones básicas de Python que permiten al usuario interaccionar con el programa.

Una **función** es una porción de código reutilizable que se usa para hacer una tarea determinada. Si tenemos un trozo de código que tenemos que utilizar repetidamente, podemos emplear la función que contiene el código dentro, en lugar de tener que escribirlo todo otra vez.

A continuación, tenéis el ejemplo de la función suma que habíamos creado antes:

[6]:
```
# Definimos la función suma
def suma(a, b):
    """ Esta función devuelve
    la suma de los argumentos a y b """
    return a + b
```

[7]:
```
# Si queremos hacer la suma de dos números,
# no hay que escribir todo el código otra vez
suma(1, 2)
```

[7]:
```
3
```

En la unidad 3 veremos en detalle las funciones, pero, de momento, es importante conocer dos funciones básicas: la función `print` y la función `input`.

6.3.1 5.3.1. Función print

La función **print()** coge como parámetro de entrada uno o más elementos y los muestra en pantalla. Cuando se imprimen, los elementos se transforman en formato texto. Si hay más de un parámetro, se muestran por defecto separados por un espacio y, al final, se acaba con un salto de línea.

[8]:
```
# Mostramos en pantalla 3 elementos
print(1, 2, 3)

# Mostramos en pantalla dos elementos de texto
print('Hello', 'world')

# Podemos también escribir fragmentos de texto
print('Hello world')

# Si no ponemos ningún parámetro, print devuelve un salto de línea
print()

# También podemos especificar otros separadores en lugar del espacio
print(1, 2, 3, sep = ':')
```

```
1 2 3
Hello world
Hello world

1:2:3
```

Otro aspecto a destacar son las **secuencias de escape** (en inglés, *escape sequences*), conjuntos de caracteres que tienen un significado especial para el intérprete. En Python, cuando usamos la función print(), podemos especificar los saltos de línea o el tabulador mediante secuencias de escape. Por ejemplo:

[9]:
```
# Usamos \t para añadir un tabulador
print('Hello\tworld')

# Usamos \n para añadir un salto de línea
print('Hello\nworld')
```

```
Hello   world
Hello
world
```

6.3.2 5.3.2. Función input

La función **input** permite al usuario introducir un valor en pantalla que, después, puede ser guardado y empleado por el programa. Es importante destacar que el elemento que el usuario especifica mediante input() se devuelve en forma de texto.

Esta función es muy útil, puesto que permite la interacción entre el código y su usuario.

[10]:
```
# En este ejemplo, pedimos al usuario que introduzca su nombre
# Esta información se guarda en la variable "nombre"
# Aquí también podemos ver cómo usamos \n para hacer el salto de línea
nombre = input("Introduce tu nombre y apellidos\n")

# La variable "nombre" puede ser empleada después
print("El nombre introducido es el siguiente: ", nombre)
```

```
Introduce tu nombre y apellidos
Alex González
El nombre introducido es el siguiente:  Alex González
```

6.4 5.4. Tipo de datos

A continuación presentamos algunos de los tipos nativos de datos que una variable de Python puede contener: números enteros (int), números decimales (float), números complejos (complex), cadena de caracteres (string), listas (list), tuples (tuple) y diccionarios (dict).

Tipo de dato	Ejemplo
enteros (int)	num_entero = 1
números decimales (float)	num_dec = 1.5
números complejos (complex)	num_complex = 2 + 3j
cadena de caracteres (string)	text = “Hello World”
listas (list)	lista = [1, 2, 3]

Tipo de dato	Ejemplo
tuples (tuple)	peli = (‘Aladdin’, 1992)
diccionarios (dict)	fruta = {‘pera’ : 1.5, ‘manzana’ : 3.4}

Veamos unos ejemplos para cada uno de estos tipos:

6.4.1 5.4.1. Tipos numéricos y cadenas de caracteres

[11]:
```
# Un número entero
int_var = 1
another_int_var = -5

# Podemos sumarlos, restarlos, multiplicarlos o dividirlos.
print(int_var + another_int_var)
print(int_var - another_int_var)
print(int_var * another_int_var)
print(int_var / another_int_var)

# También podemos hacer la división entera.
# Como solo tratamos con números enteros, no habrá parte decimal.
print(int_var // another_int_var)
```

```
-4
6
-5
-0.2
-1
```

