DMLA 2022 - Plataformas para Machine Learning

Parte 1: Python y Ciencia de Datos

Francisca Cattan y Nicolás Alvarado

Pontificia Universidad Católica de Chile

Octubre, 2022

Objetivo del curso

Introducir y aplicar las distintas plataformas, lenguajes y librerías requeridas en los distintos pasos involucrados en la construcción de sistemas de Machine Learning, con especial énfasis en el uso del lenguaje Python.

Aprendizajes esperados

- Explicar conceptos como variables, control de flujo, funciones listas, diccionarios, strings y DataFrames.
- Aplicar el razonamiento algorítmico para generar la solución a un problema como una secuencia de pasos bien definidos, secuencias de control, utilización de funciones y librerías.
- Utilizar módulos existentes e integrar sus funcionalidades dentro de un programa propio, con énfasis en herramientas de visualización de datos.

Aprendizajes esperados

- Analizar y manipular conjuntos de datos utilizando los principios del análisis exploratorio de datos.
- Entender y aplicar los conceptos y métodos fundamentales de análisis y visualización de datos, y aprendizaje de máquina a problemas reales, interpretando adecuadamente los resultados y generando acciones de valor agregado.

Evaluación

- La evaluación del curso estará compuesta por:
  - 2 Tareas (60%).
  - 2 Controles Teóricos (40%).

Evaluaciones

- Los controles teóricos serán de alternativas mediante Google Forms, individuales y en horario de clases.

- Las tareas buscan aplicar el diseño y desarrollo en código a un caso concreto de análisis de datos.
- Las tareas duran dos semanas, deben entregarse en archivos .ipynb ejecutables en Colab y pueden desarrollarse en parejas, si así lo desean.

Contenidos del curso

- Python como entorno para Machine Learning y ciencia de datos.
- Pandas y Scikit-Learn.
- Plataforma Hadoop y Spark.
- PySparkML, SparkSQL y SparkML.
- Aspectos técnicos de las plataformas Cloud.
- Herramientas Cloud para Machine Learning.

Calendario

		Tarea 1		
miércoles 5	miércoles 12	lunes 17	lunes 24	lunes 9
Python y Pandas	**Scikit-learn y datos**	**Scikit-learn y modelos de ML**	**Control 1**	**Spark y Hadoop**

Tarea 2			
lunes 14	miércoles 23	lunes 28	martes 29
PySparkML, SparkSQL y SparkML	**Plataformas Cloud**	**Control 2**	

¿Consultas?

Primero vamos a hacer una revisión completa del lenguaje de programación en Python.


```
if awake:
    code()
elif tired:
    drink_coffee()
```

Fundamentos de Python

1. Primera parte:
   - Tipos de datos
   - Operadores
   - Estructuras de control
   - Funciones
   - Manejo de módulos
2. Segunda parte:
   - Estructuras de datos
   - Obtención de datos
   - DataFrames en Pandas

Fundamentos de Python: Contenidos generales

- ¿Qué es programación?
- ¿Qué es Python?
- Entorno de desarrollo
- Google Colab Notebook
- Estructuras de datos

¿Qué es programación?

Corresponde a la acción de entregar instrucciones claras, especificas y sin ambigüedades a una máquina. Por ejemplo:

```
1  x = 2
2  y = 5
3  suma = x + y
4  print(suma)
```

¿Qué es Python?

- Lenguaje de programación interpretado.
- Multiplataforma.
- Principalmente orientado a objetos (OOP).
- Código abierto.

Entorno de desarrollo

- Python 3, archivos .py
- Jupyter Notebook (local), archivos .ipynb
- Google Colab (online), archivos .ipynb

Cuando trabajemos con archivos .ipynb nos referiremos a ellos como notebooks.

Google Colab

Ir al siguiente link https://colab.research.google.com/

Expresiones

Todos los programas que vamos a escribir se pueden construir con:

- Datos de **entrada** *(input)* que se *leen*.
- Datos de **salida** *(output)* que se *escriben*.
- **Variables** que *recuerdan* datos.
- **Operaciones matemáticas** que *calculan* datos.
- Instrucciones **condicionales** que se ejecutan dependiendo de una condición.
- Instrucciones **repetitivas** que se ejecutan múltiples veces dependiendo de una condición.

Tipos de datos

- Numeros
  - `int...3`
  - `float...3.0`
  - `complex...3 + 0j`
- Texto
  - `str..."Texto con comillas dobles" o simples`
- Booleano
  - `bool...True,False`

Numéricos

Los datos o variables numéricas se resumen en dos:

- `int`, para denotar a los enteros, i.e.

