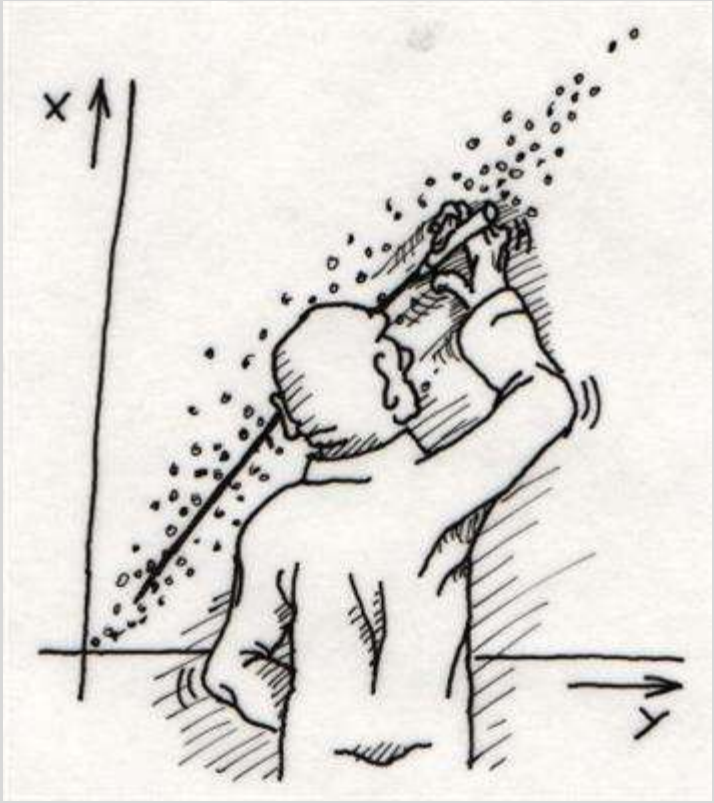


DigitalHouse > Coding School

DATA SCIENCE

UNIDAD 1
MÓDULO 1

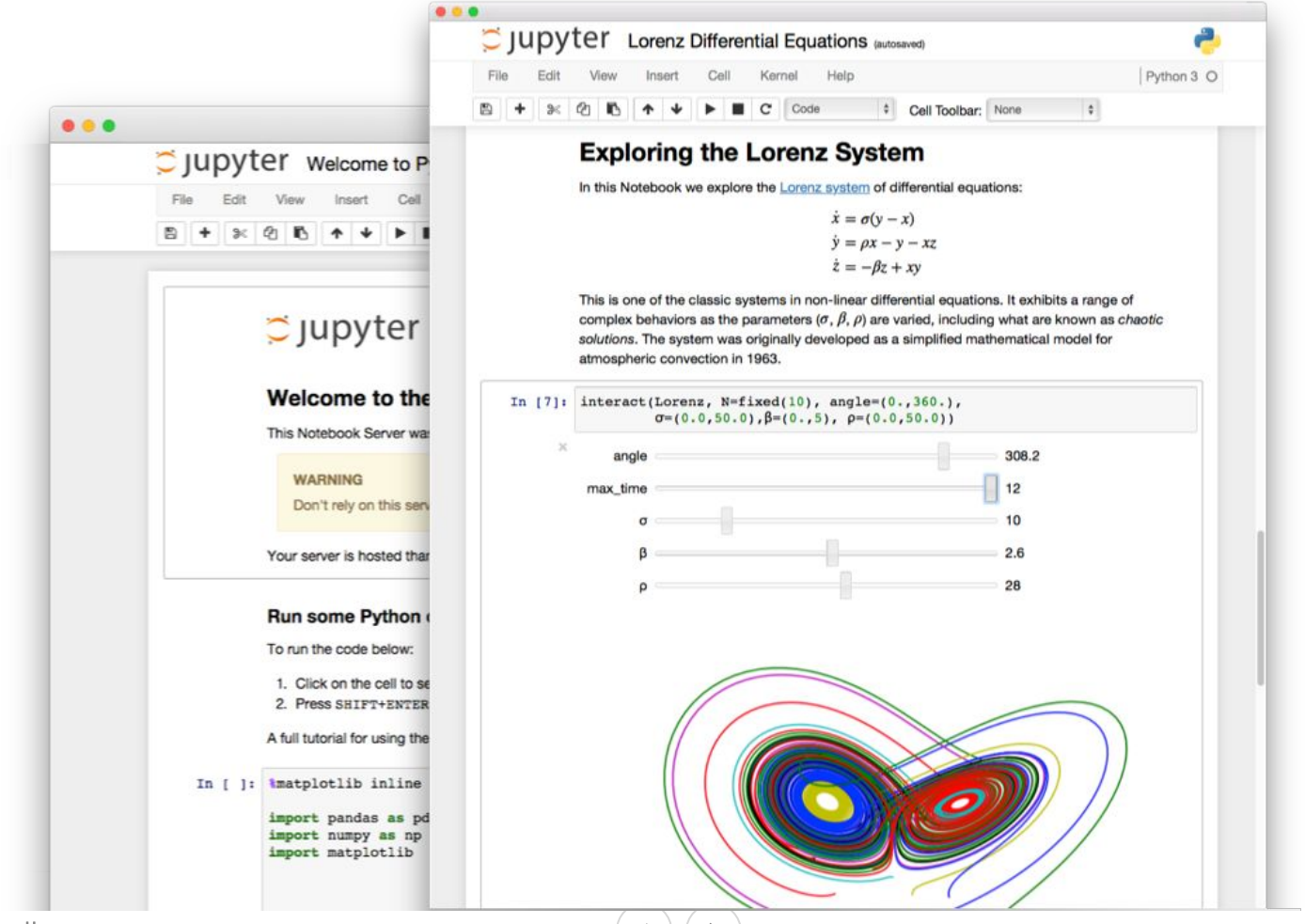
Repaso Python

TIEMPO	ACTIVIDAD	TÓPICO
5 min	Demo / Práctica	Jupiter-Notebooks
10 min	Demo / Práctica guiada	Operaciones aritméticas y con String
10 min	Demo / Práctica guiada	Tuplas
10 min	Demo / Práctica guiada	Listas
10 min	Demo / Práctica guiada	Diccionarios

TIEMPO	ACTIVIDAD	TÓPICO
10 min	Demo / Práctica guiada	if, if/else, if/elif/else
10 min	Demo / Práctica guiada	for
10 min	Demo / Práctica guiada	while
10 min	Demo / Práctica guiada	funciones

- Los participantes ya familiarizados con Python pueden utilizar la guía alternativa llamada ***Clase_1_Practica_Bonus_1.ipynb*** (disponible para descargar desde el campus) para refrescar nociones más avanzadas mientras el resto de la clase repasa los fundamentos:

TIEMPO	ACTIVIDAD	TÓPICO
5 min	Introducción	Desafío
70 min	Desafío de coding	Coding intenso
5 min	Conclusión	Conclusión del desafío


jupyter Welcome to P
File Edit View Insert Cell
jupyter
Welcome to the
This Notebook Server wa
WARNING
Don't rely on this serv
Your server is hosted thar
Run some Python
To run the code below:
1. Click on the cell to se
2. Press SHIFT+ENTER
A full tutorial for using the
In [ ]:
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib
jupyter Lorenz Differential Equations (autosaved)
File Edit View Insert Cell Kernel Help
Python 3
Code
Cell Toolbar: None
Exploring the Lorenz System
In this Notebook we explore the Lorenz system of differential equations:
$\dot{x} = \sigma(y - x)$
$\dot{y} = \rho x - y - xz$
$\dot{z} = -\beta z + xy$
This is one of the classic systems in non-linear differential equations. It exhibits a range of complex behaviors as the parameters (σ, β, ρ) are varied, including what are known as *chaotic solutions*. The system was originally developed as a simplified mathematical model for atmospheric convection in 1963.
In [7]:
interact(Lorenz, N=fixed(10), angle=(0.,360.),
σ=(0.0,50.0),β=(0.,5), ρ=(0.0,50.0))
angle 308.2
max_time 12
σ 10
β 2.6
ρ 28

- Operaciones aritméticas con diccionarios. Tipear:

```
> x = {'a':1, 'b':2}
  x['a'] + x['b']
```

- Es importante no olvidarse de usar las **comillas en las keys** salvo que se hayan usado enteros como keys. También se pueden sumar strings, tuplas y listas.

- **Pregunta:** ¿para qué se usa el operador %?

