

# Python para Análisis de datos: Introducción

## Sesión 1

Jesús Fernández (fernandez.cuesta@gmail.com)

1 Abril 2019

Introducción a Python

Python es usado en

Python como lenguaje de programación

¿?

Instalación y entorno

Entornos virtuales

Conda

Instalación

Práctica I

Práctica II

vscode

## Introducción a Python



Creado en 1990 por Guido van Rossum



Figure 1: Guido van Rossum, 1999

Definición y **evolución** del lenguaje recogido en PEPs (**Python Enhancement Proposals**)

- ▶ Énfasis en la productividad y legibilidad del código (PEP20)
  - ▶ *Beautiful is better than ugly*
  - ▶ *Explicit is better than implicit*
  - ▶ *Simple is better than complex*
  - ▶ *Complex is better than complicated*
  - ▶ *Readability counts*
  - ▶ ...
- ▶ +reusable, +fácil mantener

## Ejemplo:

```
import math

número = input("Introduce un número [0, 1, 2, ...]: ")
número = int(número)  # convierte texto a número entero

print('El factorial de', número, 'es', math.factorial(número))

# de forma alternativa:

# print('El factorial de %s es %s' % (número, math.factorial(número)))
# print('El factorial de {} es {}'.format(número, math.factorial(número)))
# print(f'El factorial de {número} es {math.factorial(número)}')
```

- ▶ Actualmente dos versiones principales activas:
  - ▶ Python 2.7 (soporte hasta 1/1/2020)
  - ▶ **Python 3.x (3.6, 3.7)** ←

python x.y.z

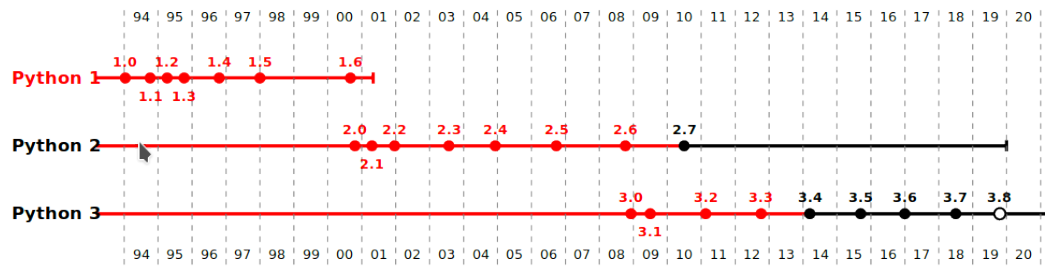
x: versión principal, incompatibles entre sí [2, 3]

y: versión secundaria, normalmente compatibles

z: versión menor (errores y seguridad)

\$ python -V

Python 3.7.3



Línea temporal de versiones, en rojo: versión obsoleta

- ▶ Lenguaje de alto nivel, interpretado, orientado a objetos
- ▶ Alta productividad, no compilado
- ▶ Gran cantidad de librerías incorporadas (*batteries included*)
- ▶ + librerías externas (>1M): Python Package Index (PyPI)

## Alto nivel

- ▶ Sencillo y comprensible
- ▶ Fácil de aprender
- ▶ Abstracción de datos (no es necesario declarar variables)
- ▶ Menos líneas de código
- ▶ Interfaces simples (*pythonic*)

```
>>> import requests
>>> r = requests.get('https://api.github.com/events')
>>> r.text
u' [{"repository": {"open_issues": 0, "url": "https://github.com/...
```

## Código ejemplo en C++

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <err.h>

char response[] = "HTTP/1.1 200 OK\r\n"
"Content-Type: text/html; charset=UTF-8\r\n\r\n"
"Hello, world!\r\n";

int main()
{
    int one = 1, client_fd;
    struct sockaddr_in svr_addr, cli_addr;
    socklen_t sin_len = sizeof(cli_addr);

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
        err(1, "can't open socket");

    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(int));

    int port = 8080;
```

## Equivalente python

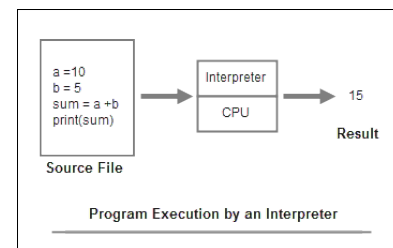
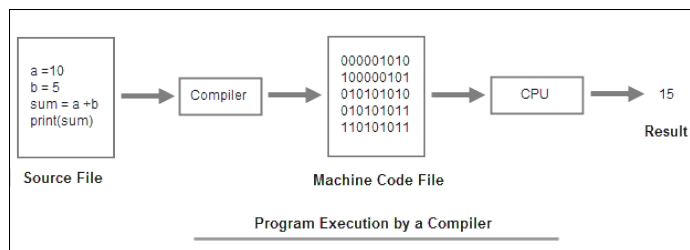
```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

Flask.run(app, port=8080)
```

## Interpretado

► No es necesario compilar. . .



... pero necesitamos tener instalado un *intérprete* `python.exe`

## Interpretado

- ▶ Compatible (en general) entre distintos sistemas operativos y arquitecturas:
  - ▶ Linux
  - ▶ MacOS
  - ▶ Windows
  - ▶ otros (AIX, AS/400, z/OS, OpenVMS, ARM, ...)
  - ▶ **Arduino/Raspberry PI (IoT)**
- ▶ Depuración de errores desde intérprete
- ▶ Gestión automática de memoria

## Práctica

### intérprete python

- ▶ Anaconda Prompt > python (> ipython)

```
(base) C:\Users\IEUser>ipython
Python 3.7.0 (default, Aug 14 2018, 19:12:50) [MSC v.1900 32 bit (Intel)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import datetime

In [2]: ahora = datetime.datetime.now()

