INGENIERÍA TÉCNICA EN DISEÑO INDUSTRIAL
509: INFORMÁTICA BÁSICA

Tema 6: Introducción a la programación. El lenguaje Python
Parte I (curso 06/07)

Índice

1. Introducción **2**

2. PythonG: sesiones interactivas **2**

3. Números y expresiones numéricas **4**
- 3.1. Operadores 5
- 3.2. Precedencia y asociatividad 5
- 3.3. Tipos de datos numéricos 7
- 3.4. Conversión implícita 8

4. Excepciones **9**

5. Cadenas y expresiones de cadenas **9**
- 5.1. Operadores 10

6. Variables y asignaciones **11**
- 6.1. Sentencia de asignación 11
- 6.2. Algunas cuestiones importantes sobre variables y asignaciones 12
- 6.3. Identificadores 12

7. Funciones internas **14**

8. Sesión de problemas 1 **16**

9. PythonG: edición y ejecución de programas **17**

10. Escritura de resultados por pantalla **19**

11. Lectura de datos del teclado **21**

12. Funciones externas: módulos **24**
- 12.1. El módulo `math` 24
- 12.2. El módulo `string` 26

13. Comentarios **27**

14. Sesión de problemas 2 **28**

Bibliografía

Libro de apuntes de la asignatura *Metodología y Tecnología de la Programación* (II04 e IG04). Temas 1, 2, 3, 4 y 5. Servicio de Reprografía de la E.S.T.C.E. y `http://marmota.act.uji.es/IG04/pdf/python.pdf`

Estos apuntes para la asignatura *Informática Básica* (509) se han basado en parte en los de *Metodología y Tecnología de la Programación*, realizados por los profesores Andrés Marzal e Isabel Gracia.

M. LUTZ Y D. ASCHER: *Learning Python*. O'Reilly & Associates, 1999.

1. Introducción

En este tema vamos a realizar una incursión en la programación de ordenadores, presentando primero sus fundamentos, sus elementos más básicos y generales (variables, sentencias de selección y repetición, etc.), e introduciendo después conceptos más avanzados y especializados (funciones gráficas), muy adecuados a la orientación profesional de un Ingeniero en Diseño Industrial.

Una ventaja práctica, profesional y personal, de tener ciertas nociones de programación es que puede ser muy útil a la hora de explotar al máximo las posibilidades de muchas herramientas informáticas, en general, y dentro de ellas las orientadas al diseño, en particular. Frecuentemente, estas herramientas permiten personalizar y hacer más cómodo y eficiente su uso mediante el empleo de macros que deben desarrollarse en algún lenguaje de programación, específico o de propósito general.

Nosotros nos introduciremos en la programación utilizando el lenguaje Python, que es un lenguaje de alto nivel de propósito general que nos ofrece las siguientes ventajas, muy interesantes para nuestros intereses:

- Su sintaxis es muy sencilla y elegante, lo que conduce a que los programas escritos en Python resulten fácilmente comprensibles.
- Es un lenguaje muy expresivo que incluye herramientas predefinidas de muy alto nivel. Los programas se diseñan y desarrollan con un nivel de abstracción elevado y se pueden describir acciones complejas utilizando muy pocas líneas de código.
- Es un lenguaje fácil de aprender, por la sencillez de su sintaxis y el elevado nivel de abstracción que permite y por la disponibilidad de herramientas de desarrollo interactivo, típicas de los lenguajes interpretados, que permiten experimentar directa e individualmente con todos los elementos del lenguaje.
- Es un lenguaje portable, multiplataforma. Python se compila y ejecuta en muchos sistemas: Unix, Linux, MS Windows, Macintosh, etc. Los programas que no utilizan una extensión específica para un sistema concreto se ejecutarán exactamente igual sobre cualquier sistema sin modificar los programas.
- Es gratuito y está desarrollado bajo el modelo de software de código abierto ("Open Source") o software libre, es decir, se proporciona el código fuente de Python para que sea posible consultarlo e incluso modificarlo.

Puedes consultar la página web `www.python.org` para obtener mucha más información sobre Python. Nosotros trabajaremos con la versión 2.4.3 sobre Windows, que es la que utilizaréis en las aulas informáticas. Para el desarrollo de programas en Python, emplearemos una herramienta elaborada por un profesor del Departamento de Lenguajes y Sistemas Informáticos de la UJI, David Llorens Piñana: *PythonG*, concretamente la versión 2.1.5. Encontrarás más información en la web de PythonG: `www3.uji.es/~dllorens/PythonG`.

Tras aprender los fundamentos de la programación con Python, trabajaremos con objetos gráficos que podremos combinar, entre ellos y con los elementos de programación de Python, para producir dibujos y animaciones. Los objetos gráficos que manejaremos son similares a algunos de los objetos gráficos que podemos encontrar al programar macros de herramientas de diseño asistido por ordenador, como AutoCAD.

2. PythonG: sesiones interactivas

Vamos a comenzar conociendo PythonG, el entorno de programación que utilizaremos para programar en Python. PythonG significa "Python Gráfico" (o sea, el lenguaje Python con funciones para gráficos) y contiene una serie de herramientas que harán cómoda nuestra tarea de realizar programas. Algunas de las herramientas disponibles en PythonG son: una ventana de ejecución interactiva, un sencillo editor de textos (que nos permitirá escribir o modificar programas) y un depurador (que nos ayudará a encontrar errores en los programas).

En esta sección veremos cómo utilizar la ventana de ejecución interactiva de PythonG, que nos permitirá escribir instrucciones individuales para que las ejecute el intérprete de Python y nos dé una respuesta inmediata a cada una de ellas. Con esta ventana podremos realizar sesiones de trabajo interactivo en las que interactuaremos con el intérprete de Python y probaremos el funcionamiento de las instrucciones.

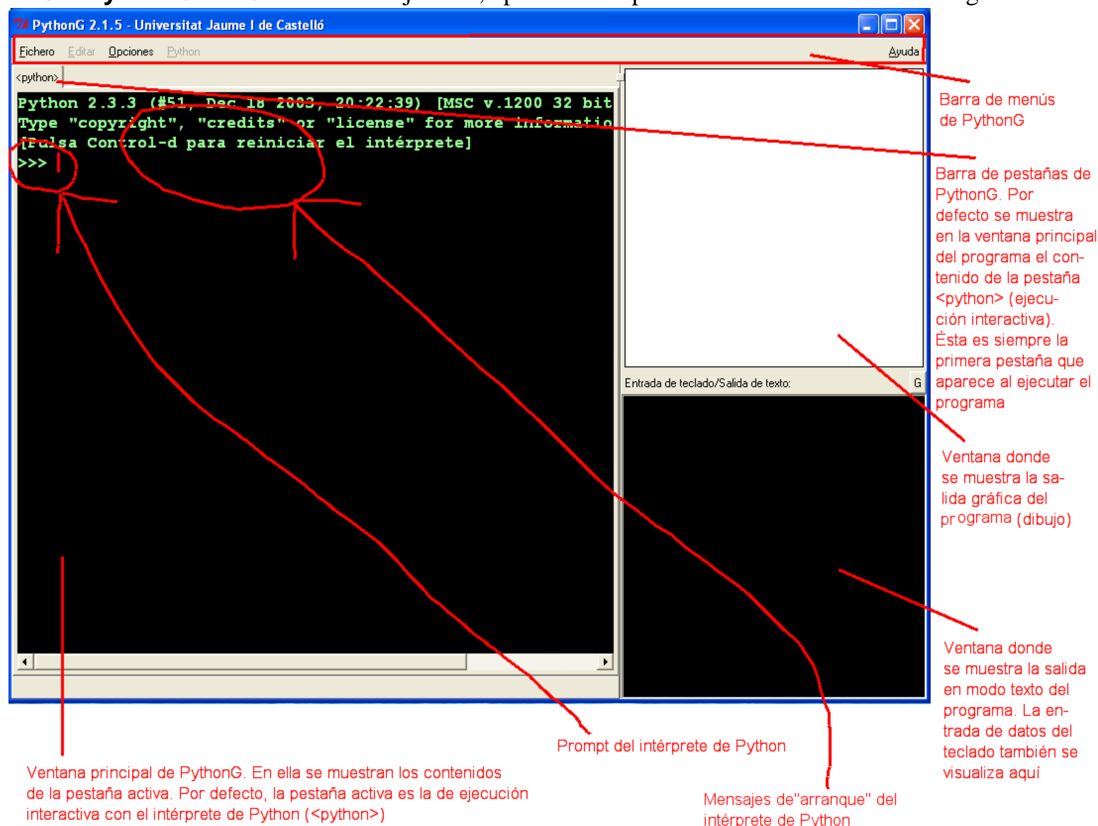
Podéis instalar PythonG desde la página web del programa y también desde la página web de la asignatura en el *Aula Virtual* de la UJI. Para instalarlo, hay que:

1.- Descargarse (p.e. en el Escritorio de Windows) el archivo `zip` correspondiente a la última versión: 2.1.5.

2.- Descomprimirlo mediante `7-zip`: aparecerá una carpeta en el Escritorio (si ahí es donde descargaste el fichero `zip`) denominada `pythonG-windows-2_1_5`.

3.- Para ejecutarlo hay que hacer doble clic en el icono del fichero `pythong.pyw` que se encuentra dentro de dicha carpeta. Se recomienda crear un *acceso directo* en el Escritorio a dicho fichero.

En las aulas informáticas, lo podréis ejecutar a través del botón **Inicio** ▷ **Programas** ▷ **Programació** ▷ **PythonG** ▷ **PythonG 2.1.5**. Cuando lo ejecutes, aparecerá en pantalla una ventana como la siguiente:



Ésta es la ventana de trabajo de PythonG. Podemos distinguir, principalmente, tres áreas en dicha ventana:

(a) Área de trabajo interactivo/edición de programas. Ésta es el área principal: fondo negro, tipo de letra en color verde. En dicha ventana aparece una sesión de trabajo interactiva con el intérprete de Python. Si creamos un nuevo fichero para la edición de un programa, se crea una nueva pestaña. Esta ventana principal mostrará, en general, el contenido de la *pestaña activa*. Podemos cambiar entre pestañas haciendo clic en la pestaña correspondiente a la ventana en la cual deseamos trabajar.

(b) Ventana de salida gráfica. Aparece en la parte superior derecha. Es una pequeña ventana con fondo blanco. En dicha ventana se muestra la salida gráfica de los programas que ejecutemos (y que, obviamente, deben usar funciones gráficas de dibujo).

(c) Ventana de salida en modo texto. Aparece en la parte inferior derecha. Es una pequeña ventana justo debajo de la anterior con fondo negro. En dicha ventana se muestran los mensajes en modo texto resultantes de la ejecución del programa (es también la ventana donde se muestran los datos que introducimos en modo texto).

También podemos observar una barra de menús. En el área de trabajo interactivo, aparecen unas líneas de información (con la versión del intérprete de Python que estamos utilizando) y el *prompt* de Python “>>>”.

Los menús de PythonG son: **Fichero**, que posibilita la gestión de ficheros (crear, abrir, guardar, cerrar, etc.), que normalmente contendrán instrucciones en Python; **Editar**, que dispone de opciones de edición y búsqueda de texto; **Opciones**, de trabajo con PythonG; **Python**, que incluye opciones de ejecución de programas, de edición de programas Python (indentación de bloques de instrucción y comentarios); y **Ayuda**, en el que encontraremos ayuda sobre Python y PythonG (funciones gráficas propias de PythonG y acceso a páginas web con documentación, principalmente). Hablaremos de algunas de las opciones de estos menús conforme vayamos encontrándoles utilidad.

