1

INTRODUCCIÓN

1.1 SISTEMAS DE NUMERACIÓN USADOS EN INFORMÁTICA

En nuestra vida cotidiana usamos, para representar los números, un sistema de numeración **decimal** ("deci", de base 10) que consiste en el empleo de 10 símbolos distintos (*0,1,..,9*) posicionalmente[2], de la siguiente manera[3]:

10^3	10^2	10^1	10^0
1000	100	10	1
7	3	9	8

$7398 = (7 * 10^3) + (3 * 10^2) + (9 * 10^1) + (8 * 10^0)$

Los ordenadores usan internamente el sistema **binario** ("bi", de base 2, donde tendremos solo dos valores posibles[4], *0* y *1*). Cada uno de esos posibles 0´s o 1´s es lo que se denomina un ***bit***, y la reunión de 8 *bits* es un ***bits***. Un ejemplo de número binario de 4 *bits*[5] es el siguiente:

2 Significa que dependiendo de la posición que ocupan se multiplicará por uno u otro valor.
3 Consideramos por simplicidad el uso de solo cuatro dígitos.
4 Estos valores, en los transistores que componen el procesador, se corresponden a que pase (1) o no (0) corriente por ellos.
5 Se denomina en inglés *nibble*.

2^3	2^2	2^1	2^0
8	4	2	1
0	1	0	1

El número binario *0101* corresponde al decimal *5*, que es el resultado de $(1*2^2)+(1*2^0)$. Los números binarios suelen ir precedidos del símbolo *0b*.

El tamaño de la memoria o de algunos elementos del procesador de un ordenador se mide en *bits*, *bytes* o múltiplos de éstos. Así podremos leer que el micro en cuestión "es de 64 *bits*" o la memoria RAM[6] de la que disponemos "es de 16GB". Dos tablas de las distintas unidades que podemos encontrar es la siguiente[7]:

Prefijo	kilo	mega	giga	tera	peta	exa	zetta	yotta
Símbolo	kB	MB	GB	TB	PB	E	Z	Y
Factor	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

Prefijo	kibi	mebi	gibi	tebi	pebi	exbi	zebi	yobi
Símbolo	KiB	MiB	GiB	TiB	PiB	Ei	Zi	Yi
Factor	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}	2^{70}	2^{80}

Por tanto si hablamos de 16GB (16 *gigabytes*) de RAM estaremos hablando de 10^9 *bytes*, y si decimos 16GiB (16 *gibibytes*) tendremos 2^{30} *bytes*. En el primer caso hablamos de 1000000000 unidades y en el segundo de 1073741824. Se introdujo la segunda tabla para terminar con la confusión que generaba el llamar *kilobyte* a 2^{10} *bytes* (y análogamente para los demás prefijos), algo que se hacía desde siempre de forma histórica.

También tenemos en informática otros sistemas de numeración: el octal (base 8) y el hexadecimal (base 16). El sistema **octal** podrá tener 8 dígitos de valores *0,1,...,7*, de la siguiente manera:

6 Los conocimientos básicos de los distintos elementos y terminologías del PC se dan por conocidos por parte del lector.

7 En la tabla *factor* se refiere a factor multiplicador.

8^3	8^2	8^1	8^0
512	64	8	1
7	1	5	0

El número *7150* en octal corresponde[8] al decimal *3688* y al número binario *111001101000*. Los números octales suelen ir precedidos del símbolo *0o*.

El sistema **hexadecimal**, pensado para representar números muy grandes de forma cómoda, tiene base 16, por lo que sus posibles valores irán de *0...9, A, B,...,F* donde *A* equivale a *10*, *B* a *11* ... y *F* a *15*.

16^3	16^2	16^1	16^0
4096	256	16	1
8	A	0	0

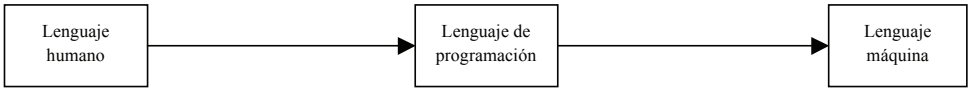
El número hexadecimal *8A00* corresponde[9] al decimal *35328*, al binario *1000101000000000* y al octal *105000*. Los números hexadecimales suelen ir precedidos del símbolo *0x*.

1.2 LENGUAJES DE PROGRAMACIÓN

Los programas informáticos, denominados *software* en inglés, son un conjunto de instrucciones que damos al ordenador para que éste las vaya realizando o *ejecutando*. El ordenador no interpreta directamente el lenguaje humano, sino que tiene sus propias instrucciones, muy sencillas comparativamente con las que solemos usar en nuestro día a día. La máquina solo "entiende" esas instrucciones, por lo que cualquier lenguaje de programación que usemos deberá ser traducido para que el procesador de nuestro ordenador[10] las pueda ejecutar. Tenemos por tanto dos extremos: por un lado el lenguaje humano y por otro el lenguaje que entiende el ordenador (denominado *lenguaje máquina*). Entre ellos, en informática se han desarrollado multitud de otros lenguajes, unos más cercanos al de la máquina (*lenguajes de bajo nivel*) y otros más cercanos al nuestro (*lenguajes de alto nivel*).

8 El resultado de (*7*512) + (1*64) +(8*5)*.
9 El resultado de (*8*4096) + (10*256)*.
10 En los países de América Latina es más común el uso del término *computadora*, que viene del latín *computare*, que significa calcular. La forma usada mayoritariamente en España es un galicismo derivado de *ordinateur*, que a su vez proviene del latín *ordinator*, que podríamos traducir como *"el que ordena"*.

En un muy simplificado esquema podríamos representarlos así:



Iremos viendo diferentes lenguajes, desde los más cercanos a la máquina a los más lejanos.

1.2.1 Lenguaje máquina y lenguaje ensamblador

El lenguaje máquina está preparado para ejecutarse directamente, de forma nativa, en el microprocesador. Son instrucciones individuales, escritas en código binario y que cambian de un procesador a otro. Por su propio diseño, el lenguaje máquina que ejecuta un microprocesador *Intel i7* no es el mismo que el de un IBM de la serie *POWER*, por poner un ejemplo. Una instrucción en *código máquina* puede ser algo en un principio tan críptico como lo siguiente:

0011000101001001

¿Qué significan cada uno de éstos ceros y unos? Dependerá del procesador concreto y de su código máquina asociado pero imaginemos un caso concreto que, aunque inventado, no deja de tener su valor didáctico. En primer lugar dividimos la instrucción en bloques de cuatro *bits*[11] cada uno, quedando así:

0011 0001 0010 0100

¿Cómo interpretar ahora estos bloques? Pues los cuatro primeros *bits* podrían ser el código binario que indique el número asociado a una determinada[12] instrucción del procesador, que pongamos por caso es ***SUM*** y se encarga de sumar determinados elementos que le indicamos posteriormente. Los cuatro *bits* siguientes podrían ser el código de uno de los *registros*[13] que vamos a sumar. En este caso es el registro *BX*. Los siguientes cuatro *bits* pueden ser el código binario del registro 2, es decir, *CX*. Y por último *0100* puede ser el código del quinto registro (*EX*). Uniendo todo ello obtenemos:

SUM BX, CX, EX

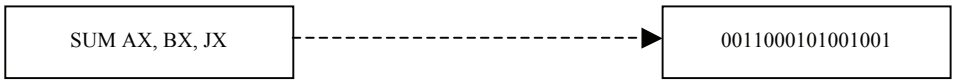
11 Un bit es cada uno de los elementos del código binario y su valor solo puede ser *0* o *1*.