In [3]: print(ahora)
2019-03-27 17:52:37.499641

In [4]: print(ahora.date())
2019-03-27

In [5]: print(ahora.time())
17:52:37.499641

In [6]: print(ahora + datetime.timedelta(days=-30))
2019-02-25 17:52:37.499641

In [7]:
```



## Diferentes paradigmas de programación

- ▶ Orientado a objetos
- ▶ Procedural
- ▶ Imperativo
- ▶ Funcional (<100%)

Python es usado en

Data Science



Machine Learning



## Desarrollo Web

(p.e. pinterest, instagram, linkedin, ...)



## Desarrollo software (scripts, prototipos, ...)

como “pegamento” entre componentes escritos en otros lenguajes

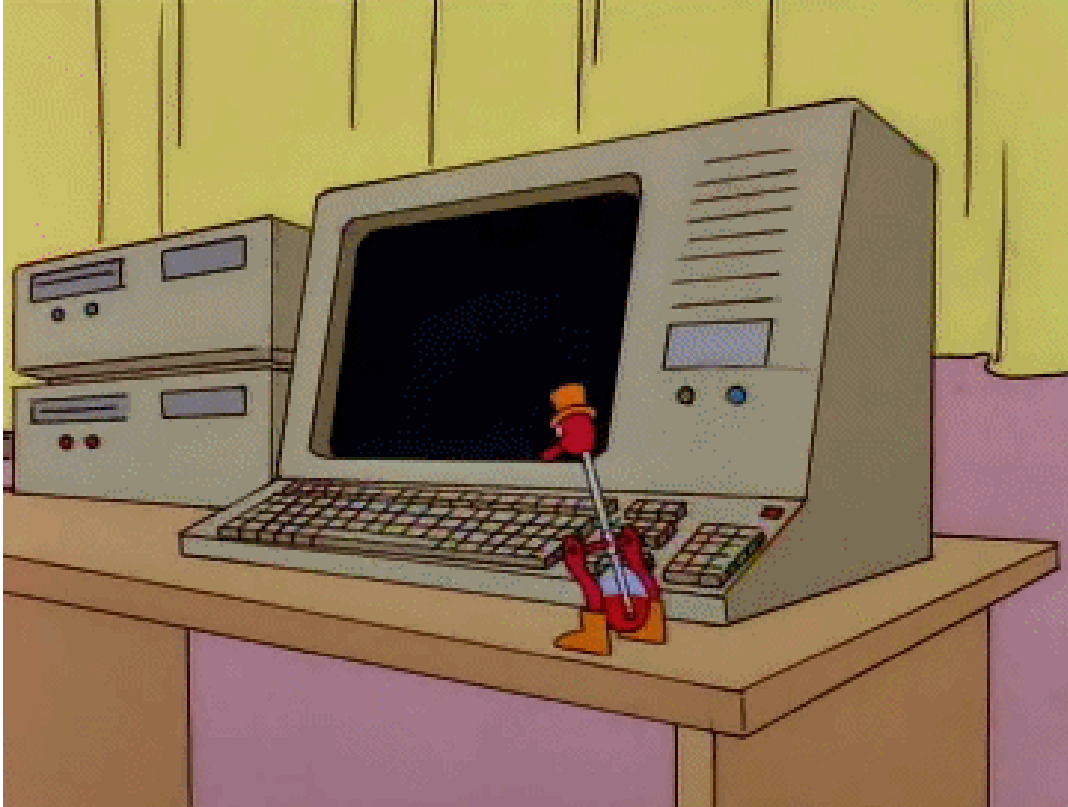


Fabric



Celery sq

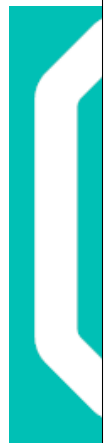
Automatización de procesos software



Automatización de procesos software

























ANSIBLE



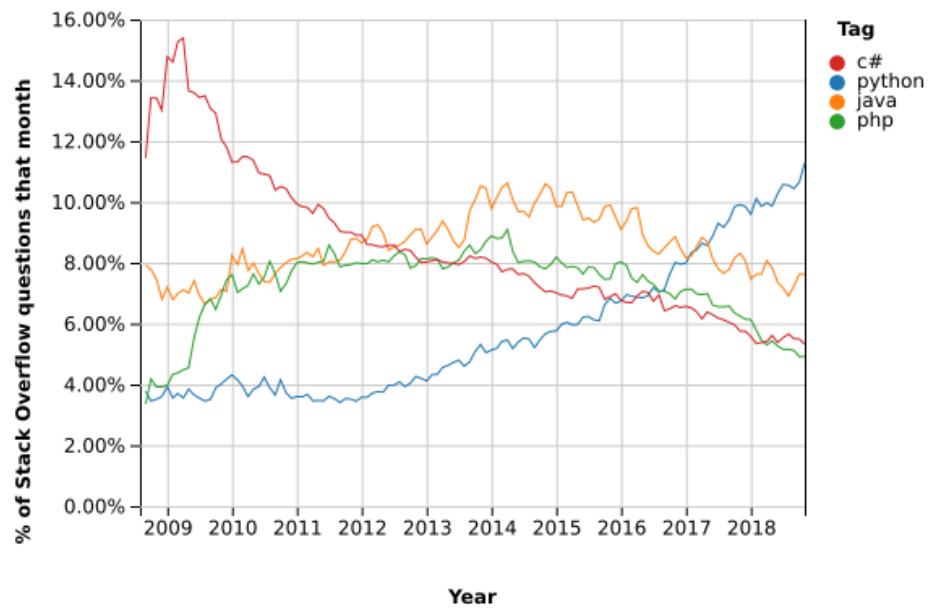
## Python como lenguaje de programación

The 2018 Top Programming Languages, IEEE

Language Rank	Types	Spectrum Ranking
1. Python	  	100.0
2. C++	  	99.7
3. Java	  	97.5
4. C	  	96.7
5. C#	  	89.4
6. PHP		84.9
7. R		82.9
8. JavaScript	 	82.6
9. Go	 	76.4
10. Assembly		74.1

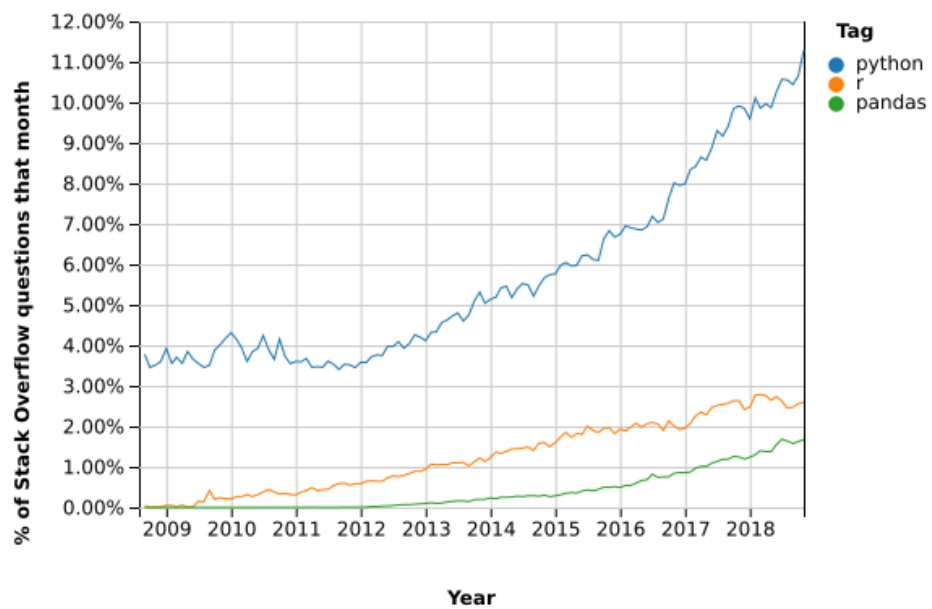
gran soporte en foros, comunidades, conferencias, ...

