



Programación en **Python**

2.1.- Contenido 1: Reconocer los conceptos fundamentales del lenguaje Python para la construcción de programas

Objetivo de la jornada

1. Reconoce conceptos básicos del lenguaje Python, sintaxis, indentación y estructura.
2. Distingue características propias del lenguaje Python versus otros lenguajes.
3. Reconoce el entorno de ejecución y las herramientas complementarias de Python para el desarrollo.

2.1.1.- Conociendo Python



RESEÑA HISTÓRICA

Python es un lenguaje de programación que fue conceptualizado en la década de los 80s por Guido Van Rossum (Holanda) luego de su experiencia implementando y utilizando el lenguaje de programación ABC. Van Rossem trabajaba en CWI (Centrum Wiskunde & Informatica) de Amsterdam en el desarrollo de un sistema operativo distribuido que del cual ABC era parte integrante. Python intentó rescatar las mejores propiedades de ABC y a la vez superar sus problemas. Las características iniciales de Python incluyeron: Sintaxis básica similar a ABC, indentación en lugar de llaves para agrupación de sentencias y un número reducido de tipos de datos muy poderosos: Tablas hash (O diccionarios), listas, strings y números. El nombre Python surge del deseo de Van Rossem de un nombre corto, único y misterioso para su nuevo lenguaje. La palabra Python proviene de la inspiración en la serie británica de humor absurdo "El Circo Volador de Monty Python". La primera versión de Python (versión 0.9.0) fue publicada en febrero de 1991, el cual ya incluía manejo de excepciones, funciones y orientación a objetos. En octubre de 2000, se introdujo Python 2.0, que incluyó comprensión de listas, un recolector de basura y soporte de unicode. Durante ocho años se difundieron las versiones 2.x de Python antes de su nuevo release Python 3.0, el que no es compatible con versiones anteriores. Python 3.0 principalmente eliminó duplicidad de código^[1].

OPENSOURCE

Python es desarrollado bajo licencias open source aprobadas por OSI (Open Source Initiative <https://opensource.org>) haciendo que sea usable de manera libre y redistribuible, incluso para uso comercial. La licencia de Python está administrado por la Python Software Foundation.

Todas las licencias de Python, a diferencia de licencias GPL, permiten distribuir versiones modificadas sin necesidad de publicar los cambios como open source.

Detalles de la licencia de Python puede ser consultada en <https://docs.python.org/3/license.html>

PROPÓSITO GENERAL

Dentro del gran ecosistema actual de lenguajes de programación disponibles, existen categorías de acuerdo al propósito de desarrollo del software. De esta forma, encontramos los Lenguajes de Ámbito Específico y los Lenguajes de Propósito General.

Los lenguajes de ámbito específico han sido diseñados para cumplir con necesidades específicas. Por ejemplo, FORTRAN es apropiado para propósitos matemáticos y Lisp para trabajos relacionados a inteligencia artificial.

Por otro lado, los lenguajes de propósito general son denominados de esta forma porque resultan ser apropiados para una amplia variedad de propósitos, por ejemplo; para cálculos matemáticos, trabajos de investigación, sistemas integrados y aplicaciones web.

Python es un lenguaje altamente modular, que ha atraído la atención de una cantidad extraordinaria de desarrolladores y usuarios en los últimos 10 años. Gracias a esto y a su amplia librería estándar, ha sido utilizado para generación de una infinidad de módulos(*) y paquetes(*) para trabajos de investigación científica en distintas áreas del conocimiento (Matemáticas, física, biología, geografía, etc.), desarrollos de muchos sistemas de hardware para emprendimientos (Robótica, Digital Signage, Sistemas de Sensores, etc.), analítica de datos e inteligencia artificial, además de estar detrás de muchos de los sistemas web de grandes empresas como: Google, Facebook, Instagram, Spotify, Netflix, Dropbox y muchos más. Esto hace que Python sea actualmente uno de los lenguajes de programación de propósito general de más amplio espectro del ecosistema tecnológico.

Nota: (*)

- Un **módulo** es un archivo que contiene definiciones y sentencias que pueden ser utilizarlas al ser importadas en un script Python.
- El concepto de **paquete** es usado en Python para estructurar y organizar módulos de manera jerarquizada.

LENGUAJE INTERPRETADO

Para ejecutar un programa en la CPU (Central Processing Unit) de un computador, éste debe estar en un formato que pueda ser entendido por la máquina, es decir, en código de máquina. Este código de máquina, o Assembler, es el código de más bajo nivel en el que podemos programar nuestro computador y depende del tipo de hardware/procesador de cada máquina específica. Actualmente existen dos grandes grupos de arquitecturas de CPU: **1) CISC** (Complex Instruction Set Computing), como los procesadores Intel x86 y sus compatibles AMD, y **2) RISC** (Reduced Instruction Set Computing), como los procesadores ARM, de bajo consumo, que hacen viable la existencia de teléfonos móviles, tablets y otros dispositivos electrónicos modernos. Esto hace que para programar cada uno de estos tipos de dispositivos se requiera hacerlo en su propio set de instrucciones.

Con el fin de desarrollar software de manera más sencilla y que pueda ejecutarse de igual forma en las distintas arquitecturas de CPU, los lenguajes de programación de más alto nivel, antes de su ejecución realizan una traducción

entre el lenguaje específico y el Assembler de la máquina particular. Existen dos formas de realizar esta traducción dependiendo del lenguaje que se utilice: Una que se realiza completamente antes de la ejecución del programa, que corresponde a **lenguajes compilados** (C, C++, Java, etc.); y otra que se realiza durante la ejecución, que corresponde a **lenguajes interpretados** (PHP, Ruby, Python, Lisp, Matlab, etc.).

Python se clasifica como un **lenguaje interpretado** o, descrito de otra forma, un lenguaje de script. Sus programas tienden a ejecutarse en el orden de 10 a 100 veces más lento que programas C. A pesar de esto, escribir código en Python significa una gran optimización del tiempo de desarrollo. Es usual que la extensión de un código en Python sea unas 2 a 10 veces menos extensa que su equivalente en C. El proceso de compilación enlentece los flujos de prueba y depuración de programas. Muchos desarrolladores y compañías de software están dispuestas a aceptar estos sacrificios en tiempos de ejecución en beneficio de los tiempos de desarrollo para tener un producto comercializable o funcional. Es sabido que el número de los errores encontrados en un código es proporcional a la cantidad de líneas de código. Por lo tanto, si un lenguaje permite escribir menos líneas de código para lograr una tarea determinada, probablemente tendrá menos errores. Además, en general, la gran velocidad de lenguajes compilados es necesaria en cierto tipo de aplicaciones más relacionadas a productos de hardware integrado (Computadoras de vehículos, cámaras inteligentes de video, robótica, etc.) y sistemas que requieren de procesamiento en tiempo real. Python, por otro lado, es suficientemente rápido para muchas aplicaciones, como aplicaciones web, utilidades de escritorio, análisis de datos, programación científica, etc. Existen esfuerzos como PyPy para contar con interpretes de mayor velocidad disponibles para Python.

TIPADO DINÁMICO

Python es un lenguaje con Tipado Dinámico. Esto significa que los errores de inconsistencia de tipos (Enteros, Strings, Tupla, Lista, etc.) son evaluados mientras el programa está siendo ejecutado, y no son un impedimento para que el programa comience a ser ejecutado. A modo de ejemplo, consideremos dos variables: **a=10 (Entero)** y **b="30" (String)**. La existencia de operaciones del tipo **a + b**, que gatillan un error de incompatibilidad de tipo en Python, no será advertida hasta el momento de ejecución de la línea de código donde esta operación se ubique. Mientras esta línea no sea ejecutada, el programa continuará su ejecución sin problemas. Otros lenguajes como Java, C o TypeScript son de Tipado Estático pues revisan consistencia en uso de tipo de datos al momento de compilar (C y Java) o antes de ejecución (TypeScript pues es interpretado).

TIPADO FUERTE.