El comportamiento del operador `/` es una de las diferencias entre Python 2 y Python 3. Mientras que en Python 3, el operador `/` hace la división real entre dos números enteros (fijaos que `1 / -5` mujer como resultado `0.2`), Python 2 hacía la división entera (por lo tanto, el resultado de ejecutar`1 / -5` en Python 2 sería `-1`). Notad que usamos el operador `//` para expresar la división entera en Python 3.

[12]:
```
# Un número decimal o 'float'
float_var = 2.5
another_float_var = .7

# Convertimos un número entero en uno decimal
# mediante la función 'float() '
encore_float = float(7)

# Podemos hacer lo mismo en sentido contrario
# con la función 'int()'
new_int = int(encore_float)
```

```
# Podemos hacer las mismas operaciones que
# en el caso de los números enteros,
# pero en este caso la división será
# decimal si alguno de los números es decimal.
print(another_float_var + float_var)
print(another_float_var - float_var)
print(another_float_var * float_var)
print(another_float_var / float_var)
print(another_float_var // float_var)
```

```
3.2
-1.8
1.75
0.27999999999999997
0.0
```

[13]:
```
# Un número complejo
complex_var = 2 + 3j

# Podemos acceder a la parte imaginaria o a la parte real:
print(complex_var.imag)
print(complex_var.real)
```

```
3.0
2.0
```

[14]:
```
# Cadena de caracteres
my_string = 'Hello, Bio! ñç'
print(my_string)
```

```
Hello, Bio! ñç
```

Fijaos que podemos incluir caracteres unicode (como `ñoç`) en las cadenas. Esto también es una novedad de Python 3 (las variables de tipos `str` son ahora UTF-8).

[15]:
```
# Podemos concatenar dos cadenas usando el operador '+'.
same_string = 'Hello, ' + 'Bio' + '!' + ' ñc'
print(same_string)

# En Python también podemos emplear wildcards
# como en la función 'sprintf' de C.
# Por ejemplo:
name = "Guido"
num_emails = 5
print("Hello, %s! You've got %d new emails" % (name, num_emails))
```

```
Hello, Bio! ñc
Hello, Guido! You've got 5 new emails
```

En el ejemplo anterior, hemos sustituido en el *string* la cadena *%s* por el contenido de la variable *name*, que es un *string*, y *%d* por *num_emails*, que es un número entero. También podríamos usar *%f* para números decimales (podríamos indicar la precisión, por ejemplo, con *%5.3f*, el número tendría una medida total de cinco cifras, y tres serían para la parte decimal). Hay otras muchas posibilidades, pero tendremos que tener en cuenta el tipo de variable que queremos sustituir. Por ejemplo, si usamos *%d* y el contenido es *string*, Python devolverá un mensaje de error. Para evitar esta situación, se recomienda el uso de la función *str()* para convertir el valor en *string*.

También podemos mostrar el contenido de las variables sin especificar el tipo, usando `formato`:

[16]:
```
print("Hello, {}! You've got {} new emails".format(name, num_emails))
```

```
Hello, Guido! You've got 5 new emails
```

6.4.2 5.4.2. Listas, tuplas y diccionarios

Ahora presentaremos otros tipos de datos nativos más complejos: listas, tuplas y diccionarios:

[17]:
```
# Definimos una lista con el nombre de los planetas (string).
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturno', 'Uranus', 'Neptune']

# También puede contener números.
prime_numbers = [2, 3, 5, 7]

# Una lista vacía
empty_list = []

# O una mezcla de cualquier tipo:
sandbox = ['3', 'a string',
           ['a list inside another list', 'second item'],
           7.5]

# Mostramos los elementos en pantalla
print(sandbox)
```

```
['3', 'a string', ['a list inside another list', 'second item'], 7.5]
```

