

Python para Ciencia y Tecnología

Fundamentos y aplicaciones

$$\int_0^{\infty} f(x) = \sqrt{a - \frac{x}{2}} dx = \frac{x}{s}$$

$$\frac{e}{s} = \sqrt{\pi} = \int \frac{dx}{dx} = \log_c \sqrt{x} = \frac{\pi}{c}$$

$$s = \frac{1}{e^2} - \frac{1}{\pi} \int_0^t$$



$$\frac{1}{t} = f(x) = \sin(x), \frac{\pi}{e} = \int_0^t$$

$$a - \frac{t}{r} = \int_0^t dlm \frac{x}{x-1} c + \frac{1}{x_0} + c$$

$$= \frac{\sqrt{1-t}}{c^2} = \sum_{n=0}^{\infty} \frac{1}{(1-\frac{x}{2})^2} dx = \frac{c}{x}$$

$$\frac{C}{O}(R) = \frac{1}{2} \int dx = C$$

$$= \frac{i}{Re} \int_{x \rightarrow \infty} \frac{dx}{e^2} - \frac{C}{2} =$$
$$\int_0^{\infty} s' x' dx = \frac{\pi}{2} \left(\frac{\sqrt{1-t} x}{x} \right)$$

$$\prod_{n=1}^{\infty} t(x)$$

$$\frac{1}{x} \int dx = \frac{1}{xi} \log(x) = \frac{C}{x}$$

$$\frac{\pi}{\sqrt{3}} \frac{\pi}{s}$$

$$\frac{1}{s} : \frac{x}{\sqrt{1-\frac{x^2}{c}}}$$

$$\frac{\pi x}{2} \frac{C}{\sqrt{n}} =$$

$$\tau = \frac{p}{C} i \sqrt{0} = \frac{R}{1_n}$$

$$\int_0^t \ln \frac{t}{t+1} \frac{\pi}{c} \Delta x$$

$$\int_0^t \frac{i n}{\sqrt{t}} \frac{\pi}{t+1}$$

$$\frac{\pi}{e} = \frac{\sqrt{v}}{n}$$

$$\log \frac{r}{s} = \frac{\pi}{s^2}$$

$$\frac{s}{r} = \frac{a+b}{l} - \frac{t}{v}$$

$$\left[\gamma = \frac{S(t)}{t^2} \sin(x), \frac{\pi}{e} = \int_0^t \frac{1}{x} \int dx = \frac{1}{xi} \log(x) \right] = \frac{C}{C} - \frac{l}{x} = a$$

Facundo Batista y Manuel Carlevaro

Python para Ciencia y Tecnología

Facundo Batista y Manuel Carlevaro

12 de junio de 2025

VERSIÓN PRELIMINAR

Título: Python para Ciencia y Tecnología
Autores: Facundo Batista y Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?
© Facundo Batista y Manuel Carlevaro
Versión 1.0
Escrito con X_ETEX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](#)
Lugar: Florida y La Plata, Buenos Aires, Argentina
Año: 2020 - 2025
Web: <http://pyciencia.taniquetil.com.ar/>

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para quienes se desempeñan en el ámbito científico o tecnológico, a partir de nuestra experiencia en el dictado del curso “Herramientas Computacionales para Científicos” que ofrecemos en la Universidad Nacional de La Plata y en la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar algunos problemas usuales en la ciencia o en la tecnología utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de quienes dan los primeros pasos en el lenguaje y abordan la búsqueda de soluciones de algunos problemas básicos (pero no por eso menos importantes) mediante técnicas numéricas y en algunos casos, también analíticas.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional ([CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)), salvo que se especifique puntualmente lo contrario.

Florida y La Plata, Buenos Aires, Argentina,

Facundo Batista y Manuel Carlevaro

Índice general

Prefacio	2
Índice general	3
I Python	4
1. Entornos de ejecución de Python	5
1.1. Por qué tener distintos entornos	6
1.2. Entornos virtuales	7
1.2.1. En detalle	7
1.2.2. Herramientas	11
1.3. Contenedores	17
1.3.1. ¿Imágenes o contenedores?	18
1.3.2. Creando imágenes	18
1.3.3. Ejecutando contenedores	20
1.3.4. Valor extra de los contenedores como entornos aislados	22
1.3.5. Compartiendo imágenes	24
II Herramientas fundamentales	26
III Métodos numéricos y técnicas computacionales	27
IV Apéndices	28
A. Zen de Python	29
Bibliografía	30

Parte I Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.



1 | Entornos de ejecución de Python

El desarrollo de programas no termina cuando escribimos los mismos, sino que es responsabilidad del desarrollador el cómo se van a ejecutar esos programas. Escribir un programa que no puede ser ejecutado no tiene sentido.

Lo más común es que tengamos distintos entornos de ejecución de nuestros programas. El más natural es el de desarrollo mismo: empezamos a trabajar en nuestra máquina, tenemos a disposición las herramientas y bibliotecas instaladas en nuestro sistema y utilizamos eso.

Cuando el desarrollo es compartido, ya se nos presenta la primer dificultad: las distintas personas trabajando en el programa van a tener distintos entornos de ejecución. Luego, siempre hay nuevas situaciones con nuevos entornos: vamos a querer correr sistemas de integración continua¹ y esos sistemas tendrán otros entornos, etc. Y claro, finalmente queremos que nuestro programa sea realmente usado, y normalmente esto implica que se instale en otras máquinas, sean locales



[Código disponible](#)

¹ Una herramienta muy usada en el mundo del desarrollo de software para validar automáticamente la calidad de los programas, mediante pruebas que se ejecutan en distintos entornos (independientes del desarrollador) para cada cambio en esos programas.

o remotas.

En la descripción anterior podemos distinguir dos mundos distintos, en función del control que tenemos como desarrolladores. Por un lado están los entornos de los desarrolladores mismos y sistemas aledaños (como los de integración continua que mencionábamos); en estos normalmente tenemos el poder de definir qué y cómo se instala y está disponible para nuestro programa. Por otro lado están aquellos entornos donde tenemos poco o ningún control: el de los servidores o instalaciones en máquinas particulares, en función del tipo de programa que estamos haciendo. En este capítulo veremos distintas formas de proveer entornos controlados de ejecución pensando en el primer grupo (entornos controlados durante el desarrollo). Aunque algunas de las técnicas y herramientas que contaremos se pueden utilizar o nos pueden facilitar el despliegue de nuestros sistemas en servidores o la distribución e instalación en máquinas clientes, no es el foco de lo que expondremos.

1.1. Por qué tener distintos entornos

Hay dos grandes características que queremos obtener cuando estamos pensando en entornos de ejecución.

La primera funcionalidad es que nuestro programa se ejecute de la misma manera por un lado en nuestro entorno de desarrollo y por el otro en los entornos de otros desarrolladores y en otros sistemas donde corran las pruebas de unidad o integración, e incluso cuando lo llevamos a los entornos finales. Para que tenga el mismo comportamiento en los distintos entornos necesitamos básicamente que los entornos sean iguales, al menos en lo que refiere al contexto de nuestro programa (por ejemplo, la versión de Python utilizada y las versiones de las bibliotecas).

