Ficha 1

Fundamentos de Programación

1.] Breve referencia histórica: el advenimiento de las computadoras.

Resulta difícil objetar que la segunda mitad del siglo XX (y lo que va del siglo XXI) se vio fuertemente influenciada por la aparición de uno de los inventos más revolucionarios de la historia de la civilización: la computadora[1]. Y es que este aparato no sólo revolucionó la vida moderna, sino que en gran medida se ha convertido también en el sostén de la misma: hoy en día las computadoras se usan en casi cualquier actividad que admita alguna clase de automatización, y cada vez con más frecuencia también en tareas que otrora eran consideradas "exclusivas" (y distintivas) de la especie humana, como la creación de arte (pictórico, literario, musical), la interpretación de textos y el procesamiento de voz e imágenes, por citar sólo algunos ejemplos comunes.

Por otra parte, la computadora como concepto tecnológico ha sido desde el primer momento de su aparición un invento que no ha parado de mostrar innovaciones y mejoras, al punto que su evolución ha tomado por sorpresa a técnicos, científicos, académicos y empresarios de todas las épocas desde la década del 40 en el siglo XX, provocando muchas frases desafortunadas y errores notables en cuanto a las predicciones efectuadas respecto del futuro del invento.

Como sea, en aquellos lugares del mundo que han alcanzado cierto grado de desarrollo tecnológico resulta casi imposible imaginar la vida cotidiana sin el uso de computadoras. Se utilizan para control de ventas y facturación, para control de stock, para apoyo en diversas áreas de la medicina, en la guerra y la defensa, en la edición de textos, en educación, en el control de comunicaciones, en navegación aérea, espacial, naval y terrestre, en el diseño arquitectónico, en el cálculo, y por supuesto, en el juego...

De hecho, es tal el impacto que las computadoras tienen en esta vida moderna, que incluso llegó a especularse con las más diversas “debacles” tecnológicas cuando estaba llegando a su fin el siglo XX, y se temía que la tradicional forma de representar fechas usando sólo dos dígitos para indicar el año pudiera causar mundialmente problemas enormes (y graves): si una computadora en un banco tomara el año “02” como “1902” y no como “2002”, los resultados en el ámbito de la liquidación de intereses en una cuenta podrían ser inimaginablemente dañinos para la economía mundial, y ni qué hablar del peligro que habría producido un error semejante en el sistema de control de lanzamiento de misiles de una superpotencia, si ésta hubiera tomado el “efecto año 2000” a la ligera. En todo el mundo, miles de horas-hombre fueron destinadas a revisar y corregir ese problema durante los años anteriores al 2000, y finalmente puede decirse que todo transcurrió sin problemas.

[1] Todas las referencias históricas que siguen en esta sección, están basadas en los libros que se citan con los números [1] (capítulo 1), [2] (capítulo 1) y [3] (capítulo 4) de la bibliografía general que aparece al final de esta Ficha.

Las primeras computadoras que pueden llamarse "operativas" aparecieron recién en la década del 40 del siglo XX, y antes de eso sólo existían en concepto, o en forma de máquinas de propósito único que pueden considerarse antecedentes válidos de las computadoras, pero no computadoras en sí mismas. Pero el hecho aparentemente sorprendente es que a pesar del corto tiempo transcurrido desde su aparición como invento aplicable y útil, la computadora ha evolucionado a un ritmo increíble y ha pasado a ser el sustento tecnológico de la vida civilizada... ¡y todo en no más de sesenta años! Esto no es un detalle menor: ningún otro invento ha tenido el ritmo de cambio e innovación que ha tenido la computadora. Si los automóviles (por ejemplo) hubieran innovado al mismo ritmo que las computadoras, hoy nuestros autos serían similares al famoso vehículo volador mostrado en la trilogía de películas de "Volver al Futuro" (quizás exceptuando el "pequeño" detalle de su capacidad para viajar en el tiempo...)

Probablemente la primera persona que propuso el concepto de máquina de cálculo capaz de ser programada para procesar lotes de distintas combinaciones de instrucciones, fue el inglés *Charles Babbage* [1] [2] [3] en 1821, luego ayudado por su asistente *Ada Byron*[2], también inglesa. Ambos dedicaron gran parte de sus vidas al diseño de esa máquina: Babbage diseñó la máquina en sí misma, sobre la base de tecnología de ruedas dentadas, y Byron aportó los elementos relativos a su programación, definiendo conceptos que más de un siglo después se convertirían en cotidianos para todos los programadores del mundo (como el uso de instrucciones repetitivas y subrutinas, que serán oportunamente presentadas en estas notas) La máquina de Babbage y Byron (que Babbage designó como "*Analytical Engine*" o "motor analítico") era conceptualmente una computadora, pero como suele suceder en casos como estos, Babbage y Byron estaban demasiado adelantados a su época: nunca consiguieron fondos ni apoyo ni comprensión para fabricarla, y ambos murieron sin llegar a verla construida.

En 1890, en ocasión del censo de población de los Estados Unidos de Norteamérica, el ingeniero estadounidense *Herman Hollerith* [2] diseñó una máquina para procesar los datos del censo en forma más veloz, la cual tomaba los datos desde una serie de tarjetas perforadas (que ya habían sido sugeridas por Babbage para su Analytical Engine). Puede decirse que la máquina de Hollerith fue el primer antecedente que *efectivamente llegó a construirse* de las computadoras modernas, aunque es notable que el único propósito del mismo era la tabulación de datos obtenidos en forma similar al censo citado: no era posible extender su funcionalidad a otras áreas y problemas (cuando una máquina tiene estas características, se la suele designar como "máquina de propósito específico"). La compañía fundada por Hollerith para el posterior desarrollo de esa máquina y la prestación de sus servicios, originalmente llamada *Tabulating Machines Company*, se convirtió con el tiempo en la mundialmente famosa *International Business Machines*, más conocida como *IBM*.