Python es un lenguaje con Tipado Fuerte. Esto significa que los tipos definidos en el código pueden ser adaptados a un contexto específico dependiendo de cómo sean utilizados en nuestro programa. Por ejemplo, si tomamos el caso del párrafo anterior con **a=10 (Entero)** y **b= "30" (String)**, si durante tiempo de ejecución Python se enfrenta a una línea de código que intente operar, por ejemplo, **a+b** se generará un error de tipo (**TypeError**) y el programa terminará repentinamente. Esto se diferencia de lenguajes Débilmente Tipados donde, por ejemplo JavaScript, hará un supuesto en tiempo de ejecución y operará exitosamente

ambas variables entregando como resultado $a + b = '1030'$. En tal caso, JavaScript asume que ambas variables pueden ser operadas como si ambas fueran de tipo String.

MULTIPARADIGMA

Python es considerado un lenguaje Multiparadigma pues soporta múltiples paradigmas de programación, tales como: Imperativo, Funcional, Procedural y Orientado a Objetos^[2].

El paradigma de programación imperativo utiliza el modo imperativo o natural del lenguaje para indicar instrucciones. Ejecuta comandos de forma secuencial, tal como si diéramos una serie de instrucciones verbales.

El paradigma de programación funcional trata los cómputos del programa como la evaluación de funciones matemáticas basadas en el cálculo lambda.

El paradigma de programación procedimental es un subconjunto de la programación imperativa, en el cual las sentencias están estructuradas en procedimientos, llamados subrutinas o funciones.

El paradigma de programación orientada a objetos considera entidades básicas como objetos, cuyas instancias pueden contener tanto datos como métodos para modificar esos datos.

SOPORTE DE BASES DE DATOS

Python provee una amplia cantidad de opciones para la interacción con bases de datos de tipo relacional SQL (MySQL, Postgres, Oracle, SQLite3, etc.) y no relacional NoSQL (MongoDB, Cassandra, Redis, etc.). Python provee una API para bases de datos (DB-API) para módulos de acceso a bases de datos. Esta API está especificada en PEP-249 (<https://www.python.org/dev/peps/pep-0249/>) y la mayoría de los paquetes o módulos de manejo de bases de datos para Python adoptan el uso de esta API. Existen muchas herramientas importables en Python para manejo de bases de datos, las que permiten conectarse, hacer consultas, inserciones y a mostrar resultados obtenidos como consecuencia de estas operaciones. Entre estas herramientas se encuentran SQLAlchemy, Records, PugSQL, ORM (Object Relational Mapper) de Django, peewee, PonyORM, SQLAlchemy y otras^[3].

DESARROLLO WEB Y FRAMEWORKS

Dada su versatilidad adaptable a la generación rápida de prototipos y grandes proyectos, Python es ampliamente utilizado para el desarrollo de aplicaciones web. Python ofrece una Interfaz Gateway hacia Web Servers (**WSGI**) como los que hemos comentado en secciones anteriores. La interfaz WSGI permite estandarizar la comunicación entre Python / Frameworks Web y el servidor web, ofreciendo la posibilidad de contar con código portable a cualquier servidor web que sea compatible con WSGI, documentado en PEP 3333 (<https://www.python.org/dev/peps/pep-3333/>). Servidores Web que comúnmente interactúan con aplicaciones web desarrolladas en Python son NGINX y Apache. Por otro lado, existe la posibilidad de utilizar servidores web

WSGI standalone en lugar de los servidores web tradicionales, obteniendo gran mejora en desempeño. En este último caso es frecuente combinar un servidor web tradicional como NGINX, actuando como Proxy inverso, y un Servidor WSGI standalone.

Un Framework Web consiste en un conjunto de librerías y un programa principal donde es posible implementar una aplicación web. Los frameworks web incluyen patrones y utilidades que permiten:

- **Enrutamiento de URL:** Relaciona un request HTTP proveniente desde el servidor web con una pieza de código en Python que es invocada para atenderlo.
- **Objetos de Request y Response:** Encapsula la información intercambiada con un navegador de internet del usuario.
- **Motor de Plantillas (O Templates):** Permite separar el código Python que implementa la lógica de la aplicación, de las salidas en HTML que ésta produce.
- **Servidor Web de Desarrollo:** Provee un servidor HTTP que puede ejecutarse en las máquinas de desarrollo para observar de forma inmediata los resultados de actualizaciones en el desarrollo de las aplicaciones.

La mayoría de las aplicaciones web desarrolladas en Python se basan en los frameworks **Django** y **Flask**. El primero es un framework ampliamente equipado con herramientas, módulos que vienen preconfigurados, base de datos preconfigurada, todos estos pensados para ser directamente utilizables. El segundo, corresponde a un Microframework, basado en la filosofía de incluir sólo lo que se necesita utilizar. Por ejemplo, Flask no considera la inclusión de base de datos como una configuración por defecto. Aunque Flask puede pensarse como un framework para proyectos pequeños, éste y Django tienen herramientas muy poderosas para cualquier tipo de desarrollo, con una gran comunidad y documentación disponible. Además de estos frameworks existen otros que también son populares en el ámbito de Python, tales como **Falcon**, **Tornado**, **Pyramid**, **Masonite** y **FastAPI**^[4].

EXTENSA LIBRERÍA ESTÁNDAR

La librería estándar de Python es muy completa y ofrece una gran cantidad de facilidades. Esta librería contiene módulos (Escritos en C) que proveen acceso a funcionalidades del sistema, tales como Entrada/Salida de Archivos; así como también módulos escritos en Python, que proveen soluciones estandarizadas para muchos problemas que se enfrentan rutinariamente en proyectos de desarrollo. Algunos de estos módulos son explícitamente diseñados para promover y mejorar la portabilidad de los programas escritos en Python, a través de la abstracción de aspectos nativos de plataformas específicas, y en lugar de ello planteando una visión de API neutral. Dentro de los módulos más comunes incluidos como parte de la librería básica están^[5]:

- **time:** Acceso y conversión de tiempos.
- **sys:** Acceso a parámetros y funciones del sistema específico.
- **os:** Varias interfaces del sistema operativo.

- **math:** Funciones matemáticas.
- **random:** Generación de números pseudoaleatorios con distintas distribuciones.
- **pickle:** Conversión de objetos Python a secuencias de bytes y viceversa.
- **urllib:** Procesamiento de URLs incluyendo request, responses, errores y otros.
- **re:** Operaciones de expresiones regulares.
- **socket:** Interfaz de bajo nivel para conexiones de red.
- **Csv:** Escribir y Leer datos desde/a archivos separados por comas y otros delimitadores.

Los instaladores para plataforma Windows usualmente incluyen la totalidad de la librería estándar y, frecuentemente, incluyen muchos componentes adicionales. Para sistemas de tipo Unix Python normalmente es provisto como una colección de paquetes, por lo que puede ser necesario utilizar herramientas para manejo de paquetes para obtener los que no estén incorporados por defecto^[6].

PEP (Python Enhancement Proposals)

Las PEP son propuestas para la mejora continua de Python en el transcurso de su evolución. Existen 3 categorías de PEP que sirven para diferentes propósitos^[7]:

- **Estándar:** Mejora el language Python con nuevas funcionalidades.
- **Informativa:** Provee información a la comunidad Python.
- **Proceso:** Modifica o hace mejoras en tópicos relevantes a la comunidad pero fuera del language Python propiamente tal.

PEP0 es un índice de todas las PEP que han sido creadas. PEP1 define qué es una PEP y sus propósitos. Dentro de todas las demás, es de especial interés PEP8, que especifica las convenciones para escribir código en Python. En esta PEP se especifican aspectos como: Máximo largo de línea, uso de TABs or espacios, indentación, forma correcta de importación de módulos, comentarios, etc. Es importante seguir las convenciones establecidas por PEP8 para hacer nuestro código más legible y estándar para ser entendido por otros desarrolladores o por nosotros mismos luego de un tiempo de alejarnos del código de un programa en particular. Esta estandarización nos permite trabajar de buena forma en proyectos de desarrollo opensource, donde debe trabajarse con otros desarrolladores y debe existir una consistencia en el estilo de la escritura de código^[8].