Esta característica que estamos buscando es “repetibilidad”. Si en las distintas computadoras usadas para desarrollar el programa se comporta de distinta manera tendremos muchos dolores de cabeza (y tiempo y esfuerzo perdidos). Entonces, tenemos que proveer entornos lo suficientemente similares para que el programa se ejecute con el mismo comportamiento.

Otra situación con la que nos encontramos es que para un determinado proyecto necesitamos una biblioteca en una versión específica pero nuestro sistema tiene una versión distinta. Si fuese más vieja, podríamos actualizar nuestro sistema, pero eso podría traernos algunos comportamientos inesperados en el mismo. Si fuese más nueva, llevar nuestro sistema a versiones anteriores podría ser incluso mas problemático. Y esto incluso podría no ser una opción, si es que para distintos proyectos necesitamos la misma biblioteca en versiones distintas, ya que no podemos estar actualizando y desactualizando nuestro sistema operativo cada vez que ejecutamos uno u otro programa.

En este caso lo que estamos buscando es “aislamiento”: que las bibliotecas que usa nuestro programa sean distintas que las que tenga el sistema (en verdad, sin importarnos qué versiones tiene el sistema, o incluso si tiene esas bibliotecas). De esta manera cada proyecto indicará cuales bibliotecas y en qué versiones las necesita, y al estar aislado del sistema base tendremos el mismo comportamiento más allá de cual sea ese sistema.

Entonces, la idea de tener distintos entornos de ejecución es, a priori, lograr estas dos grandes ventajas, repetibilidad y aislamiento. A continuación veremos dos formas de lograr estos entornos, con mayor o menor cumplimiento de estas dos ventajas y más características, que iremos explicando en cada caso.

Para hacer los ejemplos más prácticos, ejecutaremos un programa real (el `oscilador.py` que armamos en el capítulo de Ecuaciones Diferenciales Ordinarias ??) utilizando cada una de las herramientas mostradas.

1.2. Entornos virtuales

Un “entorno virtual” (que muchas veces denominamos directamente usando su nombre en inglés, *virtual environment*, o incluso acortamos con *virtualenv*) es la solución de *virtualización* de entornos nativa a Python, y por ende es la más popular. Hay muchas herramientas para trabajar con este concepto, algunas incluso en la biblioteca estándar; veremos las principales más adelante en la subsección de Herramientas 1.2.2.

El uso de un entorno virtual es la forma más sencilla y liviana de conseguir las características que presentamos en las secciones anteriores, aunque sólo nos las ofrece para las bibliotecas que utilizaremos (sólo trabaja a nivel de dependencias de Python). Estamos atados al Python instalado en el sistema para cada entorno, y tendremos que recrear ese entorno en todos aquellos sistemas donde lo necesitemos (no es trasladable).

La funcionalidad que nos ofrece es suficiente en la gran mayoría de los casos, más allá de las limitaciones indicadas, por lo que es normalmente la solución elegida sin tener que pasar a otras más complejas como la que veremos más adelante.

1.2.1. En detalle

Para mostrar la creación y uso de los entornos virtuales vamos a utilizar la herramienta integrada en la biblioteca estándar, el módulo `venv`. Lo llamamos directamente (usando la opción `-m` de Python para ejecutar un módulo como si fuese un script), indicándole el nombre del directorio donde queremos que nos cree toda la estructura.

```
[/temp]$ python3 -m venv mivenv
[/temp]$ ls -l mivenv
total 20
drwxrwxr-x 2 facundo facundo 4096 jun  8 16:02 bin
drwxrwxr-x 2 facundo facundo 4096 jun  8 16:02 include
drwxrwxr-x 3 facundo facundo 4096 jun  8 16:02 lib
lrwxrwxrwx 1 facundo facundo   3 jun  8 16:02 lib64 -> lib
-rw-rw-r-- 1 facundo facundo  69 jun  8 16:03 pyvenv.cfg
drwxrwxr-x 3 facundo facundo 4096 jun  8 16:02 share
```

Tengamos en cuenta que esta estructura puede variar según el sistema donde estemos creando el entorno, porque justamente sigue la estructura de ese sistema. Si exploramos el contenido de este directorio veremos que tiene algunos ejecutables y bibliotecas preinstaladas (notoriamente Python mismo y `pip`, la herramienta para instalar otras bibliotecas). Un detalle importante con este directorio es que no podemos moverlo a otro lugar, ya que depende de su posición al ser creado. Pero no deja de ser un directorio normal, así que si queremos “limpiar” el entorno, borrando ese directorio eliminará todo vestigio del mismo.

Cuando ejecutemos el Python de ese directorio, el mismo tendrá acceso a la biblioteca estándar del Python utilizado más las bibliotecas que instalamos en ese entorno virtual, y no podrá

acceder a las del sistema. A nivel de sistema nada se modifica cuando instalamos bibliotecas en el entorno virtual, con lo que no hay riesgo de desestabilizarlo, y un buen efecto secundario al usar los entornos virtuales es que no dependemos de tener permisos especiales de *root* o “administrador” para instalar las bibliotecas que necesitemos.

Habiendo dicho eso, hay una manera de crear el entorno virtual rompiendo un poco la aislación natural que nos provee. Si usamos el parámetro `--system-site-packages` al momento de creación, Python buscará en el sistema las bibliotecas que no encuentre en el entorno. Esto es raramente utilizado, pero muy necesario en caso de bibliotecas muy difíciles de instalar (pensemos el caso donde el proceso de instalar una biblioteca en particular lleva mucho tiempo, y no queremos pagar ese costo en cada entorno virtual que generemos). Hay otras opciones más para explorar, les recomendamos profundizar en [la documentación del módulo](#), o pedir de forma rápida la ayuda con `python3 -m venv --help`.

La forma habitual de usar los entornos virtuales es activándolos primero, lo que se hace de forma muy parecida (pero no igual) en los distintos sistemas.

Por ejemplo, en Linux y los sistemas *UNIX-like*:

```
[/temp]$ source mivenv/bin/activate  
[(mivenv) /temp]$
```

O en Windows:

```
[C:]>mivenv\Scripts\activate.bat  
[(mivenv) C:]>
```

Notar como en cada caso, luego de activar el entorno, el *prompt* cambió para reflejar eso mismo.

Una vez dentro del entorno podremos proceder a instalar las bibliotecas que deseemos, usando la herramienta `pip`.

`pip` es el administrador de paquetes estándar de Python para manejar todas las bibliotecas (por fuera de la biblioteca estándar, claro), y no sólo instalará la biblioteca que le indiquemos, sino que automáticamente instalará sus dependencias (de ser necesario, ya que podrían estar instaladas de antes).

