PythonBasicCourse-es001

October 14, 2022

1 Curso básico de Python

1.1 Apuntes

Curso básico de Python. Apuntes por Marcelo Horacio Fortino. Versión 2.3. Octubre 2022.

Versión	Autor/es	Fecha	Observaciones
1.0	Marcelo Horacio Fortino	2021/Marzo	Curso Python
1.1	Marcelo Horacio Fortino	2021/Junio	Convertido a markdown - ipynb
1.2	Marcelo Horacio Fortino	2021/Agosto	Actualizados contenidos
1.3	Marcelo Horacio Fortino	2021/Octubre	Agregado Flask microframework
1.4	Marcelo Horacio Fortino	2021/Noviembre	Agregado Pandas - datascience
1.5	Marcelo Horacio Fortino	2021/Diciembre	Agregado Devops - Ansible
2.0	Marcelo Horacio Fortino	2022/Abril	Nueva estructura: core / module
2.1	Marcelo Horacio Fortino	2022/Junio	Módulo apuntes intermedio
2.2	Marcelo Horacio Fortino	2022/Agosto	Actualizado temario y ejercicios
2.3	Marcelo Horacio Fortino	2022/Octubre	Actualizado despliegue a render.com

cluso sin la garantía MERCANTIL implícita o sin garantizar la CONVENIENCIA PARA UN PROPÓSITO PARTICULAR. El autor no asume ninguna responsabilidad si el lector hace un mal uso de la misma.

- Estos apuntes se basan en:
  - La documentación oficial de Python, https://docs.python.org/es/3/tutorial/index.html,
  - La bibliografía presentada al final de este documento, y
  - Documentación propia recogida a lo largo de los años de diversas fuentes.

1.2 Objetivo del curso

- Objetivo general del curso:
- Aprender a usar Python para crear scripts y programas simples plenamente funcionales, desarrollar aplicaciones web y realizar análisis de datos.

- Objetivos específicos:

- Reconocer las características principales de Python y su utilidad práctica.
- Identificar tipos de datos (simples y compuestos) y operadores.
- Aplicar variables y estructuras de control de flujo.
- Construir funciones y clases (POO).
- Clasificar los módulos y paquetes por sus funcionalidades y objetivos.
- Comparar el lenguaje con otros similares de scripts, procedimentales y orientados a objetos.
- Probar bibliotecas para conexiones REST a aplicaciones web y bases de datos.
- Resolver problemas y errores en el código fuente proponiendo soluciones alternativas (refactoring).
- Utilizar la biblioteca pandas junto con matplotlib y numpy para realizar análisis estadísticos de datos y gráficos.

- Como resultado práctico al final del curso cada estudiante habrá creado una aplicación web utilizando el microframework de Python Flask.

1.3 Temario

- Introducción, instalación y compilación
- Datos, expresiones y sentencias
- Variables y funciones, control de flujo
- Clases y objetos, herencia, polimorfismo
- Entradas y salidas con Python
- Gestión de módulos, paquetes y bibliotecas
- Servicios y programas en red, REST API
- Desarrollo de aplicaciones web con Flask
- Análisis de datos con pandas, matplotlib y numpy

- Módulos opcionales
  - SQL ejemplos con pandas
  - Plotting con Python
  - Machine Learning con Python
  - DevOps con Ansible
  - SDK de GCP para Python

  - Kubernetes con Python y Docker

1.4 Bibliografía

- Downey, A., Elkner, J., Meyers, C. Aprenda a Pensar Como un Programador con Python. (2015).
  Recuperado de https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf
- Kent D. Lee. Python, Programming Fundamentals Second Edition. 2014.
- Marzal Varó, A., Gracia Luengo, I., García Sevilla, Pedro. Introducción a la programación con Python 3. (2014).
  Recuperado de http://repositori.uji.es/xmlui/handle/10234/102653
- Miller, B., Ranum, D. Solución de problemas con algoritmos y estructuras de datos usando Python. Traducido por Mauricio Orozco-Alzate, Universidad Nacional de Colombia - Sede Manizales.
  Recuperado de https://runestone.academy/ns/books/published/pythoned/index.html?mode=browsing
- Shaw, Z. A., Learn Python 3 the Hard Way. (2016).
  Recuperado de https://learnpythonthehardway.org/
- Van Rossum, G. and the Python development team. Documentación de Python en español. (2020).
  Recuperado de https://python-docs-es.readthedocs.io/_/downloads/es/pdf/pdf/

