ACÁMICA

¿Qué hicimos hasta ahora?

- Vimos un poco de programación con Python en dos entornos: Jupyter y Colab
- Aprendimos a trabajar de forma eficiente con números y la computadora usando Numpy
- Aprendimos cómo abrir y operar con un Dataset usando Pandas
- Creamos (lindos) gráficos con Matplotlib y Seaborn
- Además, repasamos algunos conceptos de estadística: variables aleatorias, distribuciones, correlación, etc.

Vamos a seguir profundizando en herramientas (programación y librerías) y en estadística a lo largo de las clases.

Cuéntanos cómo vienes

Desde Acamica apostamos por una carrera intensiva, con el objetivo de que sea un antes y un después en tu profesión...

¿Qué opinas del ritmo de los contenidos vistos?

¿Tienes dudas con el trabajo de un Data Scientist?

¿Algún feedback, comentario, crítica?

TEMA DEL DÍA

Funciones

Las funciones son una de las bases en la programación. Hoy veremos en detalle sus características y cómo crear nuevas.

Agenda

Daily

Explicación: namespaces y funciones Lambda

Break

Hands-on training

Dudas con el proyecto

Cierre

Daily

Daily

Sincronizando...

Bitácora

¿Cómo te ha ido?
¿Obstáculos?
¿Cómo seguimos?

Challenge

¿Cómo te ha ido?
¿Obstáculos?
¿Cómo seguimos?

Repaso de la bitácora
Programación: funciones

Funciones

Una función es un bloque de código que sólo *corre* cuando es *llamado*.

Funciones

Ya hemos trabajado con varias...

```
In [3]: np.zeros(8)
Out[3]: array([0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [4]: np.ones(8)
Out[4]: array([1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [6]: data_pandas.head()
        data_pandas.head(3)
        data_pandas.tail()
        data_pandas.count()
        data_pandas.shape

        data_pandas.shape[0]
```

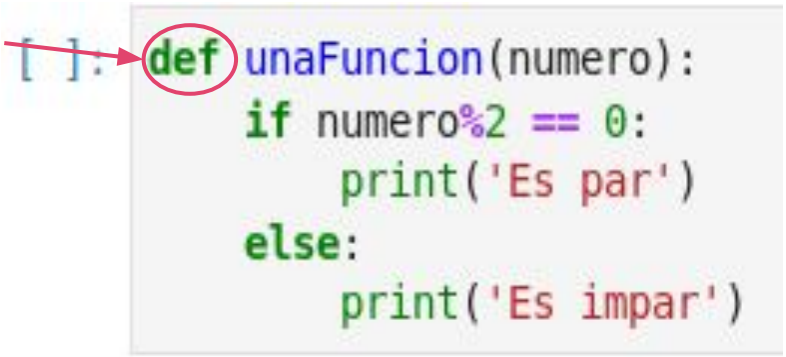
```
[ ]  data = sns.load_dataset('iris')
     data.head()
```

```
[ ]  sns.pairplot(data)
```

Funciones

¿Cómo podemos crear nuestras propias funciones?

Le dice a Python que estamos creando una función

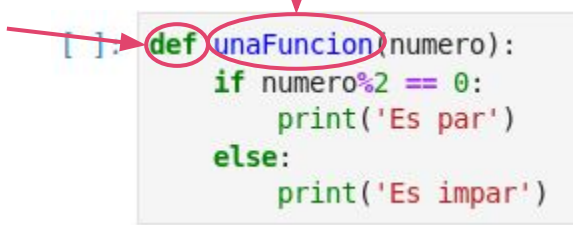


Funciones

¿Cómo podemos crear nuestras propias funciones?