COMUNIDAD

Python posee una gran comunidad global con millones de desarrolladores que interactúan a diario virtual o físicamente. Muchos de estos desarrolladores son miembros de la Python Software Foundation (PSF) que promueve, protege e impulsa el avance de Python, además de dar soporte y facilitar el crecimiento de la comunidad internacional de programadores de este lenguaje. Se sugiere a todos los interesados en este lenguaje de programación, que se encuentren activamente involucrados en desarrollos donde éste se incluya, inscribirse como miembro de esta comunidad^[9].

Algunos recursos destacados que se encuentran disponibles en la web relacionados con Python son:

- Página oficial de la comunidad python.org^[10]
(<https://www.python.org/community/>)
- Repositorio oficial de Python en GitHub^[11].
(<https://github.com/python/cpython>)
- Comunidad online de Python en Reddit^[12].
(<https://www.reddit.com/r/python>)
- The Hitchhiker's Guide to Python!^[13]
(<https://docs.python-guide.org/>)
- Canales IRC **#python**, **#python-dev** y **#distutils**
- Liderazgo de Python por su creador Guido Van Rossum^[14].
(<https://www.artima.com/weblogs/viewpost.jsp?thread=235725>)
- Tendencias comunidad Python^[15].
(<https://opensource.com/article/18/5/numbers-python-community-trends>)
- Comunidad impulsora de mujeres en Python^[16].
(<http://www.pyadies.com/>)

2.1.2.- Entorno de ejecución

2.1.2.1. Instalación de Python.

A continuación nos centraremos en la forma de instalar el interprete de Python, y sus herramientas asociadas, para las tres familias más comunes de sistemas operativos: Windows, MacOS y GNU/Linux.

WINDOWS

Para instalación de Python en Windows debemos referirnos al sitio oficial (<https://www.python.org/>) y acceder a la página de descargas, donde podemos ver la última versión liberada de Python. Al momento de escribir este documento, ésta corresponde a la versión 3.8.4. Escogemos la versión para Windows correspondiente a nuestro sistema: **"Windows x86-64 executable installer"** o **"Windows x86 executable installer"**. Luego ejecutamos este archivo y obtendremos una ventana como la siguiente:



Debemos seleccionar “Add Python 3.8 to PATH” con el propósito de que Python quede disponible en nuestra ruta de ejecución. Luego procedemos con “Install Now” y estaremos en condiciones de utilizar Python3.8.4 en nuestro sistema.

MacOS

La mejor forma de de instalar Python3 en el sistema operativo MacOS es a través del manejador de paquetes Homebrew. Para esto debemos ir a <https://brew.sh/> y copiar el código bajo Install Homebrew:

Install Homebrew

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Luego debemos pegar el texto copiado en un terminal de MacOS. Si el sistema solicita instalar las Herramientas de Línea de Comando para Desarrollador debe aceptarse y presionar **Install**. Este proceso tardará unos minutos y Homebrew estará instalado.

Para continuar con la instalación de Python3, debemos ejecutar el siguiente comando en el terminal:

```
$ brew install python3
```

Este comando instalará la última versión disponible de Python3. Una vez finalizada la instalación puede verificarse el éxito de ésta escribiendo **python3 + Enter** en un terminal y entrando a la consola de comandos de python. Podemos salir de ésta con el comando **quit() + Enter**.

GNU/LINUX

En alguna de las distribuciones GNU/Linux, por ejemplo Ubuntu; que tomaremos como referencia en esta sección, lo más probable es que ya venga instalada una versión de Python3, por ejemplo 3.6. Como esta no es la última versión de Python, instalaremos la versión 3.8.4 teniendo cuidado de no desinstalar la versión

existente. Esto debido a que varios componentes de la distribución de GNU/Linux dependen de ella y podría ocasionar problemas de sistema. Para comprobar cuál es la versión de Python3 instalada, podemos ejecutar el comando **python3 --version**

Asumiendo la distribución Ubuntu de GNU/Linux, y asegurándonos que el siguiente procedimiento sea válido para versiones de Ubuntu desde 14.04 en adelante, ejecutaremos:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.8
```

La instalación estará completada luego de unos minutos. Podemos comprobarlo ejecutando el comando **python3.8 + Enter** y verificando que se activa la consola de python como se ve en la figura siguiente. Podemos salir de ésta con el comando **quit() + Enter**.

```
$ python3.8
Python 3.8.4 (default, Jul 14 2020, 01:31:35)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.1.2.2. Modos de funcionamiento del interprete Python.

Python tiene dos modos básicos de ejecución: Script y Sesión Interactiva. El modo más utilizado para desarrollo de programas en python es el modo por Script, donde se genera un archivo de extensión .py que luego es llamado por el interprete Python para su ejecución. El modo por Sesión Interactiva se efectúa a través de una consola de línea de comandos la cual permite obtener una respuesta inmediata a cada una de las líneas de código que vayamos ingresando, además de acumular en el entorno de la sesión los objetos vayamos creando, para utilizarlos agregadamente.

MODO INTERACTIVO

El Modo Interactivo es una buena forma de verificar el funcionamiento de sintaxis o probar instrucciones. En MacOS o GNU/Linux basta con iniciar el interprete interactivo con **python3 + Enter** o **python3.8 + Enter**, respectivamente. Para el caso de Windows, en una consola de comandos de Windows debemos escribir **python3.exe**, o en su defecto utilizar el interfaz gráfica IDLE, instalada junto con Python, aplicación gráfica que permite tanto sesiones interactivas como trabajo en modo script. En Windows existe además la posibilidad de ejecutar archivos .pyw con la aplicación **pythonw.exe** (Clic derecho y ejecutar con aplicación **pythonw**

que reside en directorio Python3) En este caso se mantiene una ejecución en ambiente gráfico y en ningún momento es abierto un terminal de línea de comandos como ocurre siempre al utilizar **python3.exe**.

Un ejemplo de sesión interactiva en Python3 se ve como sigue:

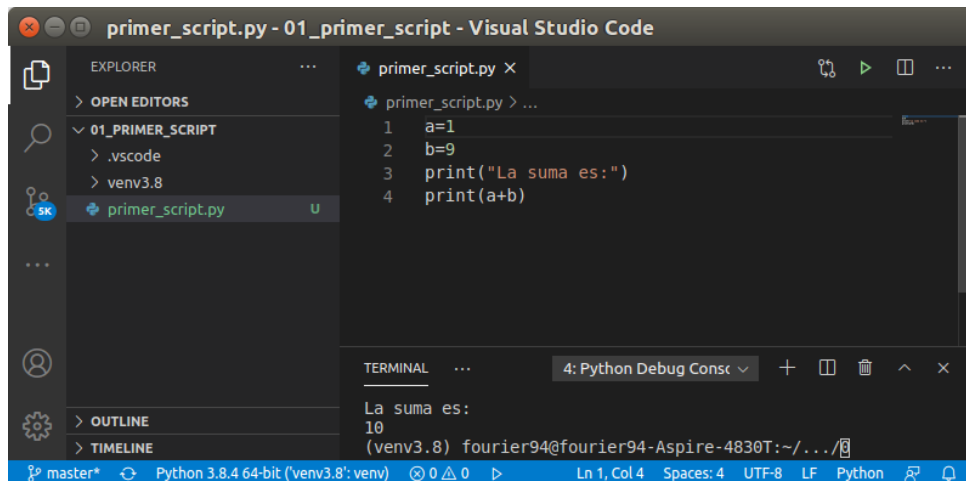
```
$ python3.8
Python 3.8.4 (default, Jul 14 2020, 01:31:35)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a=1
>>> b=9
>>> print("La suma es:")
La suma es:
>>> a+b
10
>>>
```

Hemos realizados las acciones básicas de crear dos variables **a** y **b**, asignándoles los valores enteros **1** y **9**, respectivamente; además imprimir en pantalla el string **"La suma es:"** y obtener la suma de ambas variables creadas. Hemos obtenido interactivamente los resultados de las instrucciones que generan una salida por pantalla: la función **print()** y el operador **+**.

