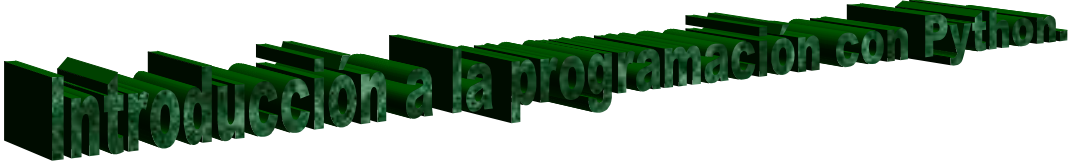


(Emilio S. Grisolia) – Cítera Software.

2:25 de la mañana... Noche lluviosa, 4 abril de 2013. ¿Qué mejor que escribir sobre programación? Si, dormir o consumir marihuana, nBoom o cualquier sustancia. Lo sé, pero tengo una cama chica y dormir con mi novia en una noche tan pesada, además no le gustan los alucinógenos, ¿qué mas puedo hacer?

Este escrito va dirigido a personas novatas en materia de informática, específicamente en la programación de software. Mi gran amigo Python será quien me acompañe a mi y a ustedes a lo largo del tutorial. Algún conocedor (o quizás no tanto) se preguntará "¿por qué Python? ¿por qué no Java, C++, C#, VB.Net, PHP, etc?". ¿Qué tiene Python frente a estos lenguajes tan potentes, robustos y conocidos? He aquí los motivos por los cuáles elijo a mi querido amigo para enseñar, y para llevar a cabo mis proyectos:

1) Sencillo de aprender.
2) Interactivo: Posee un intérprete que nos facilitará enormemente el testeo y aprendizaje.
3) Gratuito y multiplataforma: Es gratis y no nos ata (como .Net/VB) a Windows.
4) Productivo: Desarrollo de aplicaciones de manera rápida.
5) Fácil traslado del algoritmo al código en si mismo.
6) Extendible mediante módulos y código de otros lenguajes como C.
7) Gran cantidad de implementaciones: Desde PyPy hasta extensiones para el ámbito científico, numérico, multimedia (Audio, imágenes, videojuegos 2D/3D, DirectX, OpenGL), inteligencia artificial, bindings gráficos como Qt, wxWidget (y mas), implementaciones en .NET, Java (Jython), ASM, Fortran, C, y muchas cosas mas.
8) WEB: Enorme potencia a la hora del desarrollo web con uno de los frameworks mas potentes de la actualidad, Django.
9) Dispositivos móviles: Capacidad de programar para dispositivos móviles, especialmente para aquellos con Android.
10) Multiparadigma: Varias maneras de abordar los problemas a resolver.
11) Documentación extensa, oficial, y toneladas de código (cómo de módulos).
12) Prometedor y exitoso futuro.

Y muchas cosas mas (Si no entendiste mas del 90% de lo anterior, no te preocupes, que el Coco ya se encargará de vos).

¿Y si suena tan bueno, por qué no todos usan Python? Bueno, porque aún así, hay diversos campos en los que otros lenguajes predominan. Python tiene su hueco, al igual que C++, Java, PHP, Lisp, VB, y demás. Grandes sistemas (como en Google, YouTube, NASA, Facebook, NYTimes) utilizan Python, con un papel muy importante e incluso vital. Definitivamente, es un lenguaje muy atractivo, que reditúa mas que bien tanto en tiempos de producción como de depuración y mantenimiento. Fué creado en los 90, por Guido Van Rossum, y es actualmente, el lenguaje predilecto de Google.

También dejaré enlaces necesarios y de interés. Voy a hacer mucho énfasis en teoría (como orientación a objetos, algoritmos, etc), ya que son conceptos fundamentales a la hora de programar.

Sin mas rollos, pasamos directo a lo que nos importa (No, nos vamos a acostar con mi novia, ni a consumir drogas... Quizás lo último si). Cualquier cosa: grisolia.emilio94@gmail.com.

ÍNDICE:

Parte I

PARTE I

1.1-Arquitectura del computador y Sistema Operativo.

Es necesario conocer al ordenador, en mayor o menor medida. Al ser esto una introducción pequeña, no necesitaríamos grandes especificaciones, mas que lo básico e indispensable para saber con qué trabajamos.

A menudo nombramos la palabra "ordenador", "computador", "computadora", "cpu". Pero, ¿qué es en realidad? ¿cuál es su objetivo?

Tanto "ordenador", "computador" como "computadora" son sinónimos. Es un dispositivo, cuyo objetivo es procesar datos e información, realizando diversas operaciones (aritmético-lógicas) a velocidades insospechadas, para devolvernos un resultado. Este dispositivo posee su estructura propia. Estructura que (a grosso modo) define su modo de trabajo, organización e interrelación de los pequeños dispositivos que lo componen. Dicha estructura se conoce como "arquitectura del computador".

Existen diferentes arquitecturas de ordenador, la mas común es la denominada "arquitectura Von Neumann". Esta arquitectura es la que se utiliza en la mayoría de los ordenadores (Notebooks, Netbooks, Dispositivos móviles, PCs de escritorio, y casi cualquier dispositivo digital). La misma define las siguientes partes escenciales:

1. UCP (O CPU, por sus siglas en inglés: "Central Process Unity", "Unidad Central de Procesos")
   -UAL (o ALU, por sus siglas en inglés), la Unidad Aritmético-Lógica:
   Todas las operaciones aritméticas (suma, resta, por ejemplo) y lógicas (and, or, not) son llevadas a cabo en esta parte de la UCP. Esta le trasmite a la UC todos los resultados e información necesarias.
   -UC (o CU, por sus siglas en inglés), la Unidad de Control: Esta unidad se ocupa de controlar las acciones del hardware del ordenador, enviando señales y datos a donde sea necesario para realizar las diversas tareas.
2. Dispositivos de Entrada/Salida (E/S), en inglés Input/Output (I/O): Estos dispositivos se ocupan (valga la redundancia) se obtener o enviar datos e información a diversas partes del sistema, o al usuario. Por ejemplo, el mouse es un dispositivo de ENTRADA, mediante el cuál envíamos (entrada) datos (click, movimiento, presionar botones, girar la rueda) al sistema. Por otra parte, un módem es un dispositivo de ENTRADA/SALIDA, ya que es quien nos conecta a la red, y es por donde entra y sale información. Y por último, un parlante podria considerarse de SALIDA, ya que de SALE información (en forma de sonido). Todos estos dispositivos visibles se los conoce como PERIFÉRICOS.

Internamente, dentro de los circuitos del ordenador, también existen estos dispositivos, como "BUSES" por ejemplo. Para mas información, pueden dirigirse al siguiente sitio:

3. Dispositivos de memoria (RAM y ROM): La memoria es fundamental en la informática y computación. Gracias a ella, se puede almacenar millones de datos, que pueden ser utilizados (por el usuario, o cualquier dispositivo, o programa ejecutado) en otro momento. Básicamente podemos hablar de la memoria RAM ("Random Acess Memory", "Memoria de Acceso Aleatorio"), en la cuál se almacena todo lo necesario para que la UCP (UC/UAL) realicen su trabajo, accediento a los datos guardados cuando lo necesiten. En ausencia de energía, esta información se pierde.

Por otra parte, podemos hablar de la memoria ROM ("Read Only Memory", "Memoria de Solo Lectura"). Podemos englobar dentro de esto, a dispositivos como los discos rígidos; donde la información se guarda de manera "permanente", y aún en ausencia de energía, la información permanece, y no puede ser borrada (sin métodos "complejos") y sólamente puede ser leída o consultada.

Para mas información sobre la memoria, puede visitar el siguiente enlace, ya que es un tema muy extenso y complejo: http://es.wikipedia.org/wiki/Memoria_de_acceso_aleatorio

Sistema Operativo

Ahora bien, todos esos componentes internos entienden su propio lenguaje, el lenguaje máquina. En ese lenguaje, solo existen dos valores, los cuáles representamos como 0 y 1, y son conocidos como "bits" ("Binary digITS" o "dígitos binarios"). En realidad, representan niveles de voltaje (ya que internamente, toda información se trasmite mediante pulsos eléctricos), siendo 0 los pulsos aproximados a 0.5V, y 1 los pulsos aproximados a 5V. Es decir, cada cosa que el ordenador haga (Y hace millones por segundo), es una secuencia de impulsos, los cuáles representamos con 0 y 1. Imaginen si tenemos que hacer por nuestra cuenta que la computadora haga lo que hace, sería tedioso y súmamente complejo. Es por eso, que existen los Sistemas Operativos.

El Sistema Operativo ("SO" u "OS" por sus siglas en inglés), es quien conecta al usuario con la máquina. Se encarga de mostrarnos, en un lenguaje entendible, toda la información que necesitamos. Es decir, nosotros, al clickear el video de Lisa Ann que ~~me~~ te bajaste de algún sitio para adultos, no estamos mas que indicando al Sistema Operativo "qué queremos", y el se encargará de que el ordenador haga lo necesario para que ~~puedas disfrutar de un hermoso trasero~~ la acción deseada se lleve a cabo. No, el ordenador no te masturbará.

El SO, no es mas que un enorme programa.

1.2- *Programa*.

Un programa es un conjunto o secuencia de instrucciones. ¿Qué significa esto? Que el programa no es mas que una lista de cosas que el ordenador deberá hacer, y para lograr ese objetivo, seguirá esas instrucciones. Nativamente, esas instrucciones varían según el microprocesador (UCP), lo cuáles poseen su propio lenguaje, denominado Assembly, y que, aún acercándonos mas a los circuitos, no es mas que, como dije arriba, una secuencia de bits. Los programas se alojan en dispositivos de memoria como los discos rígidos (o en otro dispositivo, como sucede con la BIOS), y toda la información que necesitan y manejan, se alojan en la memoria RAM. Cuando nosotros vemos un programa con Windows, no es que se encuentre "dentro" de Windows (El cuál es un Sistema Operativo), si no que Windows nos muestra (de manera "humana") al programa alojado en el disco duro.

El proceso por el cuál se indican y guardan instrucciones (creación de un programa) se conoce como programación.

1.3-Lenguajes informáticos

Es imposible "dialogar" con la computadora tal como lo hacemos con una persona. Necesitamos hablar en su lengua, o por lo menos, en una lengua intermedia. Es debido a esta necesidad de conectarse con la máquina que surgen los "lenguajes informáticos" (pertenecientes a los lenguajes formales, http://es.wikipedia.org/wiki/Lenguaje). Dentro de estos lenguajes informáticos, se pueden encontrar otros tipos de lenguajes:

-Lenguajes de programación.

-Lenguajes de marcas.

-Lenguajes de consulta.

Entre otros. (Recomiendo la lectura de este artículo: http://www.muytranquilo.es/2012/01/informacion-sobre-lenguajes.html)

A grandes rasgos, estos lenguajes poseen sus reglas sintácticas y semánticas propias, y son utilizados tanto para indicar al ordenador que hacer (lenguajes de programación), cómo representar información (HTML, CSS), cómo consultar dicha información (SQL), y mas. Python es un lenguaje de programación (Tal como VB, C, Pascal, PHP, Java, Ruby, Fortran, y una interminable lista de etcéteras).

1.4-Tipos de lenguaje y paradigmas

Los lenguajes de programación pueden clasificarse según diversos criterios.

Según cercanía a la máquina o abstracción: En base a qué tan cercano es a la máquina o a nosotros, se clasifican en:

Lenguajes de bajo nivel: Lenguajes que se pueden comunicar (o necesitan hacerlo) estrechamente con la máquina, son mas complejos y necesitan mas instrucciones para realizar una acción.

Lenguajes de alto nivel: Son lenguajes "de nueva generación". Dejan de lado las especifiaciones propias del ordenador (a nivel hardware) y son mas "naturales" o entendibles para nosotros, a fin de facilitar la programación al intentar enfocar al programador en la resolución del problema en si.

Según forma de ejecución: Ya sabemos que el ordenador solo entiende su lenguaje. Ahora bien, ¿cómo es que si escribimos en otro lenguaje, aún así el ordenador puede hacer lo que deseamos? Esto se debe a que existen métodos de traducción. Es decir, las instrucciones que escribimos se traducen al lenguaje de la máquina. Podemos definir dos grupos bien marcados:

Lenguajes típicamente compilados: Los programas escritos en este tipo de lenguajes, deben ser traducidos a lenguaje máquina (lo que se conoce como "compilar") antes de ser ejecutados. Por ejemplo, C o C++ son típicamente compilados. Sus tiempos de producción son mas lentos, debido a que deben ser compilados en cada ejecución para probar el software.

