

# Introducción a Python

**Herramientas de Procesamiento para Grandes  
Volúmenes de Datos**

Juan Martín Pampliega

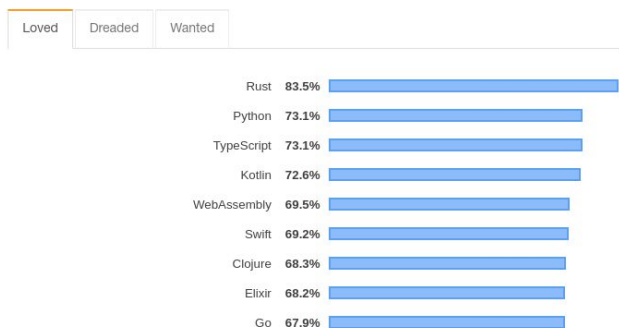
# Agenda

- Qué es Python
- Uso del lenguaje:
  - Tipos básicos: números, strings
  - Colecciones: list, dict, tuple, set
  - Variables
  - Estructuras de control
  - Funciones
  - Clases y Objetos
  - Módulos y Paquetes
  - Excepciones
  - Manejo de Archivos

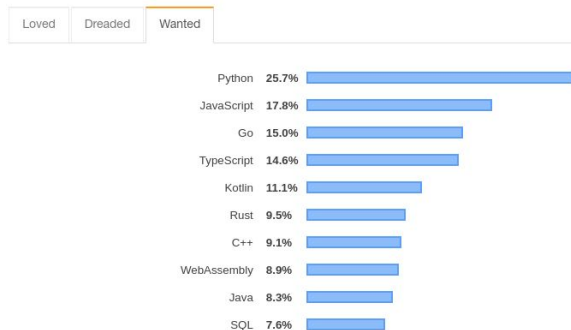
# ¿Por qué Python?

- Es un lenguaje dinámico de alto nivel ampliamente utilizado.
- Permite hacer desarrollos o exploración de datos de manera rápida y sencilla.
- Cuenta con una enorme variedad de librerías de terceros que aportan funcionalidades para casi cualquier tarea.

Most Loved, Dreaded, and Wanted Languages



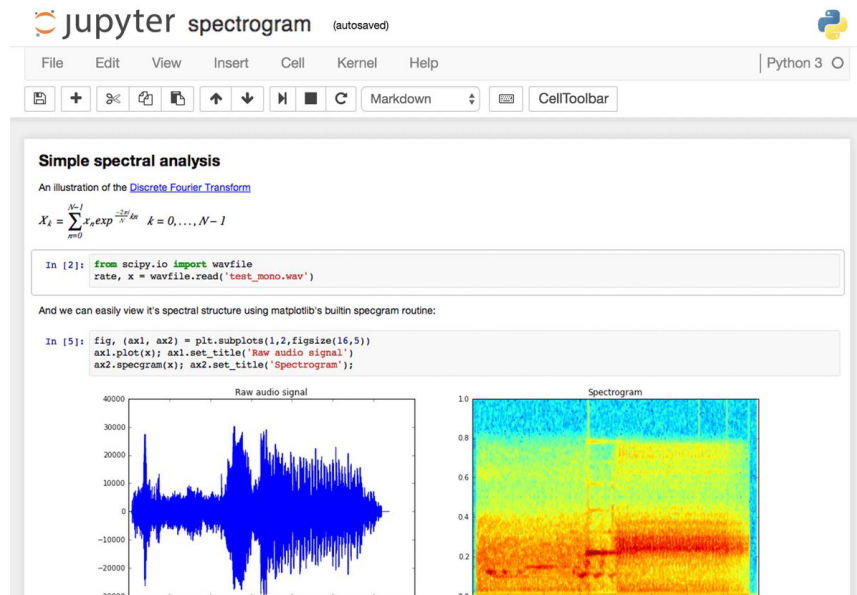
Most Loved, Dreaded, and Wanted Languages



# ¿Qué es Jupyter Notebook?

Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen código fuente, ecuaciones, visualizaciones y texto explicativo.

Entre sus usos está la limpieza y transformación de datos, la simulación numérica, el modelado estadístico, el aprendizaje automático y mucho más.