12 En nuestro caso el número binario *0011* es *3* en decimal, por lo que se ejecutaría la instrucción número *3*.

13 Un registro es una pequeña zona de memoria que tiene el procesador donde almacena los datos que va a procesar o los resultados de operaciones. Hay varios de ellos (que se suelen denotar *AX*, *BX*, *CX*…) y cuyos códigos binarios asociados serían respectivamente *0000*, *0001*, y así sucesivamente.

En nuestro código máquina inventado esta instrucción significaría "suma el contenido del registro *BX* con el contenido del registro *CX* y el resultado almacénalo en el registro *EX*". Esta forma, a pesar de seguir siendo lejana al lenguaje humano, es más cómoda que la sucesión de *0´s* y *1´s*, a la par que más fácil de leer, y sería un ejemplo de *lenguaje ensamblador*[14]. Este lenguaje fue creado para hacer menos tedioso tanto la escritura como la lectura o la modificación del código máquina. Usamos un nemónico para representar cada una de las instrucciones del set de ellas que tiene el procesador, de forma que podamos saber qué operación hace: *SUM* para sumar, *SUB* para restar, *MUL* para multiplicar…

Pero recordemos que el procesador solo lee directamente el código máquina, por lo que para poder ejecutar código en ensamblador necesitamos un programa que lo traduzca a ceros y unos. Ese programa se llama también *Ensamblador* (*Assembler*). En nuestro ejemplo, y solo para una línea, actuaría esquemáticamente así:



Un programa escrito en lenguaje ensamblador consta de muchas instrucciones secuenciales que se almacenan en un fichero, denominado *fichero fuente*. Tras actuar sobre él, el ensamblador genera un fichero totalmente en código máquina, que ya está preparado para ejecutarse directamente en el procesador. El proceso real es más complejo y lleva consigo operaciones intermedias, pero usaremos este modelo simplificado ya que solo nos interesa tener un conocimiento genérico.

Escribir programas en ensamblador, a pesar de ser más fácil y legible que en código máquina, sigue teniendo los siguientes inconvenientes:

- Largo y tedioso, ya que cualquier operación sencilla de las que estamos acostumbrados en nuestro ordenador requiere de múltiples operaciones en ensamblador.
- Debemos conocer la estructura interna del microprocesador con el que estemos trabajando, ya que, como comentamos, varía de un modelo a otro.

Es por ello que en la práctica se desarrollaron lenguajes más próximos al humano, los denominados *lenguajes de alto nivel*[15]. En ellos las instrucciones (cada una de ellas puede equivaler a muchas en ensamblador) son mucho más fáciles de entender y manejar. El coste de todo ello es una menor velocidad de procesado.

---

14 *Assembly language* (o más coloquialmente *Assembler*) en inglés.
15 *High level language* en inglés.

1.2.2 Lenguajes compilados y Lenguajes interpretados

Un *compilador* es un programa que traduce un determinado lenguaje a otro, que suele ser (aunque no es obligatorio[16]) el lenguaje máquina. De ser así tendríamos, muy esquemáticamente ya que el proceso es más complejo y con elementos intermedios, lo siguiente:

El código fuente será el de nuestro lenguaje de alto nivel y el código ejecutable el código máquina para el microprocesador. El proceso se llama *compilación* y los lenguajes que hacen uso de ello se llaman *lenguajes compilados*.

Un *intérprete* ejecuta las instrucciones del código fuente directamente, una a una, sin compilarlas en el momento, sino haciendo uso de elementos ya compilados previamente. Los lenguajes que hacen uso de él se denominan *lenguajes interpretados*. A pesar de que hay muchos matices en la distinción compilado/interpretado, usaremos lo comentado como una aproximación básica para su distinción.

1.3 GENERALIDADES SOBRE PYTHON

Python fue creado en Holanda por Guido van Rossum en 1990, en principio como un pasatiempo aunque poco a poco, debido a las características que veremos, fue ganando adeptos en todos los ámbitos hasta extenderse rápidamente, tanto a nivel de usuarios como en el de personas que desarrollaban el lenguaje. Como curiosidad, decir que el nombre de *Python* lo puso Guido en honor de la compañía de cómicos británicos *"Monty Python´s Flying Circus"*.

Tres características principales que definen a *Python* son: lenguaje de propósito general, interpretado y orientado a objetos. Su filosofía se basa en una sintaxis simple y limpia (lo cual facilita enormemente su lectura, mantenimiento y extensión, algo extremadamente agradable y aconsejable), y en potentes y extensibles librerías.

16 Podría ser un código intermedio o incluso texto.

Analizaremos algunas de estas características:

1. Lenguaje de propósito general.

   En la actualidad *Python* se aplica en muchos campos de muy diferente naturaleza, en gran parte debido a su flexibilidad para incorporar código escrito en otros lenguajes[17] y a unas bibliotecas[18] muy potentes que le permiten extender sus capacidades fácilmente. Especialmente interesante es su crecimiento en el área científica, donde podemos encontrarlo en proyectos del más alto nivel.

2. Lenguaje interpretado.

   Que *Python* sea interpretado significa que el código que escribimos es traducido y ejecutado instrucción por instrucción mediante su intérprete, como comentamos con anterioridad. No obstante *Python* permite, mediante el uso de *scripts*[19], una programación similar a la de un lenguaje compilado, por lo cual podríamos decir que *Python* es *pseudocompilado*.

3. Lenguaje orientado a objetos.

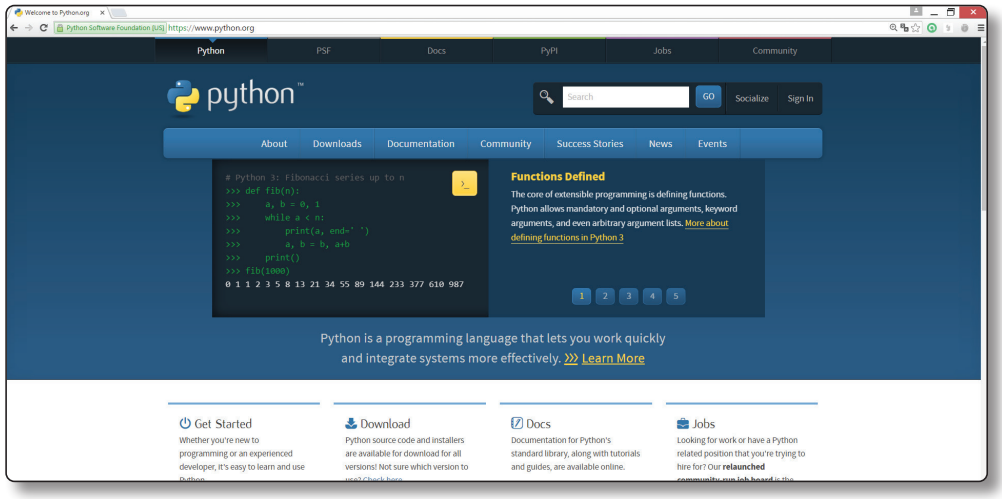
   Aunque *Python* permite también la programación funcional y la imperativa, la orientada a objetos es en la que está basada del lenguaje (por ejemplo todos los datos en *Python* son objetos) y la que le confiere gran potencia. Daremos unas nociones básicas de este tipo de programación al inicio del capítulo 3 y dedicaremos por completo el capítulo 5 para profundizar en sus conceptos fundamentales.

Podríamos añadir más características interesantes, como el hecho de ser multiplataforma. Existen versiones para los sistemas operativos más usados en la informática personal (*Windows*, *GNU/Linux y Mac OS X*) además de otras no tan conocidas en ese ámbito(*BeOS, AS/400, HP/UX, Solaris*...).