Lenguajes típicamente interpretados: Los programas escritos en este tipo de lenguajes, se traducen al tiempo que se ejecutan. Esto conlleva una pérdida de velocidad (generalmente imperceptible) en tiempo de ejecución, ya que cada instrucción, antes de ser ejecutada, se traduce. Esto se lleva a cabo mediante una aplicación llamada intérprete. Por ejemplo, lenguajes como Python o Java poseen un intérprete o máquina

virtual. Aunque existen métodos que aligeran enormemente la velocidad de ejecución. Y su tiempo de producción es mas rápido, ya que no necesitamos compilar todo el código para ejecutarlo, y podremos probar pequeños fragmentos o instrucciones simples.

Según portabilidad: Dependiendo bajo que sistemas operativos y arquitecturas se puedan ejecutar los programas, los lenguajes con los cuáles se realizan pueden ser o no multiplataforma. Por ejemplo, Visual Basic ata al programador a que los programas solo sean escritos para Windows, mientras que con Python, solo basta con que el ordenador en cuestión tenga instalado el intérprete (Aunque existen otros métodos) para que el programa se ejecute, independientemente de si el programa corre en Linux, MacOs o Windows.

Según tipo de aplicaciones escritas: Las aplicaciones pueden ser de escritorio o web. En los lenguajes WEB encontramos los lenguajes client-side y server-side. Lenguajes como Python no son lenguajes WEB, pero si poseen frameworks para el desarrollo web del lado del servidor (server-side).

Según sus paradigmas: Encontramos acá un concepto importantísimo: Paradigma. ¿Qué es un paradigma en informática? Se podría decir que es una *filosofía* adoptada a la hora de programar. Una forma de pensar, visualizar, abstraer, proceder y resolver diferentes tipos de problemas. Existen diversos paradigmas, entre los mas "conocidos" se encuentran:

Paradigma estructurado: Este paradigma sostiene que los programas deban ser organizados en subrutinas, a fin de mejorar el entendimiento y mantenimiento de los mismos.

Paradigma procedimental: Los lenguajes procedimentales (y los de alto nivel denominados funcionales) establecen que el código debe ser englobado en lo que se conoce como procedimiento o función, que son fragmentos de código que se ejecutan cuando es necesario. De esta manera de puede reutilizar el código.

Paradigma de Orientación a Objetos: Este paradigma utiliza los denominados objetos como algo central. Un objeto es una abstracción de una entidad, la cuál posee atributos, comportamiento y un identificador, y se relaciona con otros objetos.

Podríamos hablar de muchos otros paradigmas. Muchos de los lenguajes actuales soportan varios paradigmas de programación, lo cuál se conoce como multiparadigma. Python es un lenguaje multiparadigma, al permitir programación orientada a objetos (POO), funcional, estructurada y modular.

Pero, no importa si todavía no entendés todo lo anterior. Programar es algo que se aprende con tiempo, y solo fué una introducción que mas adelante quedará mas que aclarado todo.

2- Descargando e instalando Python.

Llegamos entonces, luego de tanto rollo, a comenzar a comprender lo antes dicho, y conocer lo que no dije :-). Lo primero que debemos hacer, es descargar Python. Actualmente existen dos versiones oficiales soportadas, las cuales son la 2.7 y la 3x. Hay algunas variantes fundamentales. Soy devoto a la versión 2.7, pero, como programadores, debemos estar alerta a los cambios rápidos que se van dando. Es por eso, que recomiendo la descarga de ambas versiones. ¿Por qué? ¿Por qué mierda me hacés bajar las dos si acabás de decir que tenemos que seguirle el paso a los nuevos cambios, la conch* de tu hermana? Bueno, hay toneladas

de código Python ahí afuera, así como tutoriales (buenos, pero no oficiales), escritos para Python 2.7... Lo ideal, es que se apoyen en otras lecturas. Una vez alcanzado cierto nivel podrán -por su cuenta- informarse acerca de las diferencias, y trasladar su código a la última versión.

Web oficial de Python: http://www.python.org/
Encontraremos aquí varias secciones de interés, aunque debemos conocer algo de inglés para entenderlas.

DOCUMENTATION » http://www.python.org/doc/

DOWNLOAD » http://www.python.org/download/

Documentación no oficial en Español: http://wiki.python.org/moin/SpanishLanguage
Guía para principantes (inglés): http://wiki.python.org/moin/BeginnersGuide
Sitio Python Argentina: http://python.org.ar/pyar/
Sitio Python Hispanoparlante: http://python-hispano.org/

Descargamos la versión deseada. En nuestro caso, 2.7.3 y 3.3 (actual).
En caso de tener Windows y un procesador de 32bits
Python 3.3.0 Windows x86 MSI Installer
Python 2.7.3 Windows Installer
En caso de tener Windows y un procesador 64bits (AMD64 / Intel 64 / X86-64)
Python 3.3.0 Windows X86-64 MSI Installer
Python 2.7.3 Windows X86-64 Installer
Viene instalado por defecto en MacOs y Linux. Para saber tus especificaciones:

Para Windows XP: Click derecho sobre MI PC>Propiedades.
En Windows Vista/7: Click derecho sobre Equipo>Propiedades. Lamento no tener fotos, pero no tengo como

sacarlas.

Una vez descargado Python, comenzamos la instalación. En caso de haber algún problema relacionado al MSI Installer, acá hay información. Si no encuentran solución, pueden enviarme un correo a grisolia.emilio94@gmail.com , Twitter: Grisem94S, Facebook: Emilio Grisolia. De todas maneras, solo una vez en la vida tuve un error de este tipo... Pero prevenir nunca está de mas.

La instalación es sencilla, recomiendo instalarla en C:\ o unidad por defecto. Se instalará en la carpeta Python27 y Python33 (Python 2.7 y Python 3.3 respectivamente).

3- Algoritmos.

Antes de comenzar (Ajá, bajaron el Python pero todavía no lo van a tocar, ¿no soy muy malo?), tenemos que detenernos en algo, que lo voy a hacer lo mas corto y concreto posible, pero que lo profundizaré mas adelante. Esta palabra que suena a un algodón musical (¿?), se refiere a una lista de pasos que describen como solucionar un problema. Le llamamos problema a cualquier situación que se nos plantee resolver, hasta la mas mínima, a diferencia de lo "cotidiano" en lo cuál llamamos problema a, por ejemplo, cuando hay entrega de boletines, digamos, una situación que nos puede perjudicar.

Un algoritmo podría ser una lista de pasos para instalar un programa en Windows:

```
1- Abrir el instalador del programa.
2- Click en siguiente.
3- Click en siguiente.
4- Click en siguiente.
5- Click en siguiente.
[...]
24- Click en siguiente
25- Destildar "ver léame".
26- Click en finalizar.
27- Borrar todos los complementos extras indeseados que se nos instalaron en el Chrome.
```

Otro algoritmo podría ser una receta, de como cocinar fideos:

```
1- Poner agua en una olla.
2- Calentar el agua hasta que hierva.
3- Poner los fideos.
4- Esperar 8 minutos.
5- Colar los fideos.
6- Agregar aceite.
7- Servir los fideos.
```

Y así también podríamos especificar aún mas:

```
1- Agarrar la olla.
2- Abrir la canilla.
3- Poner la olla debajo.
4- Esperar a que cargue la olla.
5- Apoyar la olla en la ornalla.
6- Encender la ornalla.
7- Tapar la olla.
8- Salar el agua.
9- Esperar que hierva el agua.
10- Ir a la baulera.
11- Agarrar el paquete de fideos.
12- Abrir el paquete de fideos.
13- Poner los fideos en la olla.
14- Esperar 8 minutos.
15- Poner el colador en la pileta.
16- Volcar los fideos en el colador.
[...]
```

Y así podríamos seguir, especificando aún mas o menos los pasos a seguir. Podríamos hacer una analogía

entre los lenguajes de alto nivel y bajo nivel con éstos dos últimos ejemplos. En el primer algoritmo observamos que nos enfocamos directamente en el problema en si, dejando de lado los movimientos y detalles que pueden variar. Es decir, en el primer algoritmo vemos especificado “ir a la baulera” para agarrar el paquete de fideos, pero, es probable que uno no los tenga guardados ahí. Lo mismo sucede con los lenguajes de alto nivel, en donde dejamos de lado las especificaciones de hardware o detalles del ordenador, y nos enfocamos en el problema en si mismo.

Sencillamente, los algoritmos deben contener un número FINITO (no infinitos) de pasos correctamente ordenados, lo mejor descritos posibles, de tal manera que cualquiera pueda resolver el problema o situación que se plantea (Aunque algunos algoritmos pueden no devolver la solución al problema). La importancia de los algoritmos radica en qué, una vez escrito, podemos trasladarlo a otro lenguaje de programación, o realizar dichos pasos sin problema alguno. Se debe desintegrar cada acción hasta el punto mas mínimo necesario. Para mas información (recomendado): http://informaticafrida.blogspot.com.ar/2009/03/algoritmo.html

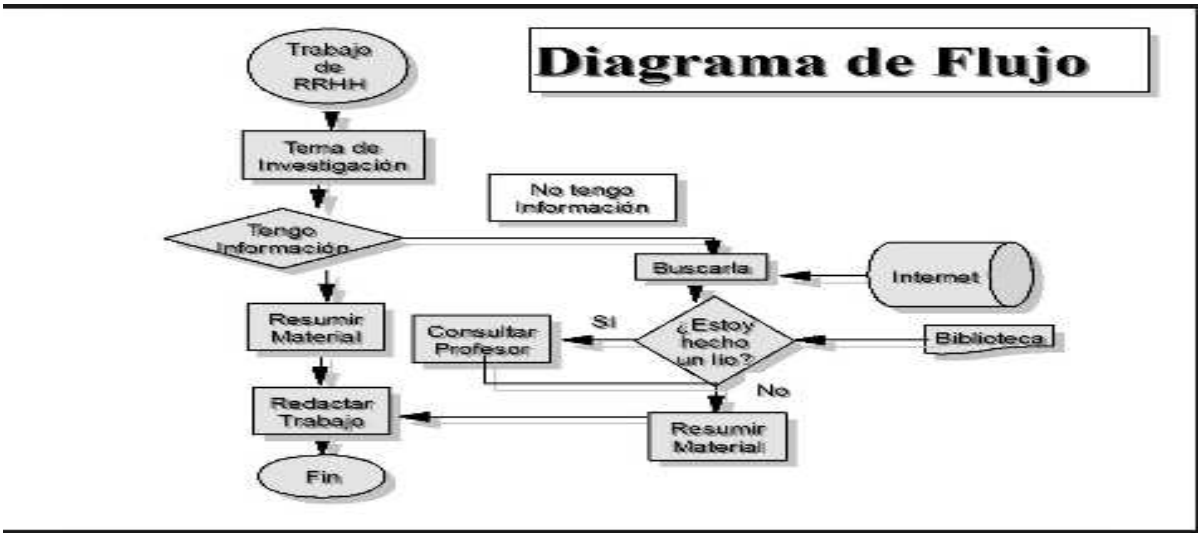
Otra cosa interesante, son los diagramas de flujo. Si alguno cursa/cursó en una escuela técnica, es casi seguro que los hayan visto. Los diagramas de flujo son una representación gráfica de un algoritmo. La misma gráfica debe especificar los pasos, de manera clara. Cada paso debe ser representado con el gráfico acorde. Básicamente encontramos (aunque hay mas) cuatro tipos de símbolos:

-Elipse: Indica inicio/fin del diagrama.

-Rectángulo: Cualquier acción llevada a cabo.

-Rombo: Evalúa una condición o figura una pregunta.

-Círculo: Continúa la lectura/ejecución en otro punto indicado.