2 Introducción a Python

- Conceptos generales de programación, El lenguaje Python, Versiones de Python, Principales ventajas de Python, Características de Python, Aplicaciones y uso, La función print, Palabras reservadas, Propuestas de mejoras y buenas prácticas - PEP, Instalación de Python, Creación de ambientes virtuales, La biblioteca virtualenv, IDE - modo interactivo, Interpretado en fichero, Compilado a bytecode, Compilado a ejecutable del sistema.

2.1 Conceptos de programación

- Los ordenadores tienen su propio lenguaje llamado *lenguaje máquina*.
- Está compuesto por una lista de instrucciones (IL) conforme al estándar IEC 61131-3.
- https://es.wikipedia.org/wiki/IEC_61131-3
- La IL es parecida al lenguaje ensamblador o *assembly*.
- Todo lenguaje (máquina o natural) consta de los siguientes elementos:
  - Un alfabeto: conjunto de símbolos utilizados para formar palabras.
  - Un léxico o diccionario: conjunto de palabras.
  - Una sintaxis: conjunto de reglas para formar cadenas de palabras.
  - Una semántica: conjunto de reglas para determinan si una frase tiene sentido.
- Tanto el código máquina como el ensamblador son denominados lenguajes de bajo nivel porque interactúan directamente con el hardware.
- La complejidad para programar en estos lenguajes hace que existan lenguajes más “amigables” al usuario categorizados como de alto nivel.

- Estos lenguajes permiten al programador enviar con una sola declaración decenas de instrucciones al ordenador.
- El programa escrito en un lenguaje de programación de alto nivel se llama código fuente.
- El fichero que contiene el código fuente se llama archivo fuente.
- Los lenguajes diseñados para ser interpretados se llaman lenguajes de *scripting* y su código fuente, *scripts*.
- Los lenguajes de alto nivel por sus características a su vez se pueden clasificar como:
- Imperativos: procedimentales (Fortran, ALGOL, COBOL, PL/I, BASIC, Pascal, C),
- Orientados a objetos - POO (Smalltalk, C++, PHP, Java), y
- Declarativos: funcionales (Haskell), lógicos (Prolog), matemáticos, y reactivos.

Fuente: https://en.wikipedia.org/wiki/Programming_paradigm.

- Igualmente hay lenguajes de programación que soportan múltiples paradigmas, como pueden ser Pascal, C++, Java, JavaScript, Scala, Visual Basic, Common Lisp, Perl, PHP, Python y Ruby, entre otros.
- Es el programador quien decide cuáles y como utilizar los elementos del paradigma.
- Los métodos mas comunes para transformar un programa de un lenguaje de programación de alto nivel a un lenguaje de máquina son:
  - COMPILACIÓN: se genera un fichero ejecutable a partir del código fuente.
  - INTERPRETACIÓN: el programa fuente se lee o interpreta cada vez que se ejecuta.
- Lenguajes compilados: C, Pascal, C++…
- Lenguajes interpretados: Bash, Python, Perl, PHP, Ruby…
- Un programa es una secuencia de instrucciones que especifican cómo se debe realizar una tarea en un dispositivo (ordenador, mainframe, smartphone, etc.) o en una infraestructura distribuida tipo Cloud computing.
- Esta tarea puede ser por ejemplo buscar y reemplazar texto, realizar una operación matemática, o procesar una imagen, entre muchas otras.

Las instrucciones básicas son similares en todos los lenguajes de programación:

- entradas (obtener datos de un teclado, fichero, etc.),
- salidas (mostrar datos en pantalla, imprimirlos, guardarlos, etc.),
- operaciones matemáticas (cálculos),
- ejecución condicional, y
- repetición.
- Un aspecto importante a tener en cuenta es que un lenguaje de programación es literal y evita la ambigüedad y la redundancia.
- Cuando surgen errores, el proceso de eliminarlos se llama depuración (*debugging*).
- Para comprobar errores de programación:
  - El compilador al encontrar un error termina su trabajo inmediatamente y muestra un mensaje de error.
  - El intérprete lee y verifica la corrección de las líneas de código e informa dónde se encuentra el error y qué lo causó.