Python en la actualidad está mantenido y desarrollado por un equipo muy amplio de personas, es *software* libre y puede ser descargado de forma gratuita (además de otras muchas herramientas y documentación) desde la web de la *Python Software Foundation*:

https://www.python.org/

17 No haremos uso de ello en este libro pero es una característica muy importante de *Python*.
18 En una primera aproximación entenderemos *bibliotecas* como código ya hecho que podemos incorporar al nuestro de forma sencilla.
19 Un *script* es una serie de instrucciones consecutivas que almacenamos en un fichero de forma unificada para posteriormente ser ejecutadas en bloque.



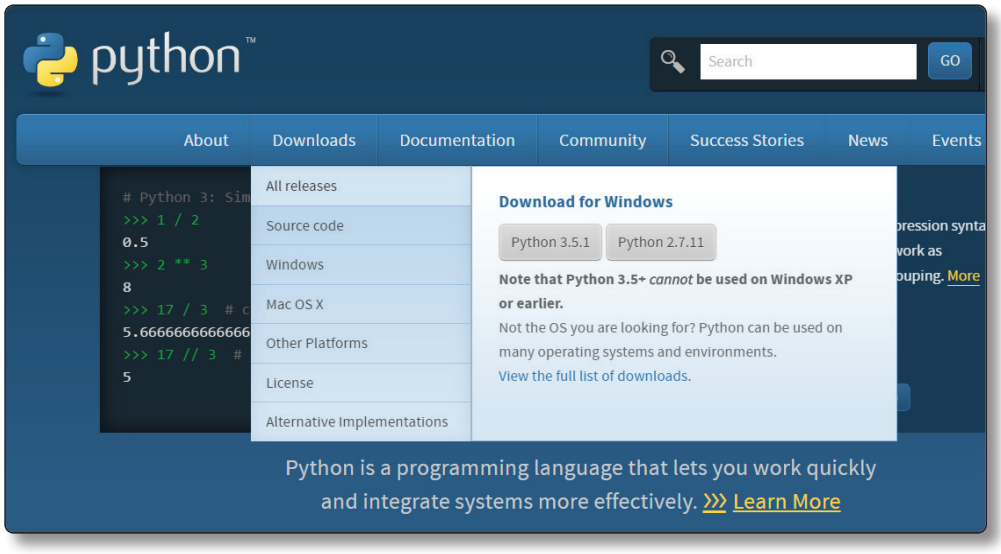
Algo curioso y no demasiado habitual en otros lenguajes es que en la actualidad coexisten dos versiones: *Python* 2 y *Python* 3. Esto genera más de un problema ya que son incompatibles entre sí. Podríamos pensar que al ser la versión 3 más reciente que la 2, tendríamos por lo menos compatibilidad hacia atrás. No es así. Un programa escrito con la sintaxis de *Python* 2 no funcionará en un intérprete de *Python* 3, y a la inversa (aunque eso parecería más lógico). Aunque el lenguaje proporciona una herramienta llamada *py2to3* que traduce código escrito en la versión 2 a la 3, no es solución suficiente. Tener dos versiones de *Python* es un problema y más si vemos que, de lejos, en la actualidad (junio de 2016) la versión 2 de *Python* es la más utilizada y que determinadas herramientas solo funcionan o están diseñadas para ella. En el futuro todo migrará a la versión 3 (en poco menos de 4 años se dejará de evolucionar la rama de la versión 2), pero no deja de ser incómoda la situación en algunos casos. Como ya comenté en el prólogo, este libro se centra totalmente en la versión[20] 3 para el sistema operativo *Windows*.

1.4 INSTALAR PYTHON EN NUESTRO ORDENADOR. PRIMEROS PASOS

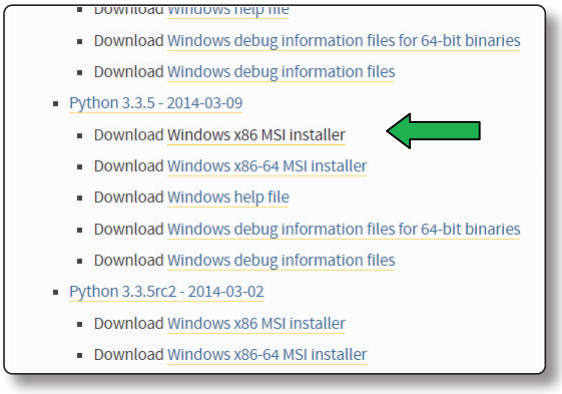
Dentro de la web *https://www.python.org/* tenemos una sección *Downloads* dedicada a la descarga de las distintas versiones de *Python*. La rama 2 y la 3 tienen a su vez distintas versiones que han ido saliendo a lo largo del tiempo, añadiendo

20 Concretamente usaremos la versión 3.3.5. Por compatibilidad con herramientas que emplearemos con posterioridad (o con sistemas antiguos) y dado que no perdemos nada de generalidad en los propósitos del libro al no usar la más actualizada (que en el momento actual es la versión 3.5.1).

parches y características a las anteriores. Nada más colocar el puntero del ratón sobre *Downloads* nos aparece (como muestra la siguiente imagen) las distintas opciones para los distintos sistemas operativos.

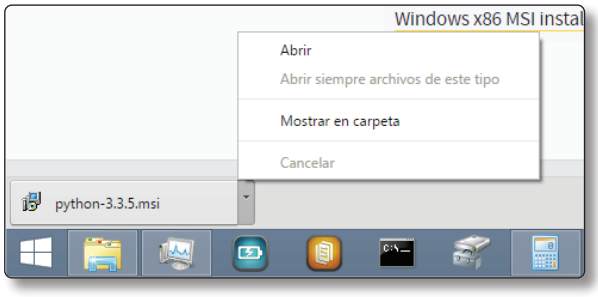


En nuestro caso, haremos clic en *Windows* y buscaremos la versión 3.3.5 (que data del 9 de marzo del 2014 y es la última versión binaria[21] de *Python* 3.3) en la lista que nos aparece[22].



21 No tendremos que compilar el código (en otras instalaciones sí) y dispone de sistema de instalación cómodo.
22 He usado la versión *x86* para una mayor compatibilidad con ordenadores antiguos.

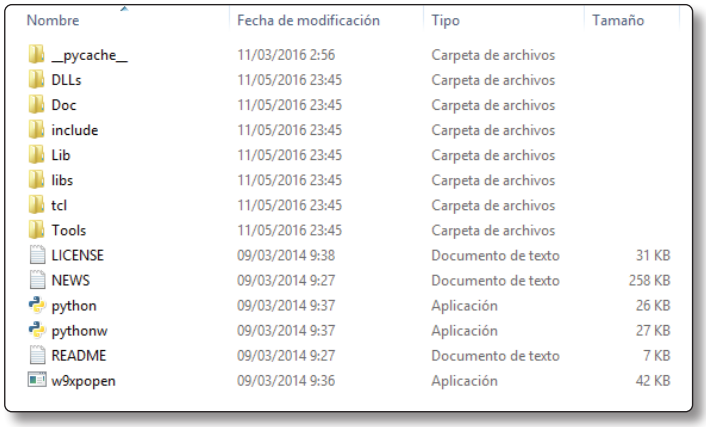
A continuación comenzará la descarga a nuestro ordenador. Al finaliza haremos clic en la flecha de la derecha de la descarga y seleccionaremos *Abrir*:



Comienza entonces un asistente de instalación donde haremos clic en *Next* u *Ok* para todos los cuadros de diálogo que nos vayan saliendo. Si todo ha salido bien, al final nos aparecerá una ventana indicando que la instalación se ha producido de forma satisfactoria. Una vez instalado, se habrá creado una carpeta en nuestro sistema con la siguiente dirección:

C:\Python33

En ella el instalador ha incluido una serie de elementos. Mediante el explorador de archivos de *Windows* podemos acceder a la citada carpeta, que tendrá un contenido similar al siguiente:



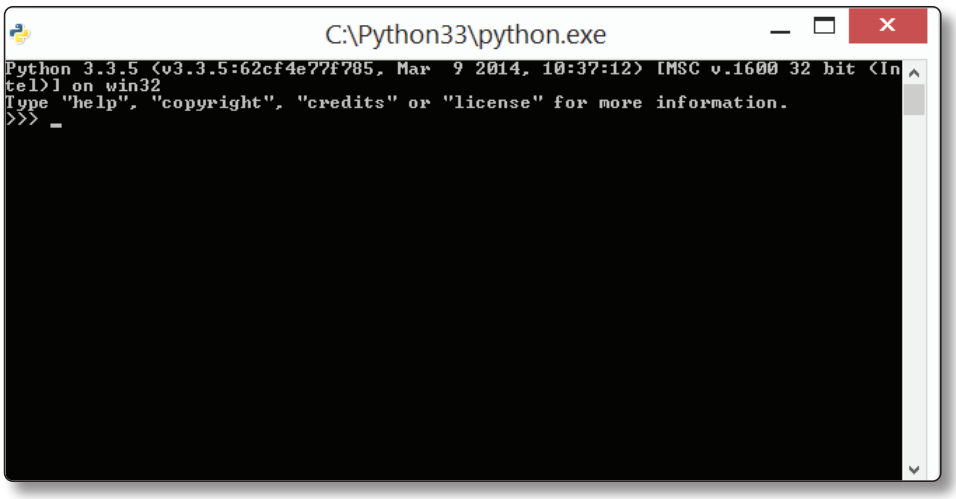
Nombre	Fecha de modificación	Tipo	Tamaño
__pycache__	11/03/2016 2:56	Carpeta de archivos	
DLLs	11/05/2016 23:45	Carpeta de archivos	
Doc	11/05/2016 23:45	Carpeta de archivos	
include	11/05/2016 23:45	Carpeta de archivos	
Lib	11/05/2016 23:45	Carpeta de archivos	
libs	11/05/2016 23:45	Carpeta de archivos	
tcl	11/05/2016 23:45	Carpeta de archivos	
Tools	11/05/2016 23:45	Carpeta de archivos	
LICENSE	09/03/2014 9:38	Documento de texto	31 KB
NEWS	09/03/2014 9:27	Documento de texto	258 KB
python	09/03/2014 9:37	Aplicación	26 KB
pythonw	09/03/2014 9:37	Aplicación	27 KB
README	09/03/2014 9:27	Documento de texto	7 KB
w9xpopen	09/03/2014 9:36	Aplicación	42 KB

Haciendo doble clic sobre el fichero de nombre *python* ejecutaríamos el intérprete y estaríamos ya en condiciones de introducir órdenes, pero lo haremos ahora de otra manera, llegando al mismo lugar. Mediante el menú *Inicio* de *Windows 8.1* accederemos a las aplicaciones, donde aparecerá un apartado para nuestro recién instalado lenguaje:

Vemos que tenemos la opción de desinstalar (*Uninstall Python*) o leer los manuales (*Python Manuals*) pero nos centraremos en las dos opciones que nos permiten ejecutar y por tanto iniciar el intérprete de *Python*. Son las siguientes:

1. *Python* (command *line*)
2. IDLE (Python GUI)

La primera opción nos permite ejecutar *Python* en la línea de comandos, también denominada *"modo consola*[23]*"*. Este modo es totalmente en modo texto, sin ayudas gráficas. Hacemos clic en ella y obtenemos la siguiente ventana flotante en nuestra pantalla:



23 El término *consola* (*console* en inglés) viene de cuando los ordenadores antiguos consistían en un gran ordenador central y muchos pequeños que solo tenían una pantalla y un teclado, sin procesador propio, que eran las *consolas*.

En ella vemos información de la versión del intérprete y sobre qué tipo de procesador y sistema operativo está trabajando. También nos informa de palabras clave que podemos introducir para obtener información sobre cosas como la licencia, el *copyright* o la ayuda. Éste último caso es muy útil para obtener información sobre varios aspectos del lenguaje. Como curiosidad, comprobaremos que la ventana no puede ser modificada en su anchura mediante el uso del ratón. Una buena práctica si vamos a usar a menudo el intérprete es, una vez abierto, hacer clic con el botón derecho del ratón en su icono de la barra de tareas de *Windows* y seleccionar *"Anclar este programa a la barra de tareas"*. De esta manera podremos acceder a él de forma cómoda con sólo un clic.

Lo primero que notamos es el símbolo >>>, que es el indicador (*prompt* en inglés) de entrada de comandos, y el cursor parpadeando. Esto indica que está a la espera de que introduzcamos alguna instrucción para ejecutarla. Teclearemos lo siguiente[24]:

```
>>> print ("Hola Python")
Hola Python
>>>
```

¡Ya hemos ejecutado nuestra primera instrucción en el intérprete de *Python*! En este caso ha sido una muy sencilla que saca por pantalla la frase *"Hola Python"*. Observamos que la instrucción necesita estar escrita de una manera determinada, con paréntesis y colocando el texto entre comillas, unas comillas que luego no aparecerán por pantalla. La instrucción que hemos ejecutado es una de las funciones incluidas por defecto en el intérprete (*built-in functions*) y tiene un formato concreto para usarse adecuadamente. Si no lo seguimos, obtendremos un error.

Una vez que ha impreso la frase por pantalla, el intérprete se queda a la espera de que se vuelvan a introducir más instrucciones. Si tecleamos *Ctrl+z* aparecerán en la pantalla los símbolos ^Z. Al pulsar *Enter* saldremos del intérprete. Es la combinación de teclas usada para ello.

Tras ver (aunque haya sido fugazmente) el uso del intérprete *Python* en *modo consola* (o desde la línea de comandos), usaremos a continuación el *IDE* (*Integrated Development Environment,* entorno de desarrollo integrado) que viene con la instalación de *Python* y que se denomina *IDLE*[25]. De forma genérica un *IDE* tiene integradas una serie de herramientas.

---

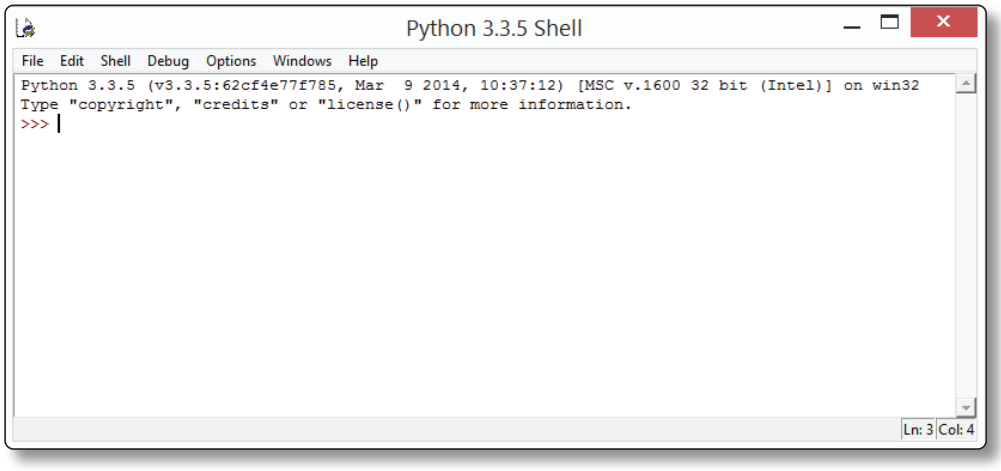
24 Para introducir correctamente el comando deberemos pulsar la tecla *Enter* tras escribirlo.