Aunque podemos indicar distintos repositorios o incluso indicarle que la fuente sea local, el procedimiento natural es que `pip` descargue e instale las bibliotecas desde el “Índice de Paquetes de Python”, más bien conocido como PyPI (por su sigla en inglés, *Python Package Index*).

En el siguiente ejemplo mostramos cómo instalar una biblioteca que usamos extensivamente en todo el libro:

```
[(mivenv) /temp]$ pip install numpy  
Collecting numpy  
...  
Successfully installed numpy-1.20.3
```

Una forma de validar que la biblioteca está instalada en el `virtualenv` es a través del *path* del módulo:

```
[(mivenv) /temp]$ python
Python 3.12.3 (main, Feb 4 2025, 14:48:35)
>>> import numpy
>>> numpy.__file__
'/temp/mivenv/lib/python3.12/site-packages/numpy/__init__.py'
```

Decíamos que una de las ventajas de la aislación del entorno virtual era poder tener la misma biblioteca con distintas versiones en distintos entornos. Para ello, vamos a necesitar que pip no instale la última versión disponible (como sucedió recien con NumPy), sino poder especificar puntualmente cual necesitamos.

Por ejemplo, instalemos otra biblioteca usada mucho en el libro, Matplotlib, pero no la última versión (que al momento de escribir estas líneas es la 3.9.2), sino una particular que podemos necesitar por algún motivo específico:

```
[(mivenv) /temp]$ pip install matplotlib==3.3.4
Collecting matplotlib==3.3.4
...
Successfully installed cylicer-0.10.0 kiwisolver-1.3.1 matplotlib-3.3.4 pillow-8.2.0 pyparsing-2.4.7
→ python-dateutil-2.8.1 six-1.16.0
```

Vemos como pip no sólo nos instaló el paquete que habíamos especificado, sino también las dependencias necesarias de forma automática. Cabe mencionar también que hay que tener cuidado cuando restringimos qué versión pip instalará, ya que como podemos utilizar los comparadores `>`, `<`, `>=`, `<=` para indicar esas restricciones, podemos meternos en problemas porque esos caracteres tienen un significado especial en la línea de comandos; la solución es sencilla, con utilizar comillas es suficiente (por ejemplo `pip install "matplotlib>3"`).

Otra funcionalidad muy utilizada de pip es la de devolvernos todos los paquetes que están instalados en el entorno virtual, lo cual obviamente no solo incluye los que hayamos especificado al momento de la instalación, sino también las dependencias que se hayan instalado correspondientemente.

Vemos eso mismo en el siguiente ejemplo:

```
[(mivenv) /temp]$ pip freeze
cylicer==0.10.0
kiwisolver==1.3.1
matplotlib==3.3.4
numpy==1.20.3
Pillow==8.2.0
pyparsing==2.4.7
python-dateutil==2.8.1
six==1.16.0
```

Es normal en un proyecto tener anotadas todas las dependencias en un archivo, normalmente llamado `requirements.txt`, de manera de indicarle a pip que instale todas ellas en el entorno virtual usando el parámetro `--requirement`.

Recomandos revisar [la documentación de referencia de pip](#) para explorar otras funcionalidades que provee.

Finalmente, para salir del entorno debemos ejecutar el comando `deactivate` (que tenemos a nuestro alcance justamente porque estamos dentro del entorno):

```
[(mivenv) /temp]$ deactivate  
[/temp]$
```

Realicemos todo este mismo procedimiento para ejecutar un programa ejemplo: el `oscilador.py` que mencionamos arriba, que podemos descargar [desde el repositorio del libro](#).

Primero entonces creamos el entorno virtual, activémoslo e instalaremos las dependencias necesarias.

```
[/temp]$ python3 -m venv oscilador-venv  
[/temp]$ source oscilador-venv/bin/activate  
[(oscilador-venv) /temp]$ pip install matplotlib numpy scipy PyQt5  
Collecting matplotlib  
  Downloading matplotlib-3.4.3-cp38-cp38-manylinux1_x86_64.whl (10.3 MB)  
   |██████████| 10.3 MB 2.4 MB/s  
Collecting numpy  
  Downloading numpy-1.21.2-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)  
   |██████████| 15.8 MB 2.0 MB/s  
Collecting scipy  
  Downloading scipy-1.7.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (28.4 MB)  
   |██████████| 28.4 MB 2.0 MB/s  
Collecting PyQt5  
  Using cached PyQt5-5.15.4-cp36.cp37.cp38.cp39-abi3-manylinux2014_x86_64.whl (8.3 MB)  
Collecting python-dateutil>=2.7  
  Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)  
Collecting pillow>=6.2.0  
  Using cached Pillow-8.3.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (3.0 MB)  
Collecting cycler>=0.10  
  Using cached cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)  
Collecting pyparsing>=2.2.1  
  Using cached pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)  
Collecting kiwisolver>=1.0.1  
  Using cached kiwisolver-1.3.1-cp38-cp38-manylinux1_x86_64.whl (1.2 MB)  
Collecting PyQt5-Qt5>=5.15  
  Using cached PyQt5_Qt5-5.15.2-py3-none-manylinux2014_x86_64.whl (59.9 MB)  
Collecting PyQt5-sip<13,>=12.8  
  Using cached PyQt5_sip-12.9.0-cp38-cp38-manylinux1_x86_64.whl (332 kB)  
Collecting six>=1.5  
  Using cached six-1.16.0-py3-none-any.whl (11 kB)  
Installing collected packages: six, python-dateutil, numpy, pillow, cycler, pyparsing, kiwisolver,  
  ↪ matplotlib, scipy, PyQt5-Qt5, PyQt5-sip, PyQt5  
Successfully installed PyQt5-5.15.4 PyQt5-Qt5-5.15.2 PyQt5-sip-12.9.0 cycler-0.10.0  
  ↪ kiwisolver-1.3.1 matplotlib-3.4.3 numpy-1.21.2 pillow-8.3.1 pyparsing-2.4.7  
  ↪ python-dateutil-2.8.2 scipy-1.7.1 six-1.16.0
```

A veces es complicado saber cuales son las dependencias que necesita un programa. Con los proyectos más grandes, como mencionábamos antes, se acostumbra tener un archivo comúnmente llamado `requirements.txt`, pero muchas veces no lo encontramos cuando estamos lidiando con un script “suelto”. En el caso del ejemplo, una simple inspección del archivo nos indica que necesitamos `matplotlib`, `numpy` y `scipy`, pero en realidad también necesitamos `PyQt5` para que `matplotlib` pueda abrir una ventana con el resultado.

Todo lo que nos queda es ejecutar el archivo utilizando directamente `python3`, ya que como tenemos activado el entorno virtual estaremos ejecutando el script ahí dentro:

```
[(oscilador-venv) /temp]$ python3 oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[(oscilador-venv) /temp]$ deactivate
[/temp]$
```

1.2.2. Herramientas

Hay muchas herramientas para trabajar con entornos virtuales, ya que es la metodología más establecida (e incluso soportada en la biblioteca estándar) para lograr los dos objetivos discutidos arriba.