En 1943, con la Segunda Guerra Mundial en su punto álgido, el matemático inglés *Alan Turing* [2] [3] estuvo al frente del equipo de criptógrafos y matemáticos ingleses que enfrentaron el problema de romper el sistema de encriptación de mensajes generado por la máquina *Enigma* usada por los alemanes. Para esto, el equipo de Turing trabajó en secreto para desarrollar una máquina desencriptadora que se llamó *Bombe*, gracias a la cual finalmente lograron romper el código *Enigma*. Sin embargo, cuando los alemanes cambiaron

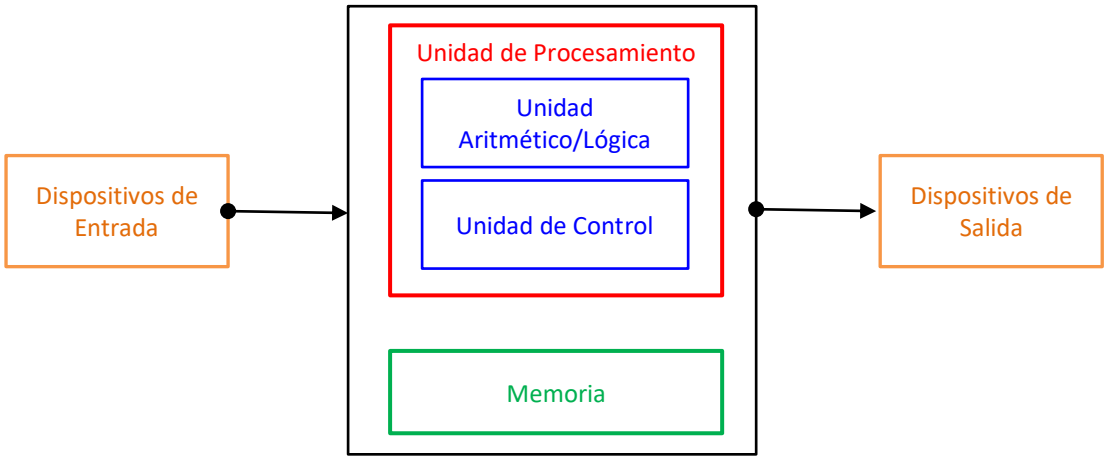
[2] Ada Byron, hija del célebre poeta inglés Lord Byron, era Condesa de Lovelace, y por este motivo suele aparecer en diversos textos y fuentes de consulta como *Ada Lovelace* en lugar de Ada Byron.

el código *Enigma* por el código *Lorenz SZ42*, Turing recurrió a *Tommy Flowers* para que diseñe y construya una nueva máquina, a la cual llamaron *Colossus*. Gracias a esta máquina finalmente se lograron descifrar los mensajes *Lorenz SZ42* y eso definitivamente llevó a los Aliados a la victoria. La historia y el aporte tanto de *Turing* como de *Flowers*, permanecieron en estricto secreto de estado hasta la década de 1970: ni siquiera los familiares y amigos cercanos de ambos estaban al tanto de lo que habían logrado[3].

La primera máquina que podemos llamar computadora en el sentido moderno, apareció en 1944 de la mano del ingeniero estadounidense *Howard Aiken* y su equipo de colaboradores de la firma IBM. La máquina se llamó *Mark I* y fue construida para la Universidad de Harvard, aunque quedó obsoleta casi en paralelo con su desarrollo, pues usaba conmutadores mecánicos controlados en forma electrónica, mientras que al mismo tiempo otros investigadores ya estaban desarrollando la tecnología de tubos al vacío que derivaría en computadoras totalmente electrónicas. De hecho, en 1945 apareció la *Electronic Numerical Integrator and Calculator*, más conocida como *ENIAC*, que fue la primera computadora electrónica de gran escala. La ENIAC fue diseñada por los ingenieros estadounidenses *John Mauchly* y *Presper Eckert* para la Universidad de Pensilvania [2].

En el mismo proyecto ENIAC trabajó *John von Neumann*, quien realizó aportes para el diseño de un nuevo tipo de máquina con programas almacenados designada como *EDVAC* (*Electronic Discrete Variable and Calculator*) [2]. El diseño de *von Neumann* incluía una *unidad de procesamiento* dividida en una *unidad aritmético-lógica* y una *unidad de control*, además de contar con una *unidad de memoria* para almacenar tanto al programa como a sus datos, y también permitía *periféricos de entrada y salida* (ver Figura 1). Este esquema con el tiempo se convirtió prácticamente en un estándar, al que se conoció como el *Modelo de von Neumann* o también *Arquitectura von Neumann* y muchas computadoras diseñadas desde entonces se basaron en el.

Figura 1: Esquema general de la Arquitectura von Neumann.



[3] La historia de *Alan Turing* al frente del equipo que desarrolló la maquina *Bombe*, ha sido llevada al cine en 2014 a través de la película dirigida por *Morten Tyldum* cuyo título original es "*The Imitation Game*" (y que en Latinoamérica se conoció como "*El Código Enigma*"). En diversos pasajes de la película pueden verse reproducciones de las máquinas *Enigma* y *Bombe*. La película incluso estuvo nominada al *Oscar*, así como el actor *Benedict Cumberbatch* que interpretó al propio *Alan Turing*.

A finales de la década de 1950 y principios de la de 1960 los tubos al vacío fueron reemplazados por los transistores, lo que permitió abaratar costos y aumentar potencia. La aparición de los circuitos integrados hacia finales de la década del 60 permitió reunir muchos transistores en un solo circuito, abaratando aún más el proceso de fabricación, y permitiendo reducir cada vez más el tamaño (y no sólo el costo) del computador. Esto abrió el camino que llevó a nuestros días: a finales de la década de 1970 aparecieron las primeras computadoras personales (*Apple II* e *IBM PC*), cuyo costo y tamaño las hacían accesibles para el público promedio... y el resultado está a la vista.