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

Numéricos

Los datos o variables numéricas se resumen en dos:

- `int`, para denotar a los enteros, i.e.

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

- `float`, para denotar a los decimales, i.e.

$$\mathbb{R} = (-\infty, \infty).$$

Tipos de datos

```
5                  # int
5.0                # float
"Hello world"      # string
'Hello world'      # string
True               # boolean
False              # otro boolean
True or False    # concatenacion de operadores logicos
true             # esto no es nada por si solo
```

Tipos de datos

En caso de dudas, podemos verificar de qué tipo es un dato utilizando la función `type()`

```
1  type(5)          # verificar que es un entero
2  type(5.0)        # verificar que es un float
3  type("texto")    # verificar que es un string
4  type(True)        # verificar que es un boolean
```

Tipos de datos

También podemos convertir datos de un tipo a otro utilizando la funciones `int()`, `float()`, `str()`. A esto se le denomina **casting**, o castear informalmente en español.

```
1  int("5")          # transforma de string a int
2  float(5)          # transforma de entero a float
3  str(543)          # transforma de int a string
```

Operadores

- Nos permiten realizar operaciones con los distintos tipos de datos o variables.
- Generan un nuevo valor luego de la operación.
- Un mismo operador puede tener más de un comportamiento en función de los tipos de datos relacionados.

Operadores sobre números: Aritméticos

Tabla de resumen

Operador	Descripción	Aridad	Precedencia
**	Exponente	Binario	1
+	Identidad	Unario	2
-	Negación	Unario	2
*	Multiplicación	Binario	3
/	División	Binario	3
//	División entera	Binario	3
%	Módulo	Binario	3
+	Suma	Binario	4
-	Resta	Binario	4

Obs: para ahorrarse problemas, usen paréntesis.

Operadores de comparación

Estos operadores se utilizan para comparar dos valores, retornando como resultado de esa comparación un valor del tipo `bool`.

Operador	Descripción
a == b	Retorna `True` ssi a es **igual a** b
a != b	Retorna `True` ssi a es **distinto a** b
a < b	Retorna `True` ssi a es **menor a** b
a <= b	Retorna `True` ssi a es **menor o igual a** b
a > b	Retorna `True` ssi a es **mayor a** b
a >= b	Retorna `True` ssi a es **mayor o igual a** b

Operadores lógicos

Estos operadores se utilizan en las expresiones condicionales, para agregar más detalle a las condiciones. Nos ayudan a concatenar operadores de comparación o datos del tipo `bool`.

Operador	Descripción
a and b	Retorna `True` ssi a **y** b son `True`
a or b	Retorna `True` ssi a **ó** b son `True` (o ambos)
not a	Retorna `True` ssi a **es False**

Ejemplos de operadores

Comparaciones y lógica

```
8 == 8          # True
8 == 9          # False
8 != 9          # True
8 <= 9          # True
8 > 8           # False
'hola' == 'Hola'          # False
not True                  # False
not False                 # True
True and True             # True
True and False            # False
True or False             # True
False or True             # True
```

Instrucciones básicas en Python

- `print()`
- `input(text)`: Nos permite recibir un input de texto del usuario.
- `round(num, n)`: Redondea el número num a n decimales.
- `len(text)`: los textos son cadena de caracteres, por ende len() retorna el número total de caracteres incluyendo espacios.

Listas

Definición: Una lista es una serie mutable e indexable de elementos.

Sintaxis

```
lista = [elemento1, elemento2, elemento3, ...]
```

Ejemplos

```
lista_1 = []                          # lista vacia
lista_2 = [3,5,7,9,42]                # lista de numeros
lista_3 = ["a","B","Hola","dd"]       # lista de strings
lista_4 = ["a",3.4,True,42]             # lista mixta
lista_5 = [[1,2,3],[4,5],["6"]]        # lista de listas
```

Listas: operadores básicos

Tabla de operadores básicos

Operador	Operación
$l_1 + l_2$	Concatena l_1 y l_2
$n * l$	Concatena n veces l
i in l	es True ssi $i \in l$
i not in l	es True ssi $i \notin l$

```
1  l1 = [1,2,3]
2  l2 = [4,5]
3  print(l1 + l2)
4  print(3 * l2)
5  print(2 in l1)
6  print(2 not in l2)
```

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

Listas: sublistas

Sublistas: Para obtener una sublista `l` se usa `l[i:j:k]`.