En esta subsección no pretendemos cubrir todas las herramientas existentes, sino una selección de las mismas, algunas muy utilizadas, otras con funcionalidades específicas, con la idea de mostrar en qué casos conviene utilizar alguna u otra. Por supuesto, no hablaremos del módulo `venv`, ya que utilizamos este mismo arriba cuando mostramos genéricamente qué eran los entornos virtuales.

1.2.2.1. `virtualenv`

Esta herramienta fue durante mucho tiempo la única forma de crear entornos virtuales. Hoy en día, con parte de su funcionalidad integrada en la biblioteca estándar, deja de ser imprescindible.

Hay algunas diferencias a favor y en contra con respecto al módulo integrado `venv`, que pueden ser más o menos importantes dependiendo de nuestro contexto, aquí mencionamos las principales. Entonces, `virtualenv`...

- es varias veces más rápido en la creación del entorno: esto quizás no es importante cuando lo utilizamos manualmente, pero puede ser un factor si estamos generando muchos entornos automáticamente en algún sistema
- necesita instalarse por separado a Python, previamente a ser utilizado
- nos puede llegar a confundir con qué versión de Python está creando el entorno (en cambio cuando usamos el módulo `venv` nosotros explicitamos qué Python usamos)
- soporta versiones de Python en las que el módulo `venv` todavía no existía (aunque dicho módulo está en la biblioteca estándar desde Python 3.3)

Toda su [documentación](#).

1.2.2.2. `virtualenvwrapper`

Esta es una de las primeras herramientas que aparecieron alrededor de `virtualenv` y le agrega una capa de funcionalidad arriba, orientada a el manejo de los entornos virtuales como "directorios".

Por ejemplo, cuando armamos un entorno virtual nuevo, en vez de crearlo en el directorio donde estamos parados, lo creará en un lugar central (configurable con la variable de entorno `WORKON_HOME`), y permitirá listarlos, duplicarlos, y especialmente utilizarlos recordando solamente el nombre.

Entonces, cuando queremos activar un entorno virtual, en lugar de tener que recordar cómo se llamaba y dónde lo habíamos creado, sólo con tener el nombre alcanza.

La instalación no es trivial y lleva cierta configuración antes de que funcione, pero si la funcionalidad buscada es esta centralización de los entornos, se puede explorar más en [su documentación](#).

1.2.2.3. fades

Hasta ahora vimos el manejo de entornos virtuales como un proceso con beneficios, pero también con un costo de administración: debemos recordar cómo se llaman los `virtualenvs` que creamos, y (a menos que usemos `virtualenvwrapper`) donde los tenemos. Esto no es mayor problema si usamos el entorno para desarrollo de un proyecto: normalmente el lugar es en ese proyecto y el directorio tendrá un nombre conocido como `env` o `venv`. Así y todo, debemos recordar de activarlo antes de empezar a trabajar con el proyecto, y desactivarlo luego.

Ese costo de administración tiene sentido en un proyecto, pero se vuelve inmanejable cuando necesitamos entornos virtuales para simple scripts o programitas utilitarios que tengamos en nuestro sistema.

Por ejemplo, tenemos un pequeño script que nos muestra la metadata de un mp3, algo simple como lo siguiente:

```
1 #!/usr/bin/env python3
2
3 import sys
4
5 import tinytag
6
7 tag = tinytag.TinyTag.get(sys.argv[1])
8 print("{} por {}".format(tag.title, tag.artist))
```

Para que este script funcione, tendríamos que crear un `virtualenv` en algún lado, e instalar `tinytag` allí. Esto no es problema. La dificultad aparece cuando queremos volver a usar este script unas semanas o meses después. ¿Dónde teníamos el `virtualenv`? ¿Cómo se llamaba? O incluso, ¿este script necesitaba un `virtualenv` o lo podía ejecutar y ya?

Fades viene a solucionar este problema, justamente, ya que nos permite usar todo el poder de los entornos virtuales sin tener que preocuparnos por ellos. Fades creará automáticamente un nuevo entorno virtual e instalará las dependencias necesarias (o reusará uno creado previamente), y ejecutará el script dentro de ese entorno.

Modifiquemos levemente el script para usar esta nueva herramienta:

```
1 #!/usr/bin/env fades
2
3 import sys
```

```
4
5 import tinytag # fades
6
7 tag = tinytag.TinyTag.get(sys.argv[1])
8 print("{} por {}".format(tag.title, tag.artist))
```

Dos cambios. El primero es que indicamos que el intérprete del script es Fades, y el segundo es que pusimos un comentario al importar `tinytag`, de manera que Fades reconozca que esa es la biblioteca que tiene que proveer dentro de un entorno virtual.

Entonces, la primera vez que ejecutamos el script Fades armará el entorno virtual necesario y ejecutará el script allí. Y nos podemos olvidar. Volvemos al script luego de un tiempo, y sólo tenemos que ejecutarlo, no hace falta recordar el nombre o la ubicación del entorno virtual, ni siquiera si necesitábamos uno. Y si llevamos este script a una máquina nueva, automáticamente Fades se encargará de proveer el entorno necesario nuevamente.

Otro caso de uso muy común de Fades es probar bibliotecas en el intérprete interactivo. Para ello utilizamos Fades desde la línea de comando y especificamos la(s) dependencia(s) necesaria(s), pero no un script a ejecutar, entonces Fades creará el entorno virtual correspondiente y ejecutará el intérprete interactivo allí, con lo cual tendremos acceso a esas bibliotecas:

```
[/temp]$ fades -d tinytag --autoimport
Python 3.8.6 (default, May 27 2021, 13:28:02)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
::fades:: automatically imported 'tinytag'
>>> tinytag.__file__
↪ '/home/facundo/.local/share/fades/c7b5b66b-0923-4866-b4e5-6d84924737b1/lib/python3.8/site-packages/tinytag/__init__.py'
```

En el caso mostrado, Fades ya tenía el entorno virtual con `tinytag`, y lo reutilizó, pero si ejecutamos esa misma orden en una computadora diferente Fades nos informará que está creando el entorno e instalando esa dependencia.

Usemos Fades para ejecutar nuestro programa de ejemplo:

```
[/temp]$ fades -d matplotlib -d numpy -d scipy -d PyQt5 oscilador.py
*** fades *** 2021-08-21 10:41:37,143 INFO      Hi! This is fades 9.0.1, automatically
↪ managing your dependencies
*** fades *** 2021-08-21 10:41:37,143 INFO      Checking the availability of dependencies in
↪ PyPI. You can use '--no-precheck-availability' to avoid it.
*** fades *** 2021-08-21 10:41:41,115 INFO      Installing dependency: 'PyQt5'
*** fades *** 2021-08-21 10:41:44,816 INFO      Installing dependency: 'numpy'
*** fades *** 2021-08-21 10:41:47,650 INFO      Installing dependency: 'scipy'
*** fades *** 2021-08-21 10:41:51,216 INFO      Installing dependency: 'matplotlib'
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[/temp]$
```

Como puede ser un incordio escribir todas las dependencias en la línea de comandos, es natural tener un archivo `requirements.txt` con las dependencias especificadas allí (el nombre de

cada paquete en cada línea es suficiente si queremos la última versión disponible en cada caso, pero también podemos especificar qué versión queremos):

```
[/temp]$ cat requirements.txt
PyQt5
matplotlib
numpy
scipy
[/temp]$ fades -r requirements.txt oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[/temp]$
```

Presten atención como en este segundo caso no tenemos mensajes de Fades instalando dependencias, porque al ser exactamente las mismas que antes, Fades ya encuentra un entorno virtual con esas mismas, entonces lo reutiliza.