Sin embargo, el hecho que las computadoras hayan ocupado el sitio que ocupan en tan breve plazo, y que evolucionen en la forma vertiginosa que lo hacen, no es en realidad una sorpresa: simplemente se debe al hecho de su propia *aplicabilidad general*. Las computadoras aparecieron para facilitar tareas de cálculo en ámbitos académicos, científicos y militares, pero muy pronto se vio que podrían usarse también en centenares de otras actividades. Invertir en mejorar el invento valía la pena, y el invento mismo ayudaría a descubrir y definir nuevas áreas de aplicación.

2.] Algoritmos y programas.

En nuestros días las computadoras conviven con nosotros. Donde miremos las encontraremos[4]. Es común que haya incluso más de una en cada hogar con nivel de vida promedio. Estamos tan acostumbrados a ellas y tan acostumbrados a usarlas para facilitar tareas diversas, que ni siquiera pensamos en lo que realmente tenemos allí. Si se nos pregunta: ¿qué es una computadora? la mayor parte de las personas del mundo dará una respuesta aproximada, intuitivamente correcta, pero casi nunca exacta y completa. Puede decirse que eso mismo ocurrirá con la mayor parte de los inventos que hoy usamos o vemos a diario, y sin embargo hay una sutil diferencia: sabemos qué es un avión, aunque no sabemos bien cómo funciona ni cómo logra volar. Sabemos qué es un lavarropas, aunque no sabemos bien cómo funciona. Lo mismo se puede decir de los automóviles, las heladeras, los relojes, etc.

Pero con las computadoras es diferente. Podemos darnos cuenta que una computadora es un aparato que puede aplicarse para resolver un problema si se le dan los datos del mismo, de forma que procesará esos datos en forma veloz y confiable y nos entregará los resultados que buscamos. Pero el hecho es que nos servirá para *muchos problemas diferentes*, incluso de áreas diferentes. La computadora no es una herramienta de propósito único o limitada a una pequeña gama de tareas. Si la proveemos con el *programa* adecuado, la computadora resolverá cualquier problema. Y aquí está la gran cuestión: podemos ser usuarios de un automóvil sin saber cómo funciona, pues sólo necesitamos que funcione y cumpla su único cometido: trasladarnos donde queremos ir. Pero si queremos que una computadora resuelva los problemas que enfrentamos día a día en nuestro trabajo, o en nuestro ámbito de estudio, o en nuestro hogar, no nos basta con prenderla y limitarnos a verla funcionar. De una forma u otra deberemos cargar en ella los programas adecuados, y a través de esos programas realizar nuestras tareas. Un usuario común de computadoras se limitará a cargar programas ya desarrollados por otras personas. Pero un usuario avanzado, con

[4] Una referencia complementaria general para esta sección, incluye los libros que aparecen con los números [1] (capítulo 14) y [4] (capítulo 1) en la bibliografía disponible al final de esta Ficha.

conocimientos de programación, podrá formular sus propios programas, o desarrollarlos para terceros (percibiendo eventualmente una remuneración o pago por ello...) [4].

La tarea de programar un computador no es sencilla, salvo en casos de problemas o ejemplos muy simples. Requiere inteligencia, paciencia, capacidad de resolver problemas, conocimientos técnicos sobre áreas diversas, creatividad, auto superación, autocrítica, y muchas horas de estudio.

El punto de partida para el largo camino en la formación de un programador, es la propia definición del término *computadora*: se trata de un aparato capaz de ejecutar una serie de órdenes que permiten resolver un problema. La serie de órdenes se designa como *programa*, y la característica principal del computador es que el programa a ejecutar puede ser cambiado para resolver problemas distintos.

La definición anterior sugiere que la computadora obtendrá los resultados esperados para un problema dado, pero sólo si alguien plantea y carga en ella el programa con los pasos a seguir para resolverlo. En general, el conjunto finito de pasos para resolver un problema recibe el nombre de *algoritmo* [1] [4]. Si un algoritmo se plantea y escribe de forma que pueda ser cargado en una computadora, entonces tenemos un *programa*.

En última instancia una computadora es lo que conoce como una "*máquina algorítmica*", pues su capacidad esencial es la de poder seguir paso a paso un algoritmo dado. Notemos que la tarea del computador es básicamente ejecutar un algoritmo para obtener los resultados pedidos, pero el planteo del algoritmo en forma de programa que resuelve un problema es realizado por una persona, llamada *programador*. En ese sentido, las computadoras no piensan ni tienen conciencia respecto del problema cuyos resultados están calculando: simplemente están diseñadas para poder ejecutar una orden tras otra, hasta llegar al final del programa.

Los algoritmos entonces, son parte fundamental en el proceso de programación, pero no se encuentran sólo en el mundo de la programación de computadoras: los algoritmos están a nuestro alrededor, y nos permiten llevar a cabo un gran número de tareas cotidianas: encontramos algoritmos en un libro de recetas de cocina, en un manual de instrucciones, o en los gráficos que nos permiten ensamblar un dispositivo cualquiera, por citar algunos ejemplos. También ejecutamos algoritmos cada vez que hacemos un cálculo matemático[5], o conducimos un automóvil, o grabamos un programa de televisión.