# Instalación Local de Python y Jupyter

En Windows se sugiere utilizar el entorno de conda:

<https://anaconda.org/anaconda/python>

En Linux el intérprete del lenguaje viene instalado por default.

Para correr Jupyter correr este comando: "jupyter-notebook"

Luego abrir el navegador en "http://localhost:8888/"

Nota: podemos evitar estos pasos usando collaboratory (ver más adelante)

# Instalación de Paquetes

En Windows se utilizará el gestor de paquetes de conda.

En linux: `pip3 install jupyter`

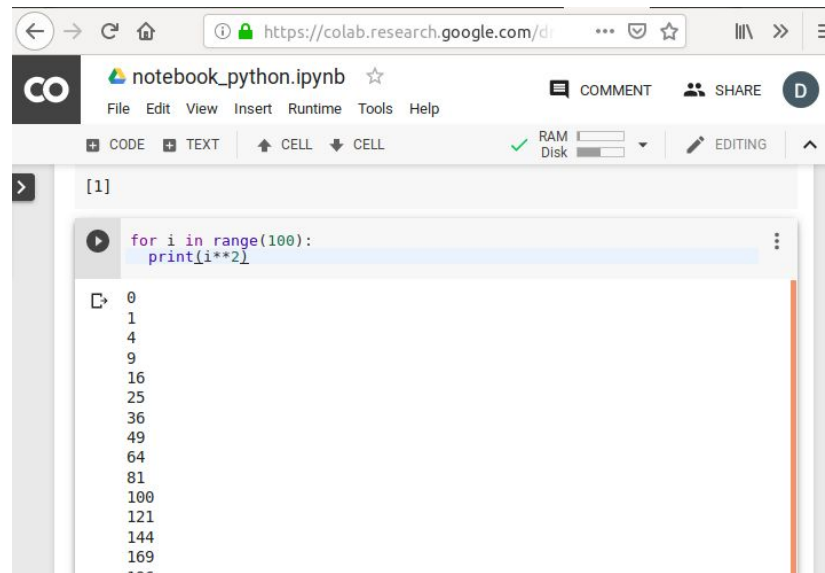
# ¿Que es collaboratory?

Es un servicio de google al que podemos acceder desde un navegador y que ejecuta las notebooks en la nube (es decir sin necesidad de hacer una instalación local).

Se puede acceder desde <https://colab.research.google.com/>

O desde el google drive:

nuevo> más >conectar aplicaciones



The screenshot displays the Google Colaboratory web interface. At the top, the browser address bar shows the URL <https://colab.research.google.com/>. Below the browser, the Colab logo is visible on the left, and the notebook title "notebook\_python.ipynb" is centered. To the right of the title are buttons for "COMMENT" and "SHARE", along with a user profile icon labeled "D". A menu bar below the title includes "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". Below the menu bar, there are tabs for "CODE" and "TEXT", and buttons for "CELL" and "DOWN CELL". On the right side of this bar, there is a status indicator for "RAM Disk" with a green checkmark and a progress bar, and an "EDITING" button. The main area of the interface shows a code cell labeled "[1]" containing a Python loop: 

```
for i in range(100):  
    print(i*2)
```

. Below the code cell, the output is displayed as a list of even numbers from 0 to 198, with a scroll bar on the right side.

# Características generales

A diferencia de otros lenguajes en Python los espacios (la indentación) son relevantes para marcar la estructura del programa.

```
def factorial(x):  
    if x == 0:  
        return 1  
    else:  
        return x * factorial(x - 1)  
  
print(factorial (10))
```



# Variables

Tipos básicos soportados: bool, int, float, string

Se definen de la siguiente manera al asignarles un valor y se puede averiguar su tipo mediante "type".

```
a = 1
b = "Hola"
print(type(a)) # int
print(type(b)) # str
```

Es una buena práctica que los nombres de las variables sigan ciertos criterios:

- Estén solo en minúsculas
- Usen guiones bajos para separar palabras: received\_data
- No empiecen con números

# Comentarios

Los comentarios se una sola línea se definen anteponiendo “#” al texto del mismo.