La visión del prompt en el área de trabajo interactivo y del cursor parpadeando a continuación de él nos indica que el intérprete de Python está ocioso, esperando que le demos órdenes para ejecutar. Inicialmente el prompt aparecerá tras las líneas informativas, y después aparecerá nuevamente al acabar la ejecución de cada orden que le demos, indicando que vuelve a quedar a la espera de que le transmitamos otra orden.

Para darle una orden al intérprete de Python, la escribiremos mediante el teclado a continuación del prompt, finalizando con la pulsación de la tecla de retorno de carro. Observa en la siguiente ventana de ejecución interactiva cómo hemos ejecutado sucesivamente una serie de órdenes y el intérprete nos ha devuelto a continuación de cada una de ellas, en la línea siguiente, el resultado de su ejecución.

PythonG 2.1.5 - Universitat Jaume I de Castelló

Fichero Editar Opciones Python Ayuda

```
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1200 32 bit
Type "copyright", "credits" or "license" for more informatio
[Pulsa Control-d para reiniciar el intérprete]
>>> 5 + 4.25
9.25
>>> 31 - 14
17
>>> 7 * 12
84
>>> 19.5 / 3
6.5
>>>
```

Entrada de teclado/Salida de texto:

En este caso, las órdenes que le hemos dado al intérprete de Python han sido expresiones numéricas, las cuales el intérprete evalúa y devuelve su resultado. Vamos a hablar de las expresiones numéricas inmediatamente, en la siguiente sección. Antes, vamos a comentar algunas posibilidades de edición de la línea de órdenes, para que nos pueda resultar más cómodo escribir las órdenes que iremos estudiando.

Cuando estamos escribiendo una orden podemos modificarla antes de pulsar la tecla de retorno de carro, si detectamos algún error de tecleo o simplemente cambiamos de idea respecto a la orden que queremos ejecutar. Las teclas de desplazamiento del cursor a la izquierda y a la derecha nos permiten situar el cursor en la posición que deseemos de la orden. La tecla de retroceso nos permite borrar el carácter situado a la izquierda del cursor y la de borrado el de la derecha. Y en cualquier posición que esté situado el cursor, podremos insertar nuevos caracteres siempre a su izquierda.

La ventana de ejecución interactiva "recuerda" toda la historia de órdenes ejecutadas, y podemos retroceder y avanzar en dicha historia mediante las teclas de desplazamiento del cursor hacia arriba y hacia abajo, respectivamente. Una vez seleccionada la orden que se desea "recuperar", podremos modificarla y ejecutarla pulsando el retorno de carro. Así podremos escribir más rápidamente órdenes muy parecidas a otras ejecutadas con anterioridad.

Cuando deseemos acabar una sesión interactiva y salir de PythonG podemos elegir la opción **Fichero** ▷ **Salir** o pulsar ante el prompt `Ctrl+x` y `Ctrl+c` de forma consecutiva.

3. Números y expresiones numéricas

Mediante las expresiones numéricas podremos calcular valores partiendo de otros valores previos y combinándolos con los operadores numéricos. De momento utilizaremos sólo números para expresar los valores que incluimos en las expresiones numéricas. Más adelante incorporaremos variables y resultados de funciones a las expresiones.

Al escribir expresiones numéricas tenemos que ceñirnos a las reglas sintácticas que exige Python. Así el intérprete podrá entenderlas correctamente y calcular su resultado. En esta sección vamos a conocer esas reglas y otros conceptos que influyen en cómo el intérprete evalúa las expresiones numéricas, para así poder escribir correctamente las expresiones y que éstas "expresen" el cálculo que deseamos realizar.

En la ventana de ejecución interactiva podemos escribir expresiones numéricas directamente ante el prompt,

para que el intérprete las evalúe y conocer su resultado. Posteriormente, cuando escribamos programas, las expresiones numéricas no las utilizaremos como instrucciones por sí mismas sino que formarán parte de otras instrucciones, como ya veremos.

3.1. Operadores

En el ejemplo de la sección anterior, has podido observar que podemos realizar operaciones de suma, resta, producto y división. Para ello, utilizamos los operadores `+`, `-`, `*` y `/`. Estos operadores actúan sobre dos valores numéricos (operandos), por lo que decimos que son *binarios*.

Los símbolos `+` y `-` también se usan como operadores *unarios*, es decir, que actúan sobre un único operando. En este caso, representan las operaciones identidad y cambio de signo.

Ejemplo:
```
»> +27
27
»> -8.625
-8.625
»> ----13
13
»> -(5 + 11)
-16
»> +(2.5 - 15)
-12.5
```

Podemos emplear los paréntesis para encerrar una subexpresión. De esta manera su resultado constituirá un operando para la expresión que la contenga.

Otros dos operadores binarios que debemos conocer son `%` y `**`. El primero es el operador *módulo* y obtiene el resto de la división entera de su operando de la izquierda entre el de la derecha.

Ejemplo:
```
»> 23% 4
3
»> 10.5% 3
1.5
»> 4.5% 1.25
0.75
```

El operador `**` representa la exponenciación o potencia, es decir, obtiene el resultado de elevar su operando de la izquierda al de la derecha.

Ejemplo:
```
»> 3**4
81
»> 81**0.5
9.0
»> 2.3**8.1
851.12916368918115
```

3.2. Precedencia y asociatividad

Podemos escribir expresiones que contengan varios operadores, que además pueden ser distintos. Pero cuando combinamos varios operadores en una expresión, su evaluación se ve afectada por la *precedencia* y *asociatividad* de los operadores. La precedencia y asociatividad establecen en qué orden se aplican los operadores para evaluar la expresión.

Unos operadores tienen prioridad sobre otros en la evaluación de expresiones, es decir, tienen mayor precedencia, con lo que al evaluar una expresión su aplicación siempre "precederá" a la de los otros. En la siguiente tabla se indican los niveles de precedencia de los operadores, siendo el nivel 1 el más prioritario y el nivel 4 el menos prioritario.

operación	exponenciación `**`	identidad `+`	cambio de signo `-`	producto `*`	división `/`	módulo `%`	suma `+`	resta `-`
precedencia	1	2		3			4	

Ejemplo:
```
»> 36.0 / 4 + 5 * 3
24.0
»> -5**2 + 17.5% 11.5
-19.0
»> +6.125 * -3 - 20 / 5
-22.375
```

Cuando, al aplicar la precedencia de operadores en la evaluación de una expresión, ocurre que se deben evaluar dos o más operadores con un mismo nivel de precedencia, entonces se tiene en cuenta la asociatividad de los operadores. La asociatividad indica cómo se agrupan las operaciones con un mismo nivel de precedencia para su realización, y puede ser:

- *de izquierda a derecha* para las operaciones de producto, división, módulo, suma y resta, es decir, con estas operaciones se comienza a operar desde la izquierda; o
- *de derecha a izquierda* para la operación de exponenciación, es decir, con esta operación se comienza a operar desde la derecha.

Las operaciones unarias de identidad y cambio de signo no se ven afectadas por la asociatividad.

Ejemplo:
```
»> 28 / 7 * 2
8
»> 2.5 + 7 - 1.25 - 3
5.25
»> 13 * 3% 10
9
»> 2**3**3
134217728
```

Fíjate que el cálculo realizado en esta última expresión es $2^{(3^3)} = 2^{27}$ debido a que la asociatividad de la exponenciación es de derecha a izquierda

Las precedencias y asociatividades establecidas se pueden "romper" empleado paréntesis. Así podremos expresar cualquier cálculo que deseemos realizar. De hecho, hay cálculos que requieren el empleo de paréntesis en las expresiones para expresarlos adecuadamente.

Ejemplo:
```
»> 36.0 / (4 + 5) * 3
12.0
»> 28 / (7 * 2)
2
»> 13 * (3% 10)
39
»> (2**3)**3
512
```

Cuando los paréntesis denotan el mismo orden de evaluación que se obtiene al aplicar las precedencias y asociatividades, entonces son innecesarios. Compara los cálculos que se realizan en el siguiente ejemplo empleando paréntesis innecesarios con los realizados sin paréntesis en los ejemplos anteriores.

Ejemplo:
```
»> ((36.0 / 4) + (5 * 3))
24.0
»> ((28 / 7) * 2)
8
»> (13 * 3)% 10
9
»> 2**(3**3)
134217728
```

3.3. Tipos de datos numéricos

Hasta ahora hemos estado empleando números en las expresiones sin preocuparnos de si éstos eran enteros o reales. Pero Python sí tiene en cuenta el hecho de que un número sea entero o real, y el empleo de unos u otros en las expresiones conlleva importantes diferencias que hay que tener en cuenta.

En general, cada dato que utilizemos en las instrucciones Python tendrá su *tipo de datos*. El tipo de un dato determina su representación interna y las operaciones que se pueden realizar con él. Más adelante estudiaremos algunos tipos de datos no numéricos, pero de momento vamos a centrarnos en los tipos numéricos.

Python tiene predefinidos cuatro tipos de datos numéricos: enteros, flotantes, enteros largos y complejos. Nosotros trabajaremos sólo con números enteros y flotantes, que introduciremos en esta sección. Los números enteros largos y complejos no los estudiaremos, ya que exceden los objetivos de nuestra introducción a la programación.

Los números flotantes son el "sucedáneo" informático de los números reales matemáticos como ya vimos en el Tema 3 de teoría[1]. Debemos tener en cuenta que los números reales son infinitos y que un número real puede tener una cantidad infinita de cifras decimales. En un ordenador tendremos que contentarnos con números finitos y cantidades finitas de cifras decimales, aunque puedan ser relativamente grandes. Recuerda lo que estudiamos en el Tema 3 sobre representación interna de los números de coma flotante. En adelante, les llamaremos simplemente *flotantes*.

De igual manera, los números enteros en un ordenador representarán un subconjunto finito de los números enteros matemáticos. Una diferencia (la más importante desde nuestro punto de vista) entre números enteros y flotantes la encontramos en su representación externa, es decir, en cómo se los escribimos al intérprete y en cómo nos los muestra él al devolvernos un resultado. Los números enteros se representan sólo con una parte entera (`23`) quizá precedida de un signo (`+23`,`-23`), mientras que los números flotantes se representan con una parte entera, precedida o no de un signo, y obligatoriamente con:

- un punto y una parte decimal (`23.847`), pudiendo ser ésta nula (`23.0`), o
- la letra `e` o `E` y un exponente entero (`23e5`, `23E13`), pudiendo precederlo de un signo (`23847e-3`), o
- ambos (`2384.7e-2`).

En un número flotante, la letra `e` o `E` y un exponente tiene el significado de multiplicar por 10 elevado al exponente; es decir, `23e5` representa al número $23 \cdot 10^5$ y `2384.7e-2` al $2384{,}7 \cdot 10^{-2}$.

La segunda diferencia entre enteros y flotantes está en el rango de números que pueden representar. Los enteros sólo pueden representar números en el rango $[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$, mientras que los flotantes representan un rango mayor. Sin embargo, cabe hacer notar que Python *convierte automáticamente* los enteros a *enteros largos* cuando nos "salimos" de este rango. Los enteros largos pueden representar un rango *mucho mayor* de números, ya que emplean internamente 64 bits en lugar de los 32 asignados a los enteros "normales".