Los algoritmos tienen una serie de propiedades importantes, que son las que permiten su uso generalizado en tareas que pueden descomponerse en pasos menores. Esas propiedades son las siguientes:

i. El conjunto de pasos definido para el algoritmo debe tener un ***final previsible***. Un conjunto de pasos infinito no es un algoritmo, y por lo tanto no es susceptible de ser ejecutado. La obvia conclusión de esta propiedad es que si un algoritmo está bien planteado, se garantiza que quien lo ejecute llegará eventualmente a un punto en el cual se detendrá. Por ejemplo, una secuencia de pasos diseñada para aplicarse sobre cada uno de los números naturales (los números enteros del 1 en adelante), no es un

[5] De hecho, la palabra ***algoritmo*** proviene del nombre del matemático árabe ***al-Jwarizmi***, que en el siglo IX dC planteó las reglas de las cuatro operaciones aritméticas para el sistema decimal. El conjunto de esas reglas se conoció en Europa como "algoritmia" (después "algoritmo"), y se generalizó para designar a cualquier conjunto de reglas para un problema dado.

algoritmo pues los números naturales son infinitos, y la secuencia planteada nunca terminaría de ejecutarse.

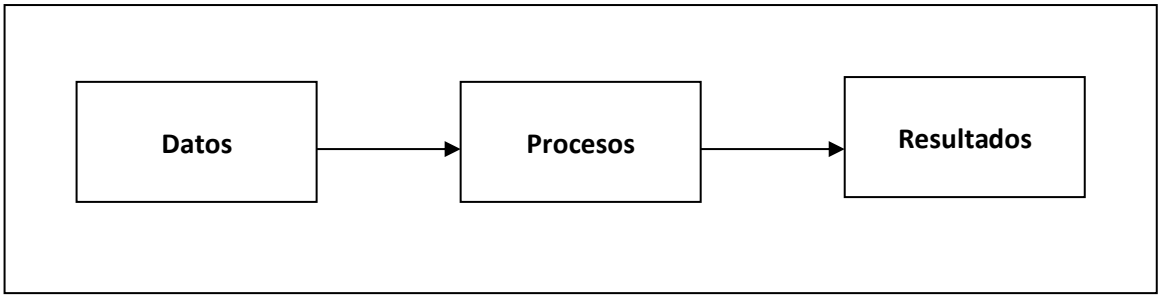
ii. Una vez que se plantea un algoritmo para una tarea o problema dado, y se verifica la corrección del mismo, ***no es necesario volver a pensar la solución*** del problema cada vez que se desee obtener sus resultados. Los algoritmos permiten ahorrar tiempo y esfuerzo a quienes los aplican. Por ejemplo, cada vez que sumamos dos números de varios dígitos, aplicamos un conocido algoritmo basado en sumar primero los dígitos de las unidades, luego los de las decenas, y así sucesivamente, acarreando los excesos hacia la izquierda, hasta obtener la suma total. *Sabemos* que ese algoritmo funciona, aunque no nos detenemos a pensar *porqué* funciona.

iii. Un algoritmo consiste en una combinación de pasos básicos más pequeños, que eventualmente pueden ser ***automatizados***. Esto implica que quien ejecuta un algoritmo no necesariamente debe ser una entidad inteligente (como una persona), sino que también puede ser ejecutado por una máquina, o por un animal convenientemente entrenado. Esa es la "magia" detrás de las computadoras: poseen un componente muy sofisticado llamado ***procesador***, electrónicamente diseñado para realizar ciertas operaciones básicas (esto es, operaciones aritméticas, comparaciones y operaciones lógicas) sobre los datos que recibe. Un programa es una combinación de esas operaciones básicas que el procesador ejecuta una tras otra, a una velocidad muy superior a la del propio cerebro humano.

Desde el momento que un algoritmo es la secuencia de pasos finita que permite resolver un problema, queda claro que un algoritmo opera sobre la base de tres elementos obvios: comienza sobre los *datos* del problema, suponiendo que se aplican sobre ellos los *pasos, procesos o instrucciones* de los que consta el algoritmo, obteniendo finalmente los *resultados* esperados (ver Figura 2).

Un requisito fundamental en el planteo de un algoritmo, es el conocimiento de cuáles son las *operaciones primarias, básicas o primitivas* que están permitidas de combinar para formar la secuencia de procesos que resuelve el problema. Este detalle subordina fuertemente el planteo. Por ejemplo, supongamos que se nos pide plantear un algoritmo que permita obtener, a cualquiera que lo siga, dibujos aceptables de una cara humana esquematizada. Es obvio que ese algoritmo podría incluir combinaciones de instrucciones tales como "dibuje un óvalo de forma que el diámetro mayor quede en posición vertical" (para representar los contornos de la cara), o como "dibuje dos pequeños óvalos (que asemejen los ojos) dentro del óvalo mayor". Pero, ¿qué pasaría si entre el juego de operaciones elementales que se suponen permitidas no estuviera contemplada la capacidad de dibujar óvalos? Si el algoritmo estuviera siendo diseñado para ejecutarse por una máquina limitada tipo "plotter" de dibujo, pero sin capacidad para desplazamientos diagonales, el dibujo de curvas sería imposible, o muy imperfecto.

Figura 2: Estructura de un Algoritmo



Por otra parte, quien diseña el algoritmo debe analizar el problema planteado e identificar a qué resultados se pretende arribar. Una vez identificado el o los resultados esperados, se procede a identificar cuáles son los datos que se disponen y por último qué acciones o

procesos ejecutar sobre dichos datos para obtener los resultados. Esto ayuda a poner orden en las ideas iniciales, dejando claros tanto el punto de llegada como el de partida. Si no se tienen claros los datos, se corre el riesgo de plantear procesos que luego no pueden aplicarse simplemente por la ausencia de los datos supuestos. Si no se tienen en claro los resultados, se puede caer en el planteo de un algoritmo que no obtiene lo que se estaba buscando... y en el mundo de los programadores, hay pocas cosas que resulten tan frustrantes e inútiles como la solución adecuada al problema equivocado [4].

