Sprint 1

MAR
2

SPRINT 1 - MEETING 9

Programación avanzada: Funciones

Unirme

La meeting comienza
Hoy a las 21:00hs.

Lista de recursos

- Notebook: DS_Bitácora_09_Funcione
- Profundiza
- Challenge
- Glosario
- Potencia tu Talento - Oportunidades de...

> ***“Primero resuelve el problema, después escribe el código.” —John Johnson-Programador.***

Es probable que hayas empezado a usar las funciones mientras entendías qué son. Cómo casi siempre en la programación, la teoría se va solidificando y simplificando a medida que la llevamos a la práctica.

Ya has usado funciones, por ejemplo, para **sumar valores** - `sum()` -; para **describir Data Frames** - `describe()` - y para **imprimir en pantalla** - `print()` -. Observa que todas terminan con paréntesis: ¡es nuestra forma de decirle a Python que estamos ejecutando una función! Además, algunas de estas funciones son nativas y otras vienen definidas en las librerías que estudiamos. Podemos pensar que las librerías son un conjunto de funciones (y clases) predefinidas que alguien programó y nosotros tomamos prestadas.

¿Para qué sirven estas funciones y qué características tienen? ¿Qué pasa si queremos utilizar una función que no existe? En esta bitácora profundizaremos sobre las principales características de las funciones en Python y veremos cómo podemos definir nuestras propias funciones.

Funciones en Python: ¡profundicemos!

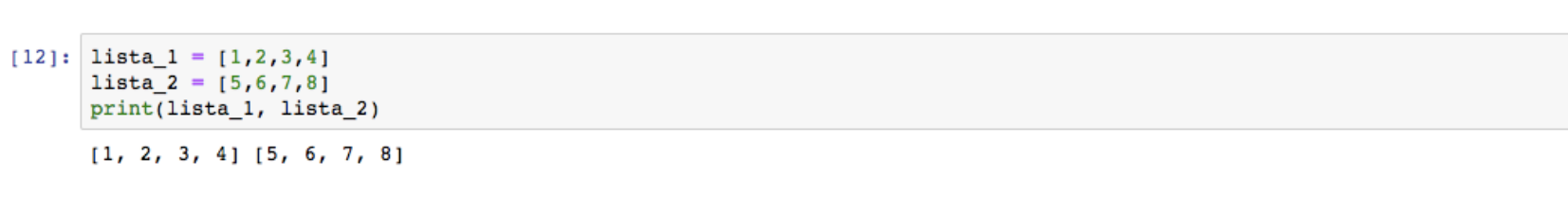
Como sabes, una función contiene un bloque de código que ejecuta una operación deseada y se la puede llamar por su nombre cuando se la necesite.

Sus principales ventajas son dos:

- **Reutilización:** permite ejecutar la operación con facilidad y repetidas veces.
- **Modularización:** permite segmentar un programa complejo para simplificar su comprensión y manipulación.

Existen todo tipo de funciones que realizan todo tipo de acción. Algunos ejemplos ya los conoces:

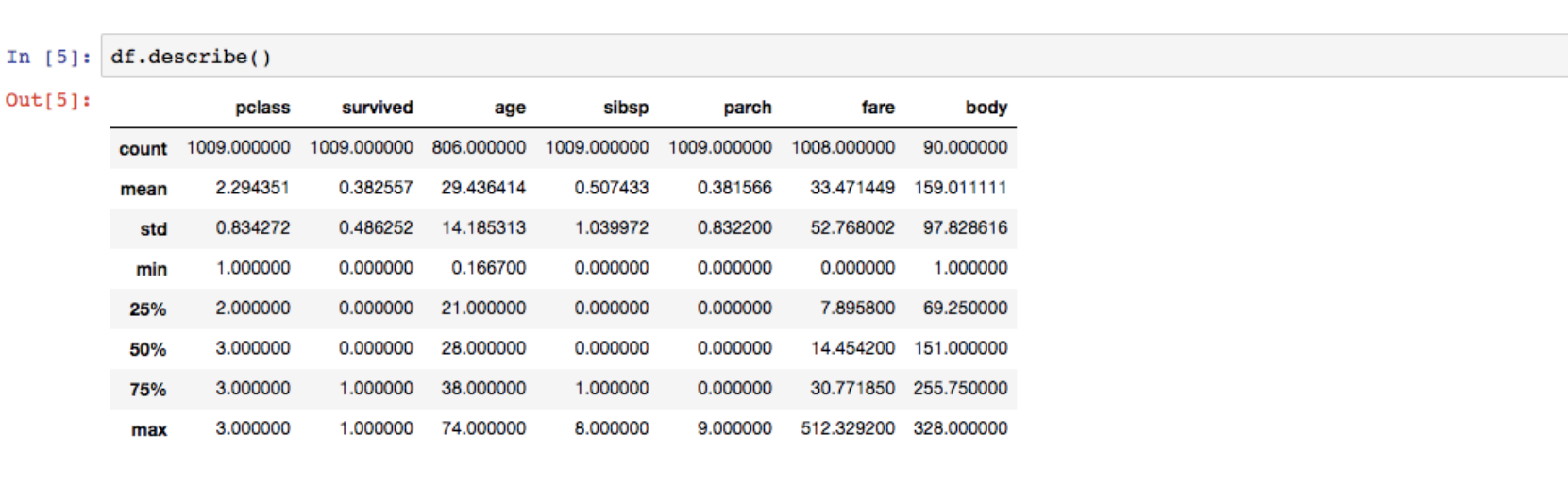
`print()` : imprime el resultado en pantalla.



```
In [12]: lista_1 = [1,2,3,4]
         lista_2 = [5,6,7,8]
         print(lista_1, lista_2)

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

`describe()` : nos brinda información relevante sobre nuestro Data Frame.



```
In [5]: df.describe()
Out[5]:
             pclass      survived    age         sibsp        parch        fare         body
      count  1009.000000 1009.000000 806.000000  1009.000000  1009.000000  1008.000000  90.000000
      mean   2.294351    0.382557    29.436414   0.507433     0.381566     33.471449    159.011111
      std    0.834272    0.486252    14.185313   1.039972     0.832200     52.768002    97.828616
      min    1.000000    0.000000    0.166700    0.000000     0.000000     0.000000     1.000000
      25%    2.000000    0.000000    21.000000   0.000000     0.000000     7.895800     69.250000
      50%    3.000000    0.000000    28.000000   0.000000     0.000000     14.454200    151.000000
      75%    3.000000    1.000000    38.000000   1.000000     0.000000     30.771850    255.750000
      max    3.000000    1.000000    74.000000   8.000000     9.000000     512.329200   328.000000
```

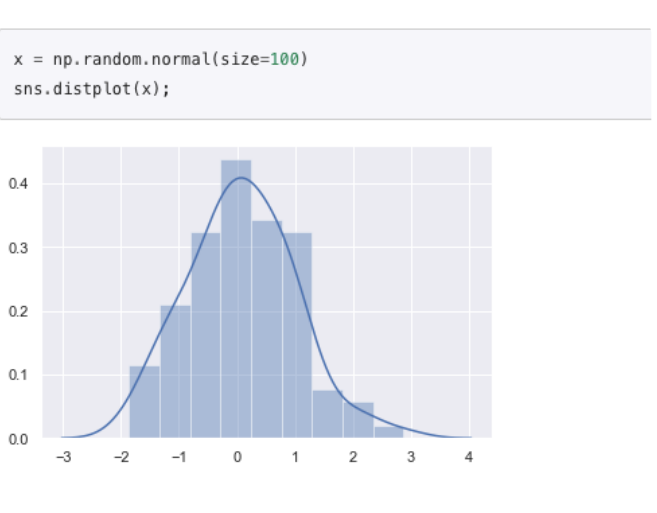
Algunas de estas funciones nos devuelven un valor, otras pueden dibujar un gráfico, realizar una suma y hasta realizar una predicción: ¡Las posibilidades son infinitas!

Una de las características principales de muchas funciones son los argumentos: la información necesaria que le damos a la función para que haga la tarea que tiene que hacer. Verás que siempre se ubican entre paréntesis. Para conocer los argumentos de una función, sólo tienes que posar el cursor sobre ella y presionar shift + tab.

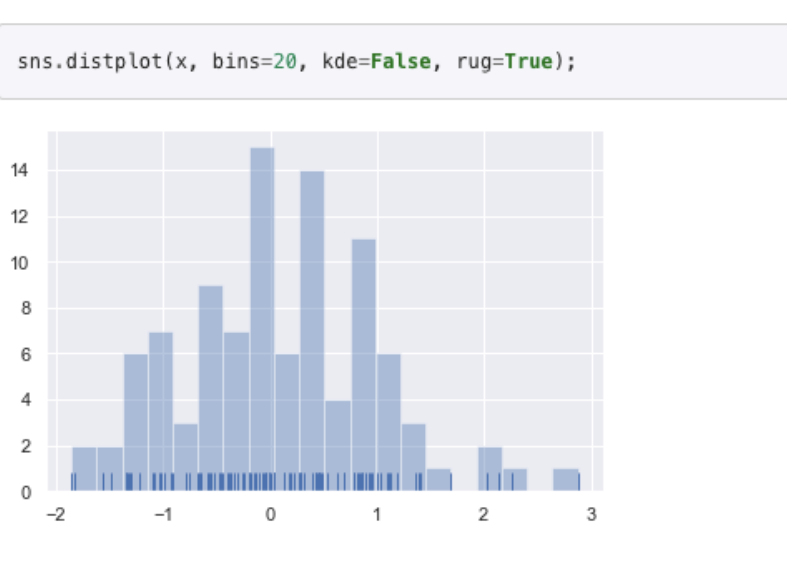
En el caso que te mostramos arriba, print() contiene dos listas como argumentos, pero podría contener otras posibilidades, como Floats o Variables, por ejemplo.

Si una función admite argumentos —y en ese caso, de qué tipo— podrás encontrarlo en su **documentación.**

Muchas de las funciones que utilizamos vienen con argumentos por default. Por ejemplo: si ejecutamos un sns.distplot(x), se mostrará en pantalla un histograma de 9 bines (recuerda que los bines son la cantidad de columnas que contiene el gráfico) y ciertos parámetros vinculados a la escala, color y tamaño, los cuales vienen predefinidos:



```
x = np.random.normal(size=100)
sns.distplot(x);
```



```
sns.distplot(x, bins=20, kde=False, rug=True);
```

Observa cómo pueden cambiar drásticamente estos gráficos con unas pocas modificaciones en sus parámetros:

Visualizing the distribution of a dataset

Definiendo una función

Esperamos que este pequeño repaso sobre conceptos y funciones ya utilizadas te haya servido. Ahora te vamos a pedir un poco más de concentración y paciencia: ¡te vamos a enseñar cómo crear tus propias funciones! A esto le llamamos definir una función.

Antes de aprender sobre la sintaxis, tómate dos minutos para pensar ¿en qué caso necesitarías definir una función?

Recuerda que una definición que dimos sobre las funciones es la de un bloque de código **encapsulado,** que se ejecuta cuando es llamado. Entre otras cosas, nos permite replicar este bloque de código con facilidad.

Generalmente, definimos una función cuando hay una operación que necesitamos ejecutar varias veces en nuestro código. Por ejemplo, imagina que tienes un Data Frame en el cual cada instancia es un alumno/a y cada columna un examen: podrías querer definir una función que te diga el/a alumno/a con mejor promedio. O quizás necesitas que te dé el promedio de todos los alumnos varones (suponiendo que cuentas con esa información). En fin, ¡hay muchas posibilidades!

Ahora sí, vayamos al código. Te vamos a mostrar cómo es la sintaxis para definir una función. Para eso vamos a usar la **palabra reservada** `def.`

```
def    nombre_funcion(argumento):
           #línea de código 1
           #línea de código 2
           #línea de código 3
           ...
           ...
           return algo
```

Así se aplicaría en un Data Frame de alumnos/as:

```
def    alumno_mejor_promedio(data_pandas):

nombre_alumno = data_pandas.Nombre[data_pandas.Promedios.idxmax()]

           return nombre_alumno
```

De ahora en más, cada vez que necesites llamar al alumno/a con mejor promedio podrás ejecutar directamente la función alumno*mejor*promedio(DataFrame) en tu código y obtendrás el nombre del alumno/a. Nota que es necesario incluir el Data Frame en el que quieres que se busque a este.

Pero ¿qué pasaría si necesitaras saber el/a mejor alumno/as de muchas escuelas/Datasets? ¿Qué le cambiarías al código? Si te animas, ¡intenta programarlo!

Veamos otro ejemplo usando el mismo Dataset: imagina que necesitas una función que retorne el promedio de un alumno/a específico/a.