25 Podría interpretarse también como *Integrated Development and Learning Environment* (entorno de desarrollo y aprendizaje integrado), aunque como curiosidad hay quien afirma que *IDLE* viene (al menos en parte) en honor a Eric Idle, uno de los miembros fundadores del grupo *Monty Python*.

Las principales son:

- Editor de texto con sintaxis resaltada, completado automático, indentado inteligente...
- Intérprete de comandos, también denominado *Shell*.
- Depurador (*debugger*) con opción de ejecución del programa paso a paso, ver valor de variables y de otros elementos del sistema como la pila[26].

Para ejecutar *IDLE* volveremos al menú *Aplicaciones* de *Windows 8.1* y haremos clic en su icono. Aparecerá la ventana[27] del *Shell*, que ahora sí podremos modificar en tamaño a nuestro gusto:

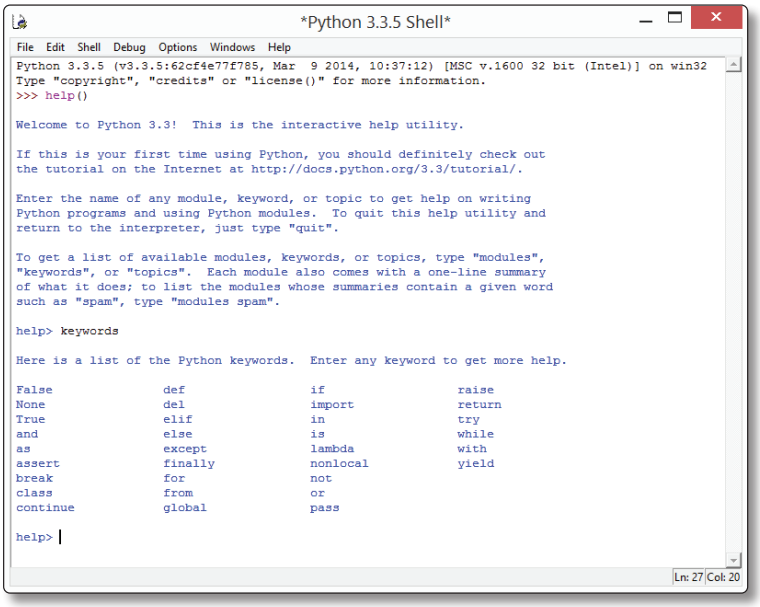


Observamos que la ventana tiene una barra de menús (*File*, *Edit*…) con multitud de opciones, algo de lo que carecíamos con anterioridad. Esto nos permitirá hacer muchas más cosas que desde el modo consola. Si repetimos el ejemplo usado anteriormente para sacar por pantalla un texto, veremos que formatea de forma especial cada una de las palabras, usando un código de colores para los elementos de distinto tipo. Si tecleamos:

```
>>> help()
help> keywords
```

26 Veremos en capítulos posteriores qué es una pila y cómo se usa.
27 Podremos igualmente anclarla a la barra de tareas de *Windows*.

Por pantalla aparece (notar que al teclear *help()* el *prompt* cambia de >>> a *help*>):



En ella aparecen todas las palabras clave (*keywords*) de *Python*. No son demasiadas. El lenguaje está diseñado desde el principio con intención de que los elementos básicos sean pocos y todo lo simples que se pueda. Iremos viendo a lo largo del libro varias de estas *keywords* y cómo usarlas. Puede que haya llamado la atención al lector que *print*, el único elemento usado hasta la fecha, no aparezca en la lista. Eso es porque *print* es una *función* que viene por defecto con *Python* (*built-in function*), y por lo tanto un elemento distinto. Veremos poco a poco qué diferencia hay entre estos elementos del lenguaje.

También, si nos fijamos más detenidamente, observaremos que salvo *False*, *None* y *True* el resto está escrito con todo minúsculas. Es importante escribir los comandos **exactamente** como nos lo indiquen ya que *Python* es un lenguaje que distingue entre mayúsculas y minúsculas[28], por lo que *If* es distinto de *IF* y a su vez de *iF*. En el momento que no escribamos la sintaxis correcta, el intérprete nos indicará que hay un error. Como ejemplo podemos teclear (estando el *prompt* como *help*>[29]*)* en *IDLE "if"* (no incluir las comillas) y obtendremos una ayuda de cómo debemos

28 Se denomina *case sensitive* en inglés.
29 Si hemos quitado accidentalmente el prompt help()>> (por ejemplo por haber pulsado *Enter*) volver a teclear help().

usar el comando de ese nombre[30]. Posteriormente si tecleamos *IF* nos indicará que no hay documentación en *Python* para él. Para seguir viendo cosas desde *IDLE*; pulsaremos *Enter* y volveremos al *prompt* habitual del intérprete (>>>).

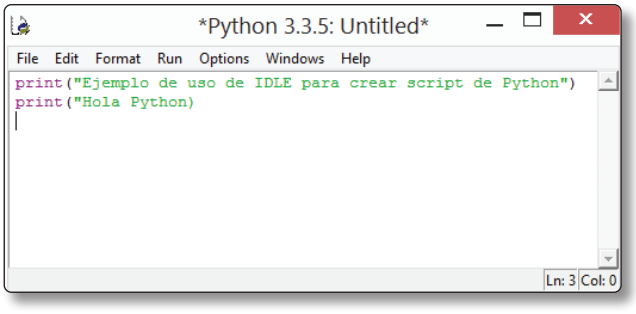
1.5 CREAR FICHEROS DE CÓDIGO PYTHON

Ya sabemos cómo se introducen instrucciones individuales en el intérprete. Una vez que ha ejecutado una de ellas, éste espera a que se introduzca otra nueva. Pero esta forma de introducir el código no es la recomendable cuando queremos desarrollar un programa complejo, ya que necesitamos poder escribir la serie de instrucciones juntas y luego ejecutarlas secuencialmente. Para ello, deberemos almacenar todas ellas en un fichero para posteriormente indicar al intérprete que queremos ejecutarlo. ¿Cómo conseguiremos esto? Veremos algunas consideraciones:

1. El código lo guardaremos en un fichero de texto con extensión *.py*, es decir, debe tener el formato *nombredelfichero.py*. Es una convención necesaria para que el intérprete lo identifique y ejecute correctamente. Este fichero se puede crear fácilmente con cualquier editor de textos, como el sencillo *Bloc de notas* de *Windows*.

2. El fichero creado de la forma indicada se denomina *script*, *módulo* o *fichero fuente*. Ejecutar un fichero de este tipo se denomina habitualmente *ejecutar un script* o ejecutar *Python* en *modo script*, en contraposición a ejecutarse en *modo interactivo*, instrucción a instrucción, como hemos visto hasta ahora.