[18]:
```
# Podemos añadir elementos a una lista.
planets.append('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturno', 'Uranus', 'Neptune',
'Pluto']
```

[19]:
```
# O  podemos eliminar.
planets.remove('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturno', 'Uranus', 'Neptune']
```

[20]:
```
# Podemos eliminar cualquier elemento de la lista.
planets.remove('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturno', 'Uranus', 'Neptune']
```

[21]:
```
# Siempre que añadimos, será al final de la lista.
# Una lista está ordenada.
planets.append('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturno', 'Uranus', 'Neptune', 'Venus']
```

[22]:
```
# Si queremos ordenarla alfabéticamente,
# podemos usar la función 'sorted()'
print(sorted(planets))
```

```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturno', 'Uranus', 'Venus']
```

[23]:
```
# Podemos concatenar dos listas:
monsters = ['Godzilla', 'King Kong']
more_monsters = ['Cthulu']

# Mostramos el resultado en pantalla
print(monsters + more_monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

[24]:
```
# Podemos concatenar una lista con otra
# y guardarla a la misma lista:
monsters.extend(more_monsters)

# Mostramos el resultado en pantalla
print(monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

[25]:
```
# Podemos acceder a un elemento en concreto de la lista:
print(monsters[0])

# El primer elemento de una lista es el 0, por lo tanto, el segundo será el 1:
print(monsters[1])

# Podemos acceder al último elemento mediante números negativos:
print(monsters[-1])
```

```
# Penúltimo:
print(monsters[-2])
```

```
Godzilla
King Kong
Cthulu
King Kong
```

[26]:
```
# También podemos obtener partes de una lista
# mediante la técnica de 'slicing'.
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturno', 'Uranus', 'Neptune']

# Por ejemplo, los dos primeros elementos:
print(planets[:2])
```

```
['Mercury', 'Venus']
```

[27]:
```
# O los elementos entre las posiciones 2 y 4
print(planets[2:5])
```

```
['Earth', 'Mars', 'Jupiter']
```

Fijaos en este último ejemplo: en la posición 2 encontramos el tercer elemento de la lista (`'Earth'`), puesto que la lista empieza a indexar en 0. Además, el último elemento indicado (la posición`5`) no se incluye.

[28]:
```
# O los elementos del segundo al penúltimo:
print(planets[1:-1])
```

```
['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturno', 'Uranus']
```

La técnica de ***slicing*** es muy importante y nos permite gestionar listas de una manera muy sencilla y potente. Será imprescindible dominarla para afrontar muchos de los problemas que tendremos que resolver en el campo de la ciencia de datos.

[29]:
```
# Podemos modificar un elemento en concreto de una lista:
monsters = ['Godzilla', 'King Kong', 'Cthulu']
monsters[-1] = 'Kraken'

# Mostramos la lista en pantalla
print(monsters)
```

```
['Godzilla', 'King Kong', 'Kraken']
```

[30]:
```
# Una tupla es un tipo muy parecido a una lista,
# pero es inmutable, es decir, una vez declarada
# no podemos añadir elementos ni eliminar:
birth_year = ('Stephen Hawking', 1942)
```

```
# Si ejecutamos la línea siguiente, obtendremos un error de tipo 'TypeError'
birth_year[1] = 1984
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-30-78ae8e82e12c> in <module>
  5
  6 # Si ejecutamos la línea siguiente, obtendremos un error de tipo␣
↪'TypeError'
----> 7 birth_year[1] = 1984


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

Los errores en Python suelen ser muy informativos. Una investigación en internet nos ayudará en la gran mayoría de problemas que podamos tener.

[31]:
```
# Un string también es considerado una lista de caracteres.
# Así pues, podemos acceder a una posición determinada
# (a pesar de que no modificarla):
name = 'Albert Einstein'
print(name[5])

# Podemos usar slicing también con las cadenas de caracteres
print(name[7:15])

# Podemos separar un string por el carácter que consideremos.
# En este caso, por el espacio en blanco, usando
# la función 'split()'.
n, surname = name.split()
print(surname)