Finalmente, el algoritmo debe plantearse de forma que pueda ser asimilado y ejecutado por quien corresponda. Si se diseña un algoritmo para ser ejecutado por una persona, podría bastar con escribirlo en un idioma o lenguaje natural, en forma clara y esquemática, aunque siempre asumiendo que la persona que lo ejecutará entiende las instrucciones primitivas de las que está compuesto. En general, una persona entenderá el algoritmo aunque el mismo tuviera alguna imprecisión o falta de claridad.

Pero si el algoritmo se diseña para ser ejecutado por una máquina, entonces debe ser planteado o escrito en términos que hagan posible que el mismo sea luego transferido a la máquina. Si la máquina es una computadora, el algoritmo debe escribirse usando un *lenguaje de programación*, generando así lo que se conoce como un programa [1] [4]. Existen muchos lenguajes de programación. Algunos de los más conocidos son *C*, *C++*, *Basic*, *Pascal*, *Java* o *Python*, siendo este último el que usaremos a lo largo del curso.

Un *lenguaje de programación* es un conjunto de símbolos, palabras y reglas para combinar esos símbolos y palabras de forma muy precisa y detallada. Los lenguajes de programación prevén un cierto conjunto mínimo de palabras válidas (llamadas "*palabras reservadas*") y un cierto conjunto de símbolos para operaciones generalmente matemáticas y lógicas (llamados "*operadores*"). Pero no cualquier combinación de esos operadores y palabras reservadas resulta en una orden válida para el lenguaje, de la misma forma que no cualquier combinación de palabras castellanas resulta en una frase comprensible en español. Los operadores y palabras reservadas deben combinarse siguiendo reglas específicas y muy puntuales. El conjunto de esas reglas se conoce como la *sintaxis* del lenguaje. Por ejemplo, en el lenguaje *Python* se puede preguntar si una proposición dada es cierta o falsa por medio de una *instrucción condicional* que se forma con las palabras reservadas *if* y *else*, junto con algunos signos especiales los dos puntos (*:*). Para escribir la condición a evaluar se pueden usar operadores como <, >, >=, etc. Pero la *sintaxis* o *conjunto del reglas del lenguaje* exige que esas palabras, signos especiales y operadores se escriban de una forma determinada para que la instrucción condicional sea válida. Una forma válida de escribirla sería:

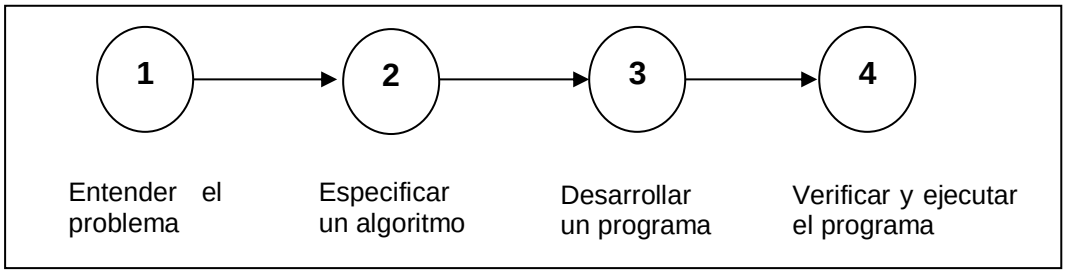
```
if x >= 10:
      # realizar aquí alguna tarea, a ejecutarse
      # si fuera cierto que x >= 10
else:
      #  realizar aquí alguna otra tarea,
      #  a ejecutarse si no fuera cierto que x >= 10
```

Sin embargo, la siguiente combinación de las mismas palabras, símbolos y operadores (que originalmente estaban en color azul en el modelo anterior) es de *sintaxis totalmente incorrecta en Python* (¡y muy posiblemente también en cualquier otro lenguaje!):

```
else x: = 10 >
: if
```

En definitiva, para que un computador pueda ser usado para resolver un problema, los pasos a seguir se ven en el siguiente esquema:

Figura 3: Pasos para resolver un problema mediante un computador.



Los pasos 1 y 2 del esquema anterior resultan esenciales para seguir adelante en el proceso. "Entender el problema" (es decir, lograr una "representación del problema") es básicamente saber identificar los *objetivos* (o *resultados*) a cumplir, junto a los *datos* de los cuales se parte, y ser capaz de plantear una estrategia general de resolución en la que se indiquen los *procesos básicos* a realizar para llegar a los resultados partiendo de los datos. En esta etapa de comprensión inicial, esa estrategia de planteo no requiere mayor nivel de detalle: basta con que se pueda simplemente comunicar en forma verbal una línea de acción general. Si bien puede parecer una etapa simple y obvia, los hechos muestran que en realidad se trata del punto más complicado y difuso en el proceso de resolver un problema. Si el problema no es de solución trivial, quien plantea el problema debe ser capaz de intuir una línea de acción que lo conduzca, eventualmente, a una solución correcta. Mientras más experimentada en resolución de problemas es la persona, mayor es la probabilidad que la línea de acción intuida sea correcta. Sin embargo, no hay garantías: quien intenta resolver el problema sigue rutas lógicas que su experiencia indica pueden ser correctas, pero el éxito no es seguro. A menudo los caminos elegidos llevan a puntos muertos, o terminan alejándose de la solución buscada. Pues bien: el conjunto de estrategias, procesos y caminos lógicos que una persona *intuye* que puede servir para dar con una solución a un problema no trivial (en cualquier campo o disciplina), se conoce con el nombre de *heurístico* (del griego *heuriskin*: lo que sirve para resolver), o más general, como *método heurístico*.

Es obvio que una persona con escasos o nulos conocimientos en el área del problema, difícilmente podrá encontrar una solución al mismo. Pero si los conocimientos se suponen dados, aun así los hechos muestran que encontrar una "estrategia resolvente" para un problema es una tarea exigente. Como vimos en la Figura 3, todo el trabajo que un programador debe realizar comienza con la inspiración de un heurístico: la *intuición de una estrategia resolvente*. Si en este punto falla o no logra ni siquiera comenzar, no podrá dar un solo paso más y la computadora no le servirá de mucho.