- `l[i:j]` retorna la sublista desde `i` hasta `j-1`.
- `l[i:j:k]` retorna la sublista l[i:j], en intervalos de `k`.

```
1  l = [3,4,2,4,9,6]    # lista de numeros
2  print(l[1:4])        # >>> [4,2,4]
3  print(l[3:])         # >>> [4,9,6]
4  print(l[:2])         # >>> [3,4]
5
6  print(l[::2])        # >>> [3, 2, 9]
7  print(l[1::2])       # >>> [4, 4, 6]
8  print(l[::-1])       # >>> [6, 9, 4, 2, 4, 3]
```

Diccionarios

Es una asociación de pares de elementos mediante la relación llave-valor (key-value).

Diccionarios

Es una asociación de pares de elementos mediante la relación llave-valor (key-value). Por ejemplo:

```
1  d1 = {'nom':'Jen', 'edad':63}
2  d2 = dict()
3  d1['nom'] # 'Jen'
4  d1.update('año': 1960) #error
```

Diccionarios

Los diccionarios son modificables, se pueden adherir o quitar elementos. No permiten duplicados, es decir, no se puede tener dos items con la misma llave.

```
d = {
    "marca": "Ford",
    "modelo": "Mustang",
    "año": 1964,
    "año": 2020
}
d #{'marca': 'Ford',
#'modelo': 'Mustang', 'año': 2020}
```

Estructuras de control

Instrucciones condicionales **`if, elif, else`**

- Permiten ejecutar una o más instrucciones **solamente** si se cumple una condición.
- Expresión cuyo valor es del tipo `bool`.
- Dos valores posibles: `True` ó `False`.

Flujos condicionales `while, for`.

- Loops **`while`**: Permite ejecutar una serie de líneas siempre y cuando se cumpla una condición.
- Loops **`for`**: Permite iterar sobre una secuencia finita.

Estructuras de control

```
1  x = 7
2  if x > 8:
3      print(x, "es mayor que 8")
4  else:
5      print(x, "es menor que 8")
```

Estructuras de control

Se puede reducir aún más el código para dar más opciones al usuario:

```
1  x = 7
2  if x < 8:
3      print(x, "Es menor que 8")
4  elif x == 8:
5      print(x, "Es igual a 8")
6  else:
7      print(x, "Es mayor a 8")
```

if, elif, else

- La instrucción **`elif`** permite ejecutar una sección de código si se cumple una condición y no se ha cumplido ningún `if` o `else` anterior.

Sintaxis if,elif,else

```
1  if condicion_1:      # Se abre la condicion
2      codigo_if
3      .
4      .
5  elif condicion_2:    # Si if es False y elif es True
6      codigo_elif
7      .
8      .
9  else:
10     codigo_else      # Si if y elif son False
11     .
12     .
13 codigo_fuera_de_if_elif_else
```

if, elif, else

¿Cuál es la diferencia entre estos dos códigos?

```
1  a = int(input("a: "))
2  if 0 < a:
3      print(1)
4  if 5 < a:
5      print(2)
6  if 10 < a:
7      print(3)
8  if 15 < a:
9      print(4)
10 else:
11     print(5)
```

```
1  a = int(input("a: "))
2  if 0 < a:
3      print(1)
4  elif 5 < a:
5      print(2)
6  elif 10 < a:
7      print(3)
8  elif 15 < a:
9      print(4)
10 else:
11     print(5)
```

Estructuras de control

Instrucciones condicionales **`if, elif, else`**

- Permiten ejecutar una o más instrucciones **solamente** si se cumple una condición.
- Expresión cuyo valor es del tipo `bool`.
- Dos valores posibles: `True` ó `False`.

Flujos condicionales `while, for`.

- Loops **`while`**: Permite ejecutar una serie de líneas siempre y cuando se cumpla una condición.
- Loops **`for`**: Permite iterar sobre una secuencia finita.

Estructuras de control

- La instrucción **`while`** permite ejecutar varias veces la misma sección de código.

Sintaxis while

```
1  while condicion:          # Se abre el ciclo
2      codigo_while          # Se repite MIENTRAS se
3      .                     # cumpla la condicion
4      .
5      .
6
7  codigo_fuera_de_while     # Se ejecuta cuando la condicion deja
   ↪  de cumplirse
```

Estructuras de control

- La instrucción **`while`** permite ejecutar varias veces la misma sección de código.