# Y podemos convertir un determinado string en una lista
# de caracteres fácilmente:
chars = list(surname)
print(chars)

# Para unir los diferentes elementos de una lista
# mediante un carácter, podemos usar la función
# 'join()':
print(''.join(chars))
print('.'.join(chars))
```

```
t
Einstein
Einstein
['E', 'i', 'n', 's', 't', 'e', 'i', 'n']
Einstein
E.i.n.s.t.e.i.n
```

[32]:
```python
# El operador ',' es el creador de tuplas.
# Por ejemplo, el típico problema de asignar el valor de una
# variable a otra, en Python se puede resolver
# en una línea de una manera muy elegante usando
# tuples (se trata de un idiom):
a = 5
b = -5
a, b = b, a
print(a)
print(b)
```

```
-5
5
```

El anterior ejemplo es un *idiom* típico de Python. En la tercera línea, creamos una tupla (a,b) a la que asignamos los valores uno por uno de la tupla (b,a). Los paréntesis no son necesarios, y por eso queda una notación tan reducida.

Para acabar, presentaremos los diccionarios, una estructura de datos muy útil a la que asignamos un valor a una clave en el diccionario:

[33]:
```python
# Códigos internacionales de algunos países.
# La clave o 'key' es el código de país, y el valor, su nombre:
country_codes = {34: 'Spain', 376: 'Andorra',
                 41: 'Switzerland', 424: None}

# Podemos buscar
my_code = 34
country = country_codes[my_code]
print(country)
```

```
Spain
```

[34]:
```python
# Podemos obtener todas las claves:
print(country_codes.keys())
```

```
dict_keys([34, 376, 41, 424])
```

[35]:
```python
# O los valores:
print(country_codes.values())
```

```
dict_values(['Spain', 'Andorra', 'Switzerland', None])
```

[36]:
```
# Podemos modificar valores en el diccionario o añadir nuevas claves.
# Definimos un diccionario vacío.
# 'country_codes = dict()' es una notación equivalente:
country_codes = {}

# Añadimos un elemento:
country_codes[34] = 'Spain'

#  añadimos otro:
country_codes[81] = 'Japan'

print(country_codes)
```

```
{34: 'Spain', 81: 'Japan'}
```

[37]:
```
# Modificamos el diccionario:
country_codes[81] = 'Andorra'

print(country_codes)
```

```
{34: 'Spain', 81: 'Andorra'}
```

[38]:
```
# Podemos asignar el valor vacío a un elemento:
country_codes[81] = None

print(country_codes)
```

```
{34: 'Spain', 81: None}
```

Los valores vacíos nos serán útiles para declarar una variable que no sepamos qué valor o qué tipo de valor contendrá y para hacer comparaciones entre variables. Habitualmente, los valores vacíos son *None* o ”, en el caso de las cadenas de caracteres.

[39]:
```
# Podemos asignar el valor de una variable a otra.
# Es importante que se entiendan las líneas siguientes:
a = 5
b = 1
print(a, b)
# b contiene la 'dirección' del contenedor al que apunta 'a'.
b = a
print(a, b)
```

```
5 1
5 5
```

[40]:
```
# Veamos ahora qué pasa si modificamos el valor de a o b:
a = 6
print(a, b)
```

```
b = 7
print(a, b)
```

```
6 5
6 7
```

Hasta aquí hemos presentado como es debido declarar y utilizar variables. Recomendamos la lectura de la documentación oficial en linea para fijar los conocimientos explicados.

7 6. Organización del código

Un módulo de Python es cualquier fichero con extensión *.py* que esté bajo la ruta del *path* de Python. El path de Python se puede consultar importando la librería sys:

[41]:
```
import sys
print(sys.path)
```