Es en el segundo paso, con el planteo del algoritmo, donde la estrategia elegida debe ser detallada con el máximo y exhaustivo rigor lógico. En esta etapa, cada paso, cada condición, cada rama lógica del proceso, debe ser expresada sin ambigüedad. Toda la secuencia de acciones a cumplir para llegar a los objetivos planteados debe quedar evidenciada en forma clara y terminante. Normalmente, se emplean en esta etapa una serie de símbolos para facilitar la tarea de mostrar el algoritmo, formando lo que se designa como un *diagrama de flujo* junto a otros diagramas específicos. Cabe aclarar sin embargo, que hasta aquí lo importante es plantear en forma rigurosa la lógica del algoritmo, y no los detalles sintácticos

del programa que finalmente se cargará en la computadora. En el momento de plantear un algoritmo, los detalles sintácticos no son importantes (de hecho, ni siquiera es relevante cuál sea el lenguaje de programación que se usará).

En el paso 3, y sólo cuando se ha logrado el planteo lógico de la solución (usando un diagrama de flujo o alguna otra técnica), es que se pasa a la "etapa sintáctica" del proceso: el paso siguiente es desarrollar un programa mediante algún lenguaje de programación, lo cual requiere de ciertos conocimientos esenciales y básicos que se adquieren sin mayores dificultades en una o dos semanas de estudio.

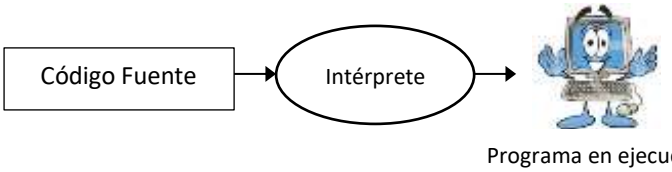
El programa que escribe un programador usando un lenguaje de programación se denomina *programa fuente* o *código fuente*. Antes de poder ejecutar un programa, se debe verificar si existen errores de escritura y/o otras anomalías en el código fuente. Para ello se usan otros programas (que vienen incluidos con el kit de desarrollo o *SDK* del lenguaje usado) llamados *compiladores* o *intérpretes* según la forma de trabajo de cada uno.

Un *compilador* es un programa que verifica el archivo de *código fuente* de otro programa detectando errores tales como palabras mal escritas, signos mal usados, etc. Como resultado, si todo anduvo bien, el *compilador* produce *otro* archivo conteniendo el llamado *programa compilado* o *código compilado* o también *código ejecutable*, y es este *programa compilado* el que *realmente* se ejecuta en la computadora (ver la Figura 4). Ahora sí aparecen las reglas del lenguaje y detalles tan triviales como la ausencia de una coma harán que el compilador no pueda *entender* el programa, acusando errores que impedirán que ese programa comience a ejecutarse.

Figura 4: Esquema general de trabajo de un *compilador*.

El esquema de trabajo de un *intérprete* es ligeramente distinto, ya que un *intérprete* ***no genera otro archivo*** con el código compilado, sino que ejecuta directamente, línea por línea, el *código fuente* (ver la Figura 5). Si la línea que actualmente está analizando no contiene errores, la ejecuta y pasa a la siguiente, continuando así hasta llegar al final del programa o hasta encontrar una línea que contenga un error de sintaxis (en cuyo caso, la ejecución se interrumpe mostrando un error) .

Figura 5: Esquema general de trabajo de un *intérprete*.



Algunos lenguajes de programación son *compilados*. Otros son *interpretados*. Y algunos trabajan en forma *mixta*, compilando el código fuente para producir un archivo *de código intermedio* no ejecutable en forma directa (llamado *código de byte*) que a su vez se ejecuta por medio de un *intérprete especial* de ese lenguaje (y que suele designarse como *máquina virtual* del lenguaje). Ejemplos conocidos de *lenguajes compilados* son *C*, *C++* y *Pascal*. Como

ejemplo de *lenguaje mixto* podemos citar a *Java*. Y en nuestro caso más próximo, tenemos a *Python* que finalmente es un ejemplo de *lenguaje interpretado*.

A pesar de todo lo dicho, cabe aclarar que por razones de comodidad y de *uso y costumbre* entre los programadores, tanto los errores de sintaxis detectados e informados por un compilador como los detectados e informados por un intérprete son designados indistintamente como *errores de compilación*: es decir, errores en la *sintaxis* de un programa fuente que impiden que ese programa comience a ejecutarse y/o finalice en forma completa (si está siendo ejecutado por un *intérprete*).

Como vimos, un programa no es otra cosa que un algoritmo escrito en un lenguaje de programación. Conocer las características del lenguaje es un paso necesario, pero no suficiente, para poder escribir un programa: alguien debe plantear antes un algoritmo que resuelva el problema dado. En todo caso, la computadora (a través del compilador o el intérprete) ayuda a encontrar *errores sintácticos* (pues si un símbolo falta o está mal usado, el compilador o el intérprete avisa del error), pero no puede ayudar con los métodos heurísticos para llegar a un algoritmo.

Cuando el programa ha sido escrito y verificado en cuanto a sus errores de compilación, el mismo puede ser cargado en la computadora y finalmente en el paso 4 se procede a solicitar a la computadora que lo *ejecute* (es decir, que lleve a cabo las tareas indicadas en cada instrucción u orden del programa). Si el programa está bien escrito y su lógica subyacente es correcta, puede confiarse en que los resultados obtenidos por el ordenador serán a su vez correctos. Si el programa contiene errores de lógica, aquí debe realizarse un proceso de revisión (o verificación) y eventualmente repetir los pasos desde el 2 hasta el 4. Veremos más adelante numerosos ejemplos de aplicación de este proceso.