Otra forma de comentar es usando string con entre comillas triples “"""”. Esto es muy útil para comentarios largos y es la forma recomendada de documentar funciones, métodos y clases.

```
a = 1      # Esto es un comentario
"""
Esto también se toma como un
comentario
que ocupa varias lineas.
"""
b = "Hola"
```

# Números

Tipos básicos: int y float

Admite las operaciones básicas de

- Suma, resta, multiplicación, división. +, -, \*, /
- División entera, módulo, valor absoluto: //, %, abs
- Exponenciación: \*\*
- Shift y operaciones lógicas de bits (and, or, not, xor): <<, >>, &, |, ~, ^
- Conversión entre tipos: int(), float()

```
a = 12
b = 3.14
c = 0xFF

c = a + b / c
print(type(float(a))) # float
print(type(int(b))) # int
```

# Strings

Las cadenas de caracteres se pueden definir usando comillas simple (') o dobles (").

```
"hello"+"world"      # "helloworld"      # concatenacion
"hello" * 3           # "hellohellohello"  # repeticion
"hello"[0]            # "h"                # indexacion
"hello"[-1]           # "o"                # indexacion desde atrás
"hello"[1:4]          # "ello"             # slicing
len("hello")          # 5                  # tamaño
"hello" < "jello"      # True               # comparacion
"e" in "hello"        # True               # busqueda
```

# Strings

Formateo de strings:

```
"hola %s" % "mundo"
```

```
"hola {texto}".format(texto="mundo")
```

```
texto = "mundo"
```

```
f"hola {texto}"
```

# Condiciones

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
>True and False
False
>True or False
True
>not False
True
```

```
a = 1
b = 2
if a > b:
    print("a es mas grande que b")
elif a < b:
    print("b es mas grande que a")
else:
    print("a y b son iguales")
```

```
a = 2
if a**a == a+a and a > 0:
    print("el cuadrado de a es igual su
suma y mayor que 0")
```

# Iteraciones

```
while condition:  
    statements  
    break # Salir del loop
```

```
for var in sequence:  
    statements  
    if condition:  
        # Saltar al siguiente del loop  
        continue
```

```
for i in range(20):  
    if i % 3 == 0:  
        print(i)  
        if i%5 == 0:  
            print("Bingo!")  
            break  
    print("---")
```

# None

Un tipo particular de valor es “None”. Es el equivalente a `null` en otros lenguajes como Java.

```
a = None
if a is None:
    print("a esta indefinido")
```



# Listas

Son arrays de tamaño variable que admiten una variedad de operaciones incluidas varias de propias de las strings (index, slicing, repetición, tamaño)

```
a = range(5)           # [0,1,2,3,4]
a.append(5)             # [0,1,2,3,4,5]   Agregar elemento al final
r = a.pop()             # [0,1,2,3,4]     Remueve el último elemento
print(r)                # 5
a.insert(2, 42)          # [0,1,42,2,3,4]   Inserta el número 42 en la posición 2
a.reverse()              # [4,3,2,42,1,0]    Invertir orden
a.sort()                 # [0,1,2,3,4,42]    Ordenar
a.extend([1,2,3])        # [0,1,2,3,4,42,1,2,3]  Agregar lista de elementos

[i*2 for i in range(5)] # [0,2,4,6,8]      Definición por comprensión
```

# Diccionarios

Sirven para mapear claves con elementos particulares de cualquier tipo.  
Un mismo diccionario puede contener valores de varios tipos.  
Tener en cuenta que el diccionario no preserva el orden de los elementos.

```
d = {"a":1, "b":2}

d["a"]           # 1
d["v"]           # Produce una excepción KeyError
d.has_key("v")   # Ver si la clave 'v' está en el dict
"v" in d         # Idem anterior

d.keys()          # ['a', 'b']           Obtener las claves
d.values()        # [1,2]              Obtener las valores
d.items()         # [('a', 1), ('b', 2)] Obtener las tuplas de claves y valores

del d["a"]        # {"b":2}             Eliminar un elemento
d["c"] = 3        # {"b":2, "c":3}      Agregar un elemento
```

# Diccionarios

Otras formas de definir diccionarios:

```
d = dict(zip("abcd",range(4))) # {'a': 0, 'c': 2, 'b': 1, 'd': 3}
```

```
d = {chr(c): c for c in range(65,70) }
```

```
# {'A': 65, 'C': 67, 'B': 66, 'E': 69, 'D': 68}
```

# Tuplas

Las tuplas son variables inmutables que aceptan un número fijo de elementos.