```
['/content', '/env/python', '/usr/lib/python37.zip', '/usr/lib/python3.7',
'/usr/lib/python3.7/lib-dynload', '', '/usr/local/lib/python3.7/dist-packages',
'/usr/lib/python3/dist-packages', '/usr/local/lib/python3.7/dist-
packages/IPython/extensions', '/root/.ipython']
```

Por defecto, Python también mira las librerías que se hayan definido en la variable del entorno $PYTHONPATH (esto puede cambiar ligeramente en un entorno Windows).

Un paquete en Python es cualquier directorio que contenga un fichero especial de nombre *__init__.py* (este fichero estará vacío la mayoría de veces).

Un módulo puede contener diferentes funciones, variables u objetos. Por ejemplo, definimos un módulo de nombre *prog_datasci.py* que contenga:

[42]:
```
# prog_datasci.py

PI = 3.14159265

def suma(x, y):
    return x + y

def resta(x, y):
    return x - y
```

Para usar desde otro módulo o *script* estas funciones, tendríamos que escribir el siguiente:

[43]:
```
# Especificamos la ruta donde está el fichero .py

from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Colab_Notebooks/prog_datasci_2
```

```
from prog_datasci import PI, suma, resta

# Y entonces podríamos usarlas normalmente.
rset = suma(2,5)
```

```
Mounted at /content/drive
/content/drive/MyDrive/Colab_Notebooks/prog_datasci_2
```

También se acostumbra a utilizar la palabra clave `as` para dar un nuevo nombre al módulo o funciones que se importan. Así, por ejemplo, podríamos hacer `from prog_datasci importe PI as mypi` para usar el nombre `mypi` para referirnos a `PI`:

[44]:
```
from prog_datasci import PI as mypi
print(mypi)
```

```
3.14159265
```

Durante el curso veremos que esta sintaxis es muy común para acortar los nombres de los módulos que importamos.

En Python también podemos usar la directiva `from prog_datasci importe *`, pero su uso está **totalmente desaconsejado**. La razón es que estaríamos importando gran cantidad de código que no usaremos (con el consiguiente aumento del uso de la memoria), pero además, podríamos tener colisiones de nombres (funciones que se llamen del mismo modo en diferentes módulos) sin nuestro conocimiento. Si no es imprescindible, no usaremos esta directiva e importaremos una a una las librerías y las funciones que necesitaremos.

Podéis aprender más sobre importar librerías y definir vuestros propios módulos aquí.

8 7. Operadores

Los operadores son símbolos que indican al intérprete de pedidos que haga una operación específica. Hay de diferentes tipos según su función y se engloban en 5 bloques: operadores aritméticos, de asignación, relacionales, lógicos y de pertenencia.

En este notebook ya hemos visto operadores, pero a continuación los describiremos más formalmente y también veremos ejemplos de cada uno:

8.1 7.1. Operadores aritméticos

Los operadores aritméticos son símbolos que nos permiten hacer una operación matemática básica. Por ejemplo, si consideramos la expresión `a=1+2`, tenemos `1` y `2` como los operandos y `+` como el operador aritmético.

Operador	Descripción	Ejemplo
+	Suma	1 + 2 = 3
-	Resta	2 - 1 = 1
	Multiplicación	2 * 3 = 6

Operador	Descripción	Ejemplo
/	División	5 / 2 = 2.5
//	División (número entero)	5 // 2 = 2
%	Módulo	16% 3 = 1
	Potencia	2 ** 4 = 16

[45]:
```
# Definimos dos elementos: a y b
a = 2
b = 3

# Hacemos la suma de a y b y mostramos el resultado
print(a + b)
```

```
5
```

8.2 7.2. Operadores de asignación

Los operadores de asignación se usan para especificar un valor a una variable. Un ejemplo ya comentado es el operador `=`. Este operador `=` se puede combinar con operadores aritméticos para poder hacer una operación y la asignación a la vez. Por ejemplo, usamos el operador básico de asignación `=` para declarar una variable: a = 5. A continuación, usamos el operador `+=` para sumar 5 a nuestra variable a y actualizar su valor: a+=5. Otra manera de escribirlo es a = a+5.

[46]:
```
# Definimos un elemento a
a = 10
print(a)