Más allá de eso, reconozcamos que también es un incordio tener las dependencias en un archivo externo, ya que al ser nuestro ejemplo un script suelto que podemos tener en cualquier lado, la obligación de recordar cual es su archivo de dependencias también es molesto.

Volvamos a hacer entonces lo mismo que con el script sencillo del principio: especifiquemos las dependencias en el mismo script. Veamos las primeras líneas de nuestro `oscilador.py`, levemente modificado:

```
1 #!/usr/bin/env fades
2
3 """Programa que integra el sistema masa-resorte-banda elástica.
4
5 fades:
6     PyQt5 # utilizado internamente por matplotlib para abrir una ventana con el resultado
7 """
8
9 import argparse
10
11 import matplotlib.pyplot as plt # fades
12 import numpy as np # fades
13 from scipy.integrate import solve_ivp # fades
```

Allí podemos ver en la línea 1 que ahora no estamos ejecutando el script con Python sino con Fades (que llamará a Python luego de disponer el entorno virtual), y en las líneas 11 a 13 que le estamos diciendo a Fades que necesitamos las dependencias para esos imports. El caso de PyQt5 es distinto, porque no tenemos un import (lo utiliza matplotlib, no nosotros directamente), entonces le indicamos esa dependencia en el docstring del módulo, en la línea 6.

Una vez modificado el script de esta manera, ya no tenemos que recordar nada sobre las dependencias del script, archivos adjuntos, entrar a directorios en particular, o activar y desactivar entornos. Directamente ejecutamos el script:

```
[/temp]$ ./oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
```

```
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Fades tiene muchas otras opciones (algunas útiles tambien para utilizarlo en proyectos grandes, no sólo scripts), recomendamos explorarlas con `fades --help`, revisar la documentación o mirar [algunas animaciones](#) que muestran las funcionalidades más comunes.

1.2.2.4. uv

uv es la más moderna de estas herramientas, focalizada en la velocidad de ejecución y en la intención de reemplazar todo el espectro de herramientas que de un modo u otro manejan entornos virtuales.

Por ejemplo, para crear un entorno virtual e instalar dependencias:

```
$ uv venv xcss
Using CPython 3.12.3 interpreter at: /usr/bin/python3
Creating virtual environment at: xcss
Activate with: source xcss/bin/activate
$ source xcss/bin/activate
$ (xcss) $ uv pip install requests
Using Python 3.12.3 environment at: xcss
Resolved 5 packages in 50ms
Installed 5 packages in 7ms
+ certifi==2025.4.26
+ charset-normalizer==3.4.2
+ idna==3.10
+ requests==2.32.3
+ urllib3==2.4.0
```

Es extremadamente rápido. Cabe acotar, sin embargo, que a veces no instala las mismas versiones que instalaría pip, ya que algunas opciones de instalación tiene defaults distintos de la herramienta original.

Es muy práctico porque tiene integrada la posibilidad de usar otro Python que no sea el del sistema:

```
$ uv venv xcs7 --python 3.7
Using CPython 3.7.9
Creating virtual environment at: xcs7
Activate with: source xcs7/bin/activate
```

En estos casos uv descargará el binario de Python que sea necesario y lo utilizará para armar el entorno. Hay que tener la precaución que uv utiliza binarios de Python pre-armadas por la empresa responsable de la herramienta (Astral), los cuales tienen algunas diferencias de comportamiento con el Python oficial, generalmente como una consecuencia de la portabilidad buscada.

Una de las grandes ventajas de uv es que soporta la [PEP 723](#), una formalización/generalización de la idea de Fades de declarar dentro del script qué dependencias deben instalarse para su funcionamiento.

Veamos el mismo ejemplo de antes, el oscilador, pero con esta metadata. Las primeas líneas entonces son:

```
1 #!/usr/bin/env python3
2 """Programa que integra el sistema masa-resorte-banda elástica."""
3
4 # /// script
5 # dependencies = [
6 #     "matplotlib",
7 #     "numpy",
8 #     "scipy",
9 # ]
10 # ///
11
12 import argparse
13
14 import matplotlib.pyplot as plt
15 import numpy as np
16 from scipy.integrate import solve_ivp
```

Entonces, luego:

```
$ uv run oscilador_con_pep723.py
Reading inline script metadata from `oscilador_con_pep723.py`
Installed 12 packages in 24ms
Opciones del script: Namespace(usar_tex=False, archivo=None)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Recomendamos revisar [la documentación correspondiente](#) para más opciones.

1.2.2.5. pipenv

Pipenv es una herramienta que apunta al manejo automático de los entornos virtuales más a nivel de proyectos. Con sólo indicarle que instale las dependencias del proyecto, creará automáticamente (de no tenerlo de antes) un entorno virtual para el proyecto en una ubicación específica, e instalará las dependencias allí.

```
[/temp/proyecto]$ pipenv install -r requirements.txt
Creating a virtualenv for this project...
Using /usr/bin/python3 (3.8.6) to create virtualenv...
  created virtual environment CPython3.8.6.final.0-64 in 104ms
creator CPython3Posix(dest=/home/facundo/.local/share/virtualenvs/trunk-qJfLQWAx, clear=False,
  ↪ global=False)
Seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy,
  ↪ app_data_dir=/home/facundo/.local/share/virtualenv)
added seed packages: pip==20.1.1, pkg_resources==0.0.0, setuptools==44.0.0, wheel==0.34.2
activators
  ↪ BashActivator,CShellActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator
Virtualenv location: /home/facundo/.local/share/virtualenvs/trunk-qJfLQWAx
Creating a Pipfile for this project...
Requirements file provided! Importing into Pipfile...
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
```

```
Updated Pipfile.lock (00d0fa)!  
Installing dependencies from Pipfile.lock (00d0fa)...  
█ ━━━━━━ 4/4 - 00:00:04  
To activate this project's virtualenv, run the following:  
$ pipenv shell
```

Tengamos en cuenta que necesitamos recordar de activar el entorno virtual antes de trabajar con nuestro proyecto:

```
[/temp/proyecto]$ pipenv shell  
Spawning environment shell (/bin/bash). Use 'exit' to leave.  
. /home/facundo/.local/share/virtualenvs/trunk-qJfLQWAX/bin/activate  
/temp/proyecto$ . /home/facundo/.local/share/virtualenvs/trunk-qJfLQWAX/bin/activate  
[(trunk-cLuRPqaE) /temp/proyecto]$
```

Vemos en la última línea como estamos “dentro” del entorno virtual, como veíamos al principio con otras herramientas.