- Los métodos de desarrollo del software en cascada (waterfall) suelen dividir la tarea de programación en etapas: Especificación, Diseño, Implementación, Validación, y Mantenimiento.
- Los métodos ágiles por su parte desarrollan en ciclos cortos iterando y de forma incremental.

2.2 El lenguaje Python

- Python es un lenguaje de programación interpretado y de alto nivel.
- Soporta múltiples paradigmas de programación, entre ellos la programación orientada a objetos (POO).
- Fue creado por Guido van Rossum a finales de los '80 y actualmente es desarrollado y mantenido por la Python Software Foundation (PSF) https://www.python.org/psf/.
- La PSF realiza entre otras cosas, las célebres conferencias Pycon alrededor del mundo https://pycon.org/.

- Su nombre proviene del programa de televisión de la BBC *Monty Python's Flying Circus*.
- La documentación oficial se encuentra en: https://docs.python.org/es/3/.

- Estadísticas:
  - PYPL PopularitY of Programming Language https://pypl.github.io/PYPL.html
  - TIOBE Programming Community Index https://www.tiobe.com/tiobe-index/
  - Stackoverflow https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

2.3 Versiones de Python

- La primera versión de este lenguaje (0.9.0) fue presentada en 1991.
- La versión 1.4 a su vez en 1994, y la 2.0 en el año 2000.
- La última versión de la serie 2.0 (2.7) llegó recién en 2010.
- De esta versión aún se conservan numerosas aplicaciones en producción.

- De manera simultánea a la versión 2, fue lanzada en 2008 la versión 3.0 para resolver ciertas fallas del diseño del lenguaje y además, *"como Python ha acumulado nuevas y redundantes formas de programar la misma tarea, Python 3.0 ha hecho énfasis en eliminar constructores duplicados y módulos, en consecuencia con 'Debe haber un— y preferiblemente solo un —modo obvio de hacerlo'."*

- El principio rector de Python 3 es: "*reducir la duplicación de funciones eliminando las antiguas formas de hacer las cosas.*"

- Fuente: https://es.wikipedia.org/wiki/Historia_de_Python

- Python 2 no es compatible con la versión 3 y además ha dejado de ser mantenida en enero del 2020.
- Al momento de actualizar este texto (Agosto 2022) la última versión estable de Python es la 3.10.

- Un ejemplo de la diferencia entre Python 2 y Python 3:

[ ]:
```
# Python 2 print statement
print '¡Hola mundo!'
# Python 3 print function
```

```
print('¡Hola mundo!')
```

2.4 Principales ventajas de Python

- Código abierto.
- Interpretable y compilable.
- Sintaxis simple y elegante.
- Gran colección de módulos estándar.
- Es un lenguaje maduro (+30 años).
- Extensible e integrable en otros lenguajes (C, C++, Java).
- Existen multitud de recursos para su aprendizaje.

2.5 Características de Python

- Interpretado, aunque también se puede compilar.
- De alto nivel: Abstacción de los detalles del SO.
- Soporta múltiples paradigmas de programación: estructurado (particularmente, procedimental), POO y programación funcional.
- Usa tipado dinámico: una variable puede tomar valores de distinto tipo.
- Es fuertemente tipado: el tipo no cambia de manera repentina, se tiene que hacer una conversión explícita.
- Multipropósito.
- Multiplataforma: Linux, Unix, macOS, Windows, etc.

- Python alias CPython: Guido van Rossum utilizó el lenguaje de programación "C" para implementar la primera versión de Python y todas las versiones de la PSF están escritas en ese lenguaje.
- Esto permite portarlo fácilmente a todas las plataformas con capacidad de compilar y ejecutar programas en lenguaje "C" (virtualmente todas las plataformas tienen esta característica).

- Otras versiones:
  - Cython: traduce automáticamente el código de Python a "C".
  - Jython: se comunica con la infraestructura Java existente de manera más efectiva.
    - La implementación actual de Jython sigue los estándares de Python 2. no hay Jython conforme a Python 3.