# Hacemos la suma de a y 5 y mostramos el resultado
a += 5
print(a)
```

```
10
15
```

Operador	Descripción	Alternativa
=	a = 10. Asignamos el valor 10 a la variable a	—
+=	a += 5. Sumamos 5 a la variable a y actualizamos el valor	a = a + 5
-=	a -= 2. Restamos 2 a la variable a y actualizamos el valor	a = a -2
* =	a * = 2. Multiplicamos la variable a por 2 y actualizamos el valor	a = a * 2

Operador	Descripción	Alternativa
/=	a /= 2. Dividimos la variable a entre 2 y actualizamos el valor	a = a / 2
//=	a //= 2. Dividimos la variable a entre 2 (obteniendo un entero) y actualizamos el valor	a = a //2
%=	a %= 3. Calculamos el módulo de a entre 3 y actualizamos el valor	a = a % 3
** =	a ** = 2. Elevamos a al cuadrado y actualizamos el valor	a = a ** 2

8.3 7.3. Operadores relacionales

Los operadores relacionales se usan para comparar dos variables según una condición específica. Por ejemplo: *Es la variable a más grande que la variable b?*.

Estos operadores devuelven un tipo de datos concretos, denominado `boolean`, según el resultado de comparación. Los `boolean` solo pueden tener 2 valores: `True` o `False`. Si la condición es cierta, se devuelve el boolean `True`. Si la condición es falsa, se devuelve el boolean `False`.

Por ejemplo, consideramos la situación siguiente:

Tenemos dos variables, con valor 2 y 4 y queremos usar los operadores `>` o `4` estamos preguntando si 2 es más grande que 4. Como esta condición es falsa, se devolverá el boolean False. En cambio, si hacemos `2 < 4`, la condición es cierta, puesto que 2 es más pequeño que 4, de forma que se devolverá el booelan True.

[47]:
```
# Las operaciones relacionales tendrán como resultado
# un valor cierto (True) o falso (False):
a = 2
b = 4

# El valor a es más grande que b?
print(a > b)
```

```
False
```

[48]:
```
# El valor a es más pequeño que b?
print(a < b)
```

```
True
```

[49]:
```
# Definimos un elemento b
b = 5
```

```
# El valor de b es igual que el de a?
print(b == a)

# El valor de b es diferente que el de a?
print(b != a)
```

```
False
True
```

[50]:
```
# Otros operadores relacionales disponibles son
# más pequeño o igual '<=',
# más grande o igual '>='
print(a <= b)
print(a >= b)
```

```
True
False
```

Operador	Descripción	Ejemplo
>	Devuelve True si el operando de la izquierda es más grande que el operando de la derecha	6 > 4 : True
<	Devuelve True si el operando de la izquierda es más pequeño que el operando de la derecha	2 < 4 : True
==	Devuelve True si el operando de la izquierda es igual al operando de la derecha	4 == 4 : True
!=	Devuelve True si el operando de la izquierda es diferente que el operando de la derecha	2 != 4 : True
>=	Devuelve True si el operando de la izquierda es más grande o igual que el operando de la derecha	10 >= 5 : True
<=	Devuelve True si el operando de la izquierda es más pequeño o igual que el operando de la derecha	7 <= 7 : True

8.4 7.4. Operadores lógicos

Los operadores lógicos nos permiten trabajar con múltiples condiciones. En Python tenemos 3: `and`, `or` y `not`.

Por ejemplo, consideramos 2 elementos:

a = 5; b = 2

Y 2 condiciones:

a > b (a más grande que b) ; a == b (a igual a b)

- Si consideramos el primer operador lógico `and`, devolverá True si se cumplen las 2 condiciones. En este caso, la 1.ª condición (a > b) se cumple, pero la 2.ª no (a == b). Por lo tanto, (a > b) and (a == b) devolverá el Boolean False.
- En segundo lugar, el operador lógico `or` devolverá True si alguna de las condiciones es cierta. En este caso, como la primera condición se cumple, (a > b) or (a == b) devolverá el Boolean True.
- Finalmente, el operador lógico `not` devolverá True si la condición es falsa. Tal como hemos comentado, la segunda condición es falsa, entonces, not (a == b) devolverá el Boolean True.

[51]:
```
# Definimos dos elementos: a y b
a = 5
b = 2