Luego de ejecutar el `install` encontramos que Pipenv nos dejó dos archivos nuevos: el `Pipfile` y el `Pipfile.lock`. Estos son archivos de control que usa Pipenv mismo para proveer la funcionalidad de entornos virtuales armados determinísticamente en distintos sistemas de trabajo. La idea es que en desarrollo especificamos las dependencias necesarias en un `requirements.txt` o similar, y Pipenv armará sus dos archivos de control; luego cuando envíemos el proyecto a otros sistemas (integración continua, servidores de producción, etc.) los entornos virtuales que se armarán en esos casos serán *exactamente iguales* que los que armamos en desarrollo, evitando posibles inconvenientes (como por ejemplo que una dependencia cambió de versión entre que probamos el proyecto localmente y que lo subimos a un servidor).

Finalmente, también para esta herramienta, recomendamos revisar [la documentación correspondiente](#).

1.3. Contenedores

Un “contenedor” es una forma de empaquetar software: el código que necesitamos para ejecutar una aplicación más todas las dependencias que necesite, incluidas las que provee el sistema operativo en la que se basa el contenedor.

Estos contenedores nos proveen el aislamiento y la repetibilidad que buscamos, pero no sólo a nivel de código Python, sino a nivel de sistema, ya que está bien separado lo que se ejecuta adentro del contenedor, del sistema donde ese contenedor corre (al que llamamos “host” o “anfitrión”). Podemos ejecutar muchos contenedores simultáneamente en el mismo anfitrión, ya que cada uno tiene todo lo que se necesita para ejecutar una determinada aplicación. Más aún, podemos compartir fácilmente estos contenedores, de manera que otras personas pueden ejecutar el mismo contenedor en otros anfitriones, con idénticos resultados.

Hay un estándar para la creación y manejo de contenedores, la [Open Container Initiative](#), pero hay que reconocer que la herramienta pionera en este concepto es [Docker](#) (y sobre la cual está fuertemente basada el estándar). Docker es un administrador de “contenedores”, entonces, y es la herramienta que vamos a utilizar en este texto. Tengan en cuenta que lo más probable

es que Docker todavía no esté en vuestro sistema, pueden instalarlo siguiendo [las instrucciones oficiales](#).

De manera similar a lo que vimos antes con los entornos virtuales, vamos a explorar cómo armar contenedores con las dependencias que necesitamos, y cómo ejecutar una aplicación dentro de este nuevo entorno. Además, vamos a ver cómo compartir estos entornos.

Una palabra sobre Kubernetes: hay muchos textos que mezclan un poco Docker y Kubernetes. Sin entrar en mayor detalle, expliquemos un poco qué es Kubernetes, para evitar la confusión. Mientras que Docker hace foco en la creación y distribución de contenedores, Kubernetes trabaja el concepto de organizar muchos contenedores para el despliegue de un servicio complejo en algún servidor (por ejemplo, mientras que un contenedor puede incluir a una base de datos, Kubernetes organizará ese contenedor, más otros con servidores web, y otros con sistemas de autenticación, para tener un servicio web completo funcionando). Kubernetes se escapa del ámbito de este libro, donde la idea es poder compartir aplicaciones, y no desplegar servicios en la nube para consumo de terceros. Pero si les interesa explorar este mundo, pueden empezar por la [página oficial](#).

1.3.1. ¿Imágenes o contenedores?

Hasta ahora estuvimos hablando de contenedores de forma genérica, porque privilegiamos hablar sobre las ventajas de los mismos (especialmente en el contexto de este capítulo) y no mezclar el tema de las “imágenes”.

Pero entremos en este detalle ahora, antes de ir a la parte más práctica, para separar bien los conceptos y poder entender mejor las próximas subsecciones. Empecemos por las imágenes, ya que sin imágenes no vamos a poder tener contenedores.

Una imagen es un archivo binario en disco, algo estático, que contiene todo lo necesario para poder ejecutar nuestra aplicación cuando corramos el contenedor.

No entraremos mucho en el detalle de la composición interna de las imágenes, pero sí es útil saber que están compuestas por “capas” (que incluso pueden reutilizarse entre imágenes), ya que es un concepto importante para entender distintos comportamientos a la hora de construir o descargarse imágenes.

Podemos crear las imágenes desde cero (como veremos abajo, con los archivos *Dockerfile*) o pueden ser una “foto” de un contenedor que está corriendo.

Los contenedores, por otro lado, son la “ejecución” de esas imágenes. A partir de una imagen podemos crear muchos contenedores similares (no digo idénticos, porque cada ejecución es independiente).

Por ejemplo, podemos tener nuestra aplicación en una imagen de Docker, junto con todo lo necesario para que esa aplicación corra como corresponde. Luego, podemos crear 10 contenedores distintos a partir de esa imagen, y ejecutar nuestra aplicación con 10 conjuntos distintos de datos. Cada contenedor será independiente del otro, más allá que hayan sido creados a partir de la misma imagen.

1.3.2. Creando imágenes

La forma primera de crear una imagen es a partir de un documento de texto (llamado *Dockerfile*) que contiene todas las órdenes que un usuario podría ejecutar en una línea de comandos.

En este archivo especificaremos el “sistema base” sobre el cual trabajaremos, y luego distintas órdenes que irán construyendo la imagen (instalando dependencias, ajustando configuraciones, etc.).

El Dockerfile tiene su complejidad, y hay órdenes de todo tipo (pueden revisar [su referencia](#)), pero las más usadas son las siguientes, que mencionaremos brevemente para poder continuar con el resto del contenido, obviamente sin entrar en todo el detalle o potencial de cada una:

- **FROM:** define la imagen base para el resto de las instrucciones (el Dockerfile *debe* comenzar con esta orden); la imagen indicada puede ser cualquiera, local o remota, al principio es muy útil usar alguna de un repositorio público (como [Dockerhub](#)).
- **RUN:** ejecuta un comando de la misma manera que si lo estuvieramos ejecutando en el sistema que estamos armando.
- **ENTRYPOINT:** el comando que ejecutará Docker cuando le hagamos run a la imagen, luego de crear el contenedor en sí; la forma preferida de escribir el comando aquí es usando una lista de cadenas (como `["python3", "script.py"]`) y no directamente `python3 script.py`, ya que esto último lo interpreta el shell del contenedor y no siempre sucederá lo que esperamos. Si queremos pasarle parámetros y opciones al script que estamos ejecutando podemos ponerlo como parte del **ENTRYPOINT**, pero se recomienda usar otra orden para ello que es **CMD** ya que de esta manera podremos luego modificar esos parámetros y opciones al momento de ejecutar el contenedor.
- **ENV:** configura variables de entorno para todas las instrucciones que se ejecutarán a continuación.
- **COPY:** copia archivos y directorios desde nuestro sistema al sistema de archivos de la imagen en el destino especificado.
- **WORKDIR:** cambia el directorio actual para todas las órdenes que vienen a continuación.