- También se pueden usar **variables** junto a **valores literales** en las **expresiones** para realizar operaciones aritméticas simples. En el notebook de iPython tipear

```
> x = 1
  y = 5
  x + y
```

- Usar elementos de una lista para realizar operaciones aritméticas. Tipear:

```
> x = [1, 2, 3]
  x[1] + x[2]
```

- También se pueden **sumar y multiplicar strings, tuplas y listas**.

Concatenar

- Sumar dos strings literales:

```
> "X Files" + " is awesome"
```

- devolverá:

```
> 'X Files is awesome'
```

- Con variables:

```
> x = "X Files"
  y = " is awesome"
  x + y
```

- devolverá:

```
> 'X Files is awesome'
```

- Con tuplas. En el notebook de iPython tipear:

```
> x = ('I', 'Love')
  y = ('True Detective Season 1',)
  print(x + y)
```

- y devolverá:

```
> ('I', 'Love', 'True Detective Season 1')
```

- Funciona de la misma forma con listas. Pero **no se pueden concatenar dos objetos de tipos diferentes**.

Indexing

- Los corchetes se usan para indexar, es decir, acceder a **un elemento** en un objeto compuesto
- Por ejemplo, con un string literal. En el notebook de iPython tipear:

```
> "I Love Spotify"[5]
```

Slicing

- El slicing se usa para acceder a un **rango de elementos** de un objeto. Tipear:

```
> x = "Spotify and Netflix are awesome"
  print(x[12:32])
```

- y devolverá

```
> Netflix are awesome
```

Tuplas

- Es una **secuencia fija e inmutable** de valores:

```
> x = ("Kirk", "Picard", "Spock")
```

- Así se crea una tupla de tres elementos. Se puede acceder a esos elementos de forma individual tipeando la variable y entre corchetes a la derecha el número de orden del elemento que se referencia. Ahora tipear:

```
> print(x[1])
```

- Recordemos que Python indexa los elementos en base 0. Es decir, comienza a contar el orden de los elementos desde el cero: el 1er elemento tiene orden 0, el segundo, orden 1 y así sucesivamente.

- También se puede acceder a los elementos en el **orden inverso**. Ahora tipear:

```
> print(x[-1])
```

- y devolverá:

```
> Spock
```

Listas

- Una lista es una **secuencia de datos mutable**. En el notebook de iPython tipear:

```
> x = ["Lord", "of", "the", "Rings"]
> x[2] = "Frodo"
> print(x)
```

- y devolverá:

```
> ['Lord', 'of', 'Frodo', 'Rings']
```

- El código anterior cambia el elemento 2 (o sea el tercero) en la lista **apuntada** por la **variable x**.

Creando listas

- Vamos a practicar creando listas. Para ello, usaremos el notebook **Clase_1_Code_along_listas_dicts.ipynb**

- Supongamos que tienen 5 amigos llamados Curly, Moe, Larry, Tweedle Dee y Tweedle Dumb. Crear una lista que se llama “friends” que contiene los 5 nombres. Asignarla a una variable que se llama *friends:*
  - friends = ['Curly', 'Moe', 'Larry', 'Tweedle Dee', 'Tweedle Dumb']
  - print(friends)

- **Las listas se indexan con enteros**. Imprimir el tercer ítem de la lista friends
  - print(friends[2])

- Se puede usar dos índices separados por dos puntos para **acceder a un rango** de una lista. Por ejemplo:
  - print(friends[2:5])

Modificando listas

- Las listas tienen métodos predefinidos que son muy útiles para manipularlas. Estos métodos funcionan “in place”, es decir, que no es necesario asignar el resultado a una variable nueva.

- El método **.append()** agrega un elemento al final de una lista.
- Agregar 'Sam' a la lista friends
  - friends.append('Sam')
  - print(friends)
  - ['Curly', 'Moe', 'Larry', 'Tweedle Dee', 'Tweedle Dumb', 'Sam’]

- También se pueden combinar listas simplemente concatenándolas
- Agregar la lista ['Bob', 'Joe'] a la lista friends
  - friends = friends + ['Bob', 'Joe']
  - print(friends)
  - ['Curly', 'Moe', 'Larry', 'Tweedle Dee', 'Tweedle Dumb', 'Sam', 'Bob', 'Joe']

- El método **.remove()** elimina un elemento específico de una lista y la función **del()** va a eliminar un item de una lista en una posición específica.