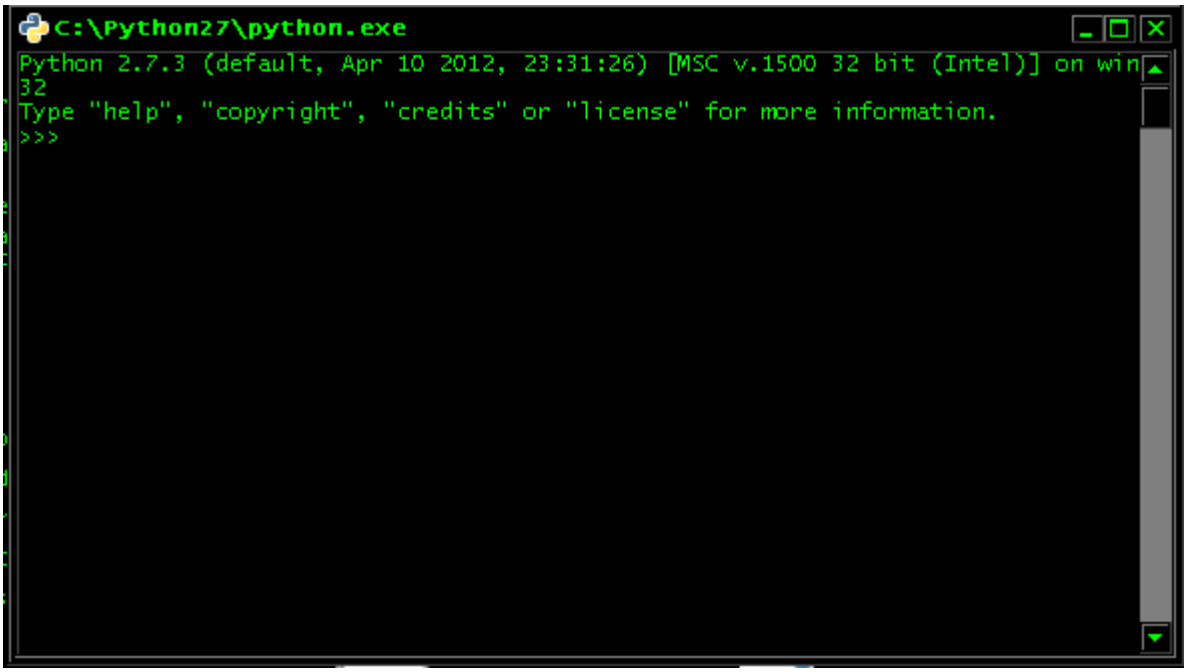


¿Qué tendrá que ver esto con programar? A medida que vayas avanzando, vas a empezar a comprender el por qué. Programar no es mas que solucionar diversos tipos de problemas. Plantear el algoritmo, de la manera que sea, facilitará enormemente la solución y codificación, ya que una vez planteado como solucionar el problema, solo basta escribir dicha solución en el lenguaje que deseamos.

3- Operaciones aritméticas y lógicas.

Vamos a interactuar ahora con el intérprete de Python. Como ya dijimos, esta aplicación se encargará de traducir lo que escribamos (en Python), para que el ordenador realice lo que deseamos. El intérprete nos será de gran ayuda, ya que en el podremos probar el código antes de escribirlo, y así evitar errores. Recomiendo ampliamente experimentar con el.

Nos dirigimos entonces a donde tengamos instalado Python, y abrimos el fichero “python”, cuyo ícono es una pantalla negra con un chinchulín azul y amarillo. Nos saldrá la siguiente pantalla (puede que con otros colores, pero es la misma):



Nos dice: “Type “help”, “copyright”, “credits” or “license” for more information.” Esto quiere decir, que si tipeamos cualquiera de esas cosas, nos desplegará información acerca del copyright, créditos, licencia o ayuda. Los tres símbolos direccionales “>>>” son el prompt. Nos indica que el intérprete está esperando que ingresemos algo. Comenzemos por operaciones aritméticas. Las operaciones aritméticas básicas son:

-Adición (suma).

-Sustracción (resta).

-División .

-Multiplicación.

-Módulo.

Realizemos algunas operaciones. Las operaciones se realizan en notación de infijo, es decir, tal como lo haríamos normalmente. Su sintáxis es:

```
>>> [Signo]<Operando> <Operador> [Signo]<Operando>
```

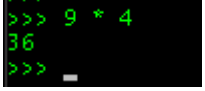
Es decir, una operación básica consta de un operando(número o variable), un operador (operación a realizar) y otro operando. Opcionalmente (Lo que está entre corchetes es opcional), se puede agregar el signo, si no se especifica, por defecto es positivo. No es obligatorio dejar un espacio entre operador-operando, pero si muy recomendable. Veamos algunas operaciones:

```
>>> 12 + 9
21
>>>
```

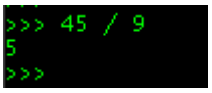
Vemos ahí el operando 12, el operador “+” y el operando 9. Al dar enter, nos dá el resultado.

```
>>> 7 - 10
-3
>>>
```

Una simple sustracción, la cuál, en este caso, nos dá un número negativo, -3. El operador es “-“ (Guión medio o signo menos).



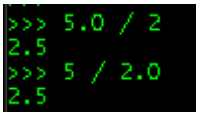
El operador de multiplicación es el "*" (Asterisco).



El operador de la división es "/" (Barra, shift 7).

Veamos ahora la siguiente división:

No, Python no está equivocado. ¿Por qué dió 2 y no 2,5? Esto se debe a que los números ingresados son números ENTEROS (integer, en inglés), y es por esto que se asume una división entera. Un número entero NO posee decimales. Para realizar una operación con números decimales (conocidos como "float" o "flotantes"), debemos especificar al menos un operando decimal, aunque su cifra decimal sea 0. Como delimitador decimal, utilizamos el PUNTO (.) a diferencia de la coma (,) utilizada por nosotros normalmente:



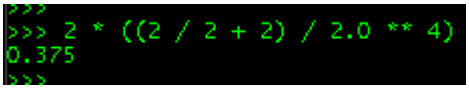
Vemos que en cualquiera de los casos, nos devuelve 2.5, sin importar cual de los dos está expresado como flotante.

Y por último, tenemos el operador "**" (doble asterisco), que es la potenciación. Implementá un ejemplo simple de cómo creés que funciona. Elevá el número 6 a la séptima potencia (7). ¿Simple, no?

Sería muy básico que solo podamos operar con dos operandos. Tenemos la capacidad de escribir cualquier cálculo. Su resultado estará sujeto al proceso de resolución que todos debemos conocer ya, el cuál consiste en resolver por prioridad de la siguiente manera:

1. Raíces y potencias.
2. División y multiplicación.
3. Suma y resta.

Si hay paréntesis se procede a resolver paréntesis internos (si los hay), siguiendo los pasos antes nombrados. A diferencia de como resolvemos normalmente, la ausencia de operador entre dos símbolos (por ejemplo, número-paréntesis NO significa multiplicación). Siempre se debe especificar el operador deseado. Veamos un ejemplo:



A) Se resuelve el primer paréntesis, que engloba: *(2 / 2 + 2) / 2.0 ** 4*

1. Primer paréntesis: *2 / 2 + 2 = 3*

1.1- Se resuelve la prioridad de la división es mas alta que la suma, por lo tanto se resuelve 2 / 2 y luego se le adiciona 2. Esto dá como resultado 3.

2- Resuelto el paréntesis quedaría: *3 / 2.0 ** 4*

3- La operación de mas prioridad es la potenciación, por lo cuál se potencia 2.0 a la cuarta, lo cuál da 16.0

Resuelto esto, el paréntesis quedaría*: 3 / 16.0*

4- Se divide *3 / 16.0* por lo cuál el resultado será flotante, y dá 0.1875

Por lo que quedaría *2 * (0.1875),* lo cuál equivale a *2 * 0.1875*

5- Resuelto eso, solo queda operar lo anterior, que es igual a *0.375*

Recordemos que, un signo menos antes de un paréntesis, invierte los signos dentro de ella:

2 -(4 +7) En donde, 4 y 7 son positivos, quedarían *(-4 -7)*, lo cuál equivale a -11, siendo finalmente *2 - 11*.

Todo número elevado a la 0, da 1.

Todo número elevado a la 1, da el mismo número.

¿Cómo resolvería esto Python?

```
2 ** 1 -(2-(7 / 2 * 2) / (5 ** 0)) =
-7.1 / 7.1 * (3 + 3 / 1.5) =
-2 + 2 / 2 -(1-(4 * 2)) =
```

Aún no terminamos acá. ¿Y si queremos conseguir la raíz cuadrada? ¿Y si quiero operar con fracciones? ¿Redondear un número? Este tipo de operaciones se lleva a cabo mediante funciones. Si bien son un concepto que se va a explicar mas adelante, es recurrente el uso de ellas. Una función, básicamente, es un conjunto de acciones que operan utilizando o sobre determinado/s valor/es. Por ejemplo, en la escuela vemos la "función lineal":

```
F(x) = ax + b
```

En programación, las funciones devuelven un valor. ¿Qué significa esto? Que reciben un dato denominado argumento. Y la función devuelve información que necesitamos. Las raíces cuadradas, por ejemplo, las conseguimos utilizando la función denominada sqrt().Esta función no se encuentra disponible de entrada. Para poder acceder a ella vamos a importar lo que se conoce como módulo. Un módulo es un fichero externo al programa desde el cuál lo importamos. Este módulo puede tener dentro desde variables, información útil, hasta funciones, etc. La función sqrt() se encuentra dentro de un módulo llamado math.py. Es decir, que antes de acceder a esta función, debemos importar dicho módulo. Para esto tipeamos lo siguiente:

```
>>>
>>> import math
>>>
```

Vemos entonces nuestra primer instrucción. La instrucción "import" importa un fichero externo especificado. Tanto las instrucciones como cualquier otra cosa, son CASE SENSITIVE. Esto quiere decir que no es lo mismo "Instrucción" que "instrucción". Veamos entonces como utilizar la función. Consigamos la raíz cuadrada de 4:

```
>>> import math
>>>
>>> math.sqrt(4)
2.0
>>>
```

Para acceder a una función de un módulo debemos expresar el nombre del módulo (en este caso math), seguido de un punto (.), y luego el nombre de la función, con su/s argumento/s en caso de necesitarlos:

```
modulo.funcion([argumentos])
```

La función sqrt() nos retorna un número tipo float. Al importar el módulo math, estamos importando TODO

su contenido, pero solo estamos utilizando UNA función. Como programadores, además de mantener la prolijidad en nuestros programas, debemos economizar los recursos del ordenador lo mas que podamos, por mas mínimo que sea el ahorro. Sería un poco desprolijo expresar un cálculo así:

```
>>> -2 + (math.sqrt(2 + (math.sqrt(2))))
-0.15224093497742652
>>>
```

Es por eso, que podemos elegir puntualmente qué funciones o que datos importar de un módulo con la sentencia o instrucción from. Su sintáxis es:

```
from modulo import funcion
```

¿Cómo importamos entonces sólamente la función sqrt() del módulo math?

```
>>>
>>> from math import sqrt
>>>
```

Entonces, para utilizar ahora dicha función, no necesitaríamos especificar de donde sale:

```
>>> sqrt(7)
2.6457513110645907
>>>
```

Fijémosnos que es prácticamente como leer en inglés. Si tradujéramos lo que dice ahí sería algo como "desde math importar sqrt". A esto me refería al decir que Python puede ser casi como leer en inglés.

Toda función que retorne valor, pasaría a ser un "equivalente". Podemos darle como argumento a una función, otra función:

```
>>> from math import sqrt
>>> sqrt(sqrt(16))
2.0
```

¿Cómo llegó a 2.0? Primero resolvió la operación dentro del primer paréntesis, la cuál consistía en obtener la raíz cuadrada de 16 (4). Posteriormente, la operación final vuelve a obtener una raíz cuadrada, en este caso, del resultado anterior, siendo 2.0 la raíz de 4.0

Se puede, también, dar como argumentos cualquier tipo de cálculo. Primero se resuelven las funciones argumento, y luego se procede a resolver como un cálculo común:

```
>>> sqrt(2 + sqrt(16) * 2 ** 2)
4.242640687119285
>>>
```

```
1- Quedan operaciones por resolver dentro de la primer función sqrt()?
   Si lo hay:
            1.1-Resolver otras funciones.
            1.2-Resolver potencias y raices.
            1.3-Resolver divisiones y multiplicaciones.
            1.4-Resolver sumas y restas.
2- Si no quedan operaciones por resolver dentro de la primer función sqrt()?:
            2.1-Resolver la primer función sqrt()
```

Concretamente, todo argumento posible de resolver, se resuelve antes de que la función que lo recibe opere sobre el. La resolución aritmética de ese argumento, se resuelve respetando las prioridades conocidas: (Paréntesis>Potencia/raíz>División/multiplicación>Suma/resta) y agregando las funciones quedando con prioridad mas alta que las potencias y raices.