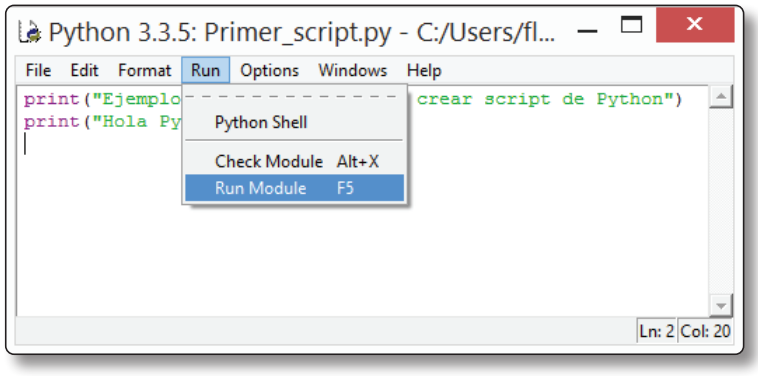
Estando en *IDLE*, vamos al menú *File®New File* y nos aparecerá una ventana flotante con varios menús y con el cursor parpadeando en una zona en blanco. Aquí es donde teclearemos el código que aparece a continuación:



30 No debemos preocuparnos por no saber interpretar de momento el formato que aparece en la ayuda. De forma progresiva comentaremos a qué se refiere cada uno de los elementos representados.

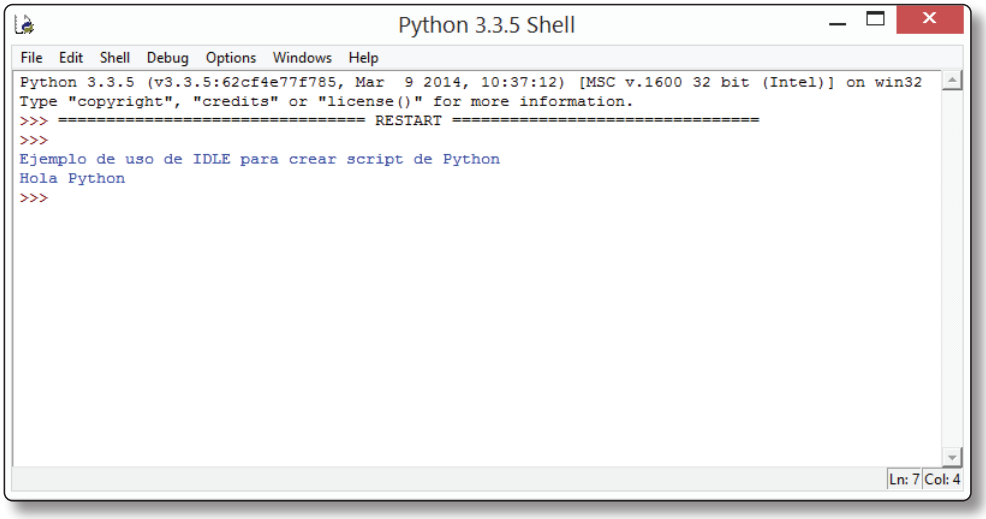
Notamos que no aparece el *prompt* del modo interactivo (>>>), sino que podemos introducir el código como en un editor de texto. Tecleamos *Enter* al final de cada instrucción. Una vez escritas las dos líneas, hacemos clic en *File®Save As,* tras lo cual nos preguntará dónde, con qué nombre y con qué extensión queremos guardar el fichero. Antes de ello crearemos una carpeta[31] en el *Escritorio* de *Windows 8.1* con el nombre *Ficheros_Python*[32]. Es muy importante, ya que en ella guardaremos todos los ficheros que iremos creando a lo largo del libro. Una vez creada accedemos a su interior y guardaremos el fichero con nombre *Primer_script* y con extensión[33] *.py*

Tras hacer clic en *Guardar* ya tenemos en nuestra carpeta el primer fichero *script* de *Python*. ¿Cómo lo ejecutamos ahora? Una opción es acceder mediante las herramientas de *Windows* a la carpeta creada y hacer doble clic sobre el fichero en cuestión. Antes de hacerlo, observamos que el sistema nos indica de forma visual (mediante un icono personalizado) que nuestro archivo es un archivo *Python*. Una vez hecho el doble clic, observamos fugazmente cómo se inicializa el intérprete de *Python* en modo consola, pero enseguida desaparece. En realidad, se ha ejecutado nuestro *script,* pero al finalizar de sacar por pantalla las dos líneas de texto, han salido del modo consola y ha desaparecido la ventana que lo contenía. ¿Cómo podemos ver la salida que ha generado nuestro programa? Como aún tenemos abierta la ventana del fichero creado (notar que una vez guardado ya aparece en la parte superior de la ventana su nombre y dirección completa) la activamos. Haciendo clic en *Run→Run Module*



31 Podemos hacerlo desde el propio cuadro de diálogo que tenemos abierto.
32 No olvidar en guion bajo.
33 Si no le indicamos extensión sino solamente el nombre, nos pondrá automáticamente la extensión *.py*.

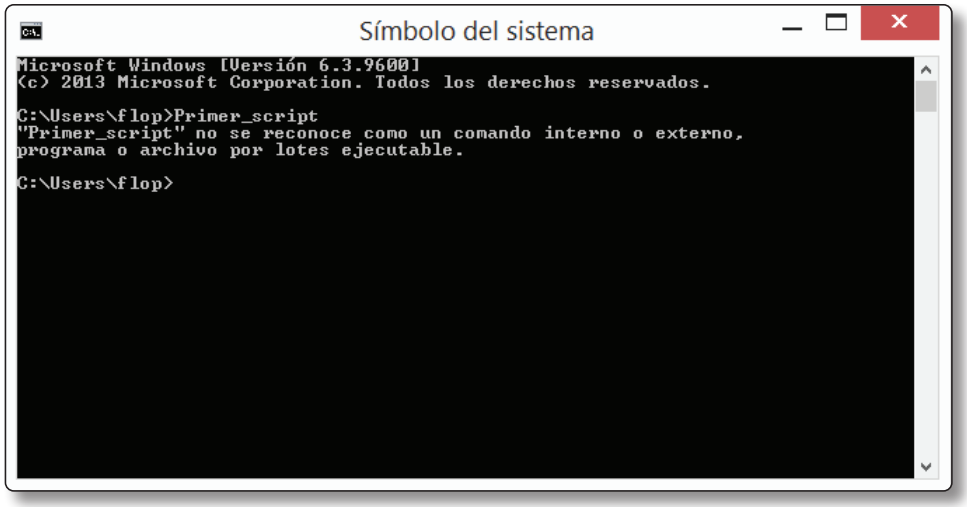
Obtendremos la salida en la ventana del editor *IDLE*:



¿Podemos ejecutar de alguna otra manera nuestro *script*? Sí. Por ejemplo, podríamos ejecutar el *símbolo del sistema* anclado a nuestra barra de tareas de Windows y teclear:

```
Python Primer_sript.py
```

¿Obtenemos algo? Sí, un error. ¿Por qué? Cuando accedemos al símbolo del sistema, lo hacemos por defecto nuestra carpeta de usuario. Pero dado que el fichero que queremos ejecutar no está exactamente en esa carpeta, el sistema no lo encuentra y manda un mensaje de error que nos indica que no ha encontrado el fichero:



Para subsanar este error tenemos las siguientes alternativas:

1. Ir mediante comandos del sistema[34] a la carpeta que hemos creado en el *Escritorio* de nuestro ordenador. Para ello bastaría con teclear secuencialmente los siguientes comandos[35]:

```
cd Desktop
cd Ficheros_Python
```

   Tras ello ejecutaremos nuestro *script,* de la siguiente manera:

```
Primer_script
```

   No hay que indicarle la extensión. Al comprobar que es un fichero de *Python* ejecuta automáticamente el intérprete. De esta manera obtendremos la salida en la propia ventana del sistema.