Para salir del interprete interactivo de Python debemos ejecutar el comando **exit()**

MODULO SCRIPT

En un escenario donde ya necesitemos escribir un programa que va más allá de unas cuantas líneas de código, necesitamos recurrir a otra forma de interactuar con el interprete de Python. En estos casos, que serán los que usualmente enfrentemos en proyectos reales, escribimos nuestros programas en un archivo de texto plano al que asignamos la extensión **.py** alusiva a Python. Este archivo o script, lo trataremos en Modo Script, lo que quiere decir que el interprete leerá este script y lo ejecutará sin necesidad de que ingresemos línea por línea las instrucciones de nuestro programa. Para esto creamos un archivo en blanco en un editor de texto, y le asignamos un nombre con extensión **.py**, por ejemplo **primer_script.py**. Podemos utilizar nuestro editor de código Visual Studio Code, y abrir el archivo creado incorporando las mismas líneas de código que utilizamos en el caso de modo interactivo:



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left shows a project named '01_PRIMER_SCRIPT' with files '.vscode', 'venv3.8', and 'primer_script.py'. The main editor window displays the file 'primer_script.py' with the following code:

```
1 a=1
2 b=9
3 print("La suma es:")
4 print(a+b)
```

Below the editor is the TERMINAL panel, which shows the output of the script:

```
La suma es:
10
(venv3.8) fourier94@fourier94-Aspire-4830T:~/.../
```

The status bar at the bottom indicates the file is 'master*', the Python interpreter is 'Python 3.8.4 64-bit (venv3.8: venv)', and the cursor is at 'Ln 1, Col 4'.

En este caso los resultados o salidas de la ejecución de nuestro script los obtenemos en la ventana de terminal inferior. Como vemos, Python entrega la misma salida, sin embargo dado que este modo no es interactivo debemos incorporar la función **print()** para hacer visible en el terminal el resultado de **a+b**.

Aunque más adelante veremos más en detalle el manejo de variables y funciones en Python, en estos ejemplos nos hemos adelantado y definimos variables numéricas (**a** y **b**) y utilizamos la función **print()**, de la forma más simplificada posible, que en python permite imprimir un objeto (En este caso el string "La suma es:") a la salida estándar del sistema, que por defecto es la pantalla.

2.1.2.3. Otras herramientas y flujo de desarrollo en Python.

A continuación presentamos dos herramientas de uso esencial para desarrollo de software utilizando Python: **virtualenv** que permite generar ambientes aislados para proyectos, y **pip** que es el manejador de paquetes más utilizado para la incorporación de módulos que no están incluidos como parte de la librería estándar instalada por defecto con Python.

VIRTUALENV

Es usual que como desarrolladores de aplicaciones en lenguaje Python, tengamos que manejar muchos proyectos a la vez. Además de esto, pasará el tiempo y generalmente utilizaremos la versión más actual de Python para los desarrollos iniciados en cada momento. De la misma forma los paquetes que incorporemos a nuestros desarrollos corresponderán a versiones específicas, que estarán determinadas por la fecha en que estemos desarrollando, por requerimientos técnicos de dependencia entre paquetes o de integración con otros sistemas.

Dado lo anterior, es muy difícil lograr consistencia entre versiones de Python y de paquetes utilizando las versiones de éstos que están instaladas como base del sistema operativo. Para facilitar el manejo de distintas versiones de Python y

paquetes en diferentes proyectos de manera aislada, han surgido herramientas como **virtualenv** alusivo a “Ambiente Virtual”.

Virtualenv permite crear el ambiente virtual de un proyecto en particular, asignando una de las versiones de Python que estén previamente instaladas en el sistema. Para crear un ambiente virtual que utilice la versión **3.8** de **Python** instalada en nuestro sistema y activarlo, debemos realizar los siguiente dos pasos:

1. Creación del ambiente virtual con Python3.8:

Ejecutamos el siguiente comando en el terminal de nuestro sistema operativo:

```
$ virtualenv -p python3.8 venv3.8
```

Donde la opción **-p** se utiliza para especificar la versión de Python que será la base de nuestro ambiente virtual. Por otro lado **venv3.8** es un nombre arbitrario que asignamos al directorio que contendrá todos los archivos que sean parte de éste ambiente virtual.

Si hacemos esto en el directorio de nuestro script **primer_script.py** de la sección anterior, tendremos lo siguiente:

```
$ virtualenv -p python3.8 venv3.8
$ ls -l
-rw-rw-r-- 1 user94 user94 41 jul 20 02:45 primer_script.py
drwxrwxr-x 4 user94 user94 4096 jul 20 03:49 venv3.8
$
```

2. Activación del ambiente virtual:

Con el ambiente virtual llamado **venv3.8** ya creado, debemos proceder a activarlo de la siguiente manera:

```
$ . Venv/bin/activate
```

Esto hará aparecer al inicio del prompt de nuestro terminal el nombre del ambiente virtual activo entre paréntesis, **(venv3.8)**:

```
(venv3.8)$ ls -l
-rw-rw-r-- 1 user94 user94 41 jul 20 02:45 primer_script.py
drwxrwxr-x 4 user94 user94 4096 jul 20 03:49 venv3.8
(venv3.8)$
```

Al ejecutar el intérprete de Python en modo interactivo veremos lo siguiente, que nos confirma que nuestro ambiente virtual está operando con la versión de Python solicitada **Python3.8**:

***Nota:** Desde este momento ya no necesitamos ejecutar el interprete con el comando **python3** sino **python** a secas, pues el ambiente virtual sabe implícitamente a qué versión nos referimos.*

```
(venv3.8) fourier94@fourier94-Aspire-4830T:~/.../01_python_basico/01_primer_script$ python
Python 3.8.4 (default, Jul 14 2020, 01:31:35)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

La desactivación de nuestro ambiente virtual se logra ejecutando en nuestro terminal el comando **deactivate**. Debemos recordar salir previamente del interprete de Python con **exit()**.

Lo anterior nos da una gran flexibilidad. Por ejemplo, consideremos una distribución Ubuntu de GNU/Linux donde tenemos instaladas las versiones de **Python 2.7, 3.5 y 3.8**. Podríamos desear probar nuestro script llamado **primer_script.py** de la sección anterior, ejecutándolo con el interprete de cada una de esas versiones por separado. Para esto podemos crear 3 ambientes virtuales con **virtualenv** de la siguiente forma:

```
$ virtualenv -p python2.7 venv2.7
$ virtualenv -p python3.5 venv3.5
$ virtualenv -p python3.8 venv3.8
```

Con esto tendremos tres directorios, uno para alojar cada uno de nuestros ambientes virtuales y los podremos activar uno a la vez para ejecutar nuestro script con el interprete que corresponda.

Cada vez que deseemos cambiar de ambiente virtual debemos desactivar el anterior con el comando **deactivate**.

PIP (Pips Installs Packages) - Gestor de Paquetes

Además de la versión de Python que estemos utilizando para desarrollar un proyecto determinado, es necesario contar con la posibilidad de instalar paquetes no estándar de Python con versiones específicas, que no necesariamente coincidirán con otros proyectos. La aislación de otros proyectos la provee **virtualenv** como lo vimos más arriba. Para instalar paquetes específicos para el proyecto en particular debemos utilizar la herramienta PIP, siempre teniendo previamente activado el ambiente virtual correspondiente a nuestro proyecto: Tomando el estado de nuestro último ejemplo, podemos decidir instalar un paquete llamado **requests**, que incluye facilidades para manejo de mensajes del protocolo HTTP (Requests, Responses, etc.) desde Python. Para esto haremos:


```
(venv3.8)$ pip install requests

Collecting requests
  Downloading requests-2.24.0-py2.py3-none-any.whl (61 kB)
    | 61 kB 13 kB/s
Collecting chardet<4,>=3.0.2
  Using cached chardet-3.0.4-py2.py3-none-any.whl (133 kB)
Collecting certifi>=2017.4.17
  Downloading certifi-2020.6.20-py2.py3-none-any.whl (156 kB)
    | 156 kB 8.4 MB/s
Collecting idna<3,>=2.5
  Downloading idna-2.10-py2.py3-none-any.whl (58 kB)
    | 58 kB 423 kB/s
Collecting urllib3[!<1.25.0,!<1.25.1,<1.26,>=1.21.1]
  Using cached urllib3-1.25.9-py2.py3-none-any.whl (126 kB)
Installing collected packages: chardet, certifi, idna, urllib3, requests
Successfully installed certifi-2020.6.20 chardet-3.0.4 idna-2.10 requests-2.24.0 urllib3-1.25.9

(venv3.8)$
```

Adicionalmente, supongamos que queremos extraer datos desde páginas de la web y utilizarlas para la aplicación que desarrollaremos. Para esto es útil contar con facilidades de lectura y extracción fácil de contenidos de archivos HTML y XML. Existe un paquete llamado BeautifulSoup4 que provee dichas características. Lo podemos instalar de la siguiente forma:

```
(venv3.8)$ pip install BeautifulSoup4

Collecting BeautifulSoup4
  Using cached beautifulsoup4-4.9.1-py3-none-any.whl (115 kB)
Collecting soupsieve>1.2
  Using cached soupsieve-2.0.1-py3-none-any.whl (32 kB)
Installing collected packages: soupsieve, BeautifulSoup4
Successfully installed BeautifulSoup4-4.9.1 soupsieve-2.0.1

(venv3.8)$
```

Tenemos así instalados dos paquetes que nos entregan facilidades para nuestro desarrollo. PIP entrega la facilidad de consultar qué paquetes se encuentran instalados en el ambiente virtual actual con el comando **freeze**. En nuestro caso, si ejecutamos este comando obtendremos lo siguiente:

```
(venv3.8)$ pip freeze

beautifulsoup4==4.9.1
certifi==2020.6.20
chardet==3.0.4
idna==2.10
requests==2.24.0
soupsieve==2.0.1
urllib3==1.25.9

(venv3.8)$
```

Podemos ver que, junto con los dos paquetes que hemos instalado (requests y beautifulsoup4) aparecen varios otros. Esto se debe a que los paquetes que

hemos instalado con PIP han instalado automáticamente otros paquetes que necesitan para operar, que se llaman dependencias. Esto explica las salidas por consola que han entregado ambas instalaciones, donde se muestran los nombres de los paquetes de dependencia.

En caso de desear portar nuestra aplicación a otra máquina, o subirla a un repositorio para ser compartida con otros desarrolladores para trabajo colaborativo, no es una buena práctica el incluir dentro del grupo de archivos en directorio de ambiente virtual. Esto pues está personalizado para la máquina en particular en que estamos trabajando y porque tiene muchos archivos de dependencias que son públicos y están disponibles abiertamente para ser obtenidos por quien requiera utilizar la aplicación. Por esto, basta sólo con almacenar la especificación de paquetes con la que pueda replicarse el ambiente virtual y los paquetes requeridos por ésta. Para ello generamos un archivo que se denomina **requirements.txt** que no es más que la versión en archivo de texto plano de los resultados entregados por el comando **freeze** de PIP. Esto lo realizamos con:

```
$ pip freeze > requirements.txt

(venv3.8) fourier94@fourier94-Aspire-4830T:~/.../01_python_basico/01_primer_script$ ls -l
total 12
-rw-rw-r-- 1 fourier94 fourier94 41 jul 20 02:45 primer_script.py
-rw-rw-r-- 1 fourier94 fourier94 117 jul 20 12:13 requirements.txt
drwxrwxr-x 4 fourier94 fourier94 4096 jul 20 03:49 venv3.8

(venv3.8)$
```

Vemos que se ha generado el archivo **requirements.txt** que contiene el conjunto de paquetes que serán requeridos por nuestra aplicación para funcionar correctamente.

Para seguir una lógica consistente en nuestra mini-aplicación de demostración **primer_script.py** cargaremos estos paquetes dentro del script.

CARGA DE MÓDULOS

La carga de módulos, que pueden provenir de paquetes como los administrados por PIP o ser simplemente otro archivo **.py**, dentro de un script Python se realiza mediante la sentencia **import**. Esto nos permite incorporar todas las funcionalidades del paquete requerido para ser usado dentro de nuestro script. Existen diferentes formas de importar módulos dependiendo de lo que nos sea más cómodo. Un módulo contiene, principalmente **objetos** y **métodos**. Podemos importarlo por completo o sólo los objetos y/o métodos que vayamos a utilizar. Por ejemplo consideremos el paquete **requests** ya instalado en nuestro ambiente virtual de demostración. Este paquete contiene métodos como **get**, **post**, **head**, **patch**, etc. Formas de importar el paquete completo son:

- `import requests`
- `import requests as rq`

La primera de estas formas carga el paquete y posteriormente podemos utilizar sus métodos con la forma **`requests.get()`**, **`requests.post()`**, etc.

A veces deseamos utilizar nombres abreviados para utilizar los módulos dentro de nuestro programa y utilizamos la segunda forma mostrada, que asigna un alias, en este caso, **`rq`** al módulo importado. Con eso, utilizaremos el módulo con **`rq.get()`**, **`rq.post()`**, etc.

Por último si queremos importar algunos o todos los métodos del módulo sin indicar de qué módulo provienen podemos utilizar otra forma de realizar la carga de éstos. Ésta forma no es recomendada pues impide trazabilidad en la comprensión del código y del origen de los métodos en uso. Se utilizarían sólo invocándolos de la siguiente forma: **`get()`**, **`post()`**, etc. Esto además puede llevar a confusiones con otros módulos que pudiesen contener métodos con nombres idénticos.

- `from requests import get, post`: Importa sólo los métodos `get` y `post`.
- `from requests import *`: Importa todos los métodos y objetos del módulo.

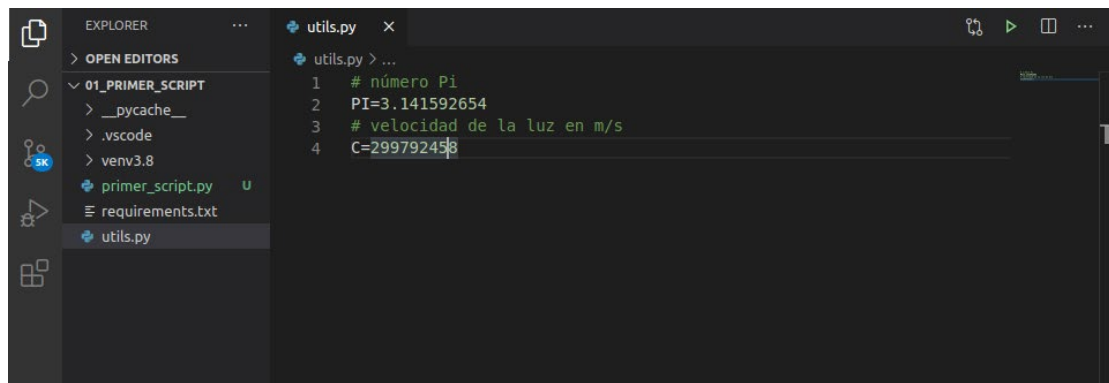
Volviendo a nuestro ejemplo de **`primer_script.py`** importamos el módulo **`beautifulsoup4 (bs4)`** y los métodos **`get()`** y **`post()`** del módulo **`requests`**. Agregamos al final de nuestro script la función **`print(bs4.__version__)`** que imprime la versión de `beautifulsoup4` que hemos importado.