```
def    promedio_alumno(nombre_alumno, data_pandas):

promedio_alumno = data_pandas.Promedios.loc[data_pandas['Nombres']
== nombre_alumno])

           return promedio_alumno
```

En este caso, la función `promedio_alumno()` retornará el promedio del alumno que le envíes como parámetro. Si no le agregaras ese argumento, la función no sabría sobre qué alumno/a tiene que operar.

Definir funciones con argumentos por default.

A veces queremos que la función ejecute ciertos argumentos por default. Este será el valor que utilizará si no le damos otro. Para convertir un parámetro en default, simplemente se le agrega un “= valor” a la hora de definir la función. Por ejemplo:

```
def    promedio_alumno(nombre_alumno = Carlos, data_pandas =
escuela_01):


promedio_alumno=data_pandas.Promedios.loc[data_pandas['Nombres']==nombre_a

           return promedio_alumno
```

En este caso, la función `promedio_alumno()` no va a necesitar un argumento para ejecutarse y siempre va a devolver el promedio de Carlos de la *escuela_01*, a menos que le incluyas como argumento el nombre de otro/a alumno/a u escuela. En ese caso, va a devolver el promedio del alumno enviado como parámetro en reemplazo del de Carlos. Este argumento por default no tiene mucho sentido y es un poco arbitrario.

¿Se te ocurre un ejemplo usando este mismo Data Frame en el que un argumento por default tenga sentido? Nota que los argumentos por default son muy comunes en los gráficos y tablas ¿por qué será?

Buenas prácticas

Para convertirte en un/a buen/a programador/a es **muy importante** tener buenas prácticas y hábitos. Esto se vuelve cada vez más relevante a medida que trabajas en códigos más complejos y con otras personas. Hay una convención de estilo llamada PEP 8, muy utilizada en la comunidad y es preciso que la conozcas:

PEP 8 -- Style Guide for Python Code

Por lo pronto, aquí van cinco recomendaciones para definir funciones que te ayudarán mucho:

- Define un nombre que empiece con **minúscula** y separa las palabras con **guión bajo.**
- **Asegúrate de que el nombre de la función sea ilustrativo de la acción que realiza.** Lo ideal es que con sólo leer la función sepas qué hace.
- **Crea funciones y documéntalas.** Si necesitas volver meses después a tu código o decides compartirlo a alguien, será más simple interpretar qué hace sin tener que leer y entenderlo completamente. Hay muchos formatos para documentar una función, pero en general incluyen: qué hace la función, cuáles son sus argumentos, y cuáles son sus *returns*.

A veces, también es conveniente incluir algún ejemplo mostrando cómo se usa. El grado de detalle depende del tiempo y de la complejidad de la función, pero es recomendable aunque sea incluir siempre por lo menos algún comentario en el código.

- **Define todas las funciones en un mismo lugar: arriba del código.** Si son muchas puedes hacerlo en un archivo nuevo (en ese caso, explora cómo importar un archivo externo).

¡Un último comentario! En el notebook sobre NumPy te mencionamos que era equivalente hacer np.max(arreglo) o arreglo.max(). En ambos casos, estamos llamando a una función que nos devuelve el máximo del arreglo. La diferencia reside en que en un la primera el arreglo es pasado como argumento (np.max(arreglo)), mientras que en la segunda forma está implícito que la función max debe operar sobre arreglo (arreglo.max()). Esta distinción la comprenderás mejor cuando conozcas otro paradigma de programación: el paradigma orientado a objetos, sobre el cual profundizaremos en la próxima bitácora.