El Dockerfile, entonces, es básicamente una secuencia de órdenes, cada una en una línea. Cada orden terminará armando una de las capas que mencionábamos antes, lo cual es importante ya que una vez creada la capa Docker la puede reutilizar en el futuro cuando rearremos la imagen.

Docker utilizará este archivo y nos dejará como resultado una imagen según lo especificado. Veamos un ejemplo con el mismo programa que utilizamos cuando aprendimos entornos virtuales (aunque lo ejecutaremos distinto: en lugar de que nos abra una ventana con el resultado, haremos que grabe esa imagen resultado a disco ²).

Nuestro Dockerfile, para el ejemplo, será el siguiente:

```
1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # instalamos las dependencias que necesitamos directamente desde PyPI
5 RUN pip install matplotlib numpy scipy
```

² Esto es porque para abrir una ventana el proceso de adentro del container se tiene que comunicar con el servidor de ventanas del host, lo cual no sólo no es sencillo, sino que depende del sistema operativo de nuestro host, y esta configuración avanzada ya se escapa del alcance del libro.

```
6  
7 # indicamos que vamos a trabajar en este directorio (se crea automáticamente)  
8 WORKDIR /oscilador/  
9  
10 # descargamos el script de prueba desde el github del libro mismo  
11 RUN wget https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/código/entornos/oscilador.py  
12  
13 # indicamos que el punto de entrada para cuando corramos el contenedor es el  
14 # script, con los parámetros por default que indican que guarde la imagen  
15 # resultante como un PDF  
16 CMD ["--archivo=imagen.pdf"]  
17 ENTRYPOINT ["python3", "oscilador.py"]
```

El siguiente paso es construir nuestra imagen. Para ello creamos un directorio nuevo y ponemos una copia de este Dockerfile, ya que a docker build le tenemos que pasar un directorio donde exista un Dockerfile (en nuestro caso será vacío, porque es un ejemplo y el script que vamos a probar lo bajamos directamente de la web, pero en la mayoría de las situaciones reales el directorio alojará a todo el proyecto que intentamos encapsular).

Ejecutamos el build, entonces, indicando el directorio actual y usando -t para indicar el nombre de la imagen creada:

```
$ docker build -t app-ejemplo .  
Sending build context to Docker daemon 2.56kB  
Step 1/5 : FROM python:3.9  
...  
Successfully built 48b2a52af8d  
Successfully tagged app-ejemplo:latest
```

Podemos ver como Docker tiene nuestra imagen recién creada:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
app-ejemplo	latest	48b2a52af8d	About a minute ago	1.16GB

1.3.3. Ejecutando contenedores

Para probar que lo que hicimos hasta ahora funciona correctamente, ejecutaremos un contenedor a partir de la imagen recién creada.

```
$ docker run app-ejemplo  
Opciones del script: Namespace(usar_tex=False, archivo='imagen.pdf')  
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75  
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Docker entonces creó el contenedor a partir de nuestra imagen, y ejecutó el script tal como indicamos en el Dockerfile con ENTRYPOINT. Cuando nuestro script terminó, automáticamente el contenedor también paró.

Podemos validar que no nos quedó un contenedor corriendo usando ps.

Contenedores

§ 1.3

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

En otras situaciones, donde el contenedor encapsula a una aplicación que queda ejecutándose (por ejemplo, una aplicación web), veremos al contenedor en ese listado.

Podemos simular este comportamiento con una aplicación Python que tenga una demora artificial, como ejemplo. Vayamos a otro directorio y pongamos allí nuestra aplicación en un archivo `demorona.py`:

```
1 import time
2
3 print(time.ctime())
4 time.sleep(30)
5 print(time.ctime())
```

En ese mismo directorio creamos el Dockerfile para esta aplicación:

```
1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # copiamos el archivo que necesitamos desde nuestro entorno a la imagen
5 COPY demorona.py .
6
7 # indicamos que ejecute ese script cuando corramos el contenedor; notar como ejecutamos
8 # Python con '-u', lo que indica que no use buffers para la salida (porque como vamos a
9 # imprimir sólo dos líneas, veríamos ambas al final del proceso)
10 ENTRYPOINT ["python3", "-u", "demorona.py"]
```

Y creamos la imagen con `docker build -t demorona .`; si ejecutamos entonces la imagen creada y esperamos los 30 segundos, veremos una salida similar a la siguiente:

```
$ docker run demorona
Wed Jul 18 20:17:20 2022
Wed Jul 18 20:17:50 2022
```

Pero si en vez de quedarnos esperando entre una línea y la otra, vamos a otra terminal y antes de que termine miramos los procesos corriendo, vamos a encontrar nuestro contenedor:

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS ...
74f26f383fb9 demorona "python3 -u demorona.py" 13 seconds ago Up 12 seconds ...
```

Volvamos a la ejecución de nuestra aplicación ejemplo (el oscilador). Como le indicamos en el Dockerfile, la opción por default al ejecutar el script es que guarde el resultado en un archivo. Obviamente queremos obtener ese archivo, ya que es el resultado del trabajo.

Esto es sencillo usando el comando `cp` de `docker`, que copia archivos y directorios desde/a contenedores. En este caso queremos traer `imagen.pdf` desde el contenedor, pero no sabemos *cual* es el contenedor en cuestión.

Veamos entonces *todos* los contenedores (incluso los parados), para encontrar el que necesitamos:

```
$ docker ps --all
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS          ...
6c3589d2933d   app-ejemplo   "python3 oscilador.p..."   An hour ago   Exited a minute ago ...
```

Luego, copiamos el archivo en cuestión. El comando `cp` (como otros comandos similares para copiar) necesita que le especifiquemos desde dónde y hasta dónde, con el detalle que tenemos que indicar el *ID* del contenedor separado por dos puntos:

```
$ docker cp 6c3589d2933d:/oscilador/imagen.pdf .
$ ls imagen.pdf
imagen.pdf
```