Ejemplo:
```
»> 2**34.0
17179869184.0
»> 2**34
17179869184L
```

Otras diferencias son que los enteros se representan internamente utilizando menos memoria (32 bits) que los flotantes (64 bits), a no ser que empleemos enteros largos (también 64 bits) y que las operaciones con enteros suelen ser más rápidas que con flotantes. Por estas razones, conviene que utilicemos números flotantes solamente cuando sean necesarias cifras decimales en los cálculos que vayamos a realizar. También hay que destacar que, utilizando los números flotantes, te encontrarás con algún resultado curioso. Fíjate en este ejemplo:

Ejemplo:
```
»> 0.2
0.20000000000000001
```

Recuerda que en el Tema 3 vimos que la parte fraccionaria de un número decimal puede no tener una equivalente binaria exacta (recuerda que la representación en binario del número decimal exacto 3,3 producía un número decimal periódico). Eso está ocurriendo aquí: `0.2` se corresponde con un número binario periódico, con lo que al representarlo en binario se trunca al número (finito) de bits disponibles. Y posteriormente, cuando el intérprete nos devuelve el resultado, nos devuelve la representación exacta en decimal del número binario que tiene almacenado. Éste será muy, muy aproximado, pero no será el mismo. Cuando conozcamos y utilicemos la sentencia de salida `print` esto no ocurrirá, ya que `print` utiliza menos cifras fraccionarias para representar los números, con lo que éstos se redondearán y en la ventana de ejecución veremos normalmente los números esperados.

[1] Codificación de la información

Para acabar hablaremos de una diferencia muy importante entre números enteros y flotantes que hay que tener siempre en cuenta. Todas las operaciones numéricas, excepto la división, producen valores equivalentes (enteros o flotantes) resultantes de las mismas si los valores de partida también son equivalentes (enteros o flotantes). La única diferencia que apreciaremos será el tipo del resultado. Fíjate en estos ejemplos.

Ejemplo:
```
»> 4.0 + 5.0 + 13.0
22.0
»> 4 + 5 + 13
22
»> 4.0 ** 6.0
4096.0
»> 4 ** 6
4096
```

En cambio, la división que se realiza cuando los valores de partida son enteros es una *división entera*, es decir, se deshecha la parte fraccionaria y se produce únicamente la parte entera como resultado (y en tipo entero). Si queremos que el resultado sea flotante e incluya la parte fraccionaria, la expresión debe incluir operandos flotantes, tal y como puedes ver en el siguiente ejemplo.

Ejemplo:
```
»> 17 / 3
5
»> 17 / 3.0
5.666666666666667
»> 17.0 / 3
5.666666666666667
»> 17.0 / 3.0
5.666666666666667
```

3.4. Conversión implícita

Podemos formar expresiones que incluyan números enteros y flotantes. En este caso, el intérprete de Python convierte implícitamente los números enteros a sus equivalentes flotantes al evaluar la expresión y produce un resultado flotante. Éste es el mismo tipo de conversión que se producía de entero a entero largo cuando, en un ejemplo anterior, calculábamos la expresión `2**34`.

Ejemplo:
```
»> 12 + 3.5 * 3
22.5
»> (2 - 0.5) / 3
0.5
```

No obstante, ten en cuenta que la expresión se evalúa siguiendo el orden que se deriva de las precedencias y asociatividades de los operadores y de los paréntesis que contenga, y la conversión se aplica en el momento en el que siguiendo este orden se encuentra el primer número flotante. Analiza los siguientes resultados teniendo en cuenta esta consideración.

Ejemplo:
```
»> 11 / 4 + 7.0
9.0
»> 11.0 / 4 + 7
9.75
```

Ejercicio 1. Evalúa a mano el resultado de las siguientes expresiones y después contrasta tus resultados con los que produce el intérprete de Python:

a) `( -5 ** 2 + 10 ) * 3`

b) `35.3 - 53 / 10`

c) `1 + 5 % 1.5 * 8 / 2`

d) `- ( (-2) ** 5 + 18 / 5 )`

e) `65 / 3 / 4.0 * 3`

f) `44 % 10 + 7 - 25 ** 50e-2`

---

4. Excepciones

¿Qué ocurre si intentamos realizar este "cálculo": `34 + "Juan"`? ¿Qué resultado obtenemos en la ventana de ejecución interactiva?

Ejemplo:
```
»> 34 + "Juan"
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Lo que ha ocurrido es que el intérprete de Python ha detectado un error al intentar evaluar la expresión y lo que vemos en la ventana es un mensaje de error que nos proporciona información del mismo. En este caso, ha encontrado un *error de tipo de dato en la operación*: se intenta sumar un número entero con una secuencia de caracteres, lo cual no es posible.

Los errores en Python se denominan *excepciones* y cuando se produzcan, tanto ahora en la ventana interactiva como cuando ejecutemos programas, el intérprete abortará su ejecución y mostrará un mensaje informativo indicando el error que se ha producido. A continuación podemos ver algunos errores que podemos encontrarnos.

Ejemplo:
```
»> 16.0% * 9
  File "<input>", line 1
    16.0% * 9
          ^
SyntaxError: invalid syntax
»> 2**
  File "<input>", line 1
    2**
      ^
SyntaxError: invalid syntax
»> 4 + 6) - 8
  File "<input>", line 1
    4 + 6) - 8
         ^
SyntaxError: invalid syntax
```

Cuando el intérprete detecta un *error sintáctico* en la línea que acabamos de enviar a ejecutar, indica el elemento que ha producido el error y nos muestra el correspondiente mensaje (en el ejemplo, hemos empleado el color gris claro para resaltar el lugar donde se produce el error; eso no lo hace PythonG). Ninguna de las tres expresiones que le hemos introducido está sintácticamente bien construida. En la primera, el intérprete encuentra un operador cuando busca un operando para `%`. En la segunda, no encuentra el operando que falta para `**`. Y en la tercera, encuentra un paréntesis cerrado, cuando anteriormente no se ha abierto ninguno.

Ejemplo:
```
»> 5 / (2 - 2)
Traceback (most recent call last):
  File "<input>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Cuando se intenta realizar una división en la que el denominador resulta ser cero, el intérprete de Python no lo permite y nos muestra un mensaje de *error de división por cero*.

Fíjate que hay errores de distinto tipo y que el intérprete nos informa exactamente de qué tipo de error se ha producido. Debemos fijarnos en los mensajes de error, cuando se produzcan, ya que nos proporcionan una gran ayuda para saber qué fallos hemos cometido y poder corregirlos adecuadamente. Conforme aprendamos más elementos de Python podremos encontrarnos otros errores distintos. Ya iremos conociéndolos.

5. Cadenas y expresiones de cadenas

Además de números, en Python podemos trabajar con información textual en la que podemos representar nombres de personas, nombres de empresas, títulos de canciones, direcciones web, etc.

La información textual la representaremos mediante *cadenas*, que es otro tipo de datos proporcionado por Python, concretamente uno de los tipos de datos *estructurados* y *secuenciales*. En esta sección presentaremos los

aspectos más básicos de las cadenas y más adelante las estudiaremos con mayor detalle. También estudiaremos otro tipo de datos estructurado y secuencial: las *listas*.

Una cadena en Python es una secuencia de caracteres (alfabéticos, numéricos, de puntuación, espacios en blanco, etc.) delimitada por comillas simples o dobles. Las comillas no forman parte de la cadena, sólo marcan su inicio y su fin. Si indicamos el comienzo de una cadena con una comilla simple, debemos finalizarla también con una comilla simple; y equivalentemente si empleamos comillas dobles. No podemos utilizar ambos tipos de comillas en la formación de una cadena.

Ejemplo:
```
"esto es un ejemplo..."       '----> y esto es otro ejemplo $#!¿?'
'Abel Lopez Fernandez'        "www.python.org"
'Computer World'              "3 + 2*5"
```

En PythonG, las cadenas se escriben en color naranja, con lo que son fácilmente identificables. Sin embargo, debes tener en cuenta que *esto sólo ocurre en las pestañas de edición de programas* (lo veremos posteriormente) y no en la pestaña (ventana) de ejecución interactiva.

5.1. Operadores

Con cadenas podemos utilizar el operador de concatenación, que es un operador binario cuyos dos operandos serán cadenas y que produce como resultado la "adición" de ambas cadenas. El símbolo empleado para representar la concatenación es +, el mismo que se emplea para la suma de números.

Ejemplo:
```
»> 'Abel ' + "Lopez" + ' ' + 'Fernandez'
'Abel Lopez Fernandez'
»> "www" + "." + "python" + "." + "org"
'www.python.org'
```

Ten en cuenta que, aunque el símbolo utilizado es el mismo, la operación suma es distinta a la operación concatenación. Además, no se puede sumar o concatenar una cadena y un número, como nos lo indica el *error de tipo* que obtenemos al intentarlo en el siguiente ejemplo (y que, recuerda, es el mismo error que obteníamos en el ejemplo que pusimos al comienzo de la sección anterior).

Ejemplo:
```
»> 1000 + 512
1512
»> '1000' + '512'
'1000512'
»> '1000' + 512
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Otro operador, que también es binario, es el de repetición de cadenas. Éste se representa con *, el mismo símbolo que el empleado en el producto de números, requiere dos operandos: una cadena y un entero, y produce como resultado la cadena formada por la concatenación repetida de la cadena consigo misma el número de veces que indique el entero.

Ejemplo:
```
»> 6 * "veces"
'vecesvecesvecesvecesvecesveces'
»> "_-_/" * 15
'_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/_-_/'
```

Si combinamos los operadores de cadenas en una expresión, el de repetición tiene mayor precedencia que el de concatenación. Para alterar la precedencia necesitamos emplear paréntesis.

Ejemplo:
```
»> "la" + "una" * 4
'launaunaunauna'
»> ("la" + "una") * 4
'launalaunalaunalauna'
```

Ejercicio 2. Evalúa a mano el resultado de las siguientes expresiones y después contrasta tus resultados con los que produce el intérprete de Python:

a) `'ta' + 'za de' + ' ...' + 'cafe'`

b) `'I' + 'E'*3`

c) `3*'w' + '.' + ('i' + 'e')*2 + '.org'`

d) `2 * ('  <<<' + '*'*4 + '>'*3)`

---

6. Variables y asignaciones

Frecuentemente, al realizar programas y también en las sesiones de trabajo interactivo con el intérprete de Python, nos interesará que ciertos valores (de partida o calculados) se recuerden para ser empleados posteriormente.

Ejemplo: Si deseamos conocer el valor en euros de dos cantidades en pesetas, primero de cada una de ellas por separado y después de su suma, podemos dar la siguiente secuencia de órdenes al intérprete:

```
»> 47923 / 166.386
288.02303078384
»> 94176 / 166.386
566.00915942447079
»> 47923 / 166.386 + 94176 / 166.386
854.03219020831079
```

En estas órdenes, estamos escribiendo varias veces los mismos valores (con el riesgo que conlleva de equivocarnos en alguna cifra) que además representan los mismos conceptos. Después de estas líneas, si deseamos calcular otro cambio de pesetas a euros deberemos escribir otra vez el valor de un euro en pesetas. O peor aún, si deseamos acumular los valores en euros que suponen estas dos cantidades en pesetas a otros valores que calculemos posteriormente, deberemos volver a teclear estas mismas expresiones completas.

6.1. Sentencia de asignación

En Python podemos asociar valores a variables, de manera que a partir de esta asociación podemos utilizar las variables en lugar de los valores en las expresiones que formemos. Para asociar un valor a una variable empleamos la sentencia de *asignación*, ejecutando una línea con la siguiente forma:

```
nombre_de_variable = expresión_que_produce_un_valor
```

En una sentencia de asignación el símbolo = separa a su izquierda y a su derecha partes de la sentencia que tienen misiones distintas, razón por la que no se puede invertir el orden de aparición en la sentencia. Es decir, `expresión_que_produce_un_valor = nombre_variable` no es equivalente. En una sentencia de asignación se realiza el siguiente proceso:

1.- Se evalúa la expresión que está a la derecha del símbolo = y se obtiene un valor resultante.

2.- Se asocia el nombre de variable a la izquierda del símbolo = con el valor obtenido.

A partir de entonces la variable hará referencia al valor que se le ha asociado. Observa como podríamos escribir el ejemplo anterior empleando variables y sentencias de asignación.

Ejemplo:

```
»> cantidad1_ptas = 47923
»> cantidad2_ptas = 94176
»> cambio_1euro_ptas = 166.386
»> cantidad1_euros = cantidad1_ptas / cambio_1euro_ptas
»> cantidad2_euros = cantidad2_ptas / cambio_1euro_ptas
»> total_euros = cantidad1_euros + cantidad2_euros
»> total_euros
854.03219020831079
```

De esta manera, hemos asociado variables a todos los valores implicados en los cálculos. A partir de este momento, si necesitamos cualquiera de ellos el intérprete los recordará y podrá referenciarlos mediante las variables.

Observa que, tras cada asignación, no obtenemos la salida por pantalla que estamos acostumbrados a ver al evaluar expresiones. La sentencia de asignación no produce ninguna salida por pantalla. Para averiguar el valor asociado a una variable, (en el ejemplo, `total_euros`) tenemos que evaluar la variable (como expresión que sólo contiene una variable) después de realizar la asignación.

6.2. Algunas cuestiones importantes sobre variables y asignaciones

Podemos asignar distintos valores a una misma variable todas las veces que queramos. Pero debemos considerar que la sentencia de asignación es una operación por un lado destructiva y por otro constructiva. Es decir, cuando hacemos una asignación el valor anterior que tuviese asociado se pierde y queda reemplazado por el nuevo valor que le asociamos, de manera que en cada instante el valor que una variable tiene asociado es siempre el último que se le ha asignado.

Ejemplo:
```
»> a = 30
»> a
30
»> a = 363
»> a
363
»> a = "Otro valor"
»> a + '.'
'Otro valor.'
```

No confundas una sentencia de asignación con una ecuación matemática. No son lo mismo. La sentencia de asignación realiza unas acciones claramente especificadas, como hemos descrito anteriormente, y aunque se emplea el símbolo = no pretende definir la "igualdad" de sus partes izquierda y derecha. Analiza el siguiente ejemplo e interprétalo en base al significado de la asignación.

Ejemplo:
```
»> contador = 10
»> contador
10
»> contador = contador + 1
»> contador
11
```

Las variables en Python *no se declaran* antes de utilizarlas (en otros lenguajes de programación sí). No obstante, debes tener cuidado de no intentar utilizar una variable (en una expresión) que aún no tenga un valor asociado. Observa el siguiente ejemplo.

Ejemplo:
```
»> entorno = 'PythonG ' + version
Traceback (most recent call last):
  File "<input>", line 1, in ?
NameError: name 'version' is not defined
```

El intérprete de Python detecta que la variable `version` no tiene ningún valor asociado, lo que impide la evaluación normal de la expresión. El mensaje de error nos indica que se ha producido un *error de nombre*, es decir, el intérprete no conoce el nombre de la variable que hemos intentado utilizar. Para que esta asignación pueda realizarse correctamente debemos asignar previamente un valor a `version`.

Ejemplo:
```
»> version = '2.1.5'
»> entorno = 'PythonG ' + version
»> entorno
'PythonG 2.1.5'
```

6.3. Identificadores

El nombre de una variable es un *identificador*. En Python utilizamos los identificadores (nombres) para hacer referencia a distintos elementos que podemos emplear en las instrucciones y los programas. Una variable es uno de estos elementos. Más adelante conoceremos otros, como las funciones, que también referenciaremos mediante sus identificadores (sus nombres).

Un identificador en Python es una secuencia de uno o más caracteres, de manera que los caracteres empleados sólo pueden ser letras, dígitos y el carácter de subrayado _ , y el primer carácter del identificador no puede ser un dígito. Las letras mayúsculas y minúsculas se consideran diferentes, y sólo podemos emplear las letras propias del alfabeto inglés (o sea, no podemos utilizar vocales acentuadas, la "ñ", la "ç", etc.)

Ejemplo: Los siguientes identificadores son válidos en Python:

```
nombre     _nombre     _nom_bre       nom_bre10    n7ombre
NOMBRE     _NOMBRE     _500Nombre     Nombre       NomBre
Primer_Apellido        SEGUNDO_APE    notas10      media
EDAD       SeXo        DNI            Apellido1    Apellido_2
```

Y los siguientes no son válidos en Python:

```
10negritos     4x4          1Apellido    Curso'3
Porcentaje%    Smith&Sons   Mañana       Canal+
```

Los identificadores válidos `nombre`, `NOMBRE`, `Nombre` y `NomBre` son distintos porque se diferencia entre mayúsculas y minúsculas.

`Primer Apellido` (con un espacio en medio) no es un identificador sino dos. Se pueden "separar palabras" en un identificador mediante el carácter de subrayado, como en `Primer_Apellido`.

No puedes utilizar como identificadores las palabras reservadas de Python. Éstas sólo pueden emplearse con el significado que ya tienen especificado en el lenguaje. No es preciso que te aprendas la lista de palabras reservadas. En las pestañas de edición de programas de PythonG (que veremos posteriormente), las palabras reservadas se resaltan en color rojo. Así es que si al escribir un identificador ves que queda resaltado en rojo, piensa otro nombre. Más adelante iremos conociendo algunas palabras reservadas. No obstante, debes tener en cuenta que en la pestaña (ventana) de ejecución interactiva no se utiliza ningún color de resaltado (siempre es fuente de color verde sobre fondo negro).

Por otra parte, es aconsejable que los nombres de los identificadores den alguna información sobre el dato que representan; por ejemplo: `nombre`, `nota`, `media`, `edad`, `sexo`, `dni`, `area`, `radio`, etc. Compara el siguiente ejemplo con el ejemplo anterior de conversión de pesetas a euros y suma de cantidades en euros. Analiza cuál resulta más comprensible a primera vista. Fíjate que en ambos se realizan las mismas operaciones; sólo cambian los identificadores empleados.

Ejemplo:

```
»> x = 47923
»> y = 94176
»> z = 166.386
»> u = x / z
»> v = y / z
»> w = u + v
»> w
854.03219020831079
```

Una última cosa sobre los identificadores: no los confundas con las cadenas. Por un lado, las cadenas requieren ser delimitadas por comillas y los identificadores no. Y por otro, sus misiones en las instrucciones y los programas son distintas. Obtendrás mensajes de error del intérprete o resultados inesperados si utilizas una cadena en lugar de un identificador o viceversa.

Ejercicio 3. Evalúa a mano el resultado de las siguientes asignaciones y después contrasta tus resultados con los que produce el intérprete de Python:

a)
```
»> x = 8
»> x = 3*x + x/4
»> x
```

b)
```
»> nota1 = 10
»> nota2 = 5
»> nota_final = 0.6*nota1 + 0.4*nota2
»> nota_final
```

c)
```
»> nombre = 'Pepe'
»> simbolo = '&'
»> frase = nombre + " " + simbolo*3 + " cia."
»> frase
```

d)
```
»> x = 5
»> x = x / 2
»> y = x**2 - 3*x + 7
»> y + 10
```

7. Funciones internas

Mediante los operadores numéricos y de cadenas realizamos "operaciones" sobre los números y las cadenas y obtenemos como "resultados" nuevos números o cadenas. Python ofrece otro medio de realizar "operaciones" y obtener "resultados" utilizando números o cadenas: las *funciones*.

Como iremos viendo, mediante funciones podremos realizar muy diversas acciones. Pero de momento, vamos a estudiar sólo unas pocas funciones que permiten realizar operaciones con los tipos de datos que conocemos: números y cadenas. Estas funciones que vamos conocer son *funciones internas*, que ya están incorporadas en el intérprete de Python y que podemos utilizar directamente, sin tener que realizar ninguna acción previa. Más adelante veremos otras funciones internas y también funciones externas (no incorporadas en principio en el intérprete). Una de estas funciones internas es la función `abs`, que obtiene el valor absoluto de un número.

Ejemplo:
```
»> abs(-8)
8
»> abs(15.25)
15.25
```

Vamos a aprovecharnos de la función `abs` para introducir algunas cuestiones importantes sobre las funciones. Una función tiene un nombre, un identificador, que sigue las mismas reglas de formación que los nombres de variables y que permite al intérprete referenciar a la correspondiente función, igual que un nombre de variable le permite referenciar a la correspondiente variable. De esta forma, `abs` es el nombre de la función valor absoluto. No utilices para tus variables nombres que sean nombres de funciones, si no quieres que el intérprete "pierda" su conocimiento de las funciones.

Ejemplo:
```
»> abs(-8)
8
»> abs = 17
»> abs(-8)
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: 'int' object is not callable
»> abs
17
```

Al asignar el valor `17` al nombre `abs` provocamos que el intérprete "olvide" que `abs` es el nombre de una función y ahora `abs` pasa a ser el nombre de una variable. Tras asignar el valor `17`, el intérprete nos devuelve un *error de tipo* al intentar calcular `abs(-8)`, puesto que ahora el intérprete ya no considera que `abs` sea una función sino que considera que es una variable. Al evaluar esta variable sí obtenemos su valor.

Tras escribir el nombre de la función, el (los) valor (valores) sobre el (los) que se va a aplicar se debe(n) encerrar necesariamente entre paréntesis. Si no, obtendremos un error.

Ejemplo:
```
»> abs -8
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: unsupported operand type(s) for -:
  'builtin_function_or_method' and 'int'
```

El valor al que se aplica la función (abs, en este caso) es su *argumento* y puede ser cualquier expresión. Esta expresión se evalúa y su resultado es el argumento (valor) que utilizará la función.

Ejemplo:
```
»> n = 6.75
»> abs(n)
6.75
»> abs(n - 10)
3.25
»> abs(2*n - 30)
16.5
```

Las funciones también pueden requerir que el argumento sea de un cierto tipo. Por ejemplo, la función abs necesita que el tipo de su argumento sea numérico; no acepta una cadena como argumento.

Ejemplo:
```
»> abs('argumento no válido')
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: bad operand type for abs()
```

Finalmente, el resultado devuelto por las funciones es un valor de un cierto tipo que puede utilizarse en cualquier contexto donde puedan aparecer valores de ese tipo. Por ejemplo, la función abs devuelve un número, por lo que podrá formar parte de cualquier expresión numérica.

Ejemplo:
```
»> num = abs(-2)
»> num
2
»> num ** abs(4 - 7)
8
»> num = num + abs(-(17% 6))
»> num
7
```

Vamos a conocer otras tres funciones internas. Dos de ellas las necesitaremos en breve, cuando veamos cómo leer datos en nuestros programas.

La función int convierte a tipo entero ("normal" o largo) su argumento. Éste puede ser un entero, un flotante o una cadena. Si su argumento es entero, devuelve el mismo número que recibe; si es flotante, lo trunca y devuelve sólo su parte entera; y si es cadena, devuelve el número entero que se corresponde con la secuencia de números especificada en la cadena (la cadena debe contener sólo caracteres númericos, y quizá un signo + o -, y éstos deben representar un número dentro del rango del tipo entero).

Ejemplo:
```
»> int(14)
14
»> int(276.987)
276
»> int(0.4567e2)
45
»> int('-32767')
-32767
»> int('1111111111111111')
1111111111111111L
»> int('diez')
Traceback (most recent call last):
  File "<input>", line 1, in ?
ValueError: invalid literal for int(): diez
»> int('34.56')
Traceback (most recent call last):
  File "<input>", line 1, in ?
ValueError: invalid literal for int(): 34.56
```

La función `float` convierte a tipo flotante su argumento. Éste puede ser un flotante, un entero o una cadena. Si su argumento es flotante, devuelve el mismo número que recibe; si es entero, devuelve un flotante equivalente (con parte decimal nula); y si es cadena, devuelve el número flotante que se corresponde con la secuencia de caracteres especificada en la cadena (la cadena debe contener sólo caracteres númericos, `+`, `-`, `.`, `e` o `E`, y éstos deben ajustarse al formato de representación de enteros o flotantes).

Ejemplo:
```
»> float(0.2345e2)
23.449999999999999
»> float(78)
78.0
»> float('234')
234.0
»> float('75.91e-3')
0.075910000000000005
»> float('0.75.91e-3')
Traceback (most recent call last):
  File "<input>", line 1, in ?
ValueError: invalid literal for float(): 0.75.91e-3
```

La función `str` convierte a tipo cadena su argumento. Éste puede ser una cadena, un entero o un flotante. Si su argumento es una cadena, la devuelve tal como la recibe; si es entero o flotante, devuelve una representación del número como cadena (que se ajustará al formato de representación de enteros o flotantes).

Ejemplo:
```
»> str('una cadena')
'una cadena'
»> str(9715)
'9715'
»> str(2.0103e5)
'201030.0'
```

Ya hemos comentado que los valores devueltos por las funciones pueden aparecer en cualquier contexto donde se espere un valor del tipo correspondiente. Así, también pueden utilizarse funciones como argumentos de otras funciones. Y en general, las funciones pueden formar parte de las expresiones de muy diversas formas.

Ejemplo:
```
»> abs(float('-2.05e-3'))
0.0020500000000000002
»> str(int(35.964))
'35'
»> abs(int(-0.1e4)+218)
782
»> 9 ** float(str(5)+'e-1')
3.0
```

Ejercicio 4. Evalúa a mano el resultado de las siguientes expresiones y después contrasta tus resultados con los que produce el intérprete de Python:

a) `int(0.7 + 0.9) + abs(-333)`

b) `abs(50 / float(str(4)))`

c) `'El numero ' + str(int(-27.00183))`

d) `float(1 + 5% int(1.5) * 8 / 2)`

---

8. Sesión de problemas 1

Ejercicio 5. Evalúa a mano el resultado de las siguientes asignaciones y después contrasta tus resultados con los que produce el intérprete de Python:

a)
```
>> x = 2
>> -2*x**3 + 5*x**2 - 15
```

b)
```
>> n = 3
>> m = 4
>> 2.0*n + (7*m + 5) / 10
```

c)
```
>> iva = 16
>> precio = 2000
>> total = precio * (1 + float(iva)/100)
>> int(total)
```

d)
```
>> tiempo_minutos = 314.5
>> horas = int(tiempo_minutos) / 60
>> minutos = int(tiempo_minutos)% 60
>> segundos = int(60*(tiempo_minutos - int(tiempo_minutos)))
>> str(horas) + " h.  " + str(minutos) + " m.  " + str(segundos) + "s."
```

Ejercicio 6. Evalúa a mano el resultado de las siguientes asignaciones y después contrasta tus resultados con los que produce el intérprete de Python:

a)
```
>> nombre = 'Javier'
>> apell1 = 'Montes'
>> apell2 = 'Berges'
>> apell1 + ' ' + apell2 + ', ' + nombre
```

b)
```
>> alerta = 'CUIDADO'
>> animal = 'PERRO'
>> '!'*5 + alerta + ' con el ' + animal + '!'*10
```

c)
```
>> separador = '<' + '-'*3 + '$'*5 + '-'*3 + '>'
>> nombre1 = ' Stanley '
>> nombre2 =' Oliver '
>> (nombre1 + separador + nombre2)*2
```

Ejercicio 7. Asigna a seis variables: `canastas1`, `canastas2`, `canastas3`, `intentos1`, `intentos2` e `intentos3`, los valores enteros que desees, teniendo en cuenta que el valor de `canastas1` debe ser menor o igual que `intentos1`, `canastas2` menor o igual que `intentos2` y `canastas3` menor o igual que `intentos3`. Estas variables representan el número de canastas de cada tipo que ha convertido un equipo de baloncesto en un partido y el número de intentos realizados. A continuación, diseña sucesivamente expresiones para obtener:

a) la cantidad de puntos obtenidos mediante canastas de 2 puntos;

b) la cantidad de puntos total;

c) el porcentaje de efectividad en canastas de 3 puntos; y

d) el porcentaje de efectividad global.

En los apartados c) y d), realiza las conversiones de tipo necesarias para que el resultado sea correcto.

Siempre que sea conveniente, almacena en variables los resultados de las expresiones y utilízalos en el diseño de las expresiones posteriores.

---

9. PythonG: edición y ejecución de programas

Hasta este momento hemos realizado solamente trabajo interactivo con el intérprete de Python. Es decir, le hemos dado órdenes individuales en una única línea y él ha ejecutado cada una de estas órdenes. Si la orden consistía en evaluar una expresión, nos devolvía su resultado. Pero si consistía en una asignación, asociaba el

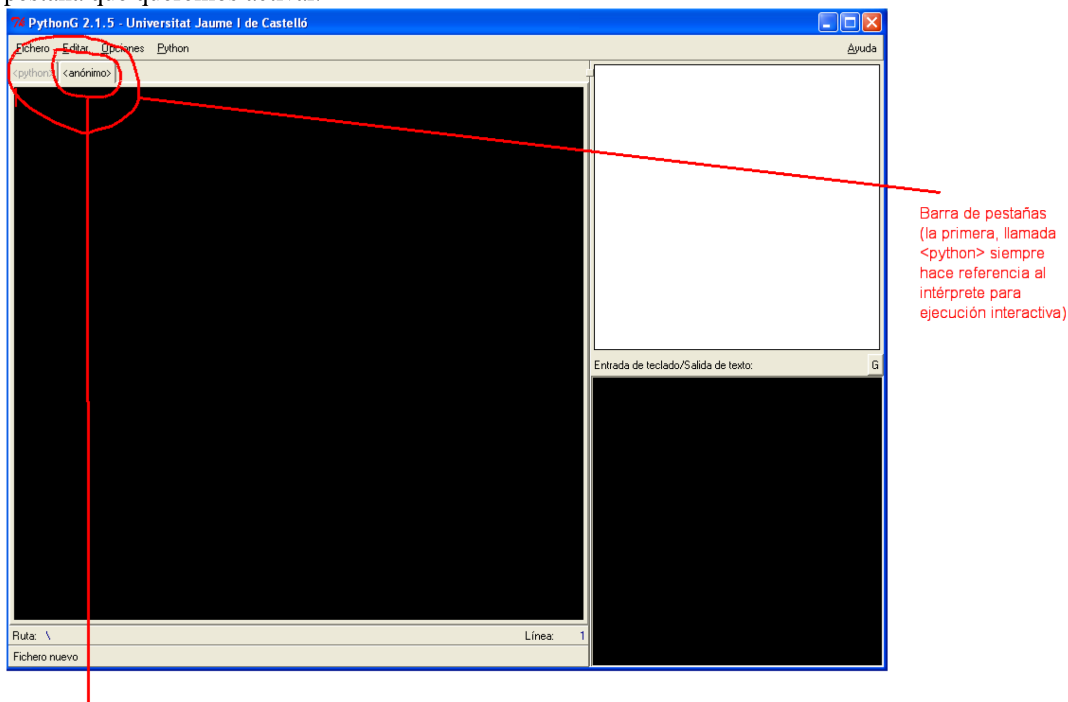
resultado de la expresión a la variable y no nos devolvía nada. Posteriormente, evaluábamos la variable para conocer el resultado que tenía asignado.

Todas las instrucciones de Python se pueden ejecutar interactivamente. Pero el modo interactivo, aunque resulta muy apropiado para experimentar con las instrucciones, no es siempre el ideal para explotar todas las posibilidades de programación en Python. Piensa en lo que pasaría si en distintas sesiones de trabajo quisiéramos ejecutar una misma serie de instrucciones. Como cada vez que abandonamos la ventana de ejecución interactiva de PythonG "perdemos" todas las órdenes que hayamos escrito, en cada sesión de trabajo tendríamos que escribir, una a una, todas esas instrucciones.

Para poder repetir la ejecución de una serie de instrucciones sin tener que escribirlas cada vez, primero tenemos que *escribirlas y guardarlas en un fichero* en memoria secundaria para conservarlas, y después, cada vez que queramos ejecutarlas, tenemos que *pedirle al intérprete que ejecute el programa almacenado en el fichero*, para que vaya tomando una a una las instrucciones que contiene y las ejecute.

PythonG integra un sencillo editor de texto, con el que podremos escribir y guardar ficheros de instrucciones en Python con cierta comodidad, y permite ejecutar estos ficheros de instrucciones. De esta forma, editaremos y ejecutaremos nuestros programas en Python.

Para crear una ventana de edición tan sólo tenemos que elegir la opción **Fichero** ▷ **Nuevo** desde la ventana de PythonG. Así aparecerá *una nueva pestaña* que se activará por defecto, es decir, que pasará a ser la pestaña activa y la ventana principal mostrará sus contenidos (en principio, el fichero está vacío, obviamente). La nueva pestaña creada se denomina `<anónimo>`. En PythonG no podemos tener dos ficheros (pestañas) con el mismo nombre y eso incluye el nombre `<anónimo>`, lo cual quiere decir que no podemos estar editando al mismo tiempo dos ficheros nuevos: antes de crear un nuevo fichero será preciso guardar (y, por tanto, darle un nombre) el fichero nuevo que estuviésemos editando. Para activar una pestaña (y eso incluye la pestaña de ejecución interactiva) y que, por tanto, la ventana principal pase a mostrar el contenido de dicha pestaña, basta con hacer clic en el nombre de la pestaña que queremos activar.



Cada vez que creamos un fichero nuevo, se crea una pestaña de edición con nombre <anónimo>. Podemos tener tantas pestañas de edición como queramos (sirven para editar programas). La pestaña de ejecución interactiva no muestra "colores" al contrario que las pestañas de edición, donde las palabras reservadas aparecen en color rojo y las cadenas en color naranja, por ejemplo

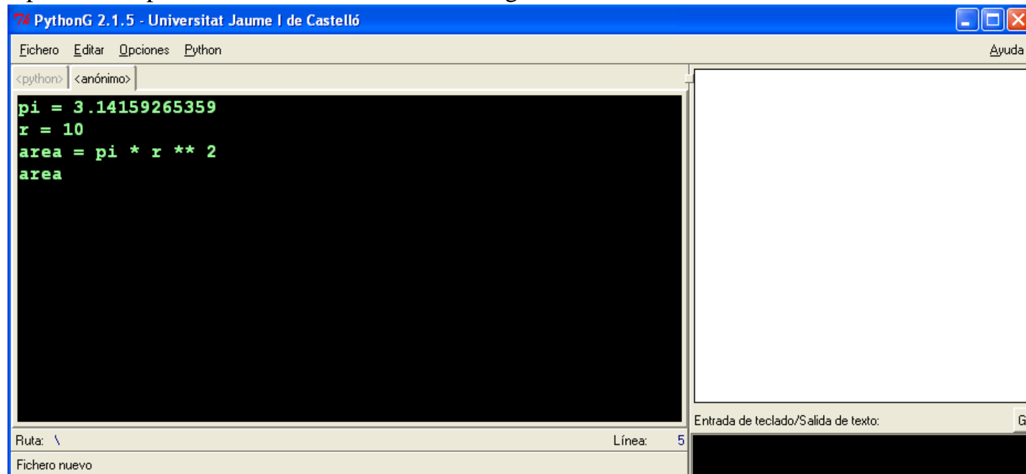
En una pestaña de edición no existe una "historia de órdenes ejecutadas", por lo que al situar el cursor en una línea y pulsar la tecla de retorno de carro no se obtiene una "copia de la orden ante el prompt" (aquí, de hecho, no aparece un prompt), como ocurre en la pestaña de ejecución interactiva.

Introduzcamos un programa en la pestaña de edición. Vamos a aprovechar lo que conocemos de Python para escribir un programa que calcule el área de un círculo de 10 centímetros de radio. Si en lugar de la pestaña de edición, utilizásemos la pestaña de ejecución interactiva, sabemos que escribiendo la siguiente secuencia de órdenes realizaríamos el cálculo pedido.