Sintaxis while

```
1  while condicion:          # Se abre el ciclo
2      codigo_while          # Se repite MIENTRAS se
3      .                     # cumpla la condicion
4      .
5      .
6
7  codigo_fuera_de_while     # Se ejecuta cuando la condicion deja
   ↪  de cumplirse
```

Obs: Necesitamos que el código modifique la condición para poder salir.

Estructuras de control

¿Que puede decir del siguiente código?

```
1  x = 5
2  while x < 8:
3      print(x, "Es menor que 8")
4  print("Terminamos")
```

Estructuras de control

¿Que puede decir del siguiente código?

```
1  x = 5
2  while x < 8:
3      print(x, "Es menor que 8")
4  print("Terminamos")
```

Cambiémoslo un poco:

```
1  x = 5
2  while x < 8:
3      x = x + 1 # equivalente a x += 1
4      print(x, "Es menor que 8")
5  print("Terminamos")
```

Estructuras de control

- La instrucción **`for`** también permite ejecutar varias veces la misma sección de código, pero combinada con la instrucción **`in range`** podemos repetirla un número fijo de veces.

Sintaxis for

```
1  for variable in range(inicial, final, step):
2      codigo_for
3      .
4      .
5      .
6
7  codigo_fuera_de_for
```

Estructuras de control

- La instrucción **`for`** también permite ejecutar varias veces la misma sección de código, pero combinada con la instrucción **`in range`** podemos repetirla un número fijo de veces.

Sintaxis for

```
1  for variable in range(inicial, final, step):
2      codigo_for
3      .
4      .
5      .
6
7  codigo_fuera_de_for
```

Obs: No necesitamos que el código modifique la condición para poder salir.

range(final)

- Cuando la instrucción **`for`** combinada con la instrucción **`in range(n)`** recibe 1 parametro, este corresponde al final del rango.
- El rango va de 0 a $n-1$.

```
1  for i in range(5):  # defino el rango de 0 a 5
2      print(i)        # e imprimo
```

range(final)

- Cuando la instrucción **`for`** combinada con la instrucción **`in range(n)`** recibe 1 parametro, este corresponde al final del rango.
- El rango va de 0 a $n-1$.

```
1  for i in range(5):  # defino el rango de 0 a 5
2      print(i)        # e imprimo
```

```
0
1
2
3
4
```

range(inicio,final)

- Cuando la instrucción **`for`** combinada con la instrucción **`in range(n,m)`** recibe 2 parametros, el primero es el inicio y el segundo el final.
- El rango va de n a $m-1$.

```
1 for i in range(5,15):   # defino el rango de inicio y fin
2     print(i)            # e imprimo
```

range(inicio,final)

- Cuando la instrucción **`for`** combinada con la instrucción **`in range(n,m)`** recibe 2 parametros, el primero es el inicio y el segundo el final.
- El rango va de n a $m-1$.

```
1  for i in range(5,15):    # defino el rango de inicio y fin
2      print(i)             # e imprimo
```

range(inicio,fin,salto)

- También podemos elegir el incremento del rango, utilizando un tercer parámetro llamado *step*.
- El rango va de n a $m-1$, con salto s.

```
1  for x in range(2, 15, 3):  # de 2 a 15 con step 3
2      print(x)
```

range(inicio,fin,salto)

- También podemos elegir el incremento del rango, utilizando un tercer parámetro llamado *step*.
- El rango va de n a $m-1$, con salto s.

```
1  for x in range(2, 15, 3):  # de 2 a 15 con step 3
2      print(x)
```

```
2
5
8
11
14
```

Estructuras de control

¿Que arroja lo siguiente?

```
1  for i in range(2):
2      print("Hola")
3  print("Adiós")
```