Dejo una lista pequeña de funciones. Estas funciones devuelven valores numéricos, es decir, que pueden recibir como argumentos cualquier operación o función que concluya en otro valor numérico.

abs(): Recibe como argumento un número y devuelve su valor absoluto.

```
>>> abs(-3)
3
```

round(): Recibe como argumento un número decimal y lo redondea hacia abajo (cuando la cifra decimal es

menor a 5) o hacia arriba (cuando la cifra decimal es mayor o igual a 5) hasta llegar a un número con decimal 0 .

```
>>> round(5.3)
5.0
>>> round(5.5)
6.0
>>> round(5.7)
6.0
>>>
```

int(): Convierte un número flotante a entero, eliminando SIN REDONDEAR su parte decimal.

```
>>> int(5.7)
5
>>>
```

float(): Convierte el valor del argumento que recibe a un valor flotante.

```
>>> float(6)
6.0
>>>
```

Podemos usar como argumento de una función cualquiera de las funciones antes nombradas:

```
>>> sqrt(round(2.0 / 0.7))
1.7320508075688772
>>>
```

Primero se resuelven la operación dada como argumento para la función round (2.0 dividido en 0.7). El resultado de esa operación devuelve un número decimal (2.85 aproximadamente), y luego la función round() lo redondea. Como su parte decimal es mayor que 5, se redondea hacia arriba, hasta el número 3.0. Entonces sqrt(round(2.0 / 0.7)) sería lo mismo que poner sqrt(3.0), lo cuál da el resultado que se ve en la imagen.

Experimenten para esclarecer sus dudas. Hay un proberbio que podría decir que es una regla de oro en programación: "Me lo dijeron, lo olvidé. Lo ví, lo entendí. Lo hice, lo aprendí."

Les planteo un ejercicio: Redondear el resultado obtenido de calcular la raíz cuadrada de la suma de dos valores absolutos de -5 y 3.3 convertidos en enteros. ¿Es correcto el siguiente planteo? ¿Si no lo es, cómo debería plantearse?

```
int(sqrt(round(abs(-5) + abs(3.3))))
```

Escríbanlo en el intérprete. El resultado correcto al que deben llegar es a 3.0 (no 3!).

Pero las operaciones aritméticas no son las únicas. Existen operaciones denominadas operaciones lógicas (que pertenecen a la ciencia fáctira denominada lógica) u operaciones booleanas. En Python tenemos tres operadores lógicos: Y (and), O (or) y Negación (not). Estos operadores comparan valores y devuelven true (verdadero) o false (falso). La siguiente tabla explica la tabla de operadores. Sacada de http://programacionpython.wordpress.com/2008/07/09/dia-12-operadores-booleanos-o-logicos-en-python/

El operador and da como resultado el valor True si y sólo si son ciertos sus dos operandos.

and		
operandos		resultado
izquierdo	derecho	
True	*True*	*True*
True	*False*	*False*
False	*True*	*False*
False	*False*	*False*

El operador or proporciona True si cualquiera de sus operandos es True, y False sólo cuando ambos operandos son False.

or		
operandos		resultado
izquierdo	derecho	
True	*True*	*True*
True	*False*	*True*
False	*True*	*True*
False	*False*	*False*

El operador not es unario, y proporciona el valor Trae si su operando es False y viceversa.

not	
operando	resultado
True	*False*
False	*True*

```
>>> True and True
True
>>> False and False
False
>>> True and False
False
>>> False and True
False
>>> True or False
True
>>> False or True
True
>>> False or False
False
>>> True or True
True
>>> not True
False
>>> not False
True
>>>
```

¿De qué sirve esto? Para lo que vos, como programador, necesites. Generalmente son utilizados en estructuras condicionales y de repetición, lo cuál se verá en poco tiempo.
Una vez visto el tema, pasamos a un nuevo concepto: variable.

5- Variables

Este es un concepto clave y fundamental. Recordemos la memoria RAM. Es donde los programas guardan la información que necesitan manejar. Esta memoria RAM posee direcciones, como una calle. Nosotros podemos guardar en ella datos. En Python (como en todo lenguaje de alto nivel) no se puede elegir en qué dirección guardar esos datos, ya que Python lo hace por nosotros. Pero aún así, podemos darle un identificador. Un identificador no es mas que nombre. Es lo que se conoce como variable. Una variable es una dirección de memoria, en la cuál se guarda un valor, y a la cuál se le asigna y a la cual se accede mediante el identificador que hayamos establecido. La sintáxis para definir una variable es la siguiente:

```
Identificador = Valor
```

Por ejemplo, una variable puede contener un número entero:

```
>>> MiVariable = 5
```

O flotante:

```
>>> MiFlotante = 5.6
```

Incluso su valor puede ser una operación aritmética, lógica, otra variable o retorno de una función:

```
>>> MiVar = 2 + round(MiFlotante)
```

En este caso, “MiVar” equivale a la suma entre 2 y el redondeo de la variable MiFlotante. Resolviendo eso sabemos que 5.6 redondeado pasa a ser 6.0. Por lo tanto MiVar = 2 + 6.0 = 8.0. Para verificar este resultado, podemos escribir en el intérprete el identificador de la variable de la cuál queremos ver su valor:

```
>>> MiVar
8.0
```

Efectivamente, vemos que MiVar contiene el valor 8.0. Aunque no solo podemos trabajar con tipos de datos numéricos o booleanos, también podemos trabajar con cadenas u otras estructuras. ¿Qué quiero decir con

“tipos de datos”?

Algunos de los tipos de datos básicos son:

```
int (entero)
float (flotante)
str (cadena)
bool (booleano)
```

5 es tipo int (entero)

6.7 es tipo float (flotante)

‘Hola’ es tipo str (cadena de texto)

True es bool (booleano)

La función type() nos devuelve el tipo de dato del argumento dado:

```
>>> type(MiVariable)
<type 'int'>
>>> type(MiFlotante)
<type 'float'>
```

Y como siempre, puede recibir una operación como argumento:

```
>>> type(int(round(MiVar / MiFlotante)))
<type 'int'>
>>>
```

¿Y el tipo cadena? ¿Qué es? Las cadenas de texto se conforman por caracteres alfanuméricos (letras, números, signos, etc). Una variable puede contener una cadena de texto. Las cadenas de texto se encierran entre comillas simples o dobles (‘’ o “”):

```
>>> MiCadena = 'Esto es una cadena!'
>>> MiCadena
'Esto es una cadena!'
>>>
```

¿Podemos almacenar una cadena numérica, solo de números o mezclar con letras? Si:

```
>>> CadenaNumerica = '6662323'
>>> CadenaNumalfa = '222texto'
>>>
```

Por otra parte, la cadena ‘1’ no es lo mismo que 1. Es por eso que si intentamos algo así, nos da error:

```
>>> CadenaNumerica + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Este tipo de error (TypeError) se refiere a un problema con tipos de datos. Nos dice que no se puede concatenar un objeto cadena (str) con un entero (int). Es decir que no se pueden operar entre si valores que NO sean numéricos. Pero, podemos convertir una cadena a otro tipo de dato, siempre y cuando la misma no contenga caracteres alfabéticos. Intentemos convertir CadenaNumerica, que contiene una cadena “6662323” a entero y sumarle 1:

```
>>> int(CadenaNumerica)
6662323
>>> CadenaNumerica + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

¿Cómo es que nos da error? Sencillamente porque CadenaNumerica sigue siendo una cadena de texto. La función int() no cambió el tipo de dato de la variable; sólamente nos mostró su valor pasado a tipo int. Es por eso que al intentar sumar, nos dió otra vez el mismo error. ¿Cómo podemos solucionar esto? ¿Cómo podemos mostrar el resultado de la adición de 1 al valor entero de CadenaNumerica? Podemos hacerlo de dos maneras:

```
>>> int(CadenaNumerica) + 1
6662324
>>>
```

Veamos que, primero se ejecuta la función, la cuál devuelve 6662323, y se le suma 1, quedando 6662324.

De esta manera, CadenaNumerica seguiría siendo tipo string (str). Pero, ¿y si queremos cambiarle el tipo de dato para esa variable? ¿O si queremos almacenar el valor entero en otra variable, para que CadenaNumerica siga siendo una cadena?

```
>>> CadenaNumerica = int(CadenaNumerica)
```

De esta manera, CadenaNumerica pasaría a valer 6662323 en vez de '6662323'. Y ahora si, podemos modificarle su valor:

```
>>> CadenaNumerica = int(CadenaNumerica)
>>> CadenaNumerica = CadenaNumerica + 1
>>> CadenaNumerica
6662324
```

Ahora bien, ¿y si queremos que vuelva a ser una cadena? Para esto, solo basta con utilizar la función str() que devuelve como cadena el argumento que reciba. Les planteo un ejercicio sencillo:

Convertir a tipo str la variable CadenaNumerica, y almacenar su valor entero convertido a flotante en otra variable nueva.

5.1 – Operando con cadenas.

Ahora bien, las cadenas no pueden operarse como números. Definamos dos nuevas variables de tipo string, e intentemos sumarlas:

```
>>> MiStr = '4'
>>> MiStr2 = '2'
>>> MiStr + MiStr2
'42'
>>>
```

Vemos que Python no sumó 4 + 2, si no que nos devolvió la cadena de texto '42'. Esto es porque, el operador +, no puede sumar dos cadenas de texto. En este caso, el signo +, concatena. Es decir, que une ambos caracteres. Pero, no es el único operador, ya que también existe el operador *, que multiplica una cadena tantas veces como el segundo operando indique:

```
>>> MiStr * 4
'4444'
```

MiStr contenía el caraceter '4', y MiStr * 4 repite 4 veces el contenido de MiStr. Podemos, a su vez, combinar operaciones:

```
>>> MiNum = 23.6
>>> MiStr3 = MiStr * 4 + str(MiNum) * 2
>>> MiStr3
'444423.623.6'
```

Las cadenas de texto son lo que se conocen como secuencia. Ya que son una secuencia de caracteres. Cada caracter tiene una posición, la cuál comienza su conteo desde 0, para el primer elemento de la secuencia. Por ejemplo, la cadena 'casa roja' tiene 9 (si, 9) caracteres:

c	a	s	a		r	o	j	a
0	1	2	3	4	5	6	7	8

El conteo comienza en 0 como ya dije, y los espacios son contados obviamente, ya que son caracteres. Podemos acceder a cualquiera de sus elementos mediante la indicación entre corchetes de la posición de dicho elemento:

```
>>> 'casa roja'[3]
'a'
>>> 'casa roja'[7]
'j'
>>> 'casa roja'[0]
'c'
>>>
```

Desde ya, lo mismo pasaría si lo hiciéramos con una variable que almacene una cadena:

```
>>> Cadena = 'hola como te va?'
>>> Cadena[-1]
'?'
>>> Cadena[2]
'l'
>>> Cadena[0]
'h'
>>>
```

Vemos que al indicar -1, nos devuelve '?'. Ese número entre corchetes se lo denomina índice y al ser negativo, "cuenta (y desde -1) de derecha a izquierda". Es decir que -2 sería 'a', -3 sería 'v' y así sucesivamente. Veamos un ejemplo:

```
>>> Cadena = 'asd123'
>>> Variable = str(int(Cadena[3]) + int(Cadena[4]) + int(Cadena[5]))
>>> Variable
'6'
>>> Variable + 'x'
'6x'
>>>
```

Se observa que Cadena contiene una cadena 'asd123', y que Variable termina conteniendo la cadena '6' ya que en primer lugar, convirtió a enteros los caracteres número 3, 4 y 5 de Cadena (los cuáles eran '123'), y luego de sumarlos, la función str() convierte el resultado de esa suma (6) a una cadena otra vez. Por último, a esa cadena se le concatena el caracter 'x'. Mas adelante, en la profundización de secuencias, veremos mas métodos y posibilidades respecto a cadenas de texto (y otro tipo de secuencias).

6- Creando un programa. Entrada/Salida de datos.