```
»> pi = 3.14159265359
```

```
»> r = 10
»> area = pi * r ** 2
»> area
314.15926535900002
```

Escribimos estas mismas cuatro instrucciones en la pestaña de edición. Ellas forman el programa que realiza el cálculo pedido. La pestaña de edición se verá de la siguiente forma:



Probemos a ejecutarlo. No hace falta guardar el contenido de la pestaña en un fichero para ejecutar el programa que hemos editado, pero *resulta muy conveniente hacerlo*.

Bueno, pues guardemos el contenido de la pestaña en un fichero. Para ello, podemos elegir la opción **Fichero** ▷ **Guardar** o **Fichero** ▷ **Guardar como...**. Cuando vamos a grabar el contenido de la pestaña por primera vez en un fichero, ambas opciones son equivalentes. Nos aparecerá el típico cuadro de diálogo para guardar ficheros de Windows. En él podemos elegir el directorio donde vamos a situar el fichero y el nombre con el que lo vamos a almacenar. Lo llamaremos circulo.py.

Es conveniente que "apellidemos" a todos los ficheros que contengan programas en Python con la extensión "py". La existencia de esta extensión en el nombre del fichero conlleva, entre otras características deseables, que Windows asocie al fichero el típico y simpático icono de los programas en Python.

Elegimos la opción **Guardar** y ya podemos ejecutar el programa. La opción **Python** ▷ **Ejecutar** del menú de PythonG inicia la ejecución del fichero. Al lanzar la ejecución del programa, observaremos el resultado en la ventana inferior de la parte derecha de PythonG (la ventana que muestra la interacción en modo texto con el programa; ya sabes, la ventana pequeña con el fondo negro). Sin embargo, no aparece el resultado del cálculo realizado, tal y como ocurría al ejecutar las órdenes interactivamente ¿Qué ha ocurrido?

Existe una diferencia entre la ejecución de programas y la ejecución interactiva, que explica el comportamiento observado y que debemos tener en cuenta: al ejecutar un programa, la evaluación de una expresión (como instrucción del programa) no conlleva la impresión de su resultado, acción que sí se produce al ordenar su evaluación interactiva. Python proporciona una instrucción específica para mostrar valores por pantalla y necesitamos utilizarla en los programas para este fin.

10. Escritura de resultados por pantalla

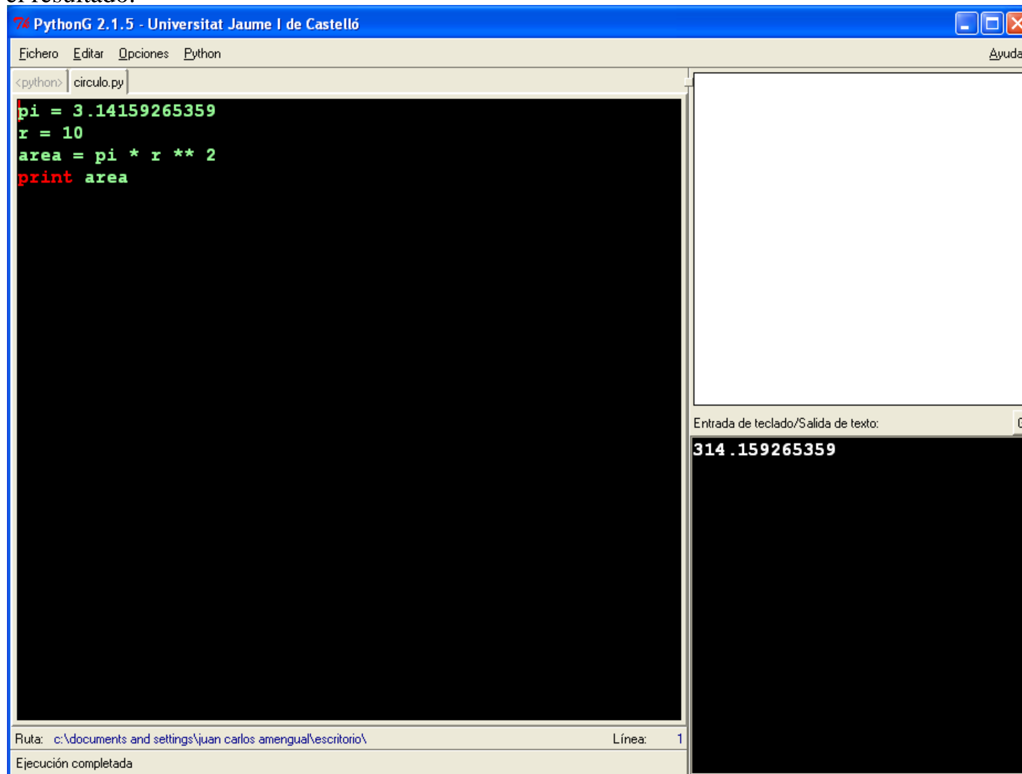
La sentencia `print` muestra por pantalla cuantos valores le indiquemos, separándolos entre sí mediante comas. La emplearemos de la siguiente forma:

`print` *`valor_1, valor_2, ..., valor_n`*

Cambiemos ahora la evaluación de la última expresión, del programa ejemplo que estamos desarrollando, por una sentencia `print`. La nueva versión del programa en la pestaña de edición quedará como sigue.

```
pi = 3.14159265359
r = 10
area = pi * r ** 2
print area
```

Guardamos (**Fichero** ▷ **Guardar**) (recuerda que ya le habíamos asignado el nombre circulo.py al fichero) y ejecutamos (**Python** ▷ **Ejecutar**) el programa. Ahora sí veremos en la ventana pequeña de la parte inferior derecha el resultado.



Hemos empleado la sentencia `print` para escribir un único valor flotante, pero podemos escribir otros tipos de valores (enteros o cadenas), podemos escribir más de un valor en la misma sentencia y podemos escribir varias sentencias `print`. Vamos a modificar el programa como sigue.

```
print "Cálculo del área de un círculo a partir de su radio."
pi = 3.14159265359
r = 10
area = pi * r ** 2
print 'Un círculo de', r, 'cm. de radio'
print 'tiene un área de', area, 'centímetros cuadrados.'
```

Guardamos, ejecutamos y en la ventanita de ejecución en modo texto obtendremos:

```
Cálculo del área de un círculo a partir de su radio.
Un círculo de 10 cm. de radio
tiene un área de 314.159265359 centímetros cuadrados.
```

Fíjate que, en la salida por pantalla, la sentencia `print` no produce las comillas delimitadoras de las cadenas y los valores que separamos por comas aparecen separados por un espacio en blanco.

Es conveniente que acompañemos la escritura de resultados con cadenas que contengan textos explícitos acerca del significado de los mismos. También podemos proporcionar mensajes que expliquen el objetivo o el funcionamiento del programa. En principio, el "usuario" de un programa desconocerá qué significan los valores que aparecen en pantalla y estos mensajes le proporcionarán ayuda para entenderlos.

Los valores que escribimos en las sentencias `print` pueden ser expresiones en general, no es preciso que sean simples números, cadenas o variables. El siguiente programa producirá exactamente la misma salida por pantalla que el anterior. Analízalo y compáralo con él.

```
print "Cálculo del área" + " de un círculo " + "a partir de su radio."
pi = 3.14159265359
r = 10
print 'Un círculo de', str(r) + ' cm. de radio'
print 'tiene un área de', pi * r ** 2, 'centímetros cuadrados.'
```

Realmente, aunque este programa produce la misma salida por pantalla que el anterior, resulta innecesariamente un poco más complejo de entender. Así pues, optaremos por quedarnos con la versión anterior. Siempre conviene expresar las sentencias de un programa (todas ellas, no sólo las sentencias `print`) de la forma más sencilla posible. De todas formas, recuerda que mediante `print` podemos escribir expresiones en general. En alguna ocasión puede resultar útil para producir la salida por pantalla que deseamos.

`print` ofrece muchas más posibilidades de formatear la salida de un programa en su ejecución, pero en principio las que hemos comentado serán suficientes para nosotros.

Ejercicio 8. Escribe un programa que asigne a dos variables, que representen dos lados consecutivos de un rectángulo, los valores $7{,}55$ y $14{,}2$, y a continuación calcule el valor del área de dicho rectángulo y lo escriba por pantalla. Recuerda que el área de un rectángulo se obtiene como el producto de dos de sus lados consecutivos.

Ejercicio 9. Escribe un programa que asigne a dos variables, que representen la base y la altura de un triángulo, los valores 5 y $18{,}7$, y a continuación calcule el valor del área de dicho triángulo y lo escriba por pantalla. Recuerda que el área de un triángulo se obtiene como el producto de la base por la altura dividido por dos.

Ejercicio 10. Escribe un programa que inicialmente asigne a tres variables las cadenas `'**'`, `'--'` y `'||||'`, y a continuación mediante expresiones de cadena obtenga las cadenas siguientes y las escriba por pantalla. Cuando lo consideres conveniente, almacena en variables los resultados de algunas expresiones y utilízalos en el diseño de expresiones posteriores.

```
---**---
---**---||||---**---||||---**---||||---**---||||---**---||||---**---
```

---

11. Lectura de datos del teclado

En la sección anterior hemos escrito un programa que, cuando se ejecuta, es muy expresivo explicando qué hace y qué resultados proporciona. Pero sólo permite conocer el área de un círculo de 10 cm. de radio, ya que el valor del radio está prefijado en el programa. Como programadores y conocedores de Python, tenemos la opción de editar el programa, cambiar el valor del radio por otros valores que nos interesen y ejecutarlo, para obtener los valores de las correspondientes áreas. Esto es posible, aunque no deseable.

Más aún, pensemos exclusivamente como usuarios del programa. En este caso, querremos limitarnos a "usar" el programa, utilizando todas las posibilidades que ofrezca cuando se ejecute pero nada más. Así, un programa que sólo pueda realizar un cálculo empleando unos valores concretos no será muy útil para un usuario. El programa tendría más utilidad si permitiese al usuario decidir qué valores se emplean en el cálculo que realiza el programa.

Esta ampliación de la utilidad del programa podríamos conseguirla con algún mecanismo de introducción de datos al programa. Así, el usuario podría introducir el valor del radio que le interesase y conocer el valor del área del círculo cuyo radio fuese el valor introducido.

Vamos a conocer un mecanismo proporcionado por Python para poder introducir a los programas datos desde el teclado: la función interna `raw_input`. En principio, utilizaremos esta función de la siguiente forma:

`raw_input(`*`cadena_mensaje`*`)`

Cuando se ejecuta así, muestra la *`cadena_mensaje`* por pantalla y a continuación queda en espera de que el usuario teclee caracteres. Cuando el usuario finaliza la introducción de caracteres pulsando la tecla retorno de carro, devuelve *una cadena* con todos los caracteres tecleados excluyendo el retorno de carro final.

Vamos a modificar el programa utilizando la función `raw_input`. Fíjate que, puesto que es una función que devuelve un valor, debemos almacenarlo en una variable si queremos utilizarlo en instrucciones posteriores. Si no conservamos el valor devuelto, no podemos emplearlo después.