stackoverflow



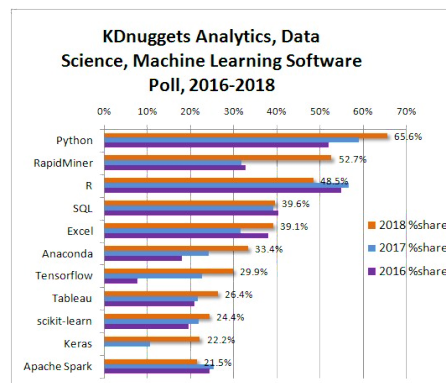
*Worldwide, Python is the most popular language, Python grew the most in the last 5 years (PYPL)*





fuelle: stackoverflow

- ▶ R: muy enfocado en análisis estadístico
- ▶ Python: generalista, con librerías especializadas (pandas, scikit-learn, scipy, ... )
- ▶ Encuesta 2017
- ▶ Encuesta 2018
- ▶ Python Data Science



i?

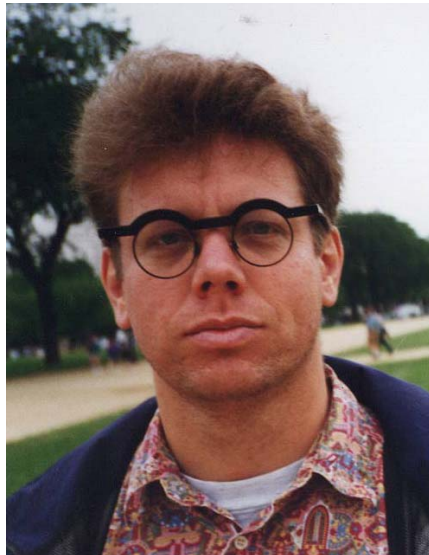


Figure 2: Guido van Rossum, 1995



## Instalación y entorno

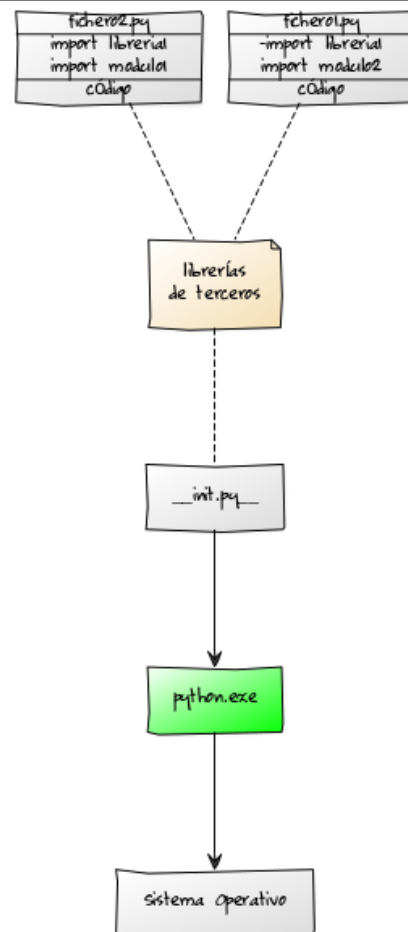
Diferentes distribuciones para Windows:

- ▶ python (cpython)
- ▶ **conda** (anaconda/miniconda)
- ▶ canopy
- ▶ winpython
  - ▶ enfocado a sistemas Windows
  - ▶ no necesita ser instalado
  - ▶ incluye compilador C/C++
- ▶ activepython (comercial)

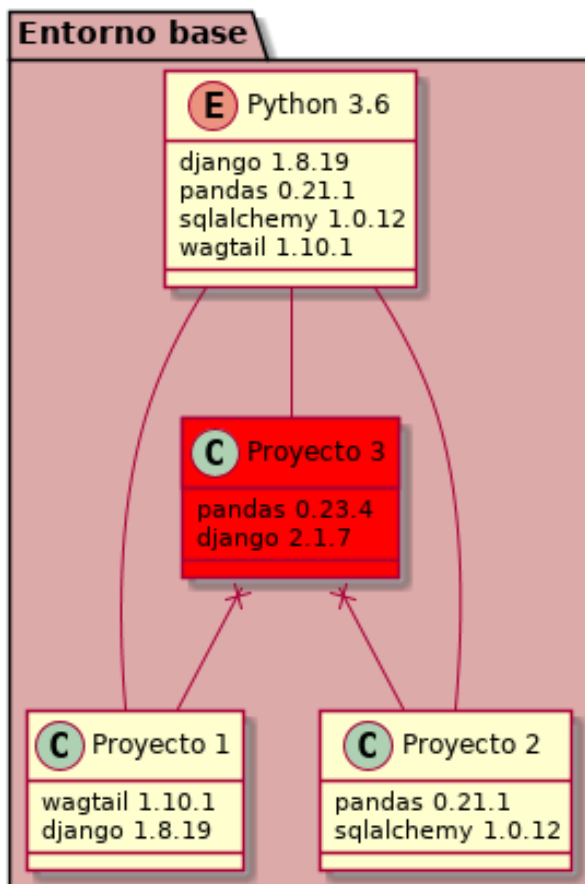
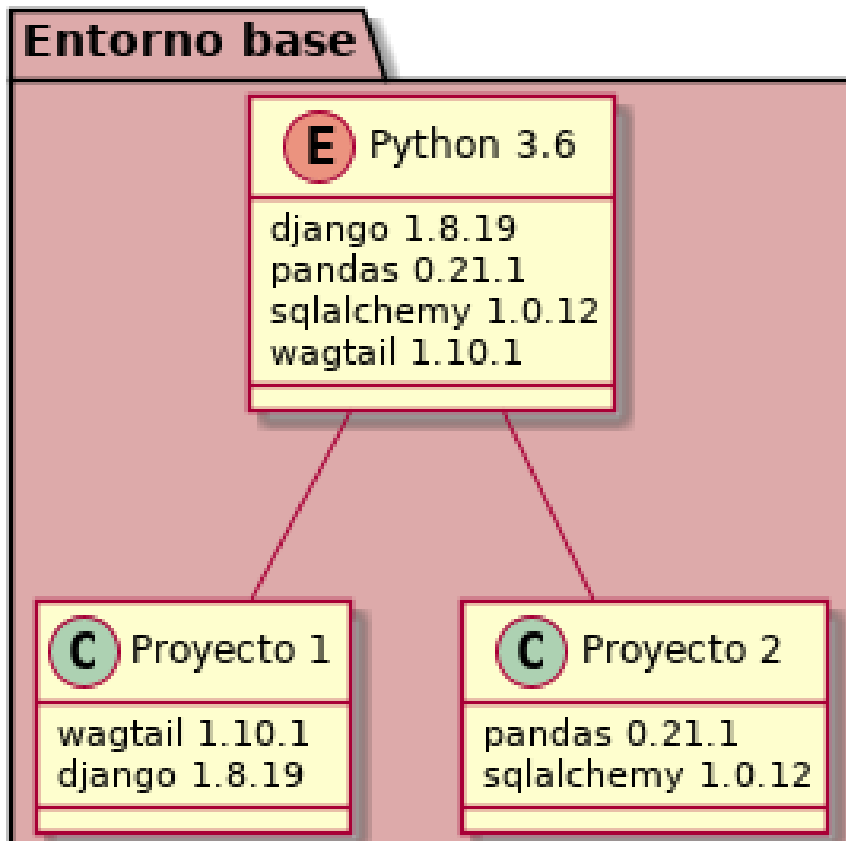
### Podemos ejecutar código **python**:

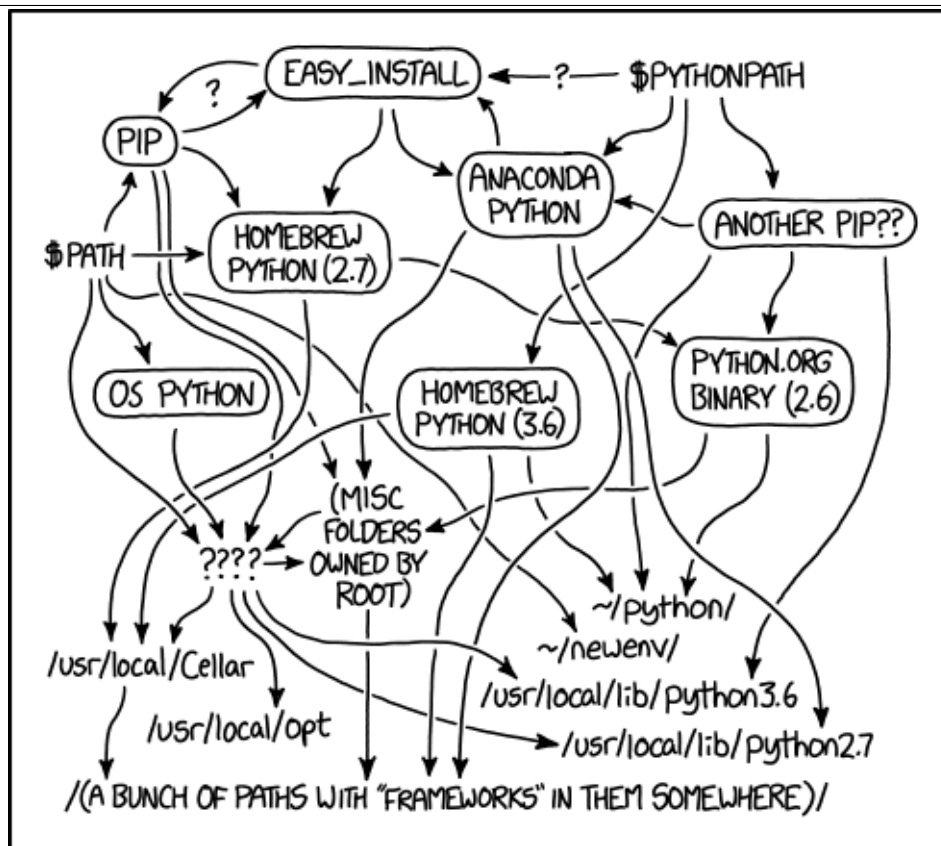
- ▶ Directamente (si el S.O. lo permite, #!)
- ▶ Desde un intérprete interactivo python (p.e. ipython, bpython)
- ▶ Desde un cuaderno jupyter

- ▶ Para ejecutar código python necesitaremos un **intérprete** (python.exe).
- ▶ Preinstalado en ciertos S.O. (p.e. Linux, Mac OS X)