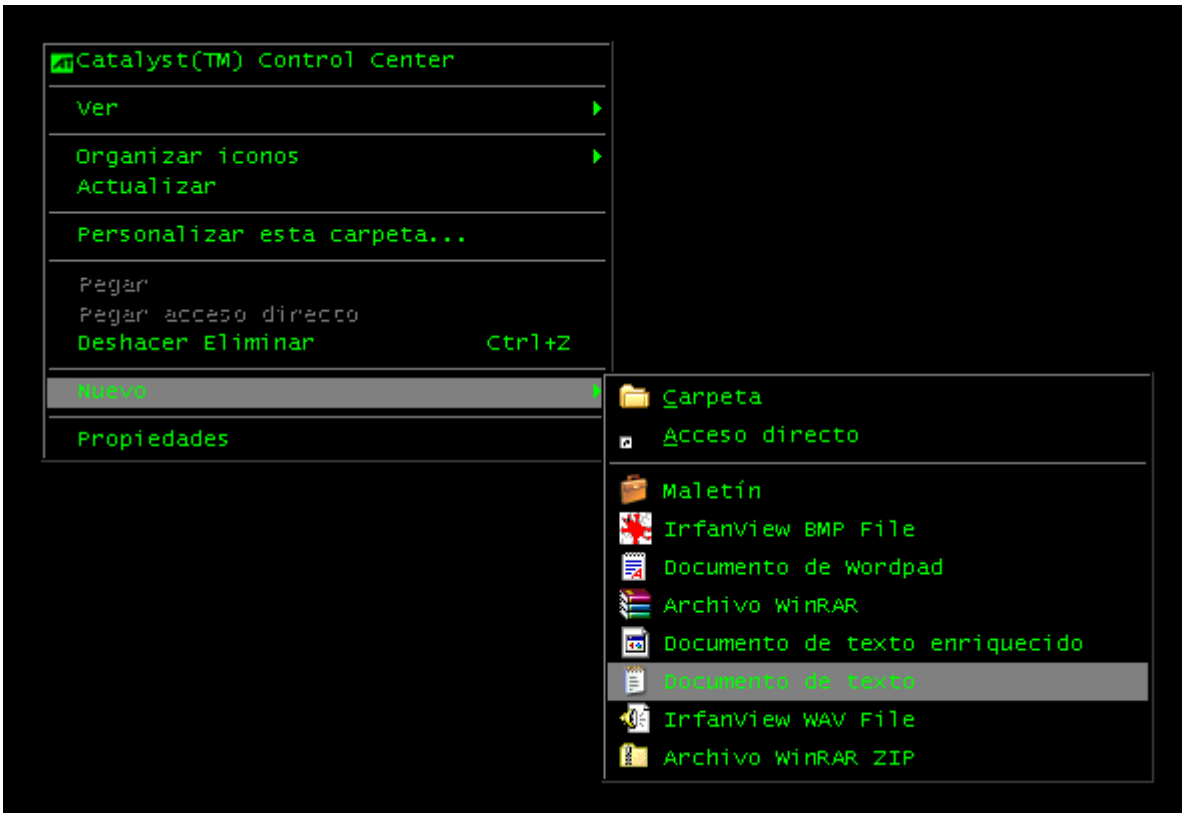
6.1- ASCII

6.2- Caracteres de escape.

Hasta ahora solo vimos cosas "descolgadas", que poca utilidad nos da. Ya que nada de lo que hicimos lo pudimos guardar, y tampoco había interacción con otra persona. Es por ello que ahora vamos a trabajar con ficheros .py, con lo que armaremos nuestro primer programa. Los ficheros .py son ficheros que serán traducidos y ejecutados por el intérprete. Cada vez que abramos uno de estos ficheros, todas las instrucciones dentro se ejecutarán, pudiendo con esto distribuir nuestros programas, etc. Para crear los ficheros .py, necesitamos símplemente un editor de texto plano, tal como el block de notas o cualquier otro. Python viene por defecto con el editor IDLE, el cuál es muy bueno. IDLE no es símplemente un editor de texto, si no que es un IDE. Un IDE es un Entorno de Desarollo Integrado. Consiste en un programa con herramientas integradas que permiten/facilitan el desarrollo. En mi caso voy a utilizar IDLE por ser defecto, y teniendo en cuenta que quizás haya gente que no tenga interés en bajarse otro editor. Pero, voy a dejar este enlace que habla de editores. Elijan el que mas les guste, e intenten no cambiarlos, ya que puede traer problemas de identación (algo fundamental, y que veremos en los temas siguientes).
http://elviajedelnavegante.blogspot.com.ar/2010/10/herramientas-ide-gui-editor-para_18.html

Repito, yo voy a usar IDLE. Aunque en lo personal, me decanto por Notepad++ y Emacs (El cuál no está mencionado ahí).

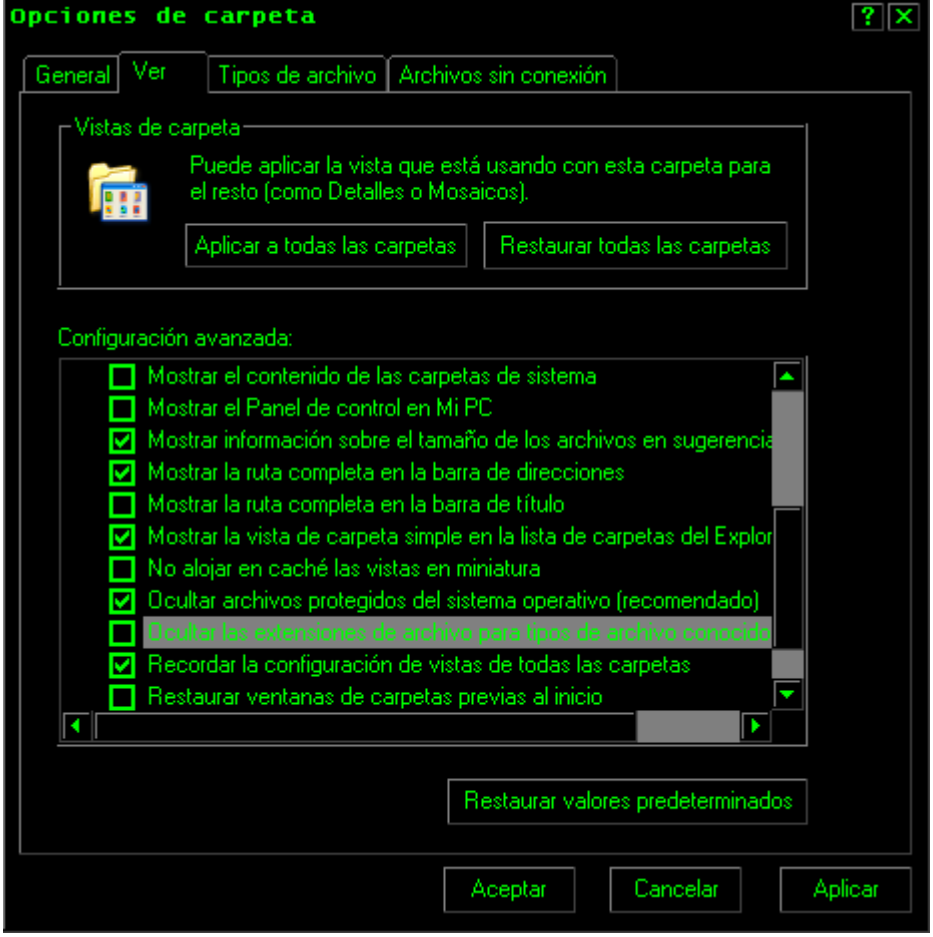
La manera mas rápida para crear un fichero es mediante la opción “nuevo” que aparece al oprimir el botón derecho sobre el escritorio o una carpeta, y creamos un fichero de texto:



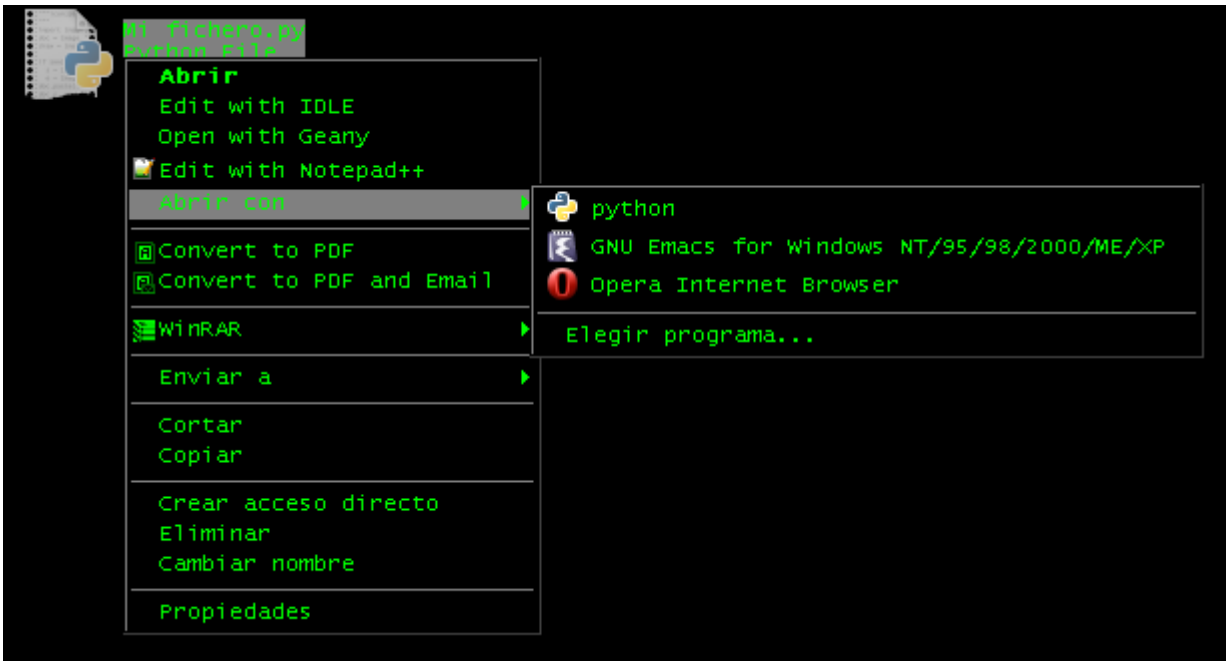
Le damos click y se nos creará un archivo con el nombre “Nuevo documento de texto.txt” seleccionado:

Reemplazamos el .txt por .py. Damos enter, y nos aparecerá un catastrófico mensaje hablando de apocalípticas posibilidades de tamaños bíblicos que podrán suceder si cambiamos la extensión de nuestro archivo. Le damos si, y esperamos a que nos invadan las ranas gigantes. No, sólamente, cambiará la extensión e ícono del fichero, debiendo quedar con este ícono:

Si no nos aparece para cambiar la extensión, nos dirijmos a herramientas>opciones de carpeta>ver y desmarcamos la opción “ocultar extensiones para archivos conocidos”:



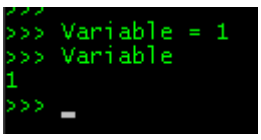
Le damos aceptar, y listo. Para comenzar a editar nuestro amado primer programa, le daremos click derecho a nuestro fichero .py y elejimos con qué programa editarlo o abrirlo:



En mi caso puedo hacerlo con IDLE, Geany, Notepad++ o Emacs. Damos click en “Edit with IDLE” y nos saldrá algo como esto:

Aquí escribiremos nuestro código Python, y también podremos ejecutar el intérprete (shell), o probar el código que vayamos creando. Recomiendo probar el programa normalmente luego de probarlo con IDLE. Esto se debe a que IDLE nos carga una shell, y ejecuta el script, el cuál al terminar, vuelve a la shell. ¿Cuál es el problema? Lo veremos en instantes. Es tradicional crear el típico programa "Hola, mundo!" cuando uno aprende un nuevo lenguaje. Y no será la excepción : -)

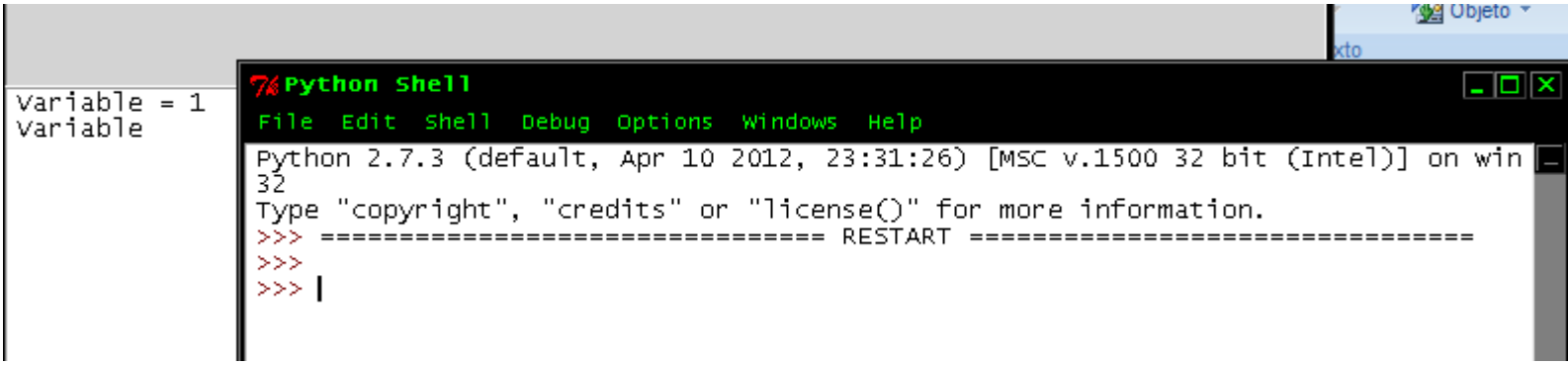
Hasta ahora, en la shell de Python, vimos que nos muestra información. Eso sucede solo de esa manera, puesto que en los ficheros .py, solo se permiten instrucciones. Fijémonos, esto sucede cuando definimos una variable en el intérprete, y tipeamos su identificador:



Nos devuelve el valor. Ahora, escribamos eso mismo en el IDLE y probemos:

```
Variable = 1
Variable
```

Para ejecutarlo, vamos a run y damos en 'run module':



No sucede nada. Nada se muestra en pantalla, y nos aparece el prompt indicando que el programa ya terminó. Entonces, vamos a file>save, y le damos doble click Mi Fichero.py. Vemos que la pantalla abre y cierra, y nada mas. ¿Por qué pasa esto? Es porque "Variable" no es una instrucción, jústamente, es una variable. Entonces, ¿cómo hago para mostrar algo en pantalla? Para ello se utiliza la instrucción print.