- Eliminar ‘Sam’ de la lista con **.remove()**
  - friends.remove(‘Sam’)
  - print(friends)
  - ['Curly', 'Moe', 'Larry', 'Tweedle Dee', 'Tweedle 9I8QW Dumb', 'Bob', 'Joe']

- Remover el primer elemento de la lista usando **del()**
  - del friends[0]
  - print(friends)
  - ['Moe', 'Larry', 'Tweedle Dee', 'Tweedle Dumb', 'Bob', 'Joe']

- Más información sobre listas

Diccionarios

- Consisten en pares de elementos que contienen una clave (key) y un valor. Las llaves **{ }** encierran diccionarios.

```
x = {'key1':'value1', 'key2':'value2'}
print(x)
```

- Pueden no aparecer en el orden exacto en el que fueron tipeados. Esto se debe a que **los diccionarios no tienen orden**. No puede usarse x[0] en un diccionario. Se indexa por la clave directamente.
- Modifiquemos el valor asociado a la primera clave:

```
x[‘key1’] = 'I love Python'
print(x)
```

- devolverá

```
{'key1': 'I love Python','key2': 'value2'}
```

- La clave se mantiene pero los valores son mutables. Las **claves son únicas** en un diccionario; **los valores no**.

- Ahora tipear:

```
> x = {'key':'value1', 'key':'value2'}
> print(x)
```

- y devolverá:

```
> {'key': 'value2'}
```

- El primer valor es sobreescrito por el segundo. Ahora tipear:

```
> x = {'key1':'value', 'key2':'value'}
> print(x)
```

- y devolverá:

```
> {'key2': 'value', 'key1': 'value'}
```

- Este ejemplo muestra que se pueden crear dos claves diferentes, con un mismo valor.

Creando y usando diccionarios

- Un aspecto potencialmente confuso de los diccionarios es que aunque **se crean con llaves { }**, para acceder a los elementos hay que **usar corchetes[ ]**, por ejemplo, my_dict['key'] devuelve el valor de 'key' en my_dict.

- Crear un diccionario "zipcodes" con los siguientes pares **distrito:codigo_postal**
  - 'sunset':94122, 'presidio':94129, 'soma':94105, 'marina':94123

- Crear el diccionario zipcodes:
  - zipcodes = {'sunset':94122, 'presidio':94129, 'soma':94105, 'marina':94123}
  - print(zipcodes)

- Como pueden ver, cuando se imprime un diccionario, los pares key:value **no tienen un orden predefinido** (a diferencia de una lista).

- Se puede acceder a los valores de una key determinada usando corchetes [ ] o con el método **.get()**.
- Acceder al código postal de soma y asignarlo a una variable

```
> soma = zipcodes['soma']
> print(soma)
```

y devolverá:

```
> 94105
```

- Se puede obtener una **lista de todas las keys** en un diccionario con el método **.keys()**. Devuelve una lista.
- Hacer un print una list de las claves en zipcodes usando **.keys()**

```
> zips = zipcodes.keys()
> print(zips)
```

y devolverá:

```
> ['soma', 'presidio', 'sunset', 'marina']
```

- El método **.items()** crea una **lista de tuplas**, en donde cada una es un par key:value del diccionario.

- Hacer un print de una lista de los pares key:value en zipcodes usando **.items()**

```
> elementos = zipcodes.items()
> print(elementos)
```

y devolverá:

```
> [('soma', 94105), ('presidio', 94129), ('castro', 94114), ('marina', 94123)]
```

Modificando diccionarios

- Para agregar un nuevo par key:value a un diccionario debe accederse a través de la key.
- Agregar **'castro':94114** a **zipcodes**
  - zipcodes['castro'] = 94114
  - print(zipcodes)
  - {'soma': 94105, 'presidio': 94129, 'sunset': 94122, 'castro': 94114, 'marina': 94123}

- Se puede remover un par key:value de un diccionario con la función **del()**, de la misma forma que una lista o con el método **.pop()** que elimina una key del diccionario y devuelve el valor para esa key.

- Eliminar el zipcode de 'sunset' con **.pop()** y asignarlo a una variable
  - sunset = zipcodes.pop('sunset')
  - print(sunset)
  - 94122

PRÁCTICA GUIADA: IF, IF/ELSE (10 min)

- En el notebook de iPython tipear. Analicemos este código:

```
weight = float(input("How many pounds does your suitcase weigh? "))
if weight > 50:
        print("There is a $25 charge for luggage that heavy.")
print("Thank you for your business.")
```

- Tipear:

```
43
```

-

- Hacerlo nuevamente, pero esta vez ingresar un número mayor a 50.