Para terminar esta introducción, debe notarse que el motivo básico por el cual una persona que sabe plantear un algoritmo recurre a un computador para ejecutarlo, es la gran velocidad de trabajo de la máquina. Nadie duda, por ejemplo, que un ingeniero sabe hacer cálculos de estructuras, pero con una computadora adecuadamente programada podrá hacerlo mucho más rápido.

3.] Fundamentos de representación de información en sistema binario.

Una computadora sólo sirve si posee un programa cargado al que pueda interpretar y ejecutar instrucción por instrucción. Además del propio programa, deben estar cargados en el interior de la computadora los datos que ese programa necesitará, y del mismo modo la computadora almacenará los resultados que vaya obteniendo.

Esto implica que una computadora de alguna forma es capaz de representar y retener información dentro de sí. En ese sentido, se llama *memoria* al dispositivo interno del computador en el cual se almacena esa información y básicamente, la memoria puede considerarse como una gran tabla compuesta por celdas individuales llamadas *bytes*, de modo que cada *byte* puede representar información usando el *sistema de numeración binario* basado en estados eléctricos.

Para entender lo anterior, digamos que esencialmente una computadora sólo puede codificar y decodificar información mediante *estados eléctricos*. Cada celda o *byte* de memoria, está compuesto a su vez por un conjunto de ocho terminadores eléctricos, que reciben cada uno el nombre de *bit* (acrónimo del inglés *binary digit* o *dígito binario*). En la

práctica, se suele reducir la idea de *bit* a la de un componente que en un momento dado puede estar *encendido* (o con *valor 1*) o bien puede estar *apagado* (o con *valor 0*) [1]. Por este motivo, decimos que una computadora sólo puede usar e interpretar el *sistema de numeración binario* (o de *base 2*), que consta únicamente de los dígitos 0 y 1.

En base a los dos únicos dígitos 0 y 1, la computadora debe codificar cualquier tipo de información que se requiera. Si se trata de representar números enteros, está claro que los dígitos 0 y 1 del sistema binario servirán sin problemas para representar el 0 y el 1 del *sistema de numeración decimal* (o de *base 10*). Por lo tanto, un único bit podría usarse para representar al 0 o al 1 en la memoria. Pero si se quiere representar al número 2 o al 3, está claro que necesitaremos *más bits* y combinar sus valores [5].

Dado que un bit permite representar dos estados posibles (esto es: $2^1 = 2$ estados), entonces dos bits *juntos* pueden representar $2^2 = 4$ estados. Con eso alcanzaría para representar en binario a los dígitos del 0 al 3 de la base 10:

Figura 6: Representación en binario de los primeros cuatro números decimales.

Dígito decimal	Representación binaria (precisión: 2 bits)
0	00
1	01
2	10
3	11

La cantidad de bits usada para representar en binario un valor cualquiera, se conoce como la *precisión* de la representación. En la tabla anterior, se representaron los cuatro primeros dígitos decimales, con precisión de dos bits.

Puede verse claramente por qué motivo en la memoria de un computador, los bits se agrupan tomados de a ocho para formar bytes: una precisión de 8 bits permitiría representar $2^8 = 256$ números enteros diferentes en un único byte. Por lo tanto, si a modo de ejemplo asumimos que *sólo se representarán números positivos*, entonces un único byte puede usarse para representar números enteros en el intervalo [0, 255] (lo cual equivale a 256 números, incluido el 0). Si se quiere representar números enteros positivos mayores a 255, entonces la computadora *agrupa dos o más bytes*, e interpreta el contenido agrupado para obtener un único valor. Así, usando *dos bytes* agrupados, se tiene un total de 16 bits que implican $2^{16} = 65536$ números enteros positivos diferentes, en el intervalo [0, 65535].

Y así continúa: mientras mayor sea la magnitud del número que se quiere representar, mayor será la cantidad de bits que requerirá para su representación interna en un computador, y mayor será entonces la cantidad de bytes que ocupará. Como los números enteros negativos también se representan en binario, usando una técnica designada como *complemento a dos*, entonces un único byte normalmente puede almacenar tanto negativos como positivos, en precisión de 8 bits, pero esto (y otros detalles técnicos) hace que el rango de valores ahora sea el del intervalo [-128, 127] (o sea, 256 números desde el -128 hasta el 127, incluyendo al 0). Si se usan dos bytes, entonces el rango de valores enteros es [-32768, 32767] (o sea, 65536 números, desde el -32768 hasta el 32767 incluido el 0). Usando 4 bytes, se tienen 32 bits de precisión, con $2^{32} = 4294967296$ valores enteros diferentes (más de 4 mil millones de números) en el intervalo [-2147483648, 2147483647] (que incluye al 0) si se admiten negativos.

La representación de caracteres (letras, símbolos especiales, etc.) también se hace en binario, asignando a cada caracter una combinación predefinida de bits. Esas combinaciones constituyen estándares muy conocidos de la industria informática, y se representan en tablas en las que a cada caracter o símbolo, se le hace corresponder una combinación de bits que representa un número. Quizás el estándar más conocido sea el designado como *ASCII* (abreviatura de las iniciales de *American Standard Code for Information Interchange* o *Código Estándar Americano para el Intercambio de Información*). En el estándar ASCII, cada caracter se representa con un único byte (o sea, ocho bits). De aquí se deduce que un caracter en memoria ocupará un byte si se usa el estándar *ASCII*, y que una palabra (o *cadena de caracteres*) ocupará tantos bytes como caracteres incluya [1].