- ▶ Diferentes tipos de intérprete (CPython, PyPy, Jython, IronPython...)
- ▶ Podemos tener "n" intérpretes distintos instalados en el sistema, cada uno con diferentes librerías
- ▶ conda: instala por defecto un entorno (intérprete) base y un conjunto de librerías



Cada entorno tiene un único intérprete python + librerías



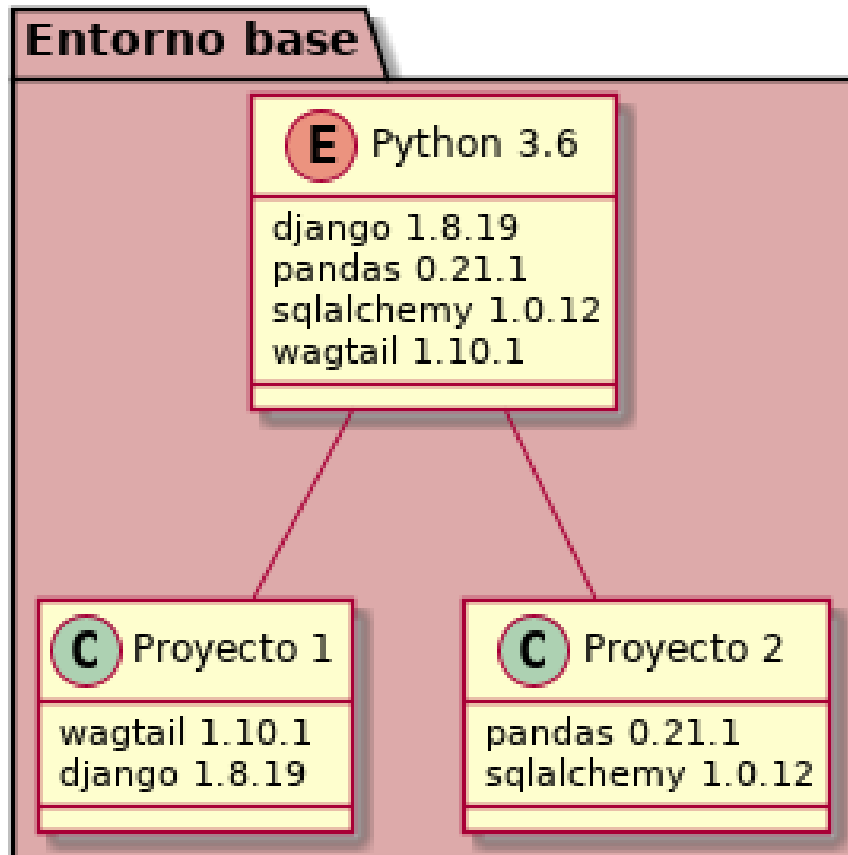


¿Solución?

## Entornos virtuales

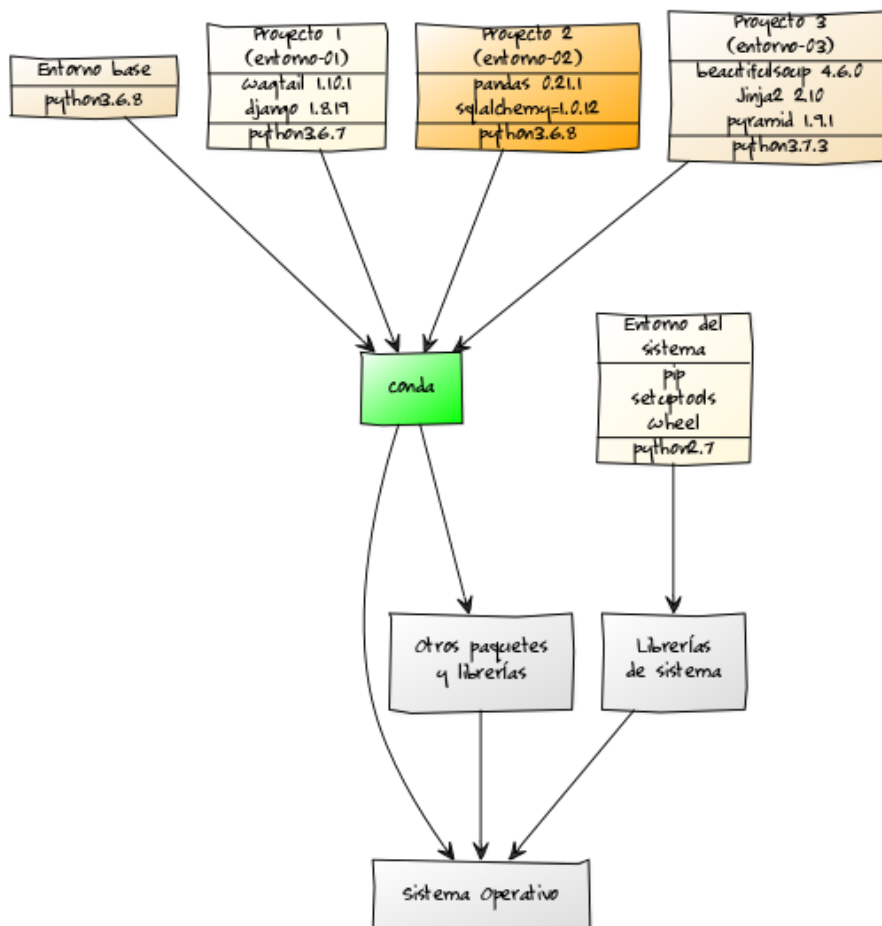
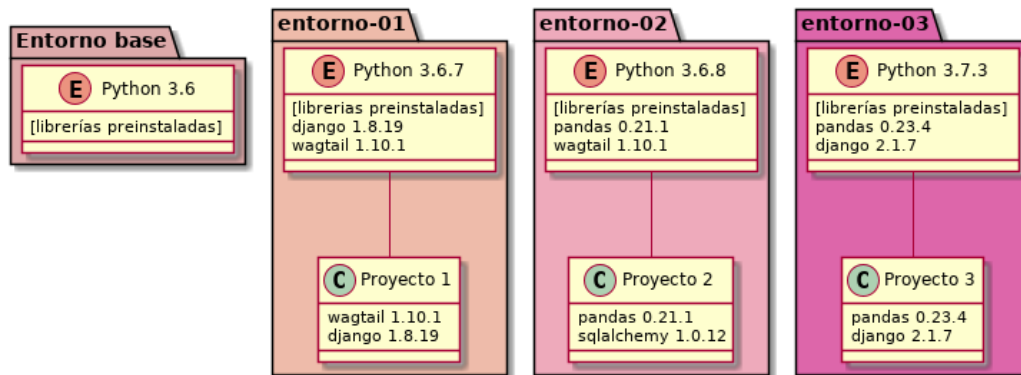
- ▶ Funcionan como un marco aislado y seguro
- ▶ Independientes entre sí
- ▶ Permiten especificar versiones de Python (2.7.10, 3.6.6, 3.6.7, ...)
- ▶ Facilmente reproducibles y exportables (**Portabilidad**)
  - ▶ p.e. replicar un entorno preexistente
- ▶ De “*usar y tirar*” (p.e. probar nuevas versiones de código o librerías)

Por defecto partiremos de un entorno global/base



**Regla general:** evitar usar el intérprete global del sistema y el entorno base

- ▶ Puede afectar a otros componentes
- ▶ Dependencias entre distintos proyectos
- ▶ Puede no ser la misma versión que la requerida en un proyecto



## Conda

### Conda (anaconda/miniconda)

- ▶ Gestor de paquetes multi-lenguaje (python/R/...)