Namespaces

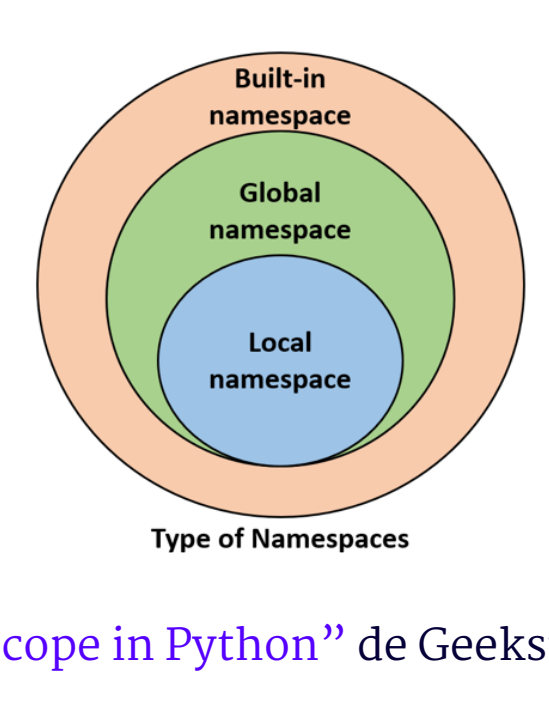
El Zen of Python (PEP 20) termina con: *“Namespaces are one honking great idea -- let's do more of those!”*. ¿De qué se trata esta declaración?

Un Namespace —espacio de nombres— es **un sistema donde cada elemento (variable, función, etc.) tiene un único nombre.**

Una analogía usual para comprender de qué se trata consiste en pensarlo como archivos en la PC. Si estamos dentro de un directorio particular, no podemos tener más de un archivo llamándose igual. De esta forma, los nombres de los archivos dentro de ese directorio actúan como un Namespace, ya que cada archivo es único. Sin embargo, podemos tener más de un archivo con el mismo nombre, pero en ese caso cada uno de esos archivos debe estar en un directorio distinto para poder diferenciarlos. Nuevamente, el nombre de los directorios es un Namespace donde cada directorio es único. Pero, otra vez, la combinación directorio + nombre de archivo debe ser única, sino no sabríamos cómo diferenciar un archivo de otro.

¿Cómo nos afecta esto cuando programamos? Si definimos una variable X = 3, y luego hacemos X = 5, no pueden convivir ambas en el mismo código. ¿Pero qué ocurre si escribimos X = 3 en nuestro código principal y X = 5 dentro de una función?

Python define los principales Namespaces, como muestra el siguiente gráfico:



Fuente: “Namespaces and Scope in Python” de GeeksforGeeks.

¿Qué significa cada uno?

- **Built-in namespace:** todas los “nombres” que quedan definidos cuando arrancamos un kernel de Python. Por ejemplo: print().
- **Global namespace:** cuando importamos módulos o trabajamos en un notebook definiendo variables, estamos trabajando en este namespace. A su vez, cada módulo crea su propio namespace.
- **Local namespace:** se crea cuando llamamos a una función y se destruye cuando termina el llamado.

Es importante notar que, por ejemplo, dos módulos pueden definir una función con el mismo nombre. Para llamar a una u otra, simplemente debes anteponer el nombre del módulo, ¡como solemos hacer con las librerías!

Dos conceptos relacionados a esto son los de *lifetime* y *scope*, que refieren a cuándo existen los nombres dentro de un Namespace y dónde podemos accederlos. No es necesario que te aprendas estos conceptos, pero si te interesa saber más, ¡observa las referencias que te dejamos en la sección Profundiza!

¡Aprende mirando!

En este vídeo continuamos algunas exploraciones sobre el dataset del video anterior.

Aclaración: los contenidos del video no están relacionados directamente con los contenidos de esta bitácora

Capítulo 3: Más allá del EDA

Empieza a trabajar en el notebook

Resuelve las secciones 1 a 4 del notebook. ¡La sección 5 la resolverás durante el encuentro!

Notebook: DS_Bitácora_09_Funciones

¡Prepárate para el próximo encuentro!

Profundiza
Te invitamos a conocer más sobre el tema de esta bitácora.

Glosario
Palabras claves para convertirte en un experto.

Challenge
Te proponemos el siguiente desafío, ¿te animas?

Potencia tu Talento - Oportunidades de Desarrollo
Tips para construir tu perfil profesional.