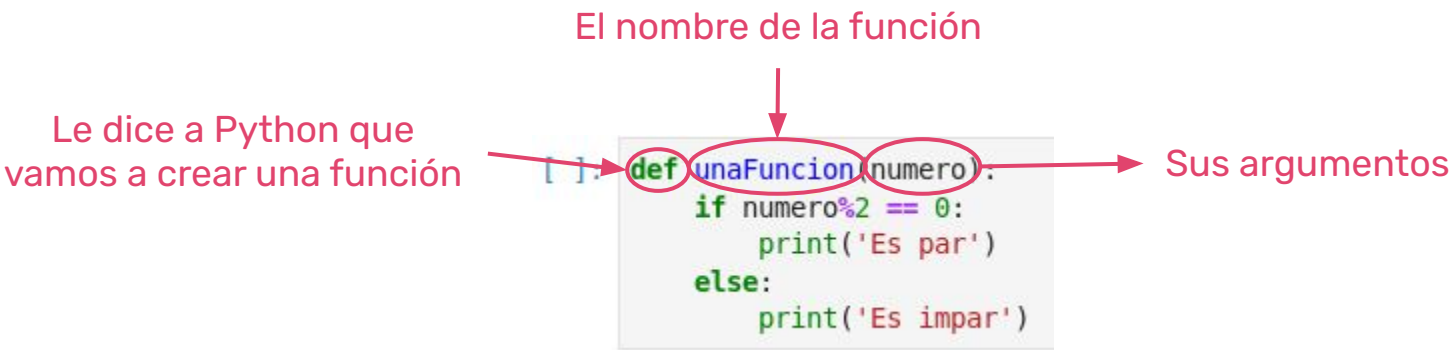
El nombre de la función

Le dice a Python que vamos a crear una función



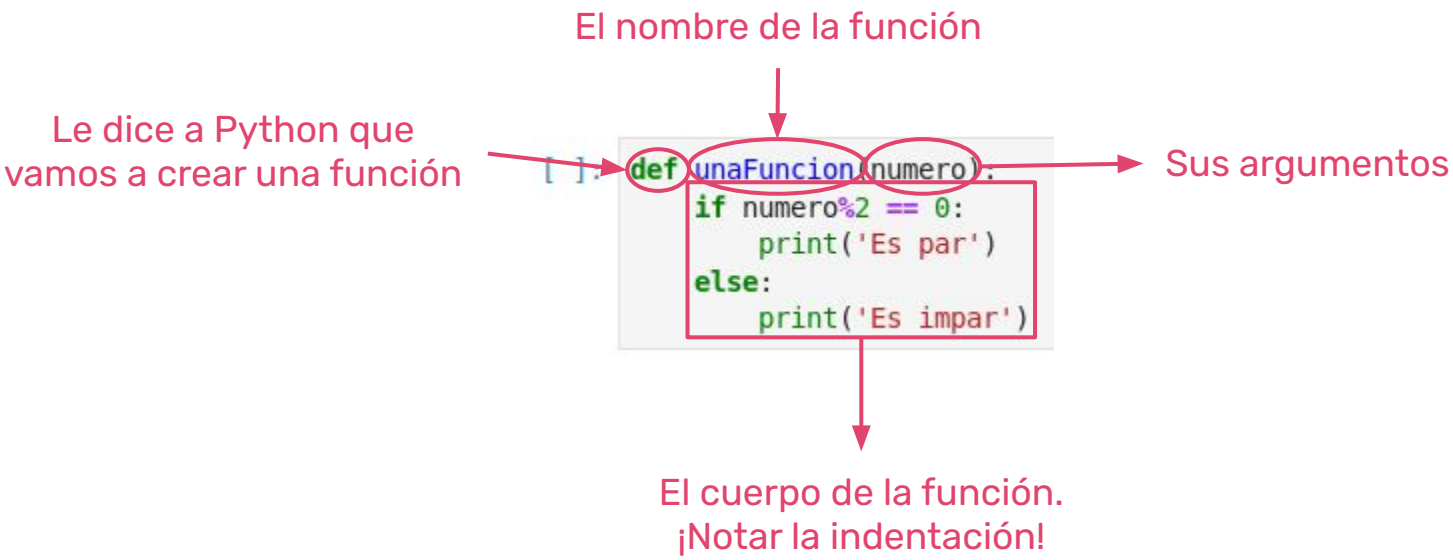
Funciones

¿Cómo podemos crear nuestras propias funciones?



Funciones

¿Cómo podemos crear nuestras propias funciones?



Funciones

Las funciones pueden tener argumentos *por default*.

```
[7]: def division(dividendo, divisor = 2):
         print(dividendo/divisor)
```

Funciones

Las funciones pueden tener argumentos *por default*.

```
[7]: def division(dividendo, divisor = 2):
         print(dividendo/divisor)
```

Salvo que le digamos lo contrario, el valor default de divisor es 2.