```
print "Cálculo del área de un círculo a partir de su radio."
pi = 3.14159265359
r = raw_input('Dime el radio (en centímetros): ')
area = pi * r ** 2
print 'Un círculo de', r, 'cm. de radio'
print 'tiene un área de', area, 'centímetros cuadrados.'
```

Guardamos y ejecutamos, y en la ventana de ejecución observaremos inmediatamente lo siguiente:

```
Cálculo del área de un círculo a partir de su radio.
Dime el radio (en centímetros):
```

En este momento, la función `raw_input` espera que tecleemos un valor del radio y pulsemos la tecla de retorno de carro. Tecleamos el valor 20 y obtenemos lo siguiente:

```
Cálculo del área de un círculo a partir de su radio.
Dime el radio (en centímetros): 20
Traceback (most recent call last):
  File "circulo.py", line 4, in ?
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Se produce un *error de tipo* al calcular el área. ¿Qué ha sucedido? Pues que intentamos realizar un cálculo numérico empleando una *cadena*. Recuerda que hemos dicho que `raw_input` devuelve una cadena con los caracteres tecleados. Aunque estos sean los dígitos de un número, el tipo del dato que devuelve esta función será cadena.

Aquí nos van a resultar de utilidad las funciones internas de conversión de tipos que vimos anteriormente: `int` y `float`. Emplearemos estas funciones para convertir el valor devuelto por `raw_input` al tipo adecuado para el uso que vayamos a realizar del dato. Solamente cuando el dato que esperemos sea una cadena emplearemos directamente la cadena devuelta por `raw_input`. Así, modificamos el programa como sigue:

```
print "Cálculo del área de un círculo a partir de su radio."
pi = 3.14159265359
r = int(raw_input('Dime el radio (en centímetros): '))
area = pi * r ** 2
print 'Un círculo de', r, 'cm. de radio'
print 'tiene un área de', area, 'centímetros cuadrados.'
```

Y al ejecutar obtendremos:

```
Cálculo del área de un círculo a partir de su radio.
Dime el radio (en centímetros): 20
Un círculo de 20 cm. de radio
tiene un área de 1256.63706144 centímetros cuadrados.
```

Realmente, el argumento *`cadena_mensaje`* es opcional en la función `raw_input`; es decir, podríamos emplear la función sin argumentos. En este caso, escribiremos `raw_input()`, ya que los paréntesis no son opcionales, son obligatorios.

Pero, como has podido comprobar en el programa que venimos realizando, hemos utilizado el argumento *`cadena_mensaje`* para proporcionar al usuario información sobre el dato que se espera que teclee. Si no lo utilizamos, la petición del dato al usuario se hará sin información alguna, con lo que un usuario que no conozca el programa no sabrá qué debe teclear en ese momento. Es muy conveniente que proporcionemos este argumento a la función `raw_input` y que lo utilicemos precisamente para especificarle al usuario qué dato espera recibir el programa.

Por otra parte, ahora que pedimos el valor del radio con un mensaje explícito y el valor tecleado por el usuario queda reflejado en la ventanita de ejecución en modo texto, resulta excesivo mostrar en la línea siguiente otra vez su valor como parte de la impresión de resultados. Vamos a corregirlo.

```
print "Cálculo del área de un círculo a partir de su radio."
pi = 3.14159265359
r = int(raw_input('Dime el radio (en centímetros): '))
area = pi * r ** 2
print 'Su área mide', area, 'centímetros cuadrados.'
```

Tenemos total libertad para elegir los mensajes que empleamos tanto en la petición de datos con `raw_input` como en la impresión de resultados o de información del programa con `print`. No obstante, conviene elegir adecuadamente estos mensajes para que la interacción del programa con el usuario resulte equilibrada, ni carente ni sobrada de información.

Estudiemos ahora algunos ejemplos de programas en Python. En los tres primeros ejemplos se presentan tres programas distintos que resuelven un mismo problema: leen dos valores enteros positivos por el teclado, calculan su suma y su promedio, y escriben el resultado por la pantalla. Pero cada uno realiza esta tarea de forma distinta. Compara los tres programas, estudiando sus semejanzas y diferencias.

Ejemplo: El primer programa realiza la tarea pedida siguiendo la estrategia de leer primero los dos valores en dos variables distintas, y realizar después la suma directa de ambas variables.

```
num1 = int(raw_input("Primer numero: "))
num2 = int(raw_input("Segundo numero: "))
suma = num1 + num2
media = suma / 2.0
print "Suma:", suma
print "Media:", media
```

Ejemplo: La estrategia del segundo programa consiste en acumular sobre una variable, cuyo valor inicial es cero, cada nuevo valor que se lee, leyendo cada valor sobre una variable distinta.

```
suma = 0
num1 = int(raw_input("Primer numero: "))
suma = suma + num1
num2 = int(raw_input("Segundo numero: "))
suma = suma + num2
media = suma / 2.0
print "Suma:", suma
print "Media:", media
```

Ejemplo: El tercer programa sigue la misma estrategia que el segundo pero utilizando una única variable para leer los dos valores. ¿Es correcto utilizar una única variable para leer? ¿Por qué?

```
suma = 0
num = int(raw_input("Primer numero: "))
suma = suma + num
num = int(raw_input("Segundo numero: "))
suma = suma + num
media = suma / 2.0
print "Suma:", suma
print "Media:", media
```

Ejemplo: Vamos a realizar un programa que, a partir de una cierta cantidad de dinero, calcula y escribe el desglose en el mínimo número de monedas de 20, 5 y 1 céntimo.

Como tenemos la moneda unidad (1 céntimo) y, además, cada moneda representa una cantidad que es un múltiplo entero de todas las anteriores, el problema se resuelve de forma exacta a partir del número de monedas que caben de la moneda de valor máximo y dejando la cantidad restante para evaluar con los restantes valores de monedas. Procediendo sucesivamente con las monedas ordenadas de mayor a menor valor y con los sucesivos restos, obtendremos la solución al problema.

Observa en el siguiente ejemplo la secuencia de divisiones que se realiza para desglosar 73 céntimos en el mínimo número de monedas de 20, 5 y 1 céntimo:

```
              73 | 20                        13 | 5
              13   3                          3   2
desglosar en monedas de 5 y 1 ctmo.   monedas de 20 ctmo.   monedas de 1 ctmo.   monedas de 5 ctmo.
```

El programa que sigue esta estrategia es el siguiente.

```
cantidad = int(raw_input("Dime una cantidad de dinero a desglosar: "))

numero20 = cantidad / 20
resto20  = cantidad% 20
print "Monedas de 20 céntimos:", numero20

numero5 = resto20 / 5
print "Monedas de 5 céntimos:", numero5

resto5  = resto20% 5
print "Monedas de 1 céntimo:", resto5
```

Ejercicio 11. Realiza un programa que lea cinco valores enteros positivos por el teclado, calcule su suma y su promedio, y escriba el resultado por la pantalla.

Ejercicio 12. Escribe un programa que, a partir de una cierta cantidad de dinero dada, calcule y escriba el desglose en el mínimo número de monedas de 50, 20, 10, 5, 2 y 1 céntimo. Básate en la estrategia empleada en el ejemplo anterior para la realización de este programa.

12. Funciones externas: módulos

Tal como ya comentamos, Python dispone de funciones externas que, en principio, no están incorporadas en el intérprete, y por lo tanto las desconoce, pero que se pueden incorporar, y entonces ya podrá utilizarlas. Estas funciones externas se agrupan en módulos según el tipo de servicio que colectivamente proporcionan. Existen multitud de módulos que abarcan un amplio rango de servicios: sistema operativo, criptografía, redes, web, gráficos, imágenes, multimedia, etc. En los módulos, no sólo se definen funciones. También pueden incluir otros elementos del lenguaje (constantes, excepciones, etc.). Podrás encontrar documentación (en inglés) sobre los módulos estándar de Python en el enlace Library Reference, accesible en C:\Python24\Doc\Python24.chm o a través de la opción del menú de PythonG **Ayuda ▷ Documentación de Python (web en inglés)...** (en este caso, se requiere conexión a Internet).

Nosotros vamos sólo a conocer como usar módulos en nuestros programas y a presentar algunas funciones y constantes de dos módulos básicos. Para incorporar una o varias funciones o constantes al intérprete debemos conocer previamente los nombres de las funciones o constantes y el módulo en el que residen. Así si queremos importar la función `sqrt`, que calcula la raíz cuadrada de un número, del módulo matemático `math`, ejecutaremos la orden:

```
from math import sqrt
```

Si la ejecutamos ante el prompt del intérprete interactivo, a partir de ese momento podremos emplear la función `sqrt` para realizar cálculos. Observa en este ejemplo que antes de importarla, el intérprete la desconoce, como deducimos del *error de nombre* que obtenemos al intentar utilizarla.

```
»> sqrt(169)
Traceback (most recent call last):
  File "<input>", line 1, in ?
NameError: name 'sqrt' is not defined
»> from math import sqrt
»> sqrt(169)
13.0
```

También podemos incluir una instrucción de importación de funciones o constantes en un programa. De igual manera que en el intérprete interactivo, a partir de entonces podremos utilizarla en el código del programa. Podemos importar más de una función o constante en una única instrucción. Con la siguiente instrucción importamos al mismo tiempo que `sqrt`, la constante `pi` y la función `sin` (seno).

```
from math import sqrt, pi, sin
```

Y también podemos importar en una sóla instrucción todas las funciones y constantes definidas en un módulo, mediante la instrucción:

```
from math import *
```

12.1. El módulo `math`

El módulo `math` contiene, entre otras, las siguientes funciones y constantes:

Funciones	
`sin(x)`	Seno de `x` (en radianes)
`cos(x)`	Coseno de `x` (en radianes)
`tan(x)`	Tangente de `x` (en radianes)
`exp(x)`	El número e elevado a `x`
`log(x)`	Logaritmo natural (en base e) de `x`
`log10(x)`	Logaritmo decimal (en base 10) de `x`
`sqrt(x)`	Raíz cuadrada de `x`
Constantes	
`pi`	Aproximación del número π, 3,14159265359
`e`	Aproximación del número e, 2,71828182846

Ahora que conocemos que la constante `pi` está definida en el módulo `math`, vamos a mejorar el programa de cálculo del área de un círculo. Lo que vamos a hacer es importar `pi` en el programa en lugar de definirlo nosotros. Siempre que podamos debemos utilizar los elementos definidos en los módulos, ya que estos se proporcionan precisamente para hacer más cómoda y rápida la labor de realizar programas. En este caso, `pi` es sólo una constante y no nos ha costado mucho definirla nosotros. Pero piensa en otros elementos un poco más laboriosos de desarrollar, como las funciones `sqrt`, `log` o `sin`.

```
print "Cálculo del área de un círculo a partir de su radio."
from math import pi
r = int(raw_input('Dime el radio (en centímetros): '))
area = pi * r ** 2
print 'Su área mide', area, 'centímetros cuadrados.'
```

Veamos un programa un poco más elaborado. Observa en el siguiente ejemplo cómo se van realizando cálculos parciales y almacenándolos en variables para después aprovecharlos en expresiones posteriores. De esta forma, se puede evitar escribir expresiones complejas descomponiéndolas en expresiones más sencillas.

Ejemplo: Vamos a escribir un programa que calcula y escribe por pantalla el volumen y el área de un cilindro. Los datos de entrada son el radio del círculo que constituye la base y la altura del cilindro.