# Mostramos en pantalla los resultados
print('Ejemplo 1: ', (a > b) and (a == b))
print('Ejemplo 2: ', (a > b) or (a == b))
print('Ejemplo 3: ', not (a == b))
```

```
Ejemplo 1:  False
Ejemplo 2:  True
Ejemplo 3:  True
```

Operador	Descripción	Ejemplo
and	Devuelve True si ambas condiciones son ciertas	a and b
or	Devuelve True si alguna de las condiciones es cierta	a or b
not	Devuelve True si la condición es falsa	not a

8.5 7.5. Operadores de pertenencia

Los operadores de pertenencia son muy empleados cuando trabajamos con listas, puesto que nos permiten comprobar si un elemento concreto está dentro de una lista o no. `in` y `not in` son los operadores de pertenencia.

Consideremos la lista siguiente: `frutas = ['pera', 'manzana', 'plátano']`

Queremos comprobar que el elemento a (a = 'pera') está a la lista. Usaremos el operador `in`. En este caso, a in frutas devolverá el boolean True. Si cogemos otro elemento b (b = 'kiwi'), como

que este elemento no está a la lista, `not in` devolverá el boolean True, mientras que `in` devolverá el boolean False.

[52]:
```
# Definimos tres elementos: frutas, a y b
frutas = ['pera', 'manzana', 'plátano']
a = 'pera'
b = 'kiwi'

# Mostramos el resultado de las operaciones de pertenencia
print(a in frutas) # pera está en la lista
print(b in frutas) # kiwi no está en la lista
print(b not in frutas) # kiwi no está en la lista
```

```
True
False
True
```

8.6 7.6. Numpy

`Numpy` es una librería de Python especializada en cálculo numérico. En esta unidad introduciremos algunos aspectos básicos de esta librería, pero, más adelante, explicaremos en detalle como trabajar con esta librería y sus funcionalidades más importantes.

Si queremos usar la librería Numpy, antes que nada, la tenemos que importar con la función **import**, tal como hemos visto en el apartado anterior. En el caso de Numpy, se ha establecido que su abreviación más común es `np`.

[53]:
```
import numpy as np
```

Numpy nos permite hacer operaciones numéricas de manera muy sencilla. Para explicar su utilidad, consideraremos la situación siguiente:

Tenemos una lista con el precio de los productos de la compra. ¿Cómo podríamos sumar todos los elementos de la lista para obtener el precio total?

Con lo que hemos visto hasta ahora, una opción podría ser acceder a cada uno de los elementos de la lista y sumarlos con el operador aritmético de la suma (+):

[54]:
```
# Creamos la lista
lista_compra = [3.5, 4.6, 8.9]

# Accedemos a cada uno de los elementos de la lista y los sumamos
suma_compra = lista_compra[0] + lista_compra[1] + lista_compra[2]

# Mostramos el resultado en pantalla
print('Suma: ', suma_compra)
```

```
Suma:  17.0
```

A pesar de que el resultado es correcto, no parece una solución muy viable, puesto que es muy manual y, si tenemos muchos elementos dentro de la lista, no sería factible escribirlos uno a uno.

Para solucionar este problema, podemos usar la librería `Numpy`.

Numpy contiene multitud de funciones. En este caso, nos interesa sumar los elementos de una lista, por lo tanto, usaremos la función sum().

[55]:
```
# Creamos la lista
lista_compra = [3.5, 4.6, 8.9]

# Aplicamos la función suma a los elementos de la lista
# y mostramos el resultado en pantalla
print('Suma: ', np.sum(lista_compra))
```

```
Suma:  17.0
```

Como podéis ver, con Numpy hemos podido sumar todos los elementos de la lista de manera fácil y rápida.

Numpy también se puede usar para calcular variables estadísticas (media o mediana, desviación estándar, máximo, mínimo, entre otros) de los elementos de una lista:

[56]:
```
# Aplicamos la función mean (media)
# a los elementos de la lista
# y mostramos el resultado en pantalla
print('Media: ', np.mean(lista_compra))