- La sintaxis general de un if-else es:

```
if condition:
    indentedStatementBlockForTrueCondition
else:
    indentedStatementBlockForFalseCondition
```

- Estos bloques pueden tener cualquier cantidad de sentencias y pueden incluir cualquier tipo de sentencia.

- En el notebook de iPython notebook tipear:

```
temperature = float(input('What is the temperature? '))
```

- Ingresar una temperatura.
- Luego tipear:

```
if temperature > 70:
    print('Wear shorts.')
else:
    print('Wear long pants.')
print('Get some exercise outside.')
```

```
if temperature > 70:
    print('Wear shorts.')
else:
    print('Wear long pants.')
```

- Estas cuatro líneas son una sentencia if-else.
- Hay dos bloques indentados:
  - uno viene luego del encabezado if y se ejecuta cuando la condición en ese encabezado es verdadera.
  - El otro es seguido por un else y sólo se ejecuta cuando la condición original es falsa.

- Probar ingresado diferentes números en la sintaxis < 70 or > 70.
- Más información sobre sentencias de control flow statements
- Check: ¿Cuál es la sintaxis general de una sentencia if? ¿Y de una if/else?

- En el notebook de iPython tipear:

```
x = int(raw_input("Please enter an integer: "))
```

- Ahora tipear un entero
- Y luego tipear:

```
> if x < 0:
     x = 0
     print 'Negative changed to zero'
  elif x == 0:
     print 'Zero'
  elif x == 1:
     print 'Single'
  else:
     print 'More'
```

- El **if**, cada uno de los **elif**, y el **else** final están alineados. Puede haber cualquier número de líneas elif, cada una seguida por un bloque indentado. Con esta construcción exactamente uno de los bloques indentados se ejecuta. Es el que se corresponde con la primera condición Verdadera.
- **Más sobre sentencias de control flow**
- Chequeo: ¿Cuántos bloques indentados en un if/elif/else son ejecutados?

- ```
  # Ejemplo básico de un for
  > for count in [1, 2, 3]:
      print(count)
      print('Yes' * count)
  ```

- Este es un **for** loop. Comienza con una sentencia **for** seguida por un nombre de variable (**count** en este caso), la palabra **in**, alguna secuencia y dos puntos. Al igual que en la definición de una función los dos puntos al final de la línea indican que sigue un bloque indentado de sentencias para completar el **for** loop.

- ¿Qué creen que va a devolver este código?
  ```

- Veamos un for loop a través de palabras:

```
> # usando un for loop para recorrer las letras en una palabra
> word = 'computer'
> for letter in word:
    print letter
```

- Usemos un for loop para imprimir una lista:

```
> shuttles = ['columbia', 'endeavor', 'challenger', 'discovery', 'atlantis', 'enterprise', 'pathfinder' ]
> for s in shuttles:
    print s
```

- Más información sobre for loops opción 1), opción 2), opción 3)

Creemos un contador simple:

```
# contador usando while loop
        count = 0
        while count < 5:
            print count
            count = count + 1
```

Problemos otro:

```
# otro contador
        count = 0
        while count < 5:
          count = count + 1

        print count
```

Uno más:

```
while True:
    reply = raw_input('Enter text, [type "stop" to quit]: ')
    print reply.lower()
    if reply == 'stop':
        break
```

Este while loop se detendrá cuando el usuario tipee "stop".

Recordar: un while loop corre hasta que la expresión es Falsa. El problema es que, a veces, no paran. Para evitar esto, es conveniente recordar estas reglas:

1. Asegurarse de usar **while** loops con precaución.
2. Revisar las sentencias **while** y asegurarse de que el test (booleano) se transformará en False en algún momento.
3. Cuando haya dudas, hacer un print de la variable test al principio y al final del **while** loop para ver qué está haciendo.

Más información sobre while loops

Ahora, traten de crear algunos while loops por su cuenta.

- La sintaxis para una función:

```
def functionname(parameters):
    "function_docstring"
    function_sentences
    return expression
```

- Creemos una función llamada **printme**:

```
def printme(str):
    "This prints a passed string into this function"
    print str
    return
```

- Ahora, llamemos a la función:

```
printme("Beltway traffic makes my head hurt.")
printme("It looks like a parking lot out there!")
```

- **Más sobre defición de funciones**
- **Chequeo:** ¿Para qué podrían ser útiles las funciones?