Si quisieramos dejar la imagen resultado en un archivo con otro formato, podemos aprovechar la flexibilidad que incluimos en el Dockerfile, donde el *entrypoint* está separado de los argumentos que recibe por default. Entonces, si especificamos parámetros al ejecutar el contenedor, estos reemplazarán ese default, lo cual podemos ver de forma sencilla para nuestro ejemplo porque el mismo muestra las opciones que recibió:

```
$ docker run app-ejemplo --archivo=imagen.png
Opciones del script: Namespace(usar_tex=False, archivo='imagen.png')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

1.3.4. Valor extra de los contenedores como entornos aislados

Incluyamos en esta subsección una complejidad que estábamos evitando a propósito y que para resolverla vamos a explotar la funcionalidad clave de los contenedores con respecto a los entornos virtuales.

Si miramos la imagen que obtuvimos en la subsección anterior (o la que nos mostraban las ventanas cuando estábamos con los entornos virtuales) veremos que los nombres de los ejes están escritos con tipografía “normal”. En realidad podríamos aprovechar que `matplotlib` es capaz de utilizar `LATEX`en determinadas situaciones, y nuestro script está preparado para ello (le tenemos que pasar la opción `--usar-tex`).

La complejidad está en que `matplotlib` utiliza el sistema `LATEX`del sistema para escribir los nombres de los ejes con la tipografía elegante que queremos. Tenemos que tener instalados las utilidades correspondientes en el sistema, y esto es algo que se escapa de los entornos virtuales, que sólo manejan las dependencias a nivel de Python, y es justamente donde podemos hacer brillar a los contenedores, ya que podemos incluir en los mismos no sólo las dependencias a nivel de Python sino también las externas al mismo.

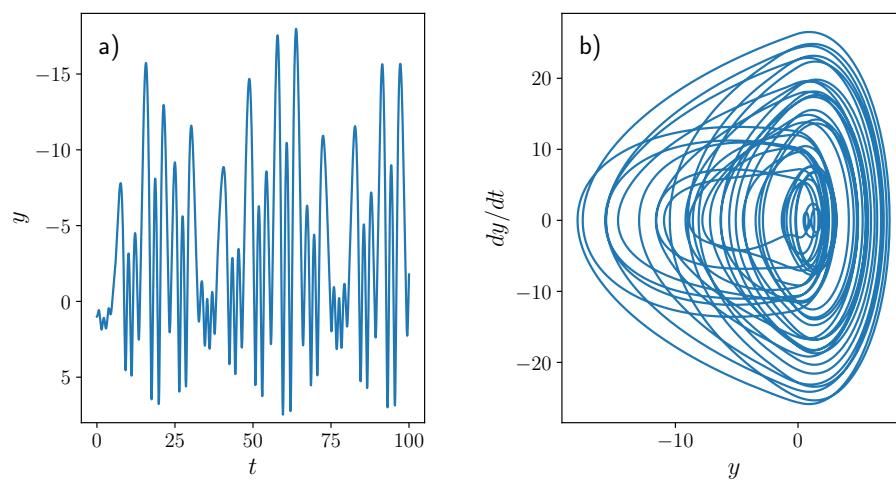
Claro, para hacer eso necesitamos instalar paquetes a nivel del sistema, lo cual dependerá de cual estamos usando como base en el contenedor. En nuestro caso estamos usando `python:3.9`, que está basado en Debian, así que tenemos que instalar los paquetes necesarios en ese sistema usando el administrador de paquetes `apt`. El Dockerfile nos quedaría:

```
1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # instalamos los paquetes necesarios para que LaTeX funcione ok
5 RUN apt update && apt install -y texlive-latex-extra cm-super dvipng
6
7 # instalamos las dependencias que necesitamos directamente desde PyPI
8 RUN pip install matplotlib numpy scipy
9
10 # indicamos que vamos a trabajar en este directorio (se crea automáticamente)
11 WORKDIR /oscilador/
12
13 # descargamos el script de prueba desde el github del libro mismo
14 RUN wget https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/código/entornos/oscilador.py
15
16 # indicamos que el punto de entrada para cuando corramos el contenedor es el
17 # script, con los parámetros por default que indican que guarde la imagen
18 # resultante como un PDF y use LaTeX
19 CMD ["--archivo=imagen.pdf", "--usar-tex"]
20 ENTRYPOINT ["python3", "oscilador.py"]
```

Atentos al detalle que cambiamos el argumento por default del script: ahora estamos incluyendo que use `\LaTeX`. Creamos la imagen como vimos anteriormente, y lo probamos:

```
$ docker run app-ejemplo
Opciones del script: Namespace(usr_tex=True, archivo='imagen.pdf')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large', 'text.usetex': True}
```

Y este es el resultado que nos trajimos del contenedor, donde podemos ver la elegancia obtenida:



1.3.5. Compartiendo imágenes

Una vez que tenemos nuestra imagen construida y probada, es momento de compartirla.

Para esto, como adelantamos antes, necesitamos un repositorio a donde subir la imagen y desde donde otras personas puedan bajarla. El más usado al momento de escribir este libro es [Dockerhub](#), una comunidad/biblioteca ofrecida por Docker mismo de imágenes de contenedores.

Obviamente, para utilizar ese recurso *online* primero deberemos preparar todo adecuadamente. Los pasos son sencillos, sin embargo (aunque sus detalles se escapan de el alcance de este libro): hay que crear una cuenta en dicho sitio, y luego crear un repositorio público.

Por ejemplo, para los casos que mostramos en este libro, el repositorio público usado es [facundobatista/app-ejemplo](#).

Una vez lista la parte remota, tenemos que configurar apropiadamente la parte local. La herramienta que utilizaremos para interactuar con Dockerhub es el mismo docker que veníamos usando antes. Tenemos que indicarle que se autentique:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: facundobatista
Password:
Login Succeeded
```

Nos falta un último paso, que es renombrar la imagen que armamos para que coincida con el repositorio remoto para la misma. Tengamos en cuenta que podríamos haberla nombrado apropiadamente al crearla (cuando usamos la opción `-t` en el `build`), pero tampoco es un incordio realizar ese paso a posteriori, usando `docker tag`:

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
app-ejemplo     latest   d727e49a4b61   2 days ago   1.83GB
$ docker tag app-ejemplo facundobatista/app-ejemplo
```

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
facundobatista/app-ejemplo    latest   d727e49a4b61  2 days ago   1.83GB
```

Entonces sólo nos queda enviar la imagen:

```
$ docker push facundobatista/app-ejemplo
Using default tag: latest
The push refers to repository [docker.io/facundobatista/app-ejemplo]
43413aee136b: Pushed
...
latest: digest: sha256:6de9f25c02e0ae924f41d3273e01e56939366eef6d830b27e19fcc17818dde04 size: 3058
```

Ya está. A partir de este momento cualquier persona puede descargar la imagen que subimos (referenciándola de la misma manera) y ejecutarla:

```
$ docker pull facundobatista/app-ejemplo
Using default tag: latest
latest: Pulling from facundobatista/app-ejemplo
30a9a79b0d7e: Pull complete
...
Digest: sha256:6de9f25c02e0ae924f41d3273e01e56939366eef6d830b27e19fcc17818dde04
Status: Downloaded newer image for facundobatista/app-ejemplo:latest
docker.io/facundobatista/app-ejemplo:latest
$ docker run facundobatista/app-ejemplo
Opciones del script: Namespace(usr_TEX=True, archivo='imagen.pdf')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large', 'text.usetex': True}
```

Esto es algo que mismo ustedes pueden probar con sólo tener docker instalado, ya que es anónimo: no hace falta sacar una cuenta en Dockerhub o autenticar docker para utilizar una imagen pública.

Parte II

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III

Métodos numéricos y técnicas computacionales

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [1].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora mismo*.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

- [1] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.