Su sintáxis es sencilla:

```
MiVar = 3
MiVariable = 4
print 'Algo a mostrar'
print 1
print MiVariable
print 'Algo' + 'Hola'
print 'Palabra' * 2
print str(MiVar) * 2
print str(float(MiVar) * (MiVariable * 3))
print MiVar, MiVariable, 'hola'
```

Vemos que podemos mostrar tanto cadenas, como números enteros o flotantes, o resultados de operaciones y funciones. Y observamos también que utilizamos el separador coma (,). Es diferente a concatenar, ya que no "concatena" si no que muestra los valores en pantalla de manera seguida. Apretamos F5 (para ejecutar el módulo) y obtenemos el siguiente resultado:

```
>>> ==========
>>>
Algo a mostrar
1
4
AlgoHola
PalabraPalabra
33
36.0
3 4 hola
>>>
```

33 y 36.0 son cadenas, al igual que 'Algo a mostrar', 'AlgoHola', 'PalabraPalabra' y 'hola'. Guardemos el fichero (ctrl s), y abrámoslo normalmente. ¿Qué ocurrió? El intérprete aparece y desaparece en un instante, sin que lleguemos a leer lo que dice. ¿Qué se necesita para que podamos leer? Necesitamos una manera de detener la ejecución justo después de que todo eso se muestre. Hagamos un pequeño algoritmo del problema:

```
Problema: Pausar para poder leer los mensajes.
1-) Mostrar mensajes.
2-) Detener ejecución.
```

¿Simple, no? Solo dos pasos. Ahora bien, ya sabemos qué hacer para solucionar el problema. Solo falta trasladarlo al lenguaje que deseamos, en este caso, Python. En algunos lenguajes existe la instrucción "pause". No es el caso acá. Pero aún así, podemos generar los mismos efectos. La forma mas sencilla es utilizando la función raw_input(). Esta función queda en espera hasta que el usuario ingrese algo por teclado, generando así un efecto de pausa indeterminada, la cual termina cuando el usuario aprete la tecla enter:

```
MiVar = 3
MiVariable = 4
print 'Algo a mostrar'
print 1
print MiVariable
print 'Algo' + 'Hola'
print 'Palabra' * 2
print str(MiVar) * 2
print str(float(MiVar) * (MiVariable * 3))
print MiVar, MiVariable, 'hola'

raw_input()
```

Guardamos, y abrimos el archivo normalmente. Eureka, el programa se detuvo y podemos leer todo. Presionamos enter, y el programa finaliza al no haber mas instrucciones que seguir. Pero, ¿y si el usuario no sabe qué hacer? Tendríamos que darle aviso al usuario, diciéndole que presione enter para terminar el programa. ¿Cómo podríamos hacerlo? Plantiemos el problema:

```
<Inicia programa>
1-) Mostrar mensajes.
2-) Detener ejecución.
3-) Mostrar aviso.
<Finaliza programa>
```

¿Es correcto este planteo? Si no te parece correcto, ¿cómo pensás que debería ser?. En caso contrario, vamos a probarlo. Traslademos esto a Python. ¿Qué maneras conocemos hasta ahora para resolver el problema? Solo print y raw_input(). Asumimos entonces, que con print debemos informar al

usuario:

```
MiVar = 3
MiVariable = 4
print 'Algo a mostrar'
print 1
print MiVariable
print 'Algo' + 'Hola'
print 'Palabra' * 2
print str(MiVar) * 2
print str(float(MiVar) * (MiVariable * 3))
print MiVar, MiVariable, 'hola'

raw_input()
print 'Presione una tecla para continuar.'
```

Guardamos, y abrimos el fichero normalmente. Vemos que se detiene, sin mostrar un mensaje. Apretamos enter, y de manera fugaz, nuestro mensaje aparece por una milésima de segundo, antes de que el intérprete se cierre, sin dejar que lo leamos. ¿Dónde está el problema? Volvamos al algoritmo:

```
<Inicia programa>
1-) Mostrar mensajes.
2-) Detener ejecución.
3-) Mostrar aviso.
<Finaliza programa>
```

Analizemos. Primero mostramos el mensaje, hasta ahí todo bien. Luego, detenemos la ejecución, y al final, mostramos el aviso. Ahí está nuestro error, claro! Para que un mensaje pueda ser leído, debe ser previo a una pausa, ya que luego de ser mostrado, lo próximo en ejecutarse es una instrucción que permita leerlo. Escribámoslo bien:

```
<Inicia programa>
1-) Mostrar mensajes.
2-) Mostrar aviso.
3-) Detener ejecución.
<Finaliza programa>
```

Ahora si! Trasladémoslo a Python. Ya mostramos los mensajes, y luego, deberíamos mostrar el aviso, y por último, permitir que sea leído:

```
MiVar = 3
MiVariable = 4
#Paso 1: Mostramos mensajes.
print 'Algo a mostrar'
print 1
print MiVariable
print 'Algo' + 'Hola'
print 'Palabra' * 2
print str(MiVar) * 2
print str(float(MiVar) * (MiVariable * 3))
print MiVar, MiVariable, 'hola'

#Paso 2: Mostramos aviso.
print 'Presione una tecla para continuar.'

#Paso 3: Pausamos para que sea leído.
raw_input()
```

Guardamos, abrimos el fichero, y el programa funciona. El aviso se muestra, y al presionar enter, termina el programa. ¿Y los numerales? Los numerales son comentarios útiles. Los comentarios no son ejecutados, son solo a modo explicativo. Son pasados por alto a la hora de la ejecución, y sirven para documentar nuestro código, ya sea para organizarnos nosotros o para que otra persona que vea nuestro código, tenga mas información al respecto. Los comentarios se delimitan por el caracter numeral (#).

Entonces, el algoritmo fué la lista de pasos para resolver el problema. Pero, la solución puede cambiar en cuánto a programación respecta. Por ejemplo, en este caso, con Python podemos realizar el paso 2 y 3 en una sola línea:

```
MiVar = 3
MiVariable = 4
#Paso 1: Mostramos mensajes.
print 'Algo a mostrar'
print 1
print MiVariable
print 'Algo' + 'Hola'
print 'Palabra' * 2
print str(MiVar) * 2
print str(float(MiVar) * (MiVariable * 3))
print MiVar, MiVariable, 'hola'

#Paso 2 y 3
raw_input('Presione enter para finalizar la ejecución')
```

Observamos que raw_input es una función que admite argumentos. En este caso el argumento es una cadena de texto, la cuál nos dice algo. raw_input() esperará a que se aprete enter, y devolverá los datos ingresados. Sería como un print incluído. Así entonces podemos también mostrar resultados de operaciones aritméticas, lógicas o resultados de alguna otra función así tanto como variables, al igual que con print:

```
raw_input('Mi cadena')
raw_input(UnNumero)
raw_input(str(UnNumero) + 'Mi cadena')
raw_input(MiVariable)
raw_input(float(UnNumero / OtroNumero))
```

Experimenten a gusto con el intérprete. Pero hagamos algo un poco mas... “””útil”””. Un programa que nos pida nombre, apellido, edad y nos diga “Felices años nombre apellido”, por ejemplo, si yo me llamo Carlos Alfombra y tengo 119 años, nos dirá “Felices 119 años, Carlos Alfombra!”.

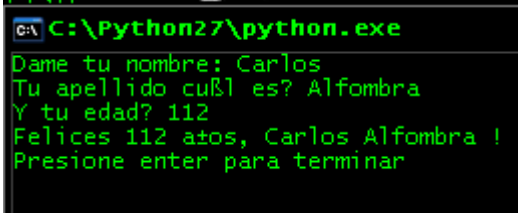
Plantiemos como solucionar el problema:

```
1  <Inicia Programa>
2  Pedir nombre.
3  Pedir apellido.
4  Pedir edad.
5  Mostrar mensaje.
6  Pausar programa.
7  <Finaliza programa>
```

Todo parece en orden... Excepto que ahora nosotros no sabemos que hacer! ¿O si? Volvamos atrás. Sabemos que necesitamos guardar información ahora, por lo cuál vamos a necesitar variables. Un máximo de 3 variables. Pero también necesitamos que el valor de dichas variables lo dé el usuario. El resto, no significaría un problema. Entonces, si lo que ingresa el usuario es lo que retorna la función raw_input() y en una variable podíamos guardar resultados de otras funciones, (como por ejemplo Variable = float(1)), ¿por qué no guardar lo que retorna raw_input? Veamos el código:

```
#Inicia programa

#2- Pedir nombre
Nombre = raw_input('Dame tu nombre: ')

#3- Pedir apellido
Apellido = raw_input('Tu apellido cuál es? ')

#4- Pedir edad.
Edad = raw_input('Y tu edad? ')

#5- Mostrar mensaje.
print 'Felices', Edad, 'años,', Nombre, Apellido, '!'

#6- Pausar.
raw_input('Presione enter para terminar')

#7- Nada mas que hacer, se cierra el programa.
```

En cada variable queda guardado lo que el usuario ingrese. La coma mostrará (separado pero seguido) toda la información. Probemos y veamos!



No es precisamente lo que esperábamos... Los pasos están bien. ¿Por qué muestra caracteres raros? Y un detalle final, el último signo de admiración debería estar pegado a 'Alfombra', no separado. Arreglemos este pequeño problema de caracteres! Uhm. Entonces, tenemos aquí nuestro primer problema de desinformación. No vimos temas que se puedan relacionar fácilmente a esto, por lo cuál debemos plantearnos qué buscar y dónde buscar. Obviamente, voy a terminar dando la respuesta, pero no sin antes pasearlos en cómo llegar a ella.

¿Cuál es el problema? El problema es que los caracteres en la consola se visualizan mal. Nuestra mejor fuente, es Google. ¿Qué tenemos que buscar específicamente? Bueno, podemos buscar "se ven mas los caracteres en Python" o algo así. Pero, es improbable que encuentres el resultado. ¿Por qué no probamos una búsqueda mas "técnica"? La acción de mostrar algo en pantalla se lo conoce como imprimir. Si, se le dice imprimir, aunque no haya impresora ni hojas de por medio. "Caracteres extraños en Python". Apenas buscamos, rozamos con gente con problemas similares, pero vemos que manejan temas que desconocemos. Y vemos conceptos como "utf", "coding cp1252", "ASCII", "caracteres de escape"... ¿Qué es todo esto? Lo vemos a continuación.

6.1 - ASCII

Algo fundamental para la computación es el hecho de representar información. Pero esa información no siempre es fácil de representar. Se inventaron diferentes métodos para poder representarla de manera mas sencilla. Imaginen expresar todo en bits. Por otra parte, lo fundamental para mostrar información es la escritura. El uso de caracteres. Para eso se diseñó el código ASCII (American Standar Code for Information Interchange, que sería algo como "Código Estándar Americano para el Intercambio de Información"). Pronunciado como "aski", es utilizado para representar caracteres de forma numérica, ya que el ordenador solo puede entender números. Hay diferentes variantes de ASCII y diversos códigos para la representación. Tan solo con poner "ASCII" en Google encontrarán la historia. Acá diré lo básico solamente.