2. Indicarle la dirección completa del fichero tras el comando *python*:

```
python C:\Users\flop\Desktop\Ficheros Python\Primer_sript.py
```

   El lector tendrá que sustituir la carpeta *flop* por su nombre de usuario en *Windows*[36].

Aún tendríamos otra forma de ejecutar el fichero. Sería accediendo a la carpeta mediante ventanas de *Windows* y una vez en ella, manteniendo la tecla de *Mayúscula*[37] (*Shift*) pulsada, hacer clic con el botón derecho del ratón. Nos aparecerá una ventana emergente en la que seleccionaremos *Abrir ventana de comandos aquí*, tras lo cual se nos abrirá una ventana del sistema estando dentro de la carpeta deseada, por lo que solo debemos ejecutar el *script*:

```
Primer_script
```

Esta forma de acceso, por comodidad, será usada habitualmente a lo largo del libro.

---

34 Son comandos tipo *DOS (Disk Operating System)*. En este caso usaríamos el comando *cd* (*change directory*) para ir cambiando de directorio.

35 Recordar que debemos pulsar *Enter* al finalizar cada línea de comandos.

36 Esto se hará de forma sistemática en el libro. Se indicará apropiadamente.

37 Es cualquiera de las dos flechas verticales situadas a ambos lados del teclado. No confundir con *Bloq Mayús*, que está encima de una de ellas.

1.6 ESTILO DE PROGRAMACIÓN EN PYTHON. INSERTAR COMENTARIOS

Cualquier persona que haya intentado leer y analizar código, seguro que se ha enfrentado al problema de inicialmente no entender bien ni cómo está estructurado ni cómo funciona. Dos problemas suelen aparecer:

1. No hay documentación de qué hace el código y de qué manera, o la que hay es tan exigua que no nos aporta demasiado.

2. El código está desordenado, lo que dificulta su lectura.

Con tiempo podremos analizarlo, ver qué hace y ordenarlo, pero esa labor puede llegar a ser larga, tediosa y frustrante. *Python* está pensado y diseñado desde su raíz para que el código esté bien estructurado y sea muy legible, obligándonos a ello. Se apoya en:

1. Indentación correcta obligatoria: las instrucciones deben tener unas sangrías determinadas en base a unas reglas que iremos viendo. De lo contrario se generaría un error.

2. Coloreado de código: en los editores el código tiene diferente color dependiendo del tipo de elemento del que se trate, lo que facilita su identificación.

Vayamos de nuevo a *IDLE* y tecleemos el siguiente comando:

```
>>> print("Hola Python")
```

Nos fijamos que el código se colorea automáticamente: *print* aparece en magenta, los paréntesis en negro y el texto entrecomillado (también valdría entrecomillarlo entre comillas simples) de verde. Estos colores (que se pueden personalizar a nuestro gusto) indican que *print* es un comando[38], que *"Hola Python"* es un texto y que los paréntesis son símbolos. Esto nos aporta claridad al código, al margen de dotarlo de un mayor atractivo visual que el aburrido blanco sobre negro al que estamos obligados si ejecutásemos el intérprete desde el símbolo del sistema.

Si por error hubiésemos colocado un espacio en blanco antes del comando, en un principio no parecería que ese detalle impidiese que se ejecutase la función *print* correctamente, pero al pulsar *Enter* nos aparece un error: *SyntaxError: unexpected indent*, que nos indica que es un error de tipo sintáctico por una indentación no

38 Aunque técnicamente es una función, hablamos genéricamente de comando como instrucción ejecutable.

esperada, es decir, por el espacio en blanco que hemos colocado. Incluso aparece una pequeña barra vertical en rojo que nos indica dónde está el error (si fuesen más espacios aparecería una barra más ancha). Si tecleamos la instrucción sin uno (o cualquier) espacio anterior, se ejecutará correctamente. Éste es un ejemplo de que *Python* no nos permite un indentado desordenado, sino que debemos seguir unas pautas determinadas para dar claridad al código y que una persona que inspeccione el código (podemos ser nosotros mismo pasado un tiempo) se encuentre una distribución limpia. Además es muy aconsejable distribuir el código con espacios (cuando se pueda ponerlos) que faciliten la lectura de la instrucción y cada uno de sus componentes. Por ejemplo sería recomendable poner:

```
>>> print ( (2 + 12) * (23 – 15) ) / 54
```

En lugar de:

```
>>> print((2 + 12)*(23–15))/54
```

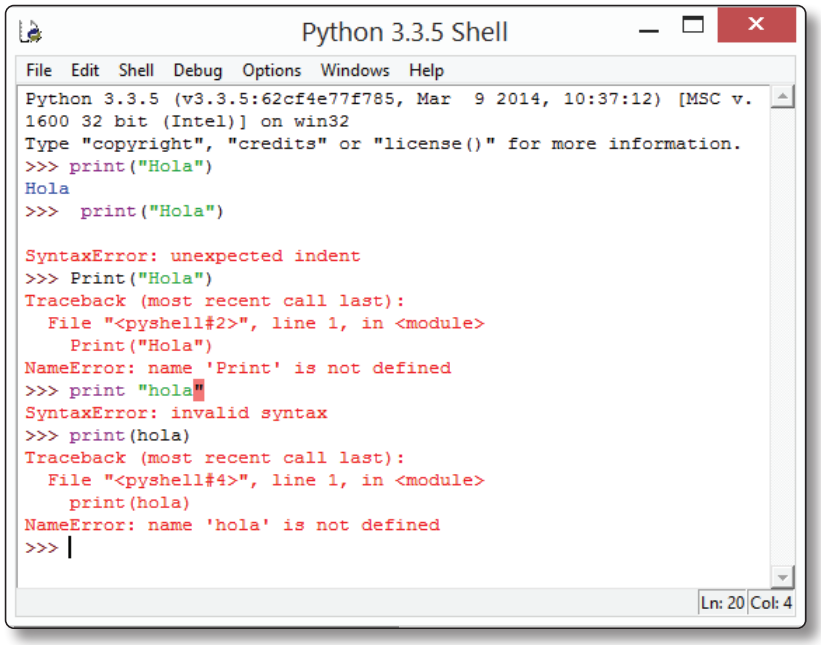
No obstante, ésta última forma es perfectamente válida. Hay incluso una guía de estilo llamada *PEP-8* para generar código en *Python* que es muy recomendable seguir. Algunos de sus puntos fundamentales[39] son:

- Usar sangrías de cuatro espacios.
- Las líneas no deben superar los 79 caracteres.
- Usar líneas en blanco para separar bloques grandes de código, funciones y clases.
- Intentar poner los comentarios en una sola línea cuando sea posible.
- Usar espacios alrededor de operadores y después de las comas, pero no nada más comenzar el paréntesis. Ejemplo: *dato = f1(19, 7) + f2(12, 4)*.
- No usar caracteres *no-ASCII* en los identificadores, salvo que sepamos claramente que la persona que va a leerlo puede implementar códigos más completos (como *Unicode*).

Con anterioridad, también comentamos que *Python* es *case sensitive*, es decir, que distingue entre mayúsculas y minúsculas. Además hablamos de que la función *print()* debe tener un cierto formato para ejecutarse adecuadamente. En la

39 No importa que aún no sepamos qué son algunos de los elementos a los que se refiere. Posteriormente podremos entenderlo completamente.

siguiente imagen aparecen varios ejemplos adicionales en los que no nos atenemos a ese formato, generando errores:



El primer comando es correcto. En el segundo cometemos un error de indentación. En el tercero el intérprete no reconoce el comando *Print,* ya que nombre correcto es *print*. En el cuarto nos aparece un error de sintaxis al olvidarnos los paréntesis; y en el quinto nos informa de que la variable *hola* no está definida. No tardaremos en ver qué son las variables para *Python* y cómo las usa.