- PyPy es un entorno de Python escrito en RPython (*Restricted Python*).
- Es un subconjunto de Python. El código fuente de PyPy no se ejecuta de manera interpretativa, sino que se traduce al lenguaje de programación "C" y luego se ejecuta por separado.
- Sirve para probar características nuevas de Python.

- Python usa una nueva línea para completar un comando.
- Utiliza el indentado y espacios en blanco para definir el alcance (*scope*) de las iteraciones, funciones y clases.
- Su sintaxis es muy parecida al lenguaje natural o pseudocódigo.

[ ]:
```
# Ejemplo de variable
variable = 10    # comentario
```

[ ]:
```
# Ejemplo de función
def funcion():
    x = 5
    return x
```

[ ]:
```
# Función print()
print(variable)    # imprime variable
print(funcion())    # imprime función
```

2.6 Aplicaciones y uso

- Desarrollo Web e Internet (Django, Flask)
- Acceso a bases de datos (SQLite3, SQLAlchemy ORM)
- Entornos gráficos de escritorio (Kivy, PQT5, Tkinter)
- Ciencia de datos (Pandas, NumPy, Matplotlib)
- Machine Learning (TensorFlow, Scikit Learn, PyTorch, Keras)
- Educación (Thonny)
- Desarrollo de software (Anvil)
- Desarrollo de juegos (Pygame, Pyglet, PyKyra, Panda3D)

- Python por sus características no suele usarse para:
  - Programación de bajo nivel (controladores, motores gráficos, etc.).
  - Aplicaciones para dispositivos móviles (Android no soporta Python apps nativas).

2.7 La función print

- Una función es un código de programación, a veces separado del código principal, que permite:
  - Enviar texto a la terminal, crear un archivo, etc.
  - Evaluar un valor (la suma de valores, la longitud de un texto, etc.) y devolverlo como resultado.

- Python cuenta con funciones integradas como por ejemplo *print*.
- Pueden provenir de los módulos integrados o instalados de Python llamados complementos.
- Se pueden programar específicamente.

- Las funciones de Python pueden aceptar cualquier número de argumentos, incluso ningún argumento.
- Necesitan de un par de paréntesis aún cuando no haya argumentos.
- Se usan comillas para tomar de manera literal (no como código) los datos.
- El nombre de la función junto con los paréntesis y argumentos, forman la invocación de la función.

[ ]:
```
print("Hola mundo!")
```

- Python explora sus datos internos para encontrar una función con ese nombre.
- Comprueba si los requisitos de la función permiten invocar la función de esta manera.
- Dentro de la función que se desea invocar toma los argumentos y los pasa a la función.
- Ejecuta y evalúa el código, finalmente terminando la tarea.
- Regresa al código y reanuda su ejecución.

- La función *print* toma los argumentos y los envía al dispositivo de salida.
- Los argumentos están separados por comas.
- Puede operar con prácticamente todos los tipos de datos:
  - Cadenas, números, caracteres, valores lógicos, y objetos.

- Los argumentos de la función print() se leen por su posición, lo que se denomina manera posicional.
- Cuenta además con los llamados argumentos de palabra clave, que es la utilizada para identificarlos.
- Todo argumento de palabra clave debe ponerse después del último argumento posicional.
- La función print() tiene dos argumentos de palabra clave: *end* y *sep*.

[ ]:
```
# Agregando end=" " cancela el salto de línea predeterminado
print("Esto es todo", end=" ")
print("amigos!")
```

- La barra diagonal inversa con la n forman un símbolo especial denominado carácter de nueva línea (*newline character*) `\n`, e inicia una nueva línea de salida.
- *sep* se utiliza para cambiar el espacio predeterminado entre los argumentos por otro carácter o por ninguno.

[1]:
```
print("Este", "lenguaje", "se", sep="_", end="*")
print("denomina", "Python.", sep="*", end="*\n")
```

```
Este_lenguaje_se*denomina*Python.*
```

- Las funciones integradas están siempre disponibles y no tienen que ser importadas.
- Python viene con más de 50 funciones integradas en la Python Standard Library.
  - https://docs.python.org/es/3/library/functions.html.