Funciones

Las funciones pueden tener argumentos *por default*.

```
[7]: def division(dividendo, divisor = 2):
         print(dividendo/divisor)
```

Salvo que le digamos lo contrario, el valor default de divisor es 2.

```
[8]: division(9)
     division(9, 3)
     division(dividendo = 9, divisor = 3)
     division(9, divisor = 3)
```

```
4.5
3.0
3.0
3.0
```

Funciones

Las funciones pueden “devolver” resultados.

```
[9]: def division(dividendo, divisor = 2):
         variable_auxiliar = dividendo/divisor
         return variable_auxiliar
```

Funciones

Las funciones pueden “devolver” resultados.

```
[9]: def division(dividendo, divisor = 2):
         variable_auxiliar = dividendo/divisor
         return variable_auxiliar
```

Con **return** le decimos qué queremos que devuelva. Puede ser más de un objeto.

Funciones

Las funciones pueden “devolver” resultados.

```
[9]: def division(dividendo, divisor = 2):
         variable_auxiliar = dividendo/divisor
         return variable_auxiliar

[10]: resultado_division = division(9,3)

[11]: print(resultado_division)

      3.0
```

Con **return** le decimos qué queremos que devuelva. Puede ser más de un objeto.

Funciones

Las funciones pueden “devolver” resultados.

Asignamos el resultado de llamar a la función a una nueva variable

```
[9]: def division(dividendo, divisor = 2):
         variable_auxiliar = dividendo/divisor
         return variable_auxiliar

[10]: resultado_division = division(9,3)

[11]: print(resultado_division)

3.0
```

Con **return** le decimos qué queremos que devuelva. Puede ser más de un objeto.

Funciones Lambda

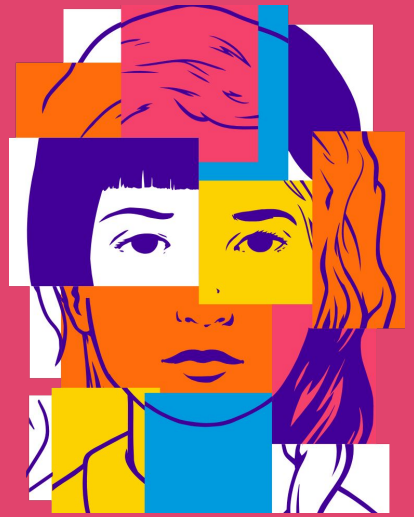
```
[18]: lambda_division = lambda x,y: x/y
      lambda_division(80,10)

[18]: 8.0
```

Una función Lambda es una forma conveniente de crear una función **en una sola línea**.
También se las conoce como funciones anónimas, ya que no tienen nombre, sino que se asignan a una variable.

Algunas características:

1. Pueden tener cualquier cantidad de argumentos, pero solo una expresión
2. No necesitan un *return*
3. Muy cómodas para crear funciones rápido
4. Se combinan muy bien con funciones como map(), filter(), apply(), applymap(), etc.

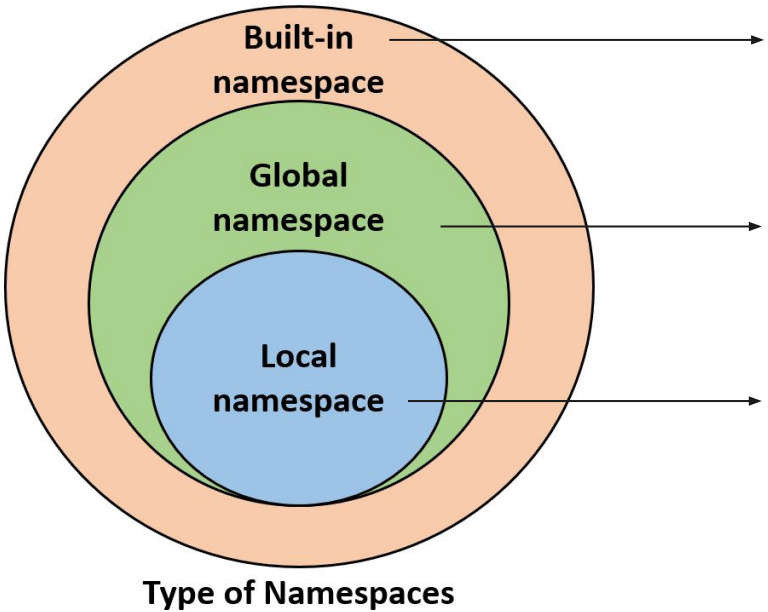
Namespaces



Namespaces (informalmente)