- ▶ Transparente: no instala ficheros fuera de su directorio
- ▶ Distribución multipropósito, gestiona paquetes adicionales (p.e. git, librerías, ...)
- ▶ Coexistencia de entornos con diferentes librerías y versiones de python
- ▶ Por defecto nos encontraremos en un entorno llamado base



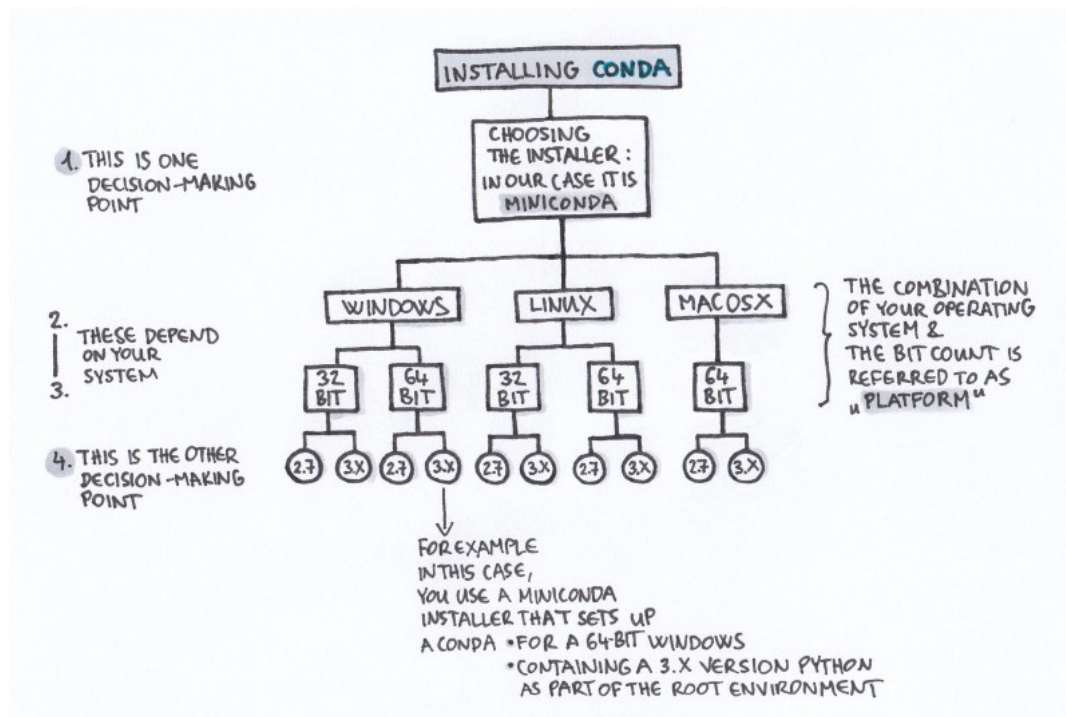
Anaconda	Miniconda
~3GB disco	<400MB
> 200 librerías	base + dependencias
+ herramientas	ciclo distribución + rápido
IDE (Spyder + <b>VSCode</b> )	
Anaconda Navigator	sin interfaz gráfico
↑ tiempo instalación	

Alternativas (más bajo nivel):

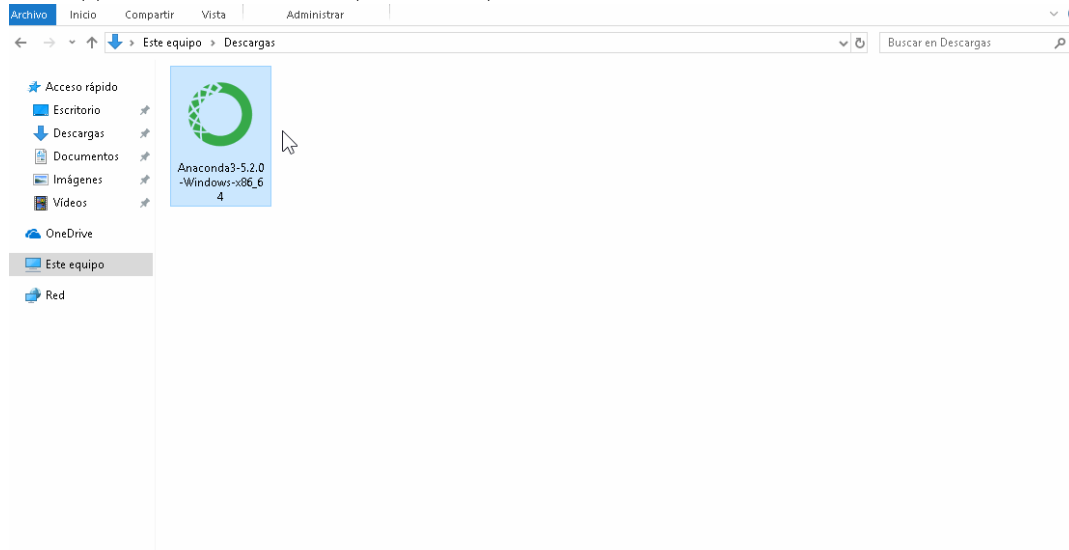
- ▶ python + pipenv
- ▶ python + virtualenvwrapper (lazy)

## Instalación

Determinar la plataforma sobre la que se va a instalar



<https://www.anaconda.com/download/>



La instalación puede tardar unos 5-10 minutos

## Práctica I

## Entorno virtual con Navigator

- ▶ Ejecutar “Anaconda Navigator” y crear un nuevo entorno (distinto a base) con diferentes librerías instaladas, p.e.:

pandas	bs4
matplotlib	jupyter
scrapy	scipy
sqlalchemy	flask
jsonschema	bokeh

## Práctica II

### Entorno virtual en modo texto

- ▶ Ejecutar Anaconda prompt
- ▶ Verificar que conda está instalado:

```
(base) conda info
```

```
(base) conda --help
```

Crear un entorno virtual con pandas, matplotlib, jupyter y pyjstat.

► Inicializar el entorno virtual:

```
(base) conda create --name entorno-01
(base) conda create --name entorno-02 python=2.7 --yes
# crea entorno con paquetes preinstalados
(base) conda create -n entorno-03 python=3.7 pandas scipy -y
```

► Acceder (activar) el entorno virtual:

```
(base) conda activate entorno-03
(entorno-03) conda list # muestra paquetes instalados
```

Crear un entorno virtual con pandas, matplotlib, jupyter y pyjstat.

► Instalar librerías adicionales dentro del entorno

```
# instala librerías gestionadas por conda
(entorno 03) conda install matplotlib jupyter --yes
# instala paquetes no gestionados por conda desde PyPI (Python Package Index)
(entorno-03) pip install -y pyjstat
```

► Exportar entorno virtual

```
# exportar definición del entorno
(entorno-03) conda list --export > requirements.txt # solo dependencias
(entorno-03) conda env export > entorno-03.yml # entorno + dependencias
```

**Distribuible y replicable**

- ▶ Acceder al entorno virtual y mostrar librerías instaladas

```
(base) conda activate entorno-03
```

```
(entorno-03) conda list # muestra paquetes instalados
```

```
# instala paquetes adicionales
```

```
(entorno-03) conda install jupyter -y
```

```
# instala paquetes desde PyPI, no gestionados por conda
```

```
(entorno-03) pip install pyjstat
```

```
(entorno-03) conda list
```

```
# instala paquetes desde github usando pip
```

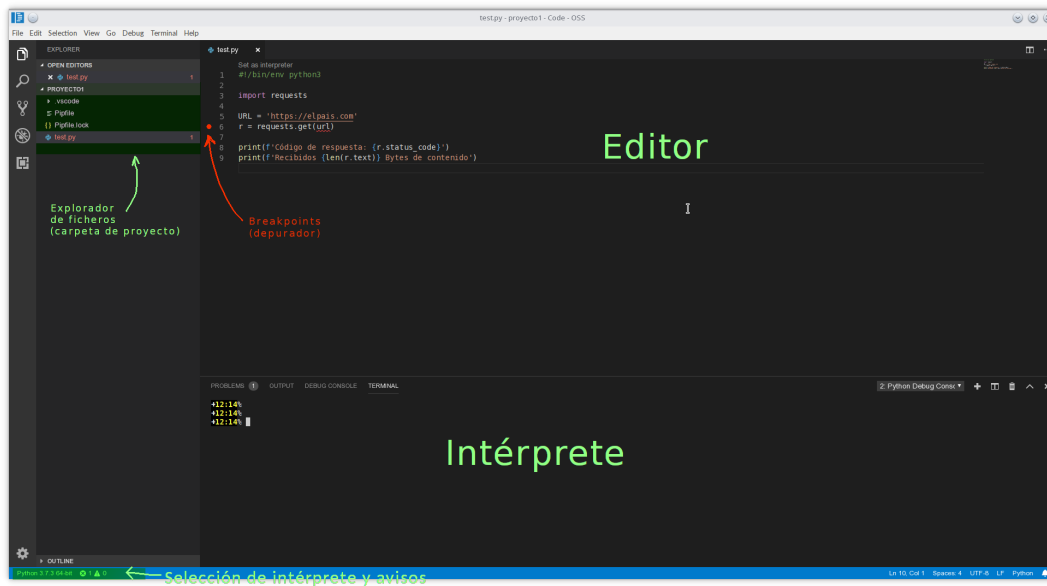
```
(entorno-03) pip install -e ^
```

```
"git+https://github.com/predicador37/pyjstat.git#egg=pyjstat-git"
```

- ▶ Resumen comandos conda
- ▶ Guía de usuario pip

vscode

- ▶ Paleta de comandos (Ctrl+Shift+P)
  - ▶ “Instalar extensiones”
  - ▶ “Python: ...”



## Integración con el entorno virtual de cada proyecto

- ▶ Seleccionar intérprete + nombre del entorno
- ▶ terminal > `ipython` / `python` o bien “Python: ejecutar REPL”



## Linter

- ▶ Analizador del código, detecta errores de sintaxis, estilo, ...
- ▶ Necesario instalar en cada entorno (preinstalado en base)
  - ▶ Paleta de comandos > *Python: seleccionar Linter* > flake8

## flake8

- ▶ Comprueba conformidad con PEP8, complejidad de McCabe, errores de sintaxis...
- ▶ Permite ignorar/silenciar diferentes avisos, excluir ficheros del análisis, ...

## Depuración de errores

- ▶ Errores de sintaxis
- ▶ Avisos del *linter*

```
#!/bin/env python3
import requests

URL='https://elpais.com'
r = requests.get(url) # crear un breakpoint aquí

print(f'Código de respuesta: {r.status_code}')
print(f'Recibidos {len(r.text)} Bytes de contenido')
```

## Estructura del código

- ▶ No hay delimitadores de línea (tipo ';' )
- ▶ Comentar código con #
- ▶ Jerarquía del código según nivel de indentación
- ▶ Indentar código con espacios
- ▶ Importar librerías al inicio
- ▶ Código agrupado con líneas en blanco
- ▶ **Recomendado:** Longitud de línea < 80 caracteres

- ▶ No hay delimitadores de línea (tipo ;)
- ▶ Comentar código con #

```
import os

home = os.path.expanduser('~') # comentario en línea
directorio_entorno = os.path.join(home,
                                   'Anaconda3',
                                   'envs',
                                   'entorno-03')

ficheros = []

# los comentarios comienzan con el caracter '#'
for f in os.listdir(directorio_entorno):
    if os.path.isfile(f):
        tamaño = os.stat(os.path.join(directorio_entorno, f)).st_size
        ficheros.append((f, tamaño))
```

- ▶ Jerarquía del código según nivel de sangría (*indent*)
- ▶ Diferenciarlo con espacios <sup>1</sup>

```
import os

home = os.path.expanduser('~') # directorio del usuario
directorio_entorno = os.path.join(
    home, 'Anaconda3', 'envs', 'entorno-03'
)
ficheros = []