En el código ASCII cada caracter es representado por un número. Utilizaba originalmente 7 bits para representar el caracter (Es decir una sucesión de siete "unos y ceros") y un bit de paridad. Con esa combinación se podían representar 128 caracteres (Del 0 al 127). Una variante conocida como ASCII extendido contaba con 256 (del 0 al 255) para representar esos 128 caracteres y mas, como las letras con tilde, la "ñ" y demás. Existen caracteres imprimibles (aquellos que se muestran) y caracteres de control, que cumplen una función y no son mostrados en pantalla (por ejemplo, esa línea que se deja al apretar enter, se produce por el caracter de control correspondiente). Los caracteres imprimibles van del 32 al 126 y para el ASCII Extendido, llega hasta el 254. Esos caracteres extendidos se los conoce como "especiales", y podemos representarlos oprimiendo la tecla alt y tecleando (en el teclado numérico) el número correspondiente al caracter. Por ejemplo, el código 65 le corresponde a la "A" (mayúscula). Apretando la tecla "alt" escribamos 65, y al soltar, aparecerá la "A". Dejo la tabla ASCII Extendida:

Caracteres ASCII de control		
00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles					
32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

ASCII extendido (Página de código 437)							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Cada número representará un caracter. Mas adelante, vamos a ver lo que se conoce como unidades de información, a fin de entender realmente el tema. Por ahora, basta y sobra saber que cada Sistema Operativo y hardware de un ordenador, puede tener diferentes configuraciones de idioma, y es por eso que puede haber problemas de comptabilidad, al no haber soporte para diferentes lenguajes. Piensen en un teclado para Japoneses, Rusos, Árabes... ¿Podrían escribir con nuestras mismas letras? ¿Y los ideogramas Chinos? Si quieren entender aún mas, y no quieren esperar, recomiendo la lectura en Internet sobre el tema.

Pasamos ahora a la representación de estos caracteres especiales, de vuelta a Python.

6.2- Caracteres de escape.

Pero, ¿qué tuvo que ver lo anterior con nuestro problema de los caracteres? ¿cómo lo solucionamos? Necesitamos utilizar lo que se conoce como caracteres de escape. Estos caracteres de escape son utilizados para representar tanto caracteres imprimibles como no imprimibles. (Break: Se murió Margaret Thatcher xD).

Todos los caracteres especiales que ingresemos NOSOTROS se mostrarán como tal, pero los que el programa muestre en pantalla, necesitarán ser representados mediante caracteres de escape. Para esto utilizamos la barra invertida "\":

Secuencia de escape para carácter de control	Resultado
\a	Carácter de «campana» (BEL)
\b	«Espacio atrás» (BS)
\f	Alimentación de formulario (FF)
\n	Salto de línea (LF)
\r	Retorno de carro (CR)
\t	Tabulador horizontal (TAB)
\v	Tabulador vertical (VT)
ooo	Carácter cuyo código ASCII en octal es *ooo*
\x*hh*	Carácter cuyo código ASCII en hexadecimal es *hh*

Otras secuencias de escape	Resultado
\\	Carácter barra invertida (\)
\'	Comilla simple (')
\"	Comilla doble (")
\ y salto de línea	Se ignora (para expresar una cadena en varias líneas).

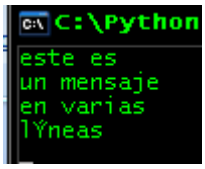
(fuente: http://www.dc.uba.ar/materias/int-com/2011/cuat1/Descargas/CadenasPython.pdf) Haremos énfasis por ahora en algunos ejemplos. Supongamos que queremos mostrar un mensaje en varias líneas:

```
print 'este es \nun mensaje \nen varias \nlíneas'
```

Lo que nos dá como salida lo siguiente:

```
...
>>>
este es
un mensaje
en varias
líneas
>>>
```

Vemos que por cada \n, se saltará una línea, como si presionásemos enter. Veamos que no separé la \n del caracter que le seguía. Eso es porque no quiero dejar un espacio antes de cada línea. Si ejecutamos eso normalmente, veremos que el problema de los acentos continúa. Agregemos una pausa al código y veamos:



Uf! ¿Y cómo voy a mostrar las letras con acentos? Para eso, utilizamos el caracter de escape \xhh, en donde "hh" se reemplaza por el código hexadecimal del caracter a mostrar. ¿Qué carajo es hexadecimal? Bueno, es otro sistema de representación de información. Por ahora, lo que nos importa es mostrar los acentos. Reemplazamos las HH por el código correspondiente. Para hacer esto, podemos utilizar la calculadora de Windows. La abrimos y la ponemos en modo científica:

Seleccionamos el modo decimal, escribimos el número y clickeamos donde dice “hex”. Nos dará el valor hexadecimal correspondiente, el cuál en el caso de la “ñ” es A4. Volvamos a nuestro programa y arreglémoslo!

```
Nombre   = raw_input('Ingrese su nombre: ')
Apellido = raw_input('Ingrese su apellido: ')
Edad     = raw_input('Ingrese su edad: ')

print 'Felices', Edad, 'a\xA4os, ', Nombre, Apellido + '!'
raw_input('Presione una tecla para continuar...')
```

```
C:\Python27\python.exe
Ingrese su nombre: Áñoñó
Ingrese su apellido: Kóriñö
Ingrese su edad: 27
Felices 27 años,  Áñoñó Kóriñö!
Presione una tecla para continuar...
```

Funciona! Vemos que en lugar de “ñ” (en “años”) pusimos \xA4. ¿Y por qué al final pusimos + y no una coma? Esto es porque el signo de admiración es seguido a la cadena, y no hay un espacio entre medio. La coma genera un espacio, mientras que el signo + con cadenas como operadores, concatena.

7- Salida con formato.

Además de poder recibir y mostrar información en pantalla, podemos darle un formato. Las cadenas en Python son inmutables. Es decir, una vez creadas, no podemos cambiar su contenido. Por lo que si intentamos hacer algo como esto:

```
Cad = 'Hola'
Cad[0] = 'B'
```

```
>>> ================================ RESTART ============
>>>

Traceback (most recent call last):
  File "C:\Python27\Jazzer\Mi fichero.py", line 2, in <mo
    Cad[0] = 'B'
TypeError: 'str' object does not support item assignment
>>>
```

Nos dará ese error. Pero, como vimos, podemos aún así, expresar cadenas que cambian según el valor de las variables.

Nos liamos mucho con el + y la coma. Hace menos legible lo que escribimos, y a menos que nuestra intención sea que el código se torne ilegible, es algo muy malo. Eso lo podemos solucionar usando el operador %. ¿Qué hace este operador? Veamos como funciona:

```
print 'Bienvenido %s, al mundo de %s' % ('Carlos', 'la inform\xA0tica')
raw_input()
```

```
C:\Python27\python.exe
Bienvenido Carlos, al mundo de la informática
```

Afortunadamente ninguno de nosotros se llama Carlos... ¿O si? Cada %s será reemplazado por su correspondiente dentro del paréntesis. El primer %s será reemplazado por ‘Carlos’, el segundo por ‘la inform\Xa0tica’, y así sucesivamente con cada argumento que agreguemos. Esos argumentos también pueden ser variables, o como siempre digo, cualquier operación. Los argumentos pueden ser tanto cadena como numéricos.

Bravo. ¿Seguimos sin hacer programas útiles, no? Esto es porque los programas empiezan y terminan siempre de la misma manera. Pero, de corazón les digo, aburrirse un poco ahora, les va a restar problemas adelante. Por ejemplo, es muy divertido entrar a un supermercado y matar a todos con un arma. Entonces, entramos a una secta extremista con el interés de disparar un arma. Nos enseñan a cargar las balas antes de disparar. Sería muy al pedo no saber apuntar/disparar si no sabemos cargar las balas. Lo mismo pasa acá.

8- Secuencias:

8.1- Listas.

8.2- Tuplas.

8.3- Diccionarios.

8.4- Métodos de secuencias.

Las secuencias son algo fundamental en Python. Son una estructura de datos. Un ejemplo de ellas son las cadenas (string). Pero no son las únicas. Gran parte de la potencia de Python radica en el uso de otras estructuras de datos y colección, como lo son las listas, tuplas y diccionarios. Las secuencias son jústamente una colección de elementos o ítems, los cuáles poseen un índice, a los cuáles se accede mediante ese índice. Pasemos a un famoso dentro de Python, la lista.

8.1- Listas.

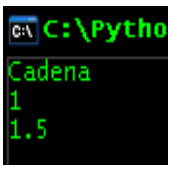
Las listas son similares a los array (si venís de otro lenguaje). Son colecciones de datos. Estos pueden ser de cualquier tipo, y variados, ya que pueden ser flotantes, enteros, cadenas, otras listas, y muchas cosas mas. Su definición es sencillísima:

```
MiLista = ['Cadena', 1, 1.5]
```

Se creó la lista de identificador “MiLista”. La misma contiene TRES elementos. Cada elemento es separado por una coma. Vemos que pusimos tres tipos de elemento, una cadena, un entero y un número decimal. Podemos acceder a cada elemento de la misma manera que lo haríamos con una cadena, mediante corchetes:

```
MiLista = ['Cadena', 1, 1.5]
print MiLista[0];
print MiLista[1];
print MiLista[2];

raw_input();
```

Y como resultado nos da:



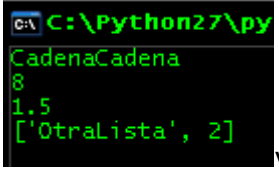
También puede contener operaciones, u otras listas dentro:

```
MiLista = ['Cadena'* 2, 1 + 7, str(1.5), ['OtraLista', 2]];
```

(El punto y coma al final es opcional, yo lo hago por costumbre, omítanlo).

Ahora, accedamos a los elementos:

```
MiLista = ['Cadena'* 2, 1 + 7, str(1.5), ['OtraLista', 2]];

print MiLista[0] # 'Cadena' * 2
print MiLista[1] # 1 + 7
print MiLista[2] # str(1.5)
print MiLista[3] # ['OtraLista', 2]

raw_input()
```

Nos mostrará esto en pantalla: 

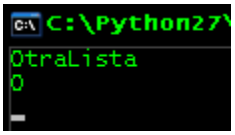
Vemos que el cuarto elemento (3) es otra lista. ¿Cómo hacemos para acceder a ella?

```
MiLista = ['Cadena'* 2, 1 + 7, str(1.5), ['OtraLista', 2]];

print MiLista[3][0] # 'OtraLista'
print MiLista[3][1] # 2

raw_input()
```

Es decir, primero accedemos al índice de la lista principal, y seguido a eso especificamos el índice de la sublista. Así también, podríamos acceder a la primer letra del primer elemento de la sublista:

```
MiLista = ['Cadena'* 2, 1 + 7, str(1.5), ['OtraLista', 2]];

print MiLista[3][0] # 'OtraLista'
print MiLista[3][0][0] # 'O'

raw_input()
```

Nos va a devolver



Ya que: 3 es la sublista. El primer 0 es el primer item de la sublista. Y el segundo 0 es el índice del elemento de la sublista al cuál queremos acceder, es decir índice 0 de la cadena 'OtraLista'. Lo importante es separar cada elemento con una coma. Las listas, a diferencia de las cadenas, son mutables. Esto quiere decir que sus elementos pueden ser cambiados una vez creada la lista:

```
MiLista = ['Cadena'* 2, 1 + 7, str(1.5), ['OtraLista', 2]];
print MiLista[0]
print 'El primer elemento de MiLista es "Cadena"*2'
MiLista[0] = 'XX'
print MiLista[0]
print 'Ahora el primer elemento es "XX"'

raw_input()
```

Lo que genera:

```
CadenaCadena
El primer elemento de MiLista es "Cadena"*2
XX
Ahora el primer elemento es "XX"
```

Lo cuál indica que la cadena no cambió, si no que se reemplazó por otro elemento, que en este caso, fué también una cadena de texto. Las listas son similares a las tuplas. Veámoslas.

8.2- Tuplas.