Un Namespace - espacio de nombres -
es un sistema donde cada elemento (variable, función, etc.)
tiene un único nombre y se le asigna un *ámbito*.

Namespaces (informalmente)



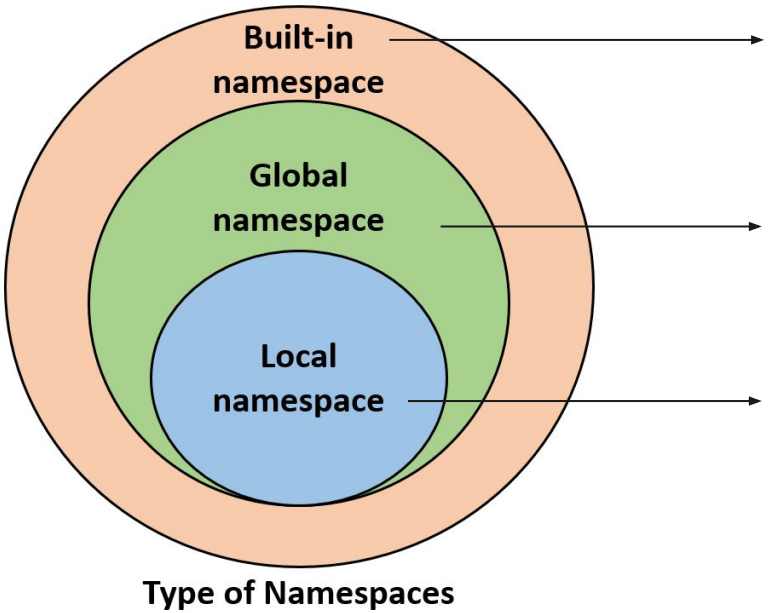
Todas los “nombres” que quedan definidos cuando arrancamos un kernel de Python. Ejemplo: `print()`.

Cuando importamos módulos o trabajamos en un notebook, estamos trabajando acá.

Se crea cuando llamamos a una función y se destruye cuando termina el llamado.

Fuente: GeeksforGeeks

Namespaces (informalmente)



Type of Namespaces

Todas los “nombres” que quedan definidos cuando arrancamos un kernel de Python. Ejemplo: `print()`.

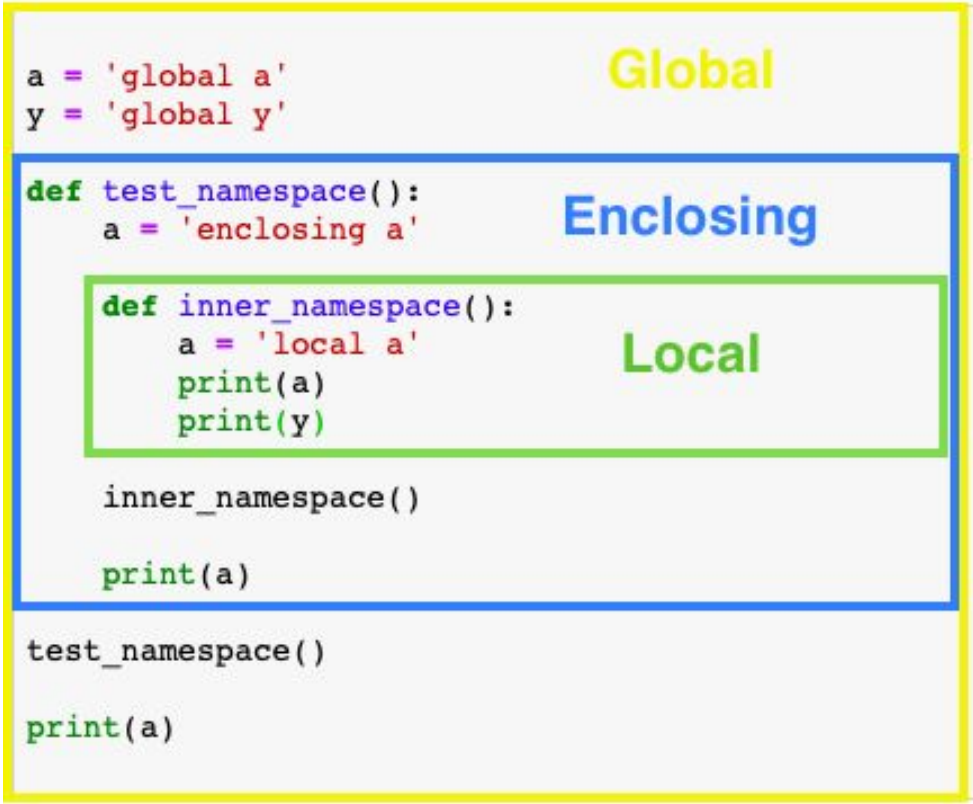
Cuando importamos módulos o trabajamos en un notebook, estamos trabajando acá.