```
key = (nombre, apellido)
point = x, y, z           # El paréntesis es opcional
x, y, z = point           # Unpack
lastname = key[0]         # Acceso a valores particulares,
                           aceptan slicing
singleton = (1,)         # Tupla de un solo elemento
```

# Sets

Son colecciones de elementos sin repeticiones.

```
a_set = {1, 1, 2}      # {1, 2}
b = [1, 2, 3, 2]
a_set = set(b)         # {1, 2, 3} Set a partir de una lista.
a_set.add(4)           # {1, 2, 3, 4}
a_set.add(2)           # {1, 2, 3, 4}
a_set.update({2, 4, 6}) # {1, 2, 3, 4, 6}
a_set.discard(6)       # {1, 2, 3, 4}
a_set.discard(10)      # {1, 2, 3, 4}
a_set.remove(10)       # KeyError

a_set = {1, 2, 3, 4, 5}
5 in a_set             # True
10 in a_set            # False
b_set = {4, 5, 6, 7, 8}
a_set.union(b_set)     # {1, 2, 3, 4, 5, 6, 7, 8}
a_set.intersection(b_set) # {4, 5}
a_set.difference(b_set)  # {1, 2, 3}
a_set.symmetric_difference(b_set) # {1, 2, 3, 6, 7, 8}
```

# Referencias a Variables

Al hacer una asignación se pasa una referencia en lugar del valor de la variable (a menos que se trate de un int o un float).

Es importante tener en cuenta esto para evitar resultados inesperados:

```
a = [1, 2, 3]
b = a
a.append(4)
print(b)          # [1, 2, 3, 4]
```

# Funciones

Como en cualquier lenguaje permiten agrupar segmentos de código para ser llamados luego.

```
def nombre(arg1, arg2, ...):
```

```
    """documentation"""
```

```
    statements
```

```
    return expression
```

Documentación de la función

Retornar valor si se omite  
"expression" se retorna None

# Funciones

Los argumentos de la función pueden tener valores predefinidos y llevar un nombre.

Definición

```
def sumar_clave(d, v=1, key="a"):
    return d[key] + v
```

Uso

```
d = {"a":10, "b": 20}
sumar_clave(d)
# 11
sumar_clave(d, v=2)
# 12
sumar_clave(d, key="b")
# 21
sumar_clave(d, key="b", v=2)
# 22
sumar_clave(*[d], **{"key":"b", "v":2})
# 22
```



# Funciones

También es posible recibir un número variable de argumentos.

```
def sumar_clave(d, key="a", *args, **kwargs):  
    # En este punto vamos a recibir un objeto list en args  
    # y un dict en kwargs.  
    # Notar que es posible usar ambos tipos de argumento a la vez.  
    print(d, key,args,kwargs)  
  
sumar_clave({"a":10}, key="a", v=10)  
# {'a': 10} a () {'v': 10}
```

# Lambdas

Las lambda son formas abreviadas (anónimas) de definir funciones.

```
>v = lambda a, b: a + b  
>v(1,2)  
3
```

# Funciones para Transformar Listas

```
lista = range(5)
# [0, 1, 2, 3, 4]

map(lambda x: x * 2, lista)
# [0, 2, 4, 6, 8]

reduce(lambda a,b: a + b, lista)
# 10

filter(lambda x: x >= 3, lista)
# [3, 4]
```

# Manejo de Archivos

Para esto se realiza una llamada a "file".

- Abrir: `f = open(filename, mode)`
  - El modo puede ser "r", "w", "a" (leer, escribir, agregar).
    - Por default es "r"
    - Agregar "b" para lecturas binarias
- Métodos:
  - Escribir datos y cerrar: `flush()`, `close()`
  - Leer: `read()`, `readline()`, `readlines()`
  - Escribir: `write(string)`, `writelines(list)`
  - Ir a una posición particular: `seek(pos)`

# Excepciones

Cuando ocurre un error se dispara una excepción. Por default esto termina la ejecución del programa pero también es posible capturarlas y decidir qué hacer.