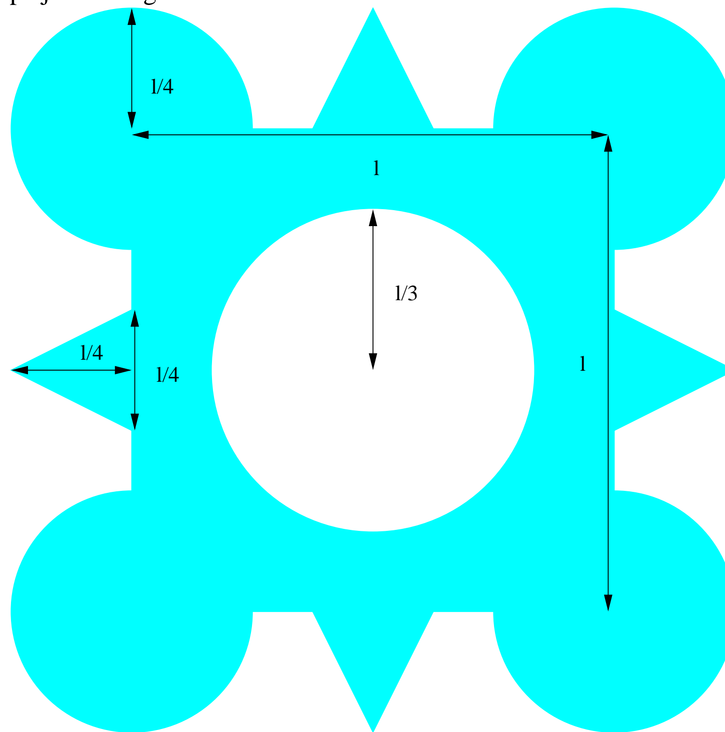
En este programa tenemos que calcular dos valores, un volumen y un área. Calcularemos primero el área del círculo que forma la base del cilindro, ya que este área se utilizará para calcular el volumen y el área del cilindro. A continuación, el volumen se puede calcular directamente, pero para calcular el área del cilindro se obtendrá previamente el perímetro del círculo de la base, el cual junto con la altura permitirá calcular la superficie rectangular del área del cilindro.

```
from math import pi

radio  = float(raw_input("Dime radio del círculo de la base (metros): "))
altura = float(raw_input("Dime la altura (metros): "))

area_base = pi * radio ** 2
volumen   = area_base * altura
peri_base = 2 * pi * radio
area      = 2 * area_base + peri_base * altura

print "Volumen del cilindro =", volumen, "metros cuadrados"
print "Área del cilindro =", area, "metros cuadrados"
```

Ejercicio 13. Modifica el programa que calcula el área de un círculo para que, además del área del círculo:

- Calcule y muestre por pantalla el perímetro de la circunferencia correspondiente $(2\pi r)$.
- Pida al usuario el valor del radio de un círculo contenido dentro del primero.
- Calcule y muestre por pantalla el área del anillo resultante, que será igual al área del círculo exterior menos el área del círculo interior.

Ejercicio 14. Escribe un programa que, a partir la lectura del valor de l, calcule y escriba por pantalla el valor del área de una figura compleja de la siguiente forma.



12.2. El módulo `string`

El módulo `string` contiene, entre otras, las siguientes funciones y constantes, que son de utilidad para manipular cadenas:

Funciones	
`lower(s)`	Convierte todas las letras mayúsculas de `s` a letras minúsculas
`upper(s)`	Convierte todas las letras minúsculas de `s` a letras mayúsculas
`lstrip(s)`	Elimina todos los espacios en blanco existentes al inicio (izquierda) de `s`
`rstrip(s)`	Elimina todos los espacios en blanco existentes al final (derecha) de `s`
`strip(s)`	Elimina todos los espacios en blanco existentes al inicio y final de `s`
`capwords(s)`	Convierte a mayúscula la primera letra de cada palabra de `s`, reemplaza espacios en blanco repetidos entre palabras por uno solo, y elimina todos los espacios en blanco existentes al inicio y final de `s`
Constantes	
`digits`	La cadena `'0123456789'`
`lowercase`	La cadena `'abcdefghijklmnopqrstuvwxyz'`
`uppercase`	La cadena `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
`letters`	La cadena `lowercase + uppercase`

Vamos a ver un ejemplo donde se utilizan estas funciones.

Ejemplo: El siguiente programa lee tres cadenas con el nombre y los dos apellidos de una persona. Estos pueden introducirse de cualquier forma (con espacios al principio, con más letras mayúsculas de las necesarias, etc.). Así, el programa muestra a continuación el nombre y los apellidos introducidos pero convertidos a un mismo formato: todas las letras minúsculas y eliminando espacios iniciales.

```
from string import lower, lstrip
nombre = raw_input("Dime tu nombre: ")
apell1 = raw_input("Dime tu primer apellido: ")
apell2 = raw_input("Dime tu segundo apellido: ")
```

```
print "Nombre" + " "*10 + ":", lower(lstrip(nombre))
print "Primer apellido :", lower(lstrip(apell1))
print "Segundo apellido:", lower(lstrip(apell2))
```

Aquí puedes observar un ejemplo de ejecución:

```
Dime tu nombre:                            TInTIn
Dime tu primer apellido:         FeRNaNDeZ
Dime tu segundo apellido:     fErnAndEz
Nombre          : tintin
Primer apellido : fernandez
Segundo apellido: fernandez
```

Fíjate en cómo se utilizan en este programa las funciones `lower` y `lstrip` para uniformizar las cadenas que pueden introducirse sin restricciones.

Ejercicio 15. Modifica el programa del ejemplo anterior para que ahora, además del nombre y los apellidos, se lean también la dirección, población y provincia de la persona. El nuevo programa debe mostrar todos los datos leídos uniformizándolos, pero no como antes. Ahora se deberán escribir las cadenas con letras minúsculas excepto las iniciales que se escribirán en mayúsculas, y se deben eliminar los espacios tanto al principio como al final de las cadenas.

13. Comentarios

En Python podemos incluir comentarios en los programas. El intérprete de Python se desentenderá de estos comentarios: no significan órdenes para él. Empleamos los comentarios para dar información a los que tengan que leer el programa, con el objetivo de aclarar diversos aspectos del mismo. Los programas podrían tener que leerlos personas distintas a quien los programó, o incluso los mismos programadores algún tiempo después de programarlos. La información que se incluya el programa, explicando su finalidad o aclarando algún fragmento que pueda resultar más confuso, será de gran ayuda.

Le indicaremos al intérprete de Python la aparición de un comentario mediante el símbolo # (almohadilla). Todo el texto que incluyamos entre una almohadilla y el final de la línea es un comentario. En realidad, la almohadilla supone una orden para el intérprete: no hacer caso de lo que haya escrito desde ese instante hasta el final de la línea.

Vamos ilustrar el empleo de comentarios, incluyendo algunos en el programa de cálculo del área de un círculo.

Ejemplo:

```
# Calculo e imprimo por pantalla el área de un círculo cuyo radio
# me proporciona el usuario por teclado

# Informo al usuario de mi propósito
print "Cálculo del área de un círculo a partir de su radio."

# Incorporo la constante pi del módulo matemático
from math import pi

# Pido y leo un valor del radio
r = int(raw_input('Dime el radio (en centímetros): '))

# Calculo el valor del área
area = pi * r ** 2

# Muestro por pantalla el área resultante
print 'Su área mide', area, 'centímetros cuadrados.'
```

En PythonG, los comentarios se escriben en color blanco, aunque *sólo en las pestañas de edición* (no en la de ejecución interactiva), con lo que podrás identificarlos fácilmente. Obviamente, aunque se pueden escribir, los comentarios no serán de mucha utilidad en la pestaña de ejecución interactiva.

14. Sesión de problemas 2

Ejercicio 16. El siguiente programa lee los valores de dos números flotantes introducidos por teclado y los almacena, respectivamente, en las variables `a` y `b`. A continuación, el programa calcula el logaritmo neperiano de `a` y después el logaritmo neperiano de `b`, almacenando el resultado en las variables `lognep_a` y `lognep_b`, respectivamente. Finalmente, el programa calcula la diferencia entre `lognep_a` y `lognep_b`, almacenando el resultado en la variable `res` y mostrando su valor por pantalla.

```
from math import log             # Importamos la función log del módulo matemático

a = float(raw_input("Dame el valor de a: "))           # Leemos los valores de a y
b = float(raw_input("Dame el valor de b: "))           # b introducidos por teclado

lognep_a = log(a)                            # Calculamos los logaritmos neperianos
lognep_b = log(b)                            # de a y b usando la función log

res = lognep_a - lognep_b               # Calculamos el logaritmo neperiano de a/b

print "El valor de log(", a, "/", b, ") es", res  # Mostramos resultado en pantalla
```

Recuerda la propiedad de los logaritmos $\log(a/b) = \log(a) - \log(b)$, la cual se utiliza en este programa.
A continuación, realiza las dos modificaciones que se indican.

1.- Modifica el programa para que, en lugar de calcular el logaritmo neperiano de `a/b`, calcule el *logaritmo en base 2* de `a/b`. Ten en cuenta que *no* existe una función que calcule el logaritmo en base 2 en el módulo `math`. Para resolver el problema, consideraremos que calcular el logaritmo en base 2 de un número x consiste precisamente en encontrar el exponente y que hace que al elevar la base 2 a dicho exponente y obtengamos como resultado x; es decir, $\log_2 x = y \Leftrightarrow 2^y = x$. Así, podremos expresar el exponente y en función de x de la siguiente manera:

$$2^y = x \quad \Leftrightarrow \quad \log 2^y = \log x \quad \Leftrightarrow \quad y \cdot \log 2 = \log x \quad \Leftrightarrow \quad y = \log x / \log 2$$

De esta forma, podemos calcular el logaritmo en base 2 de un número a partir de su logaritmo neperiano y del logaritmo neperiano de 2.

2.- Modifica el programa resultante para que calcule el *logaritmo en base* N, siendo ahora la base N un tercer valor que habrá que leer del teclado. De esta forma, generalizamos el programa y lo hacemos extensivo al cálculo de logaritmos en cualquier base.

Ejercicio 17. El siguiente programa:

```
veces = 3
frase = 'aprende a programar en Python\n'
print frase * veces
```

produce la siguiente salida por la ventana de ejecución:

```
aprende a programar en Python
aprende a programar en Python
aprende a programar en Python
```

"`\n`" es un único carácter (secuencia de escape) que no produce un símbolo visible en la pantalla, sino que provoca un cambio de línea de escritura de los caracteres que aparezcan tras él, como puedes observar en el ejemplo. Los cambios de línea producidos por "`\n`" son adicionales al que produce `print`.

Modifica este programa para que lea un nombre y también el número de veces que hay que repetir la frase, en lugar de darle un valor fijo como ahora. A continuación, añadirá el nombre a la izquierda de la cadena `'aprende a programar en Python\n'` convirtiendo todas las letras del nombre a mayúsculas, y la cadena resultante se mostrará por pantalla tantas veces como se haya indicado. Por ejemplo, si por teclado introducimos `'Mr. Bean'` y `2`, por pantalla deberá aparecer:

```
MR. BEAN aprende a programar en Python
MR. BEAN aprende a programar en Python
```

Ejercicio 18. La distancia euclídea entre dos puntos del plano, (x_1, y_1) y (x_2, y_2), se obtiene calculando:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Realiza un programa que lea las coordenadas de dos puntos cualesquiera del plano, calcule su distancia euclídea y muestre el resultado por pantalla.

Autoría:

Estos apuntes han sido editados por el profesor Juan Carlos Amengual Argudo, y su redacción supervisada por él mismo y el profesor Roberto Solana Montero, ambos profesores de la 509, a partir del material elaborado por el profesor Antonio Castellanos López, que fue profesor de esta asignatura hasta el curso 2002/2003.