```

primer_script.py - 01_primer_script - Visual Studio Code
EXPLORER
  OPEN EDITORS
    01_PRIMER_SC...
    .vscode
    venv3.8
    primer_script.py
    requirements.txt
  primer_script.py
    2 from requests import get, post
    3 a=1
    4 b=9
    5 print("La suma es:")
    6 print(a+b)
    7 print(bs4.__version__)
  PROBLEMS
  TERMINAL
    4: Python Debug Consc
    La suma es:
    10
    4.9.1
    (venv3.8) fourier94@fourier94-Aspire-4830T:~/.../01_python
  Python 3.8.4 64-bit (venv3.8: venv)
  Ln 2, Col 31 Spaces: 4 UTF-8 LF Python

```

Los módulos que importamos luego de haberlos instalados con PIP pertenecen a la categoría de módulos de terceros. Como hemos dicho, en un script Python es posible además importar como módulo otro archivo **.py** que contenga componentes de utilidad para nuestro script principal. En nuestro ejemplo incorporaremos un módulo de este tipo. Primeramente lo crearemos y lo denominaremos **utils.py**, refiriéndonos a utilidades para nuestro script principal. El archivo **utils.py** estará compuesto de la siguiente forma:

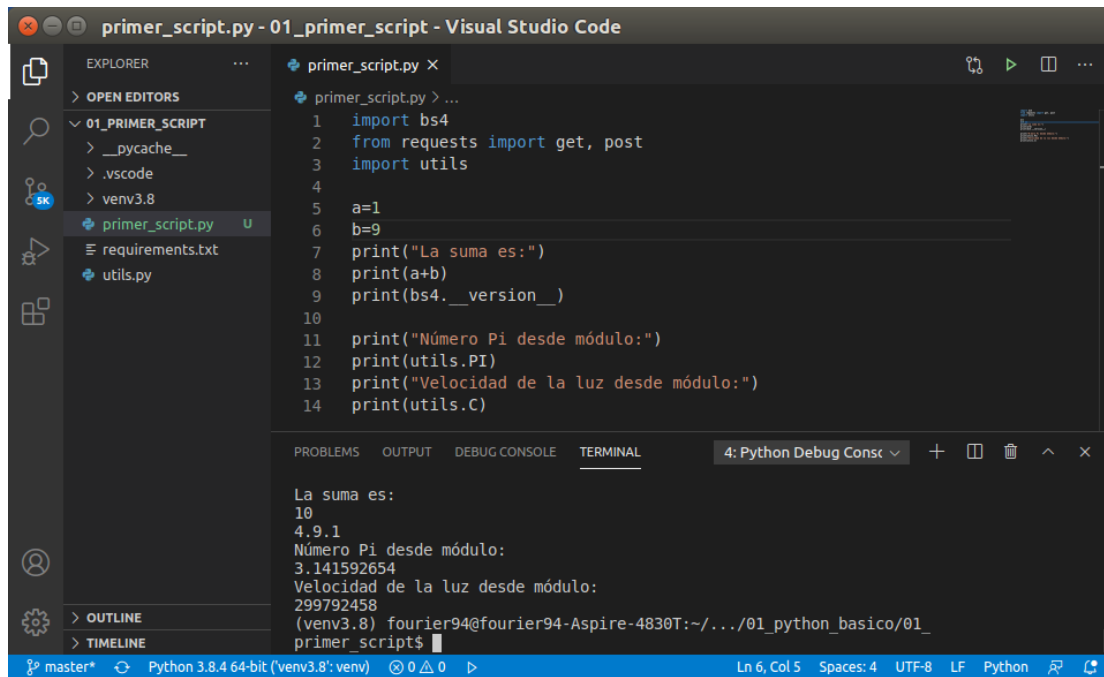
A screenshot of a code editor interface. On the left, the 'EXPLORER' sidebar shows a project structure with folders like '01_PRIMER_SCRIPT' and files like 'primer_script.py', 'requirements.txt', and 'utils.py'. The main editor window is open to 'utils.py' and contains the following Python code:

```
1 # número Pi
2 PI=3.141592654
3 # velocidad de la luz en m/s
4 C=299792458
```

COMENTARIOS

Nota: incorporamos en este caso líneas que comienzan con el símbolo **#**. Éstas corresponden a comentarios, es decir, líneas que generalmente no son instrucciones para la máquina. Al contrario, las incluimos para hacer más fácil la revisión de nuestro código por nosotros mismos y otros desarrolladores que necesiten involucrarse en el proyecto. Pueden incluirse también comentarios de múltiples líneas sin necesidad de escribir **#** en cada línea. Para esto comenzamos y terminamos nuestro texto multilínea de comentario extenso con tres comillas dobles (**"""comentario multilínea"""**). Debemos considerar que agregar comentarios bien escritos y de mensajes claros equivalen a documentar nuestro código, y es un gran valor para usuarios y desarrolladores relacionados a éste.

Este archivo **utils.py** lo emplazamos en el mismo directorio de nuestro script principal. Luego lo importamos en **primer_script.py** y podremos utilizar las constantes físicas que nos provee (Número Pi y velocidad de la luz). Agregamos algunas sentencias en nuestro script para hacer uso de éstas, y observamos los siguientes resultados luego de presionar **Ctrl + F5** para ejecutar:



```
primer_script.py - 01_primer_script - Visual Studio Code

EXPLORER
> OPEN EDITORS
01_PRIMER_SCRIPT
  > __pycache__
  > .vscode
  > venv3.8
    primer_script.py
    requirements.txt
    utils.py

primer_script.py
1 import bs4
2 from requests import get, post
3 import utils
4
5 a=1
6 b=9
7 print("La suma es:")
8 print(a+b)
9 print(bs4.__version__)
10
11 print("Número Pi desde módulo:")
12 print(utils.PI)
13 print("Velocidad de la luz desde módulo:")
14 print(utils.C)