Se crea cuando llamamos a una función y se destruye cuando termina el llamado.

En general, es fácil ir de afuera hacia adentro, pero no de adentro hacia afuera.

Fuente: GeeksforGeeks

Fuente: Artículo Medium




```
local a
global y
enclosing a
global a
```

¿Cuál es la diferencia entre estos bloques de código?

1

```
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
---------------------------------------
NameError
<ipython-input-15-28655831cd39> in <module
      3     return variable_auxiliar
      4 print(division(50))
----> 5 print(divisor)

NameError: name 'divisor' is not defined
```

2

```
divisor = 5
def division(dividendo):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
10.0
5
```

3

```
divisor = 5
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
5
```

¿Cuál es la diferencia entre estos bloques de código?

1

```
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
---------------------------------------
NameError
<ipython-input-15-28655831cd39> in <module
      3     return variable_auxiliar
      4 print(division(50))
----> 5 print(divisor)

NameError: name 'divisor' is not defined
```

2

```
divisor = 5
def division(dividendo):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
10.0
5
```

3

```
divisor = 5
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
5
```

The Zen of Python

```
Namespaces are one honking great idea -- let's do more of those!
```

¿Cuál es la diferencia entre estos bloques de código?

1

```
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
---------------------------------------
NameError
<ipython-input-15-28655831cd39> in <module
      3     return variable_auxiliar
      4 print(division(50))
----> 5 print(divisor)

NameError: name 'divisor' is not defined
```

Acá **`divisor`** es una variable local dentro de la función, deja de existir cuando termina el llamado a la función.

2

```
divisor = 5
def division(dividendo):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
10.0
5
```

Acá **`divisor`** es una variable global. En el llamado de la función, como no encuentra una variable local llamada **`divisor`**, busca una variable global llamada **`divisor`**.

3

```
divisor = 5
def division(dividendo, divisor = 2):
    variable_auxiliar = dividendo/divisor
    return variable_auxiliar
print(division(50))
print(divisor)
```

```
25.0
5
```

Acá existe **`divisor`** como variable global y **`divisor`** como variable local. El llamado a la función usa siempre primero la variable local.

¡BREAK!

Ph. Credit: Drew Coffmann

Hands-on training

Hands-on training

DS_Bitácora_09_Funciones.ipynb

Proyecto 1

Primer proyecto.

Ejercitación y dudas: 30 minutos.

Recursos

Funciones

- Te recomendamos este artículo en español. Además de ser claro e incluir ejemplos; incluye el concepto de namespace a la explicación. La próxima clase vamos a ver Objetos, un elemento que también menciona: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion5/funciones.html
- Otro artículo con contenido similar, pero en inglés: https://www.geeksforgeeks.org/functions-in-python/

Para la próxima

- Termina el notebook de hoy.
- Lee la bitácora 10 y carga las dudas que tengas al Trello.

En el encuentro que viene uno/a de ustedes será seleccionado/a para mostrar cómo resolvió el challenge de la bitácora. De esta manera, ¡aprendemos todos/as de (y con) todas/as, así que vengan preparados/as.

ACÁMICA