# Aplicamos la función median (mediana)
# a los elementos de la lista
# y mostramos el resultado en pantalla
print('Mediana: ', np.median(lista_compra))

# Aplicamos la función std (desviación estándar)
# a los elementos de la lista
# y mostramos el resultado en pantalla
print('Desviación estándar: ', np.std(lista_compra))

# Aplicamos la función max (máximo)
# a los elementos de la lista
# y mostramos el resultado en pantalla
print('Máximo: ', np.max(lista_compra))

# Aplicamos la función min (mínimo)
# a los elementos de la lista
# y mostramos el resultado en pantalla
print('Mínimo estándar: ', np.min(lista_compra))
```

```
Media:  5.666666666666667
Mediana:  4.6
```

```
Desviación estándar:  2.3299976156401727
Máximo:  8.9
Mínimo estándar:  3.5
```

En este enlace podéis consultar las operaciones matemáticas que contiene la librería Numpy.

9 8. Recursos para el programador de Python

Parte del proceso de aprender a programar Python consiste en **aprender dónde y cómo buscar** información para resolver problemas que no sabemos solucionar inmediatamente. Así, familiarizarnos con la documentación oficial, aprender a usar los buscadores (como por ejemplo Google) para ayudarnos a encontrar la información que necesitamos, y descubrir el funcionamiento de las herramientas de pregunta-y-respuesta (question-and-answer o Q&A) que usan los programadores, serán competencias a trabajar en la asignatura. Estas competencias son de vital importancia, puesto que los lenguajes de programación están en evolución continua, de forma que por mucho que conozcamos un lenguaje, es habitual que a menudo nos encontremos con la necesidad de buscar algún recurso externo que nos ayude a llevar a cabo las tareas que necesitemos hacer.

Durante el curso, iremos viendo cómo podemos consultar estos recursos, y los usaremos para resolver las actividades que se irán planteando. Veréis que los enunciados de las actividades que os proponemos hacer contienen indicaciones sobre la consulta de recursos externos. Estas indicaciones os irán guiando en la búsqueda de información para resolver los ejercicios. A principio del curso, las indicaciones serán muy explícitas, y conforme el curso va avanzando lo dejarán de ser progresivamente. De este modo, iréis adquiriendo autonomía en el proceso de búsqueda de información, cosa que os servirá para afrontar los problemas de programación que encontráis en el futuro! Ahora bien, recordad que durante todo el proceso estáis siempre acompañados por el consultor de vuestra aula, que os ayudará en todo momento (¡no dudéis en preguntar todo aquello que haga falta!).

10 9. Ejercicios y preguntas teóricas

La parte evaluable de esta unidad consiste en la entrega de un fichero IPython Notebook con extensión IPYNB que contendrá los diferentes ejercicios y las preguntas teóricas que se tienen que contestar. Encontraréis el fichero (`prog_datasci_2_python_entrega.ipynb`) con las actividades en la misma carpeta que este notebook que estáis leyendo.

10.1 9.1. Instrucciones importantes

Es muy importante que a la hora de entregar el fichero Notebook con vuestras actividades os aseguréis que:

1. Vuestras soluciones sean originales. Esperamos no detectar copia directa entre estudiantes.
2. Todo el código esté correctamente documentado. El código sin documentar equivaldrá a un 0.
3. El fichero comprimido que libráis es correcto (contiene las actividades de la PAC que tenéis que entregar).

Para hacer la entrega, tenéis que ir a la carpeta del drive Colab Notebooks, clicando con el botón derecho en la PEC en cuestión y haciendo Download. De este modo, os bajaréis la carpeta de la PEC comprimida en zip. Este es el archivo que tenéis que subir al campus virtual de la asignatura.

11 10. Bibliografía

Por un lado, os recomendamos consultar la introducción a Jupyter Notebook, para familiarizaros con el entorno a trabajo.

Por otro lado, para acabar de entender los conceptos sobre variables presentados en esta unidad, os recomendamos consultar el videotutorial sobre tipo de variables (podéis descargarlo para verlo con más buena calidad) y el videotutorial sobre *strings* y listas.