TERMINAL
4: Python Debug Console
La suma es:
10
4,9.1
Número Pi desde módulo:
3.141592654
Velocidad de la luz desde módulo:
299792458
(venv3.8) fourier94@fourier94-Aspire-4830T:~/.../01_python_basico/01_
primer_script$
```

Hemos utilizado los recursos provistos por el módulo externo **utils** que hemos creado. Esta es una forma simplificada de ilustrar lo que ocurre al recurrir a recursos que provienen desde módulos que importamos.

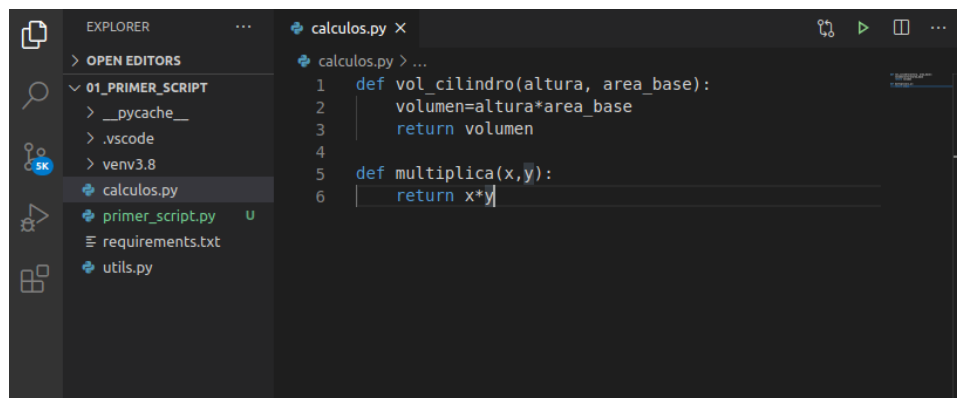
BYTECODE (Archivos .pyc y .pyo)

Si revisamos el directorio de nuestro proyecto veremos que existe un directorio que nosotros no hemos creado, denominado **__pycache__**/. Este directorio ha sido creado por el intérprete de Python (CPython) y contiene los archivos en código binario o **bytecode**. Los archivos que encontramos dentro de este directorio tendrán un nombre referenciando a los archivos **.py** originalmente creados por nosotros. Estos archivos pueden poseer la extensión **.pyc** o **.pyo**, y corresponden a **código binario compilado** o **código binario compilado optimizado**. Como desarrollador, generalmente no nos preocupamos de estos códigos. Éstos sólo corresponden a versiones binarias optimizadas que permiten al programa iniciar más rápidamente pues el intérprete no tiene que partir de la versión “humano-amigable” de nuestro código y transformarla nuevamente a códigos entendidos por la máquina. Esto a no ser que hagamos cambios en nuestros scripts, en cuyo caso los archivos correspondientes serán creados nuevamente. De la misma forma, si borramos el directorio completo **__pycache__**/ el intérprete volverá a crearlo junto con los archivos internos, al momento de ejecutar nuestra aplicación nuevamente.

PAQUETE

Hasta aquí hemos importado un módulo que contiene recursos de utilidad para nuestra aplicación, como son las constantes científicas Pi y C. Con fines ilustrativos, si pensamos que un módulo de un proyecto real puede tener muchos elementos y de distinto propósito, es sensato decidir dividir las utilidades que

proporciona en distintos archivos que son módulos individuales. Luego de alguna forma debemos organizar estos módulos para que sean parte de una unidad que los contenga y permita acceso a sus componentes de manera ordenada y lógica. En nuestro ejemplo, supondremos que el módulo **utils.py** puede crecer muchísimo incluyendo una infinidad de constantes científicas. Por otro lado, nos interesa contar con fórmulas con las que podamos hacer cálculos matemáticos y científicos, y decidimos consolidar todas estas fórmulas en otro archivo **.py** o módulo que llamaremos **calculos.py**. El módulo **calculos** contendrá una fórmula para calcular el volumen de un cilindro en función del área de su base, y otra fórmula que suma dos números. Estas fórmulas estarán implementadas como funciones, que será un tema a tratar más adelante pero que aquí veremos de forma muy superficial, sólo para los propósitos de esta sección. El archivo **calculos.py** será:

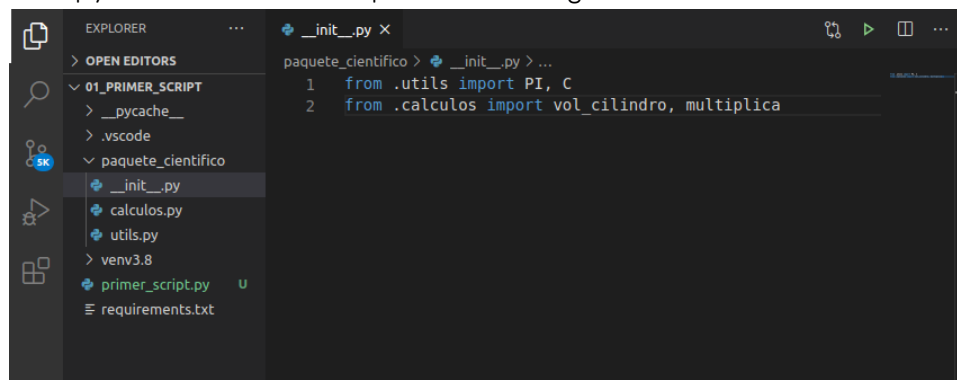


The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor window on the right. The Explorer sidebar shows a project structure with files: `01_PRIMER_SCRIPT`, `__pycache__`, `.vscode`, `venv3.8`, `calculos.py`, `primer_script.py`, `requirements.txt`, and `utils.py`. The Editor window shows the contents of `calculos.py` with the following code:

```
1 def vol_cilindro(altura, area_base):
2     volumen=altura*area_base
3     return volumen
4
5 def multiplica(x,y):
6     return x*y
```

Tenemos ambos archivos, **utils.py** y **calculos.py**, en el mismo directorio que nuestro script principal **primer_script.py**. Para organizarlos en forma de un paquete, los pondremos en un directorio llamado **paquete_cientifico**.

Crearemos el archivo que dará estructura y consolidará los dos módulos previamente creados en lo que llamamos un paquete. Este archivo se denomina **__init__.py**, el cual estará compuesto de la siguiente forma:

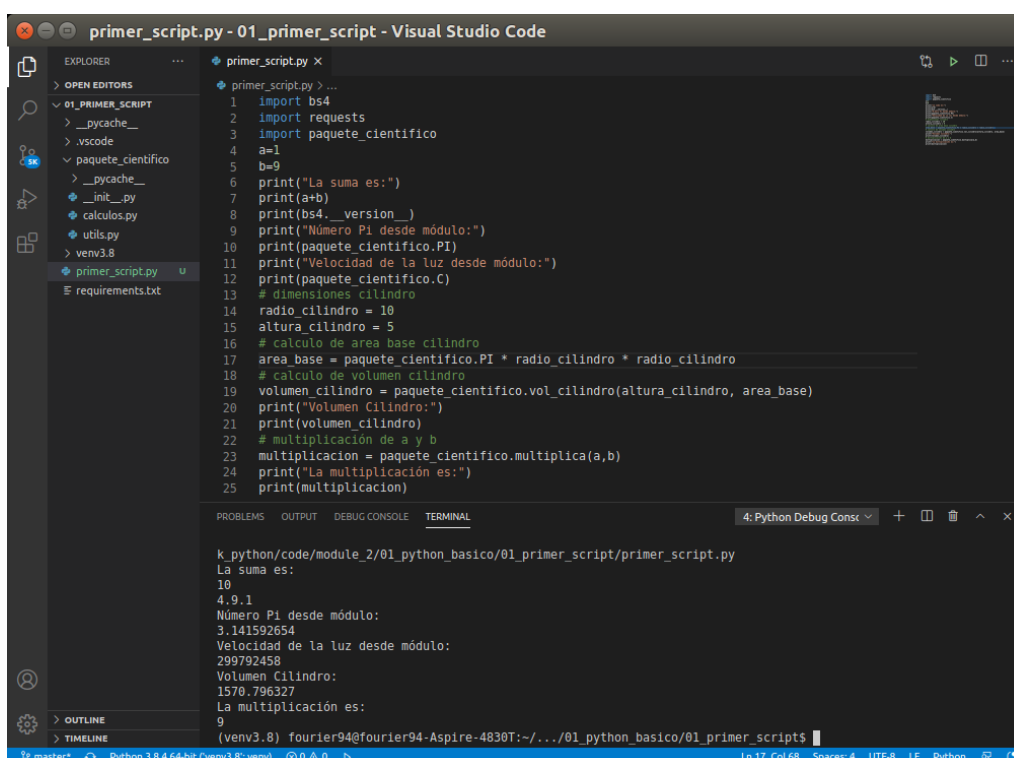


The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor window on the right. The Explorer sidebar shows a project structure with files: `01_PRIMER_SCRIPT`, `__pycache__`, `.vscode`, `venv3.8`, `paquete_cientifico`, `__init__.py`, `calculos.py`, `utils.py`, `primer_script.py`, and `requirements.txt`. The Editor window shows the contents of `__init__.py` with the following code:

```
1 from .utils import PI, C
2 from .calculos import vol_cilindro, multiplica
```


Con esto el directorio **paquete_cientifico** puede ser tratado como un paquete de python, el que será importado y visto por el intérprete como una sola unidad con distintos recursos para utilizar, tal como si todo estuviera en un único archivo **paquete_cientifico.py** y lo importaremos como módulo. Sólo que en este caso tenemos mejor organizados y estructurados los elementos del módulo.

Si importamos nuestro para utilizar nuestro **paquete_cientifico** desde **primer_script.py**, no necesitamos importar cada uno de los módulos del paquete. Sólo hacemos `import paquete_cientifico` y contaremos con todas las facilidades de los módulos individuales, tal como se muestra a continuación:



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left shows a project structure with folders like `01_PRIMER_SCRIPT`, `__pycache__`, `.vscode`, `paquete_cientifico`, and `venv3.8`. The main editor displays `primer_script.py` with the following code:

```
1 import bs4
2 import requests
3 import paquete_cientifico
4 a=1
5 b=9
6 print("La suma es:")
7 print(a+b)
8 print(bs4.__version__)
9 print("Número Pi desde módulo:")
10 print(paquete_cientifico.PI)
11 print("Velocidad de la luz desde módulo:")
12 print(paquete_cientifico.C)
13 # dimensiones cilindro
14 radio_cilindro = 10
15 altura_cilindro = 5
16 # calculo de area base cilindro
17 area_base = paquete_cientifico.PI * radio_cilindro * radio_cilindro
18 # calculo de volumen cilindro
19 volumen_cilindro = paquete_cientifico.vol_cilindro(altura_cilindro, area_base)
20 print("Volumen Cilindro:")
21 print(volumen_cilindro)
22 # multiplicación de a y b
23 multiplicacion = paquete_cientifico.multiplica(a,b)
24 print("La multiplicación es:")
25 print(multiplicacion)
```

The TERMINAL panel at the bottom shows the output of the script:

```
k_python/code/module_2/01_python_basico/01_primer_script/primer_script.py
La suma es:
10
4.0.1
Número Pi desde módulo:
3.141592654
Velocidad de la luz desde módulo:
299792458
Volumen Cilindro:
1570.796327
La multiplicación es:
9
(venv3.8) fourier94@fourier94-Aspire-4830T:~/.../01_python_basico/01_primer_script$
```

Hemos asumido un cilindro de radio 10 y altura 5. Calculamos el área de la base del cilindro en nuestro script principal y luego solicitamos a nuestro **paquete_cientifico** que calcule el volumen con su función **volumen_cilindro()**. Además utilizamos la función **multiplicacion()** para multiplicar los números que anteriormente habíamos sumado.

Nota: Exploración del Objeto **paquete_científico**

En Python todo es un **objeto**, las variables que declaramos, las funciones que definimos; incluso el módulo creado desde nuestro **paquete_cientifico** es un objeto. Existe la función **dir()** con la cual podemos ver todas las propiedades y métodos que posee un objeto determinado. En este caso, en un interprete interactivo de python, si importamos `paquete_cientifico` con `import paquete_cientifico`, podemos hacer `dir(paquete_cientifico)` y veremos todas sus

propiedades y métodos, pudiendo acceder a ellos como mostramos para la propiedad `__path__` :

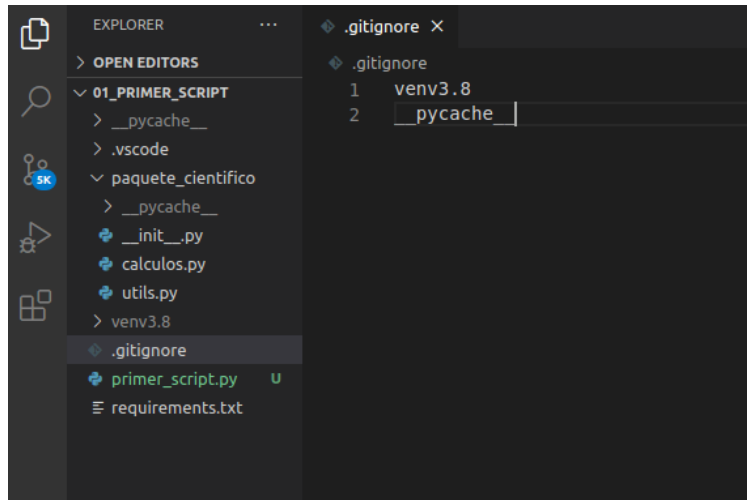
```
>>> import paquete_cientifico
>>> dir(paquete_cientifico)
['C', 'PI', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__path__', '__spec__', 'calculos', 'multiplica', 'utils', 'vol_cilindro']
>>> paquete_cientifico.__path__
['/home/fourier94/Documents/awakelab/full_stack_python/generated/full_stack_python/code/
module_2/01_python_basico/01_primer_script/paquete_cientifico']
>>>
```

Este ejemplo describe de forma muy simplificada cómo están compuestos los paquetes que instalamos en Python a través de PIP o incorporándolos en nuestro proyecto como un directorio.

REPOSITORIO GIT

Finalmente, con el fin de tener nuestro proyecto en formato de repositorio, para continuar haciendo seguimiento de versiones y contribuciones de distintos desarrolladores a éste, inicializamos un repositorio **git** en su directorio raíz. Ya hemos creado repositorios y los hemos conectado con GitHub en secciones anteriores de esta documentación. Sin embargo, revisamos aquí cómo lo haríamos para este primer proyecto en Python.

Como ya mencionamos, el directorio **venv3.8** corresponde al ambiente virtual creado con **virtualenv** que articula la versión de Python en la que está desarrollada nuestra aplicación. Además alberga los paquetes instalados con PIP desde sus repositorios originales en la web. Todo esto es código que no es necesario incorporar a nuestro repositorio pues está disponible vía internet para cualquier persona que desee replicar el proyecto en base a sus archivos la especificación de paquetes y versiones del archivo **requirements.txt**. De la misma forma, los archivos temporales de bytecode creados por el intérprete de Python (En el directorio **__pycache__**/), tampoco requieren ser parte de nuestro repositorio dado que son regenerados en cada ejecución si es necesario. Por lo tanto, debemos crear un archivo **.gitignore** que instruya a git ignorar ambos directorios. Éste será:



Hacemos **add** y **commit**, indicando que este es nuestro commit inicial, para dejar todo listo para hacer **push** a un repositorio remoto, por ejemplo en GitHub, en caso que así lo deseemos posteriormente.

```
(venv3.8)$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore
    .vscode/
    paquete_cientifico/
    primer_script.py
    requirements.txt

nothing added to commit but untracked files present (use "git add" to track)

(venv3.8)$ git add -A
(venv3.8)$ git commit -m "commit inicial proyecto primer_script"
[master (root-commit) 559d725] commit inicial proyecto primer_script
7 files changed, 50 insertions(+)
create mode 100644 .gitignore
create mode 100644 .vscode/settings.json
create mode 100644 paquete_cientifico/__init__.py
create mode 100644 paquete_cientifico/calculos.py
create mode 100644 paquete_cientifico/utils.py
create mode 100644 primer_script.py
create mode 100644 requirements.txt

(venv3.8)$ git status
On branch master
nothing to commit, working directory clean
(venv3.8)$
```

2.1.3.- Referencias

[1] Python Course

https://www.python-course.eu/python3_history_and_philosophy.php

[2] Perceiving Python programming paradigms

<https://opensource.com/article/19/10/python-programming-paradigms>

[3] The Hitchhiker's Guide to Python (Databases)

<https://docs.python-guide.org/scenarios/db/>

[4] The Hitchhiker's Guide to Python (Web Frameworks)

<https://docs.python-guide.org/scenarios/web/>

[5] Cassell, L., & Gauld, A. (Eds.). (2014). Python Projects.

<https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119207580.app2>

[6] The Python Standard Library

<https://docs.python.org/3/library/>

[7] Python Enhancement Proposals (PEPs)

<https://www.python.org/dev/peps/>

[8] PEP 8 -- Style Guide for Python Code

<https://www.python.org/dev/peps/pep-0008/>

[9] Python Community

<https://www.fullstackpython.com/python-community.html>

[10] Python.Org - Community

<https://www.python.org/community/>

[11] Official Repository of CPython Interpreter

<https://github.com/python/cpython>

[12] Reddit's Python Community

<https://www.reddit.com/r/python>

[13] The Hitchhiker's Guide to Python!

<https://docs.python-guide.org/>

[14] All Things Pythonic, Origin of BDFL, by Guido van Rossum

<https://www.artima.com/weblogs/viewpost.jsp?thread=235725>

[15] By the numbers: Python community trends in 2017/2018
<https://opensource.com/article/18/5/numbers-python-community-trends>

[16] PyLadies
<http://www.pyladies.com/>