2.8 Palabras reservadas

- Como todos los lenguajes de programación, Python tiene una serie de palabras reservadas que no se pueden utilizar para nombrar nuestras variables o funciones.
- Ellas son:

[2]:
```
import keyword # Importa módulo keyword
print(keyword) # Muestra la ruta del módulo
```

```
<module 'keyword' from '/usr/lib/python3.8/keyword.py'>
```

[ ]:
```
print(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']
```

2.9 Propuestas de mejoras y buenas prácticas

- Python cuenta con una serie de documentos denominados PEP (*Python Enhancement Proposal*) que sirven para proveer información o describir nuevos procesos/ambientes.
- Las PEPs contienen especificaciones técnicas de una característica y de porqué se ha decidido por ella.

- Entre ellas, se han creado una serie de guías de estilo y de buenas prácticas que ayudan a respaldar la filosofía de Python.

- PEP 0 – Index of PEPs https://www.python.org/dev/peps/
- PEP 8 – Style Guide for Python Code https://www.python.org/dev/peps/pep-0008/
- PEP 257 – Docstring Conventions https://www.python.org/dev/peps/pep-0257/
- PEP 20 – The zen of Python: https://www.python.org/dev/peps/pep-0020/

- Podemos ver la PEP *The Zen of Python* abriendo un terminal en Linux o la ventana del símbolo del sistema en Windows® (CMD) y ejecutando:

[ ]:
```
python3
>>> import this
```

- 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 eso no sea obvio al principio 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!

- Extraído de https://es.wikipedia.org/wiki/Zen_de_Python.

- Algunas prácticas recomendadas son:

```
- nomenclatura consistente,
- 2 espacios en blanco entre clases,
- líneas de máximo 79 caracteres,
- tabulado con espacios en vez del uso del tabulador,
```

[ ]:
```
# Alinear paréntesis con el primer argumento
foo = nombre_extenso_de_la_funcion(var_uno, var_dos,
                                   var_tres, var_cuatro)
```

- uso de comentarios para las líneas y los bloques de código,
- uso de docstrings para documentar bloques de código,

[ ]:
```
# Un comentario de una línea de texto

# Comentarios
# con
# varias
# líneas
```

[ ]:
```
"""
Docstring

Docstrings son comentarios de varias líneas de texto que se
insertan antes de un módulo, clase, etc. explicando su función.

"""
```

- importar bibliotecas sistema, de terceros, y locales en este orden:

[ ]:
```
# Uso correcto de importaciones
import os
import sys

import pandas as pd

import biblioteca_local
```

- evitar espacios alrededor de operadores binarios,
- codificación preferida UTF-8, y
- escribir código simple en vez de complejo.

- Se puede descargar una guía en castellano con ejemplos de:
- https://recursospython.com/pep8es.pdf

- Google ha hecho pública su guía de estilo de Python, que por cierto, es el lenguaje dinámico más utilizado en esa organización.
- Se puede acceder en la URL https://google.github.io/styleguide/pyguide.html

2.10 Instalación de Python

- Para programar se necesitan las siguientes herramientas:
  - Un editor para escribir el código.
  - Una consola para ejecutarlo.
  - Un depurador que permita ejecutar el código paso a paso.

- La instalación estándar de Python 3 contiene una aplicación llamada IDLE.
- IDLE por sus siglas en inglés significa Desarrollo Integrado y Entorno de Aprendizaje (*Integrated Development and Learning Environment*).

- En los sistemas operativos GNU/Linux como Debian o Ubuntu se instala y ejecuta respectivamente con:

[ ]:
```
sudo apt-get install idle3
idle
```

- El lenguaje Python viene ya instalado en muchos sistemas operativos GNU/Linux como Debian o Ubuntu.
- En Windows® 10 se puede instalar desde la tienda de Microsoft® o se puede descargar e instalar manualmente: https://www.python.org/downloads/windows/.
- Tutorial en Internet: https://techexpert.tips/es/windows-es/instalacion-del-entorno-virtual-de-python-en-windows.