Esta estructura se define igual que las listas, exceptuando que en vez de corchetes, utilizamos la coma y opcionalmente, los paréntesis:

```
MiTupla = (1, 'Casa', float(5)/2)
```

Podríamos definirla de igual manera sin paréntesis:

```
MiTupla = 1, 'Casa', float(5)/2
```

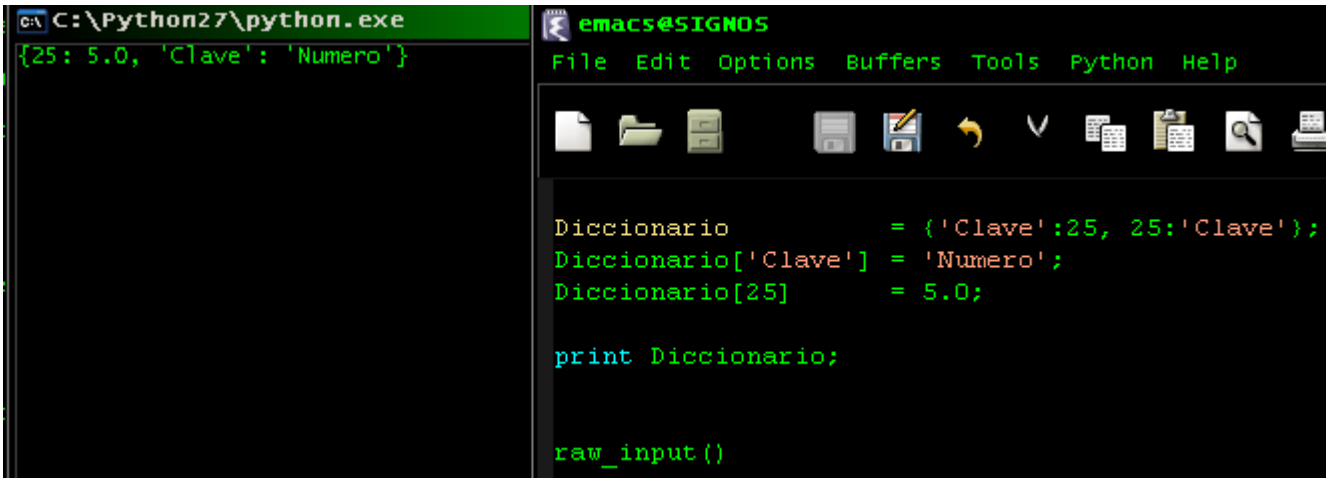
Accedemos a ella de la misma manera que a una lista, mediante corchetes. Entonces ¿cuál es la diferencia? La diferencia está en que las listas pueden ser modificadas, y las tuplas no. Una vez indicados sus elementos, no cambiarán a lo largo del programa. ¿Qué ventajas tienen frente a las listas? Su ventaja es que consumen y ocupan menos memoria. Si sabemos que una lista no será modificada a lo largo del programa, ¿por qué la usaríamos? Como programadores, economizar recursos, es fundamental. Además, son útiles para el retorno de funciones (algo que veremos mas adelante). Sin embargo, tenemos otro invitado mas. Los diccionarios.

8.3- Diccionarios.

Un diccionario es una colección identificada por lo que se conoce como par clave-valor. Para acceder a uno de sus valores, debemos hacerlo utilizando su clave. Veamos su definición:

```
MiDiccionario = {'Letra':'A', 1:'Numero', 'Numero':1}

MiDiccionario['Letra'] # ~Devolvería 'A'
MiDiccionario[1]       # ~Devolvería 'Numero'
MiDiccionario['Numero']# ~Devolvería 1
```

Para crear un diccionario, utilizamos paréntesis. Separamos mediante la coma los pares clave-valor, y a su vez, los pares clave-valor se identifican mediante dos puntos ":". Tanto la clave como el valor pueden ser de cualquier tipo de dato, exceptuado que las claves no pueden ser listas u otros diccionarios. A su vez, lo diccionarios son mutables, al igual que las listas:

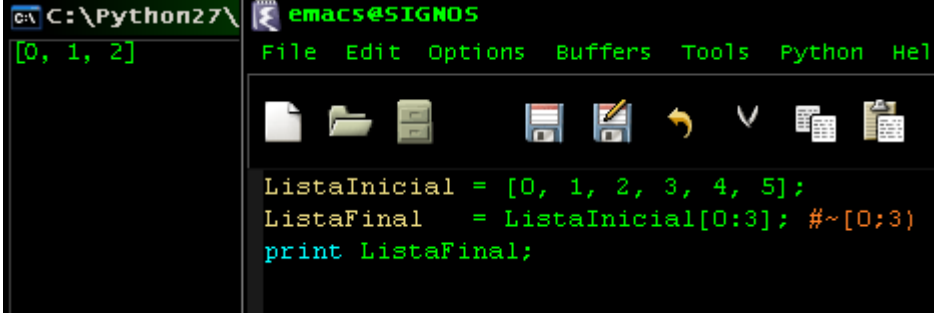


Vemos que el orden cambió, pero eso no importa, ya que accedemos a los valores mediante claves y no posiciones. Observamos que no cambiamos las claves, si no los valores a los cuáles se encuentran asociadas. Originalmente la clave 25 se asociaba al valor 'Clave' , y a la clave 'Clave' se asociaba el valor 25. Luego, la clave 25 se le asocia el decimal 5.0, y a la clave 'Clave' el valor 'Numero'.

Todo esto es muy lindo. Dijimos que los diccionarios pueden cambiar al igual que las listas (Eso se lo conoce como dinámico). Pero, no parecen muy útiles hasta ahora. Sucede que estas colecciones poseen métodos y propiedades. Pasemos al siguiente tema.

8.4- Métodos de secuencias.

Slicing: De "slice", cortar. Esta técnica nos permite "cortar" o referirnos a un fragmento de una colección. Supongamos la creación de una lista, que contenga cierto rango de elementos de otra lista:



Indicando entre corchetes el rango. A pesar de que indicamos como elemento final el 3 (Por lo que podríamos pensar que tendríamos 4 elementos), dicho elemento NO queda incluído en ListaFinal. Podemos también cambiar los valores o simplemente acceder:

```
Lista = [0, 1, 2, 3, 4, 5];
#---- Acceso a lista con slice.
print Lista[:]    #~Devuelve toda la lista.
print Lista[0:2]  #~Devuelve [0, 1]
print Lista[2:]   #~Devuelve [2, 3, 4, 5]
print Lista[:3]   #~Devuelve [0, 1, 2]

#---- Mutabilidad son slice.
Lista[0:1] = [1, 2]          #~Luego [1, 2, 1, 2, 3, 4, 5]
Lista[0:3] = [1]             #~Luego [1, 2, 3, 4, 5]
Lista[0:0] = [1, 2, 3, 4]    #~Luego [1, 2, 3, 4, 1, 2, 3, 4, 5]
```

Ejercicio: Crear una lista “L1”, y posteriormente, una lista “L2” que sea una copia idéntica. Sabiendo que al referirnos con [:] equivale a referirse a todo el contenido de una lista. Solo son necesarias dos líneas de código.

Pero, ¿y si queremos eliminar o agregar elementos?

```
Lista = []  #~ Una lista vacía.

Lista.append(elemento)    #~ Añade un elemento al final de la lista.
Lista.extend([1, 2])      #~ Añade una lista al final.
Lista.insert(1, elemento  #~ Añade un Elemento en indice indicado.
Lista.count(elemento)     #~ Cuenta las veces que Elemento se repite en la lista.
Lista.reverse()           #~ Invierte la lista.
Lista.sort()              #~ Ordena la lista.
del Lista[indice]         #~ Borra el elemento del índice especificado.
len(Lista)                #~ Devuelve la longitud de la lista.
```

Recordemos que los índices también pueden ser negativos. También es posible generar listas a partir de cadenas o valores mediante el método split():

```
Nombres = "Cholo, Pocho, Pito"
Nombres.split(",") #> nos devuelve la lista ['Cholo', 'Pocho', 'Pito']
```

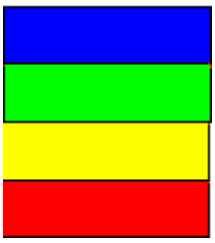
split() puede recibir como argumento qué se utilizará para separar cada elemento. Si no indicamos nada, se toma el espacio como separados. Podemos también, inicializar una variable como lista de la siguiente manera:

```
lista = list();
```

Se generará una variable de tipo lista, sin ningún contenido. También podríamos crearla de la siguiente manera, pero recomiendo hacerlo como dije recién: `lista = []`

Otra cosa interesante es su uso como PILAS o STACK. Una pila en informática es una estructura de datos, en dónde vamos apilando un dato arriba de otro. Imaginemos una pila de libros. A medida que vamos comprando libros, el mas nuevo se pone por encima del mas viejo, y así sucesivamente. Ahora, imaginemos que tenemos una sola mano y que no podemos sostener la pila de libros para que no se caiga, y queremos sacar un libro

del medio. Deberíamos sacar los de arriba, uno por uno, hasta llegar al deseado. Es por eso que las pilas o stack son estructuras denominadas LIFO (Last In, First Out. Último en entrar, primero en salir):

 < El elemento rojo fué el primero en entrar, y el azul fué el último. Si queremos acceder al rojo, debemos sacar el azul, luego el verde y seguido a eso el amarillo. Para utilizar las listas como pilas, utilizamos el método pop() y el método push().

El método push() ingresa un valor en el final de la lista, mientras que pop() devuelve el último valor de dicha lista. Pero, ¿no podríamos símplemente usar append() y del lista[-1]? Si bien cumplen la misma función, el programador debe no solo escribir de manera prolija, si no también de manera que se entienda la idea del programa, desde donde se aborda y desde qué punto de vista. Entonces, si queremos que la lista funcione como PILA, lo ideal sería usar métodos propios de las pilas (que es un concepto general, y existen en otros lenguajes), es decir push y pop. Por otra parte, si quisiésemos eliminar el último elemento de la lista, es mas sencillo poner lista.pop() que del lista[-1]. Queda en ustedes como programadores la decisión.

Con pop() a su vez podemos generar colas o queue. Suponiendo el orden de llegada de pacientes al doctor:

```
listaPacientes = ['Paciente1', 'Paciente2']
#Pacientes 1 y 2 ya están esperando... Pero llega otro mas:
listaPacientes.push('Paciente3')
#Al rato, un nuevo paciente entra al consultorio:
listaPacientes.push('Paciente4')

#El médico comienza a atender, y en este caso, el primero en entrar
#es el primero en ser atendido. pop() puede recibir como argumento la posición:
listaPacientes.pop(0) #Saca de la lista a Paciente1
listaPacientes.pop(0) #Saca de la lista a Paciente2, que pasó a estar en la primer posición de
 la lista.
print listaPacientes #~Devolvería ['Paciente3', 'Paciente4']
```

Es decir que push() y pop() agregan un elemento al final de la lista o lo sacan, pudiendo recibir argumentos.

Tuplas

De la misma manera que podemos crear una variable con formato de lista mediante list(), también podemos crear una variable con formato tupla mediate tuple(). Así como también podremos hacer definiciones múltiples en una línea:

```
x, y, z = 1, 2, 3 #x=1, y=2, z=3
```

Recordemos que las tuplas utilizan la coma como separador.

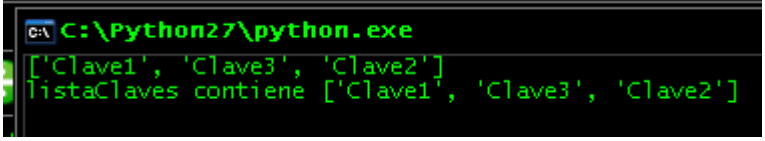
¿Se te ocurre alguna manera de intercambiar valores entre dos tuplas? ¿Cuál?

Diccionarios

Uno de los métodos de los diccionarios es el método keys(). Este método devuelve una LISTA con todas las claves en un diccionario:

```
Diccionario = {'Clave1':1, 'Clave2':2, 'Clave3':3}
print Diccionario.keys()
listaClaves = Diccionario.keys()
```

listaClaves sería una variable que contendría una lista con las claves:



El método sort() la ordena:

```
Diccionario = {'Clave1':1, 'Clave2':2, 'Clave3':3}

listaClaves = Diccionario.keys()
print 'listaClaves contiene', listaClavess.sort()
```

Así como tuple() y list(), los diccionarios se pueden definir con dict(), recibiendo como argumento UNA lista, con cada par clave-valor dentro de una tupla:

```
miDiccionario = dict([(Clave:Valor), (Clave:Valor)])
```

Otras operaciones a realizar pueden ser las siguientes:

```
miDic = {1:'Valor', 2:'Valor2', 3:'Valor3'}

miDic.has_key(1) #~ Verifica si miDic posee la clave 1. Devuelve True o False.
miDic.keys() #~ Devuelve una lista de tuplas con las llaves.
miDic.values() #~ Devuelve una lista de tuplas con los valores.
len(miDic) #~ Devuelve la longitud del diccionario.
del miDic[1] #~ Elimina el valor correspondiente a la clave 1.
```

Y por último, tenemos otro tipo de estructura. La cuál, puede resultar algo confusa: El conjunto.