```
def foo(x):  
    return 1/x  
  
def bar(x):  
    try:  
        print(foo(x))  
        print('aaa')  
    except ZeroDivisionError, message:  
        print("No se puede dividir por cero:", message)  
  
bar(0)
```

Agregando distintas cláusulas “except” es posible capturar distintos tipos de excepciones.

# Excepciones

La estructura “try” para manejo de errores es particularmente útil para hacer limpieza en caso que ocurra una error.

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()    # Esto siempre se ejecuta haya o no un error
print("OK")      # Si ocurre una excepción esta parte de no se ejecuta
```

También es posible disparar excepciones usando “raise”.

```
raise IndexError("Un error")
```

# Clases y Objetos

- **Clase.**

Es donde se define el comportamiento y las propiedades de un objeto a ser creado.

- **Atributo**

Es una propiedad del objeto, en el caso de python una variable asociada al objeto

- **Método.**

Son acciones que puede realizar el objeto, o sea funciones.

- **Objeto**

Es la instancia de la clase.

# Clases y Objetos

- En Python todo es un objeto!
  - `d` es una instancia (objeto) creada a partir de la definición dada por la “clase diccionario”.
  - `keys()` es un método (función) de la clase.
  - `__class__` es un atributo

```
> d = { 'a': 1 }  
> d.keys()  
dict_keys()  
  
> d.__class__()  
<class 'dict'>
```



# Classes

```
class Human:
    """Friendly human with a name."""


    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hi! My name is {self.name}")
```

# Classes

Nombre de  
la clase

Clases padre (herencia)



```
class Human():  
    """Friendly human with a name."""  
  
    def __init__(self, name):  
        self.name = name  
  
    def greet(self):  
        print(f"Hi! My name is {self.name}")
```

# Classes

```
class Human:  
    """Friendly human with a name."""
```

Métodos  
de la clase

```
    def __init__(self, name):  
        self.name = name  
  
    def greet(self):  
        print(f"Hi! My name is {self.name}")
```

# Classes

```
class Human:
    """Friendly human with a name."""

    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hi! My name is {self.name}")
```

Es el primer argumento que reciben los métodos, es obligatorio para los métodos declararlo.

Permite acceder a los métodos y atributos de la clase misma.

Por convención se llama self.

# Clases

```
class Human:
    """Friendly human with a name."""

    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hi! My name is {self.name}")
```

Es el primer método que se ejecuta tras la creación de una instancia de la clase.

Recibe los argumentos externos cuando se realiza la instanciación.

Los métodos tipo `__**__` suelen ser métodos especiales que dan cierta funcionalidad a los objetos

# Classes

```
class Human:
    """Friendly human with a name."""

    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hi! My name is {self.name}")
```

Es un atributo de la instancia clase

# Módulos

A medida que un proyecto va creciendo es conveniente agrupar funcionalidades similar en un mismo archivo.

Llamemoslo “foo.py”:

```
def calcular_potencia(n,p):  
    return n**p
```

En un archivo separado “main.py” podemos hacer esto entonces:

```
import foo  
foo.calcular_potencia(2, 3) # 8  
  
# Otra forma  
from foo import calcular_potencia  
calcular_potencia(2, 3) # 8
```

# Paquetes

Son colecciones de módulos en un directorio

Debe contener un archivo `__init__.py`.

Puede contener subpaquetes

Sintaxis del import:

- `from P.Q.M import foo`  
`print foo()`

- `from P.Q import M`  
`print M.foo()`

- `import P.Q.M`  
`print P.Q.M.foo()`

- `import P.Q.M as M`  
`print M.foo()`



# Librerías estándar más utilizadas

Core: os, sys, string, argparse, StringIO, struct, pickle, ...

Expresiones regulares: re

Redes: socket, httpplib, htmlplib, ftplib, smtplib, ...

Debugging y Profiling: pdb (debugger), profile

# Referencias

- [https://python.quantecon.org/python\\_essentials.html](https://python.quantecon.org/python_essentials.html)
- <https://jakevdp.github.io/WhirlwindTourOfPython/>
- <https://evanli.github.io/programming-book-3/Python/Fluent%20Python.pdf>  
(avanzado)