# Busca ficheros y guarda (nombre, tamaño)
for f in os.listdir(directorio_entorno):
    if os.path.isfile(f):
        tamaño = os.stat(os.path.join(directorio_entorno, f)).st_size
        ficheros.append((f, tamaño))
    print(f)
print(tamaño)
```

<sup>1</sup> Generalmente con 4 espacios, no mezclar con TAB

- ▶ Importar librerías al inicio
- ▶ Código agrupado con líneas en blanco

```
import pathlib
from pathlib import Path

home = Path.home() # pathlib.Path.home()
directorio_entorno = home / 'Anaconda3' / 'envs' / 'entorno-03'
ficheros = [(f.name, f.stat().st_size)
             for f in directorio_entorno.iterdir()
             if f.is_file()]

cuaderno jupyter
cuaderno jupyter (offline)
```

Palabras reservadas

```
import builtins
import keyword
```

```
print(', '.join(keyword.kwlist))
```

```
False, None, True, and, as, assert, async, await, break, class,
continue, def, del, elif, else, except, finally, for, from,
global, if, import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, while, with, yield
```

```
print(dir(builtins))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError',
'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
```

```
if __name__ == "__main__":
```

Se llama a `__main__()` cuando se ejecuta directamente:

```
(base) python circunferencia.py
```

```
import sys
import math
```

```
def area(radio):
    return math.pi * (radio ** 2)
```

```
def longitud(radio):
    return 2 * math.pi * radio
```

*# entra aquí cuando se ejecuta directamente*

```
if __name__ == "__main__":
    radio = float(sys.argv[1]) # sys.argv[] son los argumentos de entrada
    print(
        "La longitud de una circunferencia de radio {}cm es {:.2f}cm^2."
        .format(radio, longitud(radio))
    )
    print("El area de una circunferencia de radio {}cm es {:.2f}cm^2."
        .format(area(radio), radio))
```

- ▶ Un fichero python puede contener (entre otros) definiciones de constantes, variables, funciones o clases.
- ▶ Para organizar el código guardaremos los ficheros .py en un árbol de directorios:

```
principal/
  __init__.py
  practica.py
  recolector/
    __init__.py
    collector.py
    database.py
  conversor/
    __init__.py
    limpia.py
    procesa.py
  graficos/
    __init__.py
    graficos.py
    exportar.py
  string/
    __init__.py
    string.py
```

## Proyecto ejemplo

proyecto/	nombre del paquete
README.rst	descripción del proyecto
LICENSE	licencia
setup.py	distribución/empaquetado [2]
requirements.txt	descripción de las dependencias
entorno_conda.yml	descripción del entorno
ejemplo/__init__.py	el código en sí
ejemplo/core.py	" "
ejemplo/helpers.py	" "
docs/conf.py	documentación del proyecto
docs/index.rst	" "
tests/unitarios.py	funciones de test del proyecto
tests/funcionales.py	" "

<sup>2</sup> p.e. setuptools

## Práctica III

### Parte 1

- ▶ Abrir una consola ipython
- ▶ In[n] identifica las entradas de comandos
- ▶ comandos *mágicos* (solamente para ipython)
  - ▶ comienzan por el caracter '%'
  - ▶ ejemplo: %edit, %pylab
- ▶ Ctrl+R activa la búsqueda en el historial
  - ▶ %hist, %history para visualizarlo
- ▶ \_ guarda la salida del último comando <sup>3</sup>

<sup>3</sup> \_ se usa también como variable de usar/tirar, p.e. cuando una función devuelve varios resultados, pero solamente estamos interesados en uno.

- ▶ [TAB] autocompleta (p.e. import st[TAB])
  - ▶ muy util para ver los métodos/atributos de un módulo/clase

```
In[]: import st[TAB]
```

```
In[]: import string
```

```
In[]: string.[TAB]
```

```
string.Formatter      string.ascii_uppercase string.octdigits
string.Template       string.capwords         string.printable
string.ascii_letters  string.digits           string.punctuation
string.ascii_lowercase string.hexdigits        string.whitespace
```

```
In[1]: ? # muestra ayuda del intérprete ipython
```

```
In[2]: ? dict # muestra ayuda breve de un método/clase/...
```

```
In[3]: help(dict) # muestra ayuda completa
```

```
In[4]: import math
```

```
In[5]: math.sq[TAB]
```

```
In[5]: math.sqrt(94)
```

```
In[6]: a = _ # recupera última salida
```

```
In[7]: dir(math) # muestra todos los métodos
```



- ▶ Importa la librería `math`
- ▶ Convierte 67 grados a radianes (`math.radians`)
- ▶ Calcula el cuadrado ( $n^2$ ) del resultado (operador `**` o `math.pow`)
- ▶ Verifica si el resultado es mayor que 16/13

## Solución

```
import math
```

```
grados = math.radians(67)
grados ** 2 > 16/13
# math.pow(grados, 2) > 16/13
```

```
True
```

## Parte 2

- ▶ Importa la librería numpy
  - ▶ `import numpy as np`
- ▶ Importa el módulo pyplot de la librería matplotlib
  - ▶ `from matplotlib import pyplot as plt`
- ▶ Crea un array `a` de 1000 puntos entre `[0, 1]`
  - ▶ Ejecuta el método `np.linspace()`
- ▶ Calcula el seno de  $2 \cdot \pi \cdot a$ 
  - ▶ usa el método `np.sin()`
  - ▶ usa la constante `np.pi`
- ▶ Activa el modo gráfico integrado
  - ▶ `%matplotlib inline`
- ▶ Crea un gráfico con `a` en el eje `x` y  $2 \cdot \pi \cdot a$  en el eje `y`
  - ▶ emplea la función `plt.plot()`

## Solución

```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt

a = np.linspace(0, 1, 1000)
b = sin(2*np.pi*a)

plt.plot(a, b)
```

## Práctica IV

Descargar el cuaderno de ejemplo en el directorio `python` y ejecutar línea por línea.  
(o bien ejecutarlo en forma remota con Binder)

## Práctica V

Exportar entorno virtual

*# exportar definición del entorno*

*# solo dependencias*

```
(entorno-01) conda list --export > requirements.txt
```

*# entorno + dependencias*

```
(entorno-01) conda env export > entorno-01.yml
```

*# o bien desde base*

```
(base) conda env export --name entorno-01 > entorno-01.yml
```

*# Restaurar el entorno con otro nombre (clonar)*

```
(entorno-01) deactivate
```

```
(base) conda env create --name entorno-11 --file requirements.txt
```

```
(base) activate entorno-11
```

```
(entorno-11) conda list
```

*# Destruir el entorno y restaurarlo desde la copia*

```
(base) conda env remove --name entorno-01 -y
```

```
(base) conda env create -f entorno-01.yml
```

## Exportar entorno virtual (II)

- ▶ Copia y restauración de dependencias con pip (`requirements.txt`)
  - ▶ independiente de conda