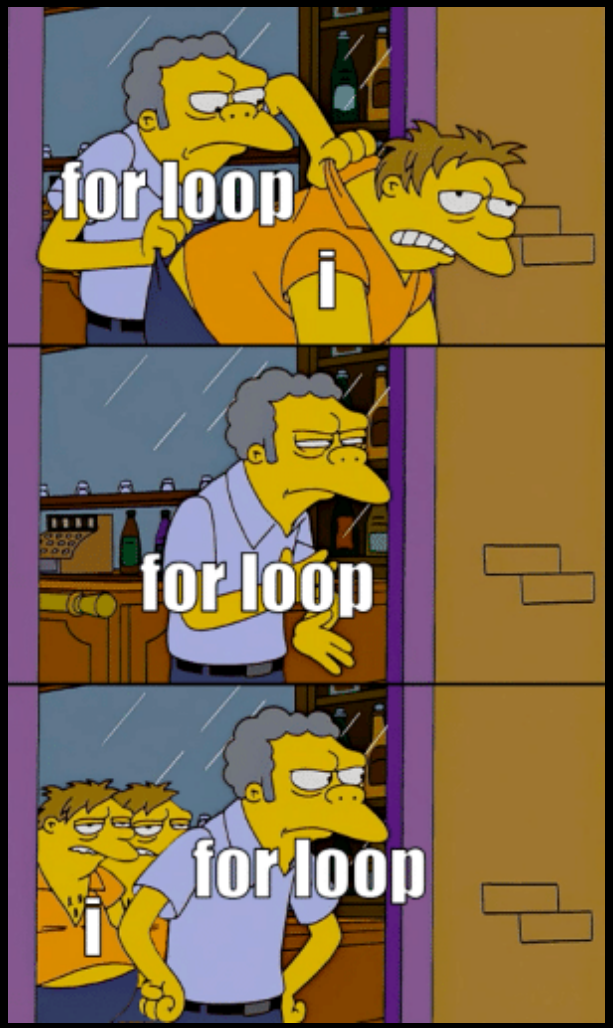
Estructuras de control

¿Que arroja lo siguiente?

```
1  for i in range(2):
2      print("Hola")
3  print("Adiós")
```

```
1  for i in range(3,7,2):
2      print("¿Qué pasa ahora?" + str(i))
3  print("Adiós")
```

Estructuras de control



Funciones

- Nos permiten reutilizar partes de código tantas veces como queramos y generalizar su estructura.
- Las funciones no necesariamente retornan un valor. Pueden no retornar y solo realizar cambios o actualizaciones en nuestro código.
- Los parámetros de entrada tampoco son obligatorios. Se pueden crear funciones que no reciban parámetros.

Funciones

Veamos como escribir una función:

Funciones

Veamos como escribir una función:

```
1  def nombre_funcion(x1, x2, ...):
2      # funcionamiento
3      return output
```

Funciones

Veamos como escribir una función:

```
1  def nombre_funcion(x1, x2, ...):
2      # funcionamiento
3      return output
```

Un ejemplo:

```
1  def suma(a,b):
2      c = a + b
3      return c
```

Funciones

Anteriormente manipulamos funciones sin saberlo:

Funciones

Anteriormente manipulamos funciones sin saberlo:

- int(), float(), str(), input()
- type(), print()

Resumen

Qué tenemos hasta ahora?

- Estructuras de control: if-else, while, for.
- Funciones: def, return.

Resumen

- if, else, elif: Control de flujo condicional.
- while: Control de flujo iterador.
- for: Control de flujo iterador con uso de range().
- range(n,m,k): genera un iterable desde n, hasta m-1 cada k.

Módulos

Qué es un módulo?

Módulos

Qué es un módulo?

- Un módulo en Python es un script (un archivo .py) que puede ser importado para el uso propio.
- Permite organizar lógicamente el código.
- Agrupar el código relacionado dentro de un módulo hace que sea mas fácil de entender y usar.
- Nos permiten incorporar lógicas y funcionalidades extra a nuestro código.
- Algunas vienen instaladas por defecto en Python y otras deberemos incorporarlas mediante una instalación.
- En la web, existe una infinidad de librerías que nos simplifican el trabajo a la hora de trabajar.

Módulos

- Para importar un módulo usamos la instrucción **import**.
- Para usar la función del módulo escribimos **modulo.función**

Módulos

- Para importar un módulo usamos la instrucción **import**.
- Para usar la función del módulo escribimos **modulo.función**

```
1  import misfunciones
2  x = misfunciones.ejemplo()
3  print(x)
```

Módulos

- Euclideana: $|x - y|$.
- Discreta: $f(x, y) = 1$ si $x \neq y$ y $f(x, y) = 0$ si $x = y$.
- Manhattan: $|x_1 - y_1| + |x_2 - y_2|$.
- Minskowski: $(|x_1 - y_1|^p + |x_2 - y_2|^p)^{1/p}$

Módulos

Probemos lo siguiente:

Módulos

Probemos lo siguiente:

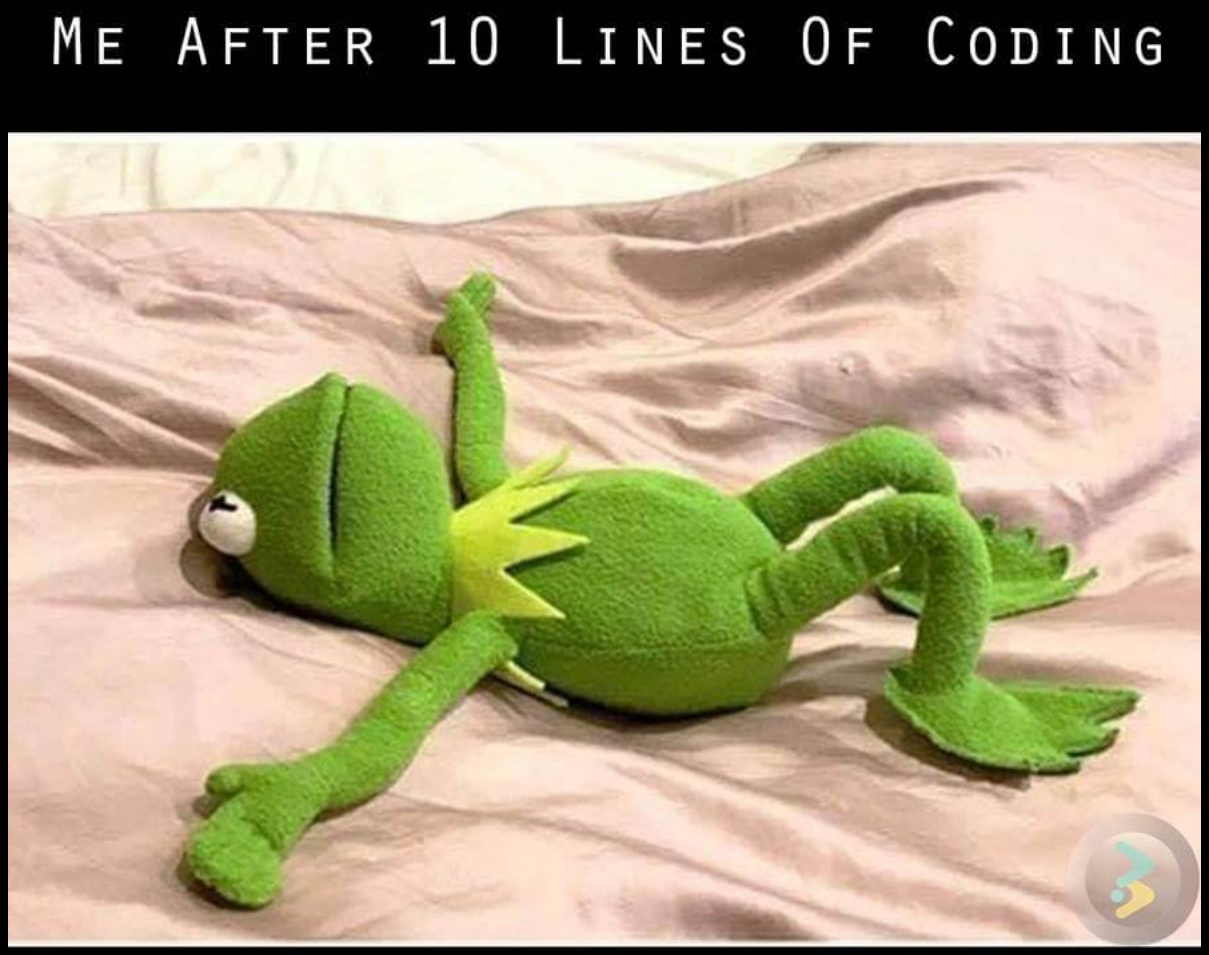
```
1  def eucl(x,y):
2      return abs(x-y)
3  def dis(x,y):
4      if x == y:
5          print(int(0))
6      else:
7          return 1
8  def mink(x1,y1,x2,y2,p):
9      a = abs(x1-y1)**p
10     b = abs(x2-y2)**p
11     return (a + b)**(1/p)
12 def man(x1,y1,x2,y2):
13     return mink(x1,y1,x2,y2,1)
```

Un paréntesis

pip es un sistema para administrar (e instalar) paquetes de Python. Pueden usarlo en sus Notebooks de Google Colar, y gran parte de las veces encontrará el paquete en la web.

- !pip install <nombre_paquete>
- !pip uninstall <nombre_paquete>

Vamos por más...



Vamos por más...

- Análisis exploratorio en Python
- Manipulación de DataFrames
- Preparación de datos parte 1: datos faltantes
- Unión y combinación