Notemos además que para ejecutar las instrucciones en *Python* no debemos acabarlas en ningún símbolo como *'.'* o *';'* al estilo de otros lenguajes de programación, ya que nos daría de nuevo error.

Observamos cómo *Python* obliga a tener distribuido correctamente el código, generando errores si no es así. A documentar correctamente el código no estamos obligados, pero es una práctica más que recomendable, no solo para que posteriormente otras personas puedan saber qué hace el código y cómo, sino para que lo sepamos nosotros mismos pasado un tiempo[40]. Para ello tenemos la ayuda de los *comentarios*, que es texto que el intérprete de *Python* no reconoce como código y lo pasa por alto cuando se ejecuta el programa.

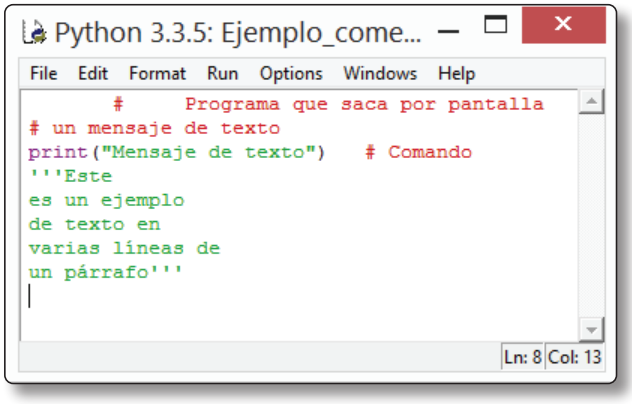
40 A veces ese tiempo es sorprendentemente corto.

Hay dos opciones para introducir esos comentarios:

1. Mediante el uso del carácter #: comentarios en una línea. Colocando el símbolo de almohadilla[41](no tiene que estar justamente al principio de la línea), el texto que aparezca después a lo largo de esa misma línea será interpretado como comentario. Esto solo valdría para una línea, por lo que si queremos poner comentarios en dos de ellas seguidas, tendríamos que colocar al inicio de ambas un carácter #. Para evitarlo usaríamos la siguiente opción.

2. Mediante el uso de los caracteres ”’: comentarios en un párrafo. Colocando el texto entre dos de éstos elementos (comillas triples), todo lo que aparezca en el medio será considerado comentario, teniendo la posibilidad de escribir cómodamente en más de una línea.

En la imagen de la izquierda aparece un ejemplo de *script* que crearemos desde *IDLE*[42] y que guardaremos en nuestra carpeta *Ficheros_Python* del escritorio con el nombre *Ejemplo_comentarios* y con la extensión por defecto *.py*. Una vez hecho podremos ejecutarlo[43] y veremos en el *shell* que solo ejecuta el comando *print*, el resto lo interpreta como comentarios.



De esta manera podremos documentar los programas que realicemos, teniendo en cuenta que debe ser una descripción concisa, destacando los aspectos fundamentales y relevantes, sin extendernos más de lo necesario, ya que eso emborronaría nuestro código.

41 En inglés es el símbolo de peso libra.
42 Estando en el *shell*, *File→New Window* o *Ctrl + n*.
43 Menú *Run→Run Module* o pulsando *F5*.

Nos hemos encontrado ya con varios de los denominados *caracteres especiales*. Una pequeña tabla de ellos será la siguiente:

Carácter/es	Nombre	Función que realizan
” ”	Comillas dobles (apertura y cierre)	Encierran cadenas de caracteres (texto).
( )	Paréntesis (apertura y cierre)	Usados en formatos de funciones.
#	Almohadilla	Precede comentarios de línea.
”’	Comillas triples	Encierran comentarios de párrafo.

1.7 ERRORES EN PYTHON. TIPOS DE ERRORES

A pesar de haber tecleado aún muy poco código en *Python,* ya hemos comprobado lo fácil que es cometer un error. Hemos visto alguno de ellos al, por ejemplo, realizar un mal indentado o cambiar una letra minúscula por una mayúscula. Los errores se dividirán en tres categorías principales:

1. Errores de sintaxis (*Syntax errors*)
2. Errores en tiempo de ejecución (*Runtime errors*)
3. Errores lógicos (*Logic errors*)

Cometeremos un **error de sintaxis** cuando no nos atengamos con exactitud al formato sintáctico que nos impone *Python*, algo que suele ser habitual. Dejarnos sin cerrar unos paréntesis o unas comillas son casos habituales. El intérprete nos informará del error con detalle, por lo que será muy fácil poder subsanarlo. Ejemplos de instrucciones que nos generarían errores de sintaxis son:

```
>>> print(“Hola)
>>> print “hola”
>>> print (“Hola”).
```

Los **errores en tiempo de ejecución** aparecen mientras se está ejecutando el programa en el intérprete y éste detecta operaciones no permitidas, del estilo de introducir un número al esperar el programa un carácter, u operaciones matemáticas no realizables como la división por cero. Al detectar este tipo de errores el intérprete para la ejecución del programa de forma instantánea e informa de ello por pantalla, por lo cual no suele ser difícil de identificar rápidamente la fuente del error. Los

errores de sintaxis en realidad son tratados como un error de tiempo de ejecución, ya que es cuando se ejecuta en el intérprete cuando nos lo indica. No obstante, cuando usemos completos editores de código en *Python,* nos los indicará con anterioridad a la ejecución en el intérprete, con el consiguiente ahorro en tiempo. Un ejemplo sería:

```
>>> print (100 / 0)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print (100 / 0)
ZeroDivisionError: division by zero
```

Los **errores lógicos** ocurren cuando las cosas no salen como nosotros queremos, es decir, si el resultado del programa no es correcto. El motivo puede ser muy variado y difícil de encontrar, dependiendo de la complejidad del código. Puede ocurrir que no obtengamos ningún tipo de error pero que tampoco consigamos el objetivo. En este caso sería necesario revisar totalmente el programa, a veces con la ayuda de un *debugger*[44], para encontrar el error. Es sin duda el tipo de error más difícil de subsanar.

1.8 EL INTÉRPRETE DE PYTHON COMO UNA POTENTE CALCULADORA

Hasta ahora solo hemos visto ejemplos muy sencillos. Antes de adentrarme en explicar detalladamente cómo funciona el lenguaje *Python*, comprobaremos que podemos visualizar el intérprete como una potente calculadora en línea con la que obtener rápidamente valores de expresiones numéricas. Por ejemplo, si necesitamos calcular el valor de la siguiente expresión:

$$\left(\frac{25 \cdot 12^2}{16}\right) \cdot 3$$

Podremos conseguirlo tecleando lo siguiente en el intérprete:

```
>>> ( ( 25 * 12 **2) / 16) * 3
675.0
```

Observamos que el símbolo de la multiplicación es el '*', el de la potenciación el '**' y que hemos usado paréntesis para más claridad. También que el resultado nos aparece con punto decimal. También podemos usar variables, asignarles un valor

44 Depurador. Es un programa que nos ayuda a, paso a paso, ver qué va haciendo nuestro programa.

y calcular en base a ellas. Manteniendo el ejemplo puesto anteriormente:

```
>>> a = 25
>>> b = 12
>>> c = 2
>>> d = 16
>>> e = 3
>>> (( a * b**c) / d) * e
675.0
```

El símbolo *'='* es el usado para asignar a cada una de las variables los valores correspondientes. En breve explicaré en detalle todos estos elementos. De momento sirvan los ejemplos para visualizar la capacidad del intérprete *Python* para calcular expresiones complejas.