```
(entorno-01) pip freeze > requirements.txt
```

```
(entorno-01) conda activate entorno-03
```

```
(entorno-03) pip install -r requirements.txt
```

Recursos adicionales

## Download

Download these documents

## Docs by version

Python 3.8 (in development)  
Python 3.7 (stable)  
Python 3.6 (security-fixes)  
Python 3.5 (security-fixes)  
Python 2.7 (stable)  
All versions

## Other resources

PEP Index  
Beginner's Guide  
Book List  
Audio/Visual Talks

# Python 3.7.3 documentation

Welcome! This is the documentation for Python 3.7.3.

## Parts of the documentation:

### What's new in Python 3.7?

or all "What's new" documents since 2.0

### Tutorial

start here

### Library Reference

keep this under your pillow

### Language Reference

describes syntax and language elements

### Python Setup and Usage

how to use Python on different platforms

### Python HOWTOs

in-depth documents on specific topics

### Installing Python Modules

installing from the Python Package Index & other sources

### Distributing Python Modules

publishing modules for installation by others

### Extending and Embedding

tutorial for C/C++ programmers

### Python/C API

reference for C/C++ programmers

### FAQs

frequently asked questions (with answers!)

Documentación oficial

- ▶ Tutorial de python
- ▶ Guías de estilo:
  - ▶ PEP8
  - ▶ Guía de estilo de Google
- ▶ Cuadernos Jupyter para Data Science

#### Otros recursos en línea

- ▶ Librería standard de Python
- ▶ Guía de referencia de Python
- ▶ Guía para principiantes
- ▶ **The Hitchhiker's Guide to Python!**
- ▶ Stackoverflow
- ▶ Stackoverflow en español
- ▶ awesome python

*Bonus:* pipenv



- ▶ Permite gestionar un entorno virtual y sus dependencias
- ▶ Instalación desde pip
- ▶ Cada carpeta de proyecto es considerada un entorno virtual
- ▶ Pipfile y Pipfile.lock definen las dependencias
  - ▶ Pipfile.lock para versiones específicas, evitando que se actualicen
  - ▶ Recomendable añadir ambos ficheros en control de versiones
- ▶ Ejecutar scripts con `pipenv run ...`

## Instala pipenv y crea un entorno virtual nuevo

```
$ mkdir proyecto
$ cd proyecto
$ pipenv --python=3.7 [--three]
```

## Instala distintos paquetes dentro del entorno virtual

```
$ pipenv install bpython
$ pip install 'pandas==' # muestra versiones disponibles [OJO! pip]
$ pipenv install 'pandas==0.23.1' 'requests==2.2.1'
$ pipenv install 'lxml==3.*' # bloquea solamente la rama 3.x
$ pipenv install 'untangle==1.1.*' # bloquea versiones [major, minor]
$ pipenv install -r requirements.txt # instala desde un fichero requirements
$ pipenv lock -r > requirements.txt # exporta a fichero requirements
```

Para comprobar los paquetes instalados, mostrar el contenido de Pipfile

```
$ cat Pipfile
$ cat Pipfile.lock
```

Otras funcionalidades útiles con pipenv:

*# muestra dónde instala realmente el entorno*

**\$ pipenv --venv**

*# muestra gráfico de dependencias*

**\$ pipenv graph**

*# muestra vulnerabilidades con los paquetes instalados*

**\$ pipenv check**

Acceder al intérprete propio del entorno:

**\$ pipenv shell**

**\$ python** *# o bpython/ipython si está instalado*

*# o bien: pipenv run python*

Ejecutar código cargando librerías propias del entorno:

```
>>> import pandas as pd
>>> URL_crypto = 'https://api.coinmarketcap.com/v1/ticker/'
>>> cc = pd.read_json(URL)
>>> cc.set_index('id', inplace=True)
>>> cc[cc['price_usd'] < 300.0]
```

```
# desinstala todos los paquetes no especificados en `Pipfile.lock`
$ pipenv clean
```

```
# destruye el entorno virtual
$ pipenv --rm
```

Restaura un entorno virtual desde los ficheros Pipfile y Pipfile.lock:

```
$ pipenv install
```

## *Bonus (II): GIL y GC*

### *Reference counting*

- ▶ Usado por python para la gestión de memoria
- ▶ Cuenta las referencias a cada objeto creado
- ▶ Cuando la cuenta es 0, la memoria reservada se libera

```
import sys
```

```
a = []
```

```
b = a
```

```
sys.getrefcount(a)
```

```
3
```

GC: *Garbage Collector*

Utiliza 2 algoritmos para la gestión de memoria:

- ▶ Reference counting: libera objetos sin referencias en un programa
  - ▶ ejecutado en "tiempo real"
- ▶ Referencias cíclicas: se ejecuta periódicamente

## Ejemplo

```
foo = []

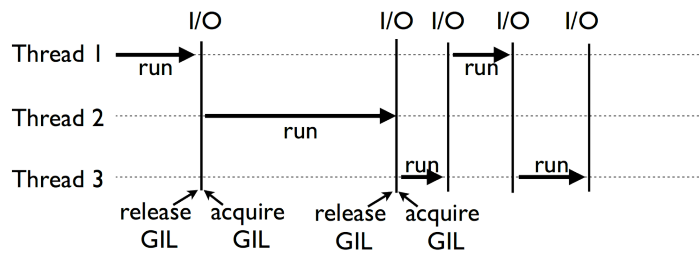
# 2 referencias, 1 de la variable foo y una de la llamada a getrefcount
print(sys.getrefcount(foo))

def bar(a):
    # 4 referencias
    # variable foo, argumento a función, getrefcount + pila interna de python
    print(sys.getrefcount(a))

bar(foo)
# 2 referencias, se limpiaron las propias de la función
print(sys.getrefcount(foo))
```

## GIL: Global Interpreter Lock

- ▶ Afecta a CPython
- ▶ Evita que la variable de conteo de referencias entre en condición de carrera
- ▶ Restringe la ejecución de código concurrente
  - ▶ ejecución secuencial dentro del intérprete
  - ▶ *multithreading*: aplicaciones no limitadas por CPU (p.e. I/O)
  - ▶ *multiprocessing*: varios intérpretes concurrentemente



## Ejemplo práctico: programa limitado por CPU, ejecución secuencial

```
# single_threaded.py
```

```
import time
```

```
COUNT = 50000000
```

```
def countdown(n):
```

```
    while n>0:
```

```
        n -= 1
```

```
start = time.time()
```

```
countdown(COUNT)
```

```
end = time.time()
```

```
print('Tiempo de ejecución {:.2f} segundos'.format(end - start))
```

```
$ python single_threaded.py
```

```
Tiempo de ejecución 2.30 segundos
```

## Programa limitado por CPU, ejecución concurrente (multithread)

```
# multi_threaded.py
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

thread_1 = Thread(target=countdown, args=(COUNT//2,))
thread_2 = Thread(target=countdown, args=(COUNT//2,))
start = time.time()
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
end = time.time()

print('Tiempo de ejecución {:.2f} segundos'.format(end - start))
$ python multi_threaded.py
Tiempo de ejecución 2.85 segundos
```

## Programa limitado por CPU, ejecución concurrente (multiprocess)

```
# multi_process.py
import time
from multiprocessing import Pool

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
with Pool(2) as p: # bypass del GIL
    p.map(countdown, 2* [COUNT//2])
end = time.time()

print('Tiempo de ejecución {:.2f} segundos'.format(end - start))
$ python multi_threaded.py
Tiempo de ejecución 1.32 segundos
```



cuaderno jupyter (offline)

cuaderno jupyter (binder)