- La instalación de Python por lo general estará en `C:\Python38` (cambiando el número de versión), aunque se puede cambiar esta ruta durante la instalación.
- Para añadir este directorio a la ruta en el prompt de DOS:

```
set path=%path%;C:\python38
```

- Resolver ERROR: https://recursospython.com/guias-y-manuales/python-no-se-reconoce-como-un-comando-interno-o-externo/.

2.11 Creación de ambientes virtuales

- La instalación de Python incluye otras herramientas como:
  - *pip* para la instalación de bibliotecas, módulos y paquetes externos, y
  - *venv* para crear ambientes virtuales.

- Un entorno virtual Python sirve para aislar el intérprete Python, las bibliotecas y los scripts en un espacio determinado (carpeta o directorio).
- Esto permite trabajar en un proyecto evitando problemas generados por actualizaciones y/o de dependencias con el intérprete y las bibliotecas.
- También facilita la compilación y que pueda ser portado a otros sistemas.

- Python a partir de su versión 3.3 incluye un módulo, *venv*, que permite la creación de entornos virtuales.
- Fuentes:
  - https://peps.python.org/pep-0405/.
  - La guía oficial de instalación y uso de *pip* y ambientes virtuales:
  - Installing packages using pip and virtual environments.

- Para crear el ambiente virtual en GNU/Linux utilizamos el siguiente comando:

```
python3 -m venv ambiente_virtual
source ambiente_virtual/bin/activate
```

En Windows el comando se ejecuta de la siguiente manera:

[ ]:
```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

- Este comando crea el directorio de destino (suele ser *.venv*) y el archivo *pyvenv.cfg* con una clave *home* que apunta a la instalación de Python desde la cual se ejecutó el comando.
- Crea un subdirectorio *bin* (o *Scripts* en Windows) con una copia o un enlace simbólico de los binarios Python dependiendo de la plataforma o argumentos utilizados al crear el entorno.
- Crea un subdirectorio (inicialmente vacío) `lib/pythonX.Y/site-packages` (`Lib\site-packages` en Windows).
- Fuente: https://docs.python.org/es/3/library/venv.html.
- Instalar un paquete dentro del ambiente virtual se realiza de la misma forma que en un sistema com Python y *pip* instalados:

```
[ ]: python3 -m pip install -U Flask           # Instala microframework Flask
     python3 -m pip install -U Flask==2.0.0    # instala versión específica
     python3 -m pip install --upgrade Flask    # actualiza Flask
```

- El fichero *requirements.txt* sirve para instalar módulos de la aplicación si ésta es cambiada de entorno (Sistema operativo, Servidor, etc.).
- Se puede crear manualmente e actualizarlo cuando es necesario escribiendo los nombres de los módulos a medida que se instalan.
- Como ejemplo, en GNU/Linux creamos el fichero y agregamos *Flask* en el mismo:

```
[ ]: touch requirements.txt
     echo "Flask" > requirements.txt
```

- Posteriormente para instalar todos los módulos en el nuevo entorno se ejecutará:

```
[ ]: python3 -m pip install -r requirements.txt
```

- La siguiente opción (alternativa a la anterior y preferible) crea el fichero *requirements.txt* automáticamente.
- Se suele ejecutar una vez completo el desarrollo de la aplicación y previo a la migración al nuevo entorno.

```
[ ]: pip freeze > requirements.txt
```

2.12 La biblioteca virtualenv

- Si se está utilizando Python 3.3 o superior, la forma preferida de gestionar ambientes virtuales es mediante el módulo *venv*.
- De todas maneras se puede contar con la biblioteca *virtualenv* como herramienta para crear ambientes virtuales, la cual agrega funcionalidades extras; entre ellas:
  - soporte a Python 2.7, configuración, mantenimiento, duplicación, y
  - resolución de problemas (*troubleshooting*).
- Fuente: https://virtualenv.pypa.io/en/latest/
- Para instalarlo en Ubuntu:

```
sudo apt install -y build-essential libssl-dev libffi-dev python3-dev
sudo apt install python3-pip
sudo apt install -y python3-venv
```

- Para realizar el mismo procedimiento en Windows 10®:
  - Descargar *pip* de https://bootstrap.pypa.io/get-pip.py
  - Guardarlo en el directorio donde está el fichero ejecutable de python.
- Luego abrimos la línea de comandos y escribimos:

```
python get-pip.py
```

- Finalmente instalamos *virtualenv* y lo activamos:

```
pip install virtualenv
virtualenv virtual_env
cd virtual_env\Scripts
activate.bat
```

- Por último es aconsejable tener instalado un sistema de control de versiones (CVS) como GIT.
- En Windows 10® se puede descargar desde https://git-scm.com/download/win.
- En GNU/Linux Ubuntu ya viene instalado, pudiéndose ver su versión con el comando:

[ ]:
```
git --version
```

- Un listado oficial de los más relevantes proyectos de creación de paquetes:
- https://packaging.python.org/en/latest/key_projects/#virtualenv.

2.13 IDE - modo interactivo

- Desde la línea de comandos se invoca escribiendo la palabra *python*, o en su defecto si tenemos instalado Python 3 escribiremos en cambio *python3*.
- Este es el denominado modo interactivo.
- Para salir utilizamos *CONTROL-D* en Unix, *CONTROL-Z* en Windows, o escribiendo el comando *quit()*.
- Para conocer la versión el comando es *python -V*.

[ ]:
```
# Versión de Python
python3 -V
# Entrar en modo interactivo
python3
```

[ ]:
```
# Crear una variable e imprimir en pantalla el resultado
nombre = "Juan"
print("Hola", nombre)
```

2.14 Interpretado en fichero

- Se leen y se ejecutan una a una todas las instrucciones del fichero.

- Abriendo un editor se creará el fichero `hola.py` con el siguiente código:

[ ]:
```
# Los comentarios al código deben comenzar con una almohadilla
# Fichero hola.py
# Imprime el nombre de Juan
nombre = "Juan"
print("Hola", nombre)
```

[ ]:
```
python3 hola.py
```

- En Linux se puede ejecutar como un script en la shell:

[ ]:
```
#!/usr/bin/python3
# Fichero hola.py
# Imprime el valor de la variable nombre
nombre = "Juan"
print("Hola", nombre)
```

[ ]:
```
which python # Muestra la ruta al ejecutable de python
chmod +x hola.py
./hola.py
```

2.15 Compilado a bytecode

- Se puede compilar el programa a bytecode usando *py_compile* https://docs.python.org/es/3/library/py_compile.html.
- El código de bytes se carga en el tiempo de ejecución de Python y es interpretado por una máquina virtual.
- El módulo *py_compile* proporciona una función para generar un archivo de código de bytes a partir de un archivo fuente y otra para invocar el archivo fuente del módulo como un script:

[ ]:
```
>>> python3
import py_compile
py_compile.compile ("hola.py")
'__pycache__/hola.cpython-36.pyc'
quit()
```

- Se ejecuta el programa:

[ ]:
```
python3 __pycache__/hola.cpython-36.pyc
```

2.16 Compilado a ejecutable del sistema

- Algunas bibliotecas de Python permiten compilar a un ejecutable del sistema operativo usado.
- Un ejemplo es pyinstaller https://www.pyinstaller.org/, que genera una aplicación junto con todas sus dependencias en un único paquete.

- En los sistemas operativos Debian/Ubuntu se instala con:

[ ]:
```
sudo apt install python3-dev
pip install -U pyinstaller
```

[ ]:
```
pyinstaller hola.py
```

- En caso de error se debe exportar la ruta al fichero *.bashrc*:

[ ]:
```
export PATH=/home/user/.local/bin:$PATH # /home/user se debe reemplazar con el␣
↪directorio del usuario
source ~/.bashrc
python -m pyinstaller
```

- Otra opción para crear el instalador:

[ ]:
```
python3 -m PyInstaller hola.py
```

- Para mostrar el directorio con el programa y luego ejecutarlo:

[ ]:
```
ls ./dist/hola/
./dist/hola/hola
Hola Juan
```

- Para instalar *pyinstaller* en Windows® y crear el fichero *.exe* ejecutable:

[ ]:
```
pip install pyinstaller
# En el directorio del fichero hola.py ejecutar:
pyinstaller --onefile hola.py
```