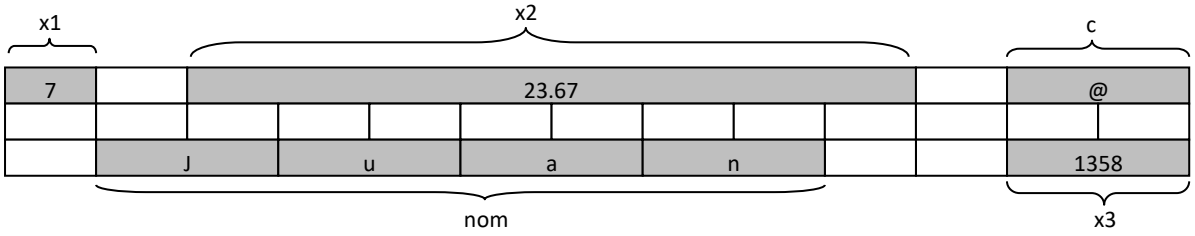
Dado que el estándar *ASCII* sólo emplea un byte por caracter, la cantidad de caracteres diferentes que pueden representarse mediante *ASCII* es de sólo 256. Eso era suficiente en las primeras épocas de la industria informática, pero a partir de los 80 en adelante comenzaron a surgir exigencias de representación de un mayor número de caracteres (como las letras de los alfabetos orientales, por ejemplo). En consecuencia, se propusieron otros estándares de mayor precisión y uno de ellos (muy difundido actualmente) es el estándar *Unicode* en el cual cada caracter puede ser representado con *uno o más bytes* (dependiendo del tipo de codificación que se use, como *UTF-8* o *UTF-16*, por ejemplo) en lugar de sólo uno.

4.] Elementos básicos de programación en *Python*: tipos, variables y asignaciones.

Cuando se desea ejecutar un programa, el mismo debe cargarse en la memoria del computador. Un programa, entonces, ocupará normalmente varios cientos o miles de bytes. Sin embargo, con sólo cargar el programa no basta: deben cargarse también en la memoria los *datos* que el programa necesitará procesar. A partir de aquí, el programa puede ser ejecutado e irá generando ciertos *resultados* que de igual modo serán almacenados en la memoria. Estos valores (los datos y los resultados) ocuparán en memoria un cierto número de bytes, que como vimos depende del *tipo de valor* del que se trate (por ejemplo, en general en distintos lenguajes de programación un valor de *tipo entero* ocupa entre uno y ocho bytes en memoria, un valor de *tipo real* o de *coma flotante*, con punto y parte decimal, ocupa entre cuatro y ocho bytes, y un *caracter* ocupa uno o dos bytes dependiendo del estándar de representación empleado) [4].

En todo lenguaje de programación, el acceso a esos datos y resultados ubicados en memoria se logra a través de ciertos elementos llamados *variables*. Una *variable* es un grupo de bytes asociado a un nombre o identificador, de tal forma que a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a la variable. En el siguiente gráfico, se muestra un esquema conceptual referido a cómo podría verse la memoria de un computador, con cuatro variables (*x1*, *x2*, x3, *c*, *nom*) alojadas en ella, suponiendo que estamos usando el lenguaje *Python*:

Figura 7: Memoria de un computador (esquema)



En el gráfico (que es sólo una *representación libre* de la idea de memoria como tabla formada por casilleros), se supone que la variable *x1* contiene el valor 7, y del mismo modo el resto de las variables del cuadro contiene algún valor. El nombre de cada variable es elegido por el programador, de acuerdo a ciertas reglas que se verán más adelante.

Está claro que lo que *realmente* se almacena en cada variable es la *representación binaria* de cada valor, pero aquí por razones se simplificación mostramos los valores tal como los entiende una persona. La cantidad de bytes que se usó para graficar a cada variable está en correspondencia con el tipo de valor representado y también de acuerdo a la forma de trabajo de *Python*: en este lenguaje, una variable que almacene un número entero ocupará *automáticamente* la cantidad de bytes que requiera para poder representar ese valor. Por lo tanto, la variable *x1* sólo ocupará un byte (ya que el número 7 representado en binario puede almacenarse sin problemas en un único byte). Pero la variable *x3*, que también almacena un número entero, en este caso ocupa dos bytes ya que el valor 1358 en binario requiere al menos de dos bytes para poder ser representado.

En la misma gráfica, la variable *x2* se representó con ocho bytes. El motivo es que *x2* contiene el valor 23.67 que no es entero, sino de *coma flotante*. En Python, *todos* los números de *coma flotante* (o *reales*) se representan con ocho bytes de precisión, sin importar la magnitud del número.

Finalmente, la variable *c*, que contiene un único caracter (el signo @) se graficó con dos bytes. El motivo: *Python* utiliza el estándar *Unicode* para representar caracteres. Por la misma razón, la variable *nom* en la gráfica ocupa ocho bytes: dos por cada caracter de la palabra "Juan".

Para darle un valor a una variable en Python (o para cambiar el valor de la misma)[6] se usa la *instrucción de asignación*, que consiste en escribir el nombre de la variable, seguido del signo igual y luego el valor que se quiere asignar. El signo igual (=) se designa como *operador de asignación* [6]. Ejemplo:

```
x1 = 7
```

La instrucción anterior, internamente, convierte el valor 7 a su representación binaria, y almacena ese patrón de bits en la dirección de memoria en la que esté ubicada la variable *x1*, manejando automáticamente *Python* la cantidad de bytes que necesitará.

La clase de valores que una variable puede contener en un momento dado, se llama *tipo de dato*. Los lenguajes de programación proveen varios *tipos de datos* estándar para manejar variables. Así, en el lenguaje *Python* existen al menos tres tipos elementales que permiten manejar números enteros, números en coma flotante, y valores lógicos (*True* y *False*). Además, *Python* permite manejar cadenas de caracteres y muchos otros tipos de datos predefinidos. El cuadro de la Figura 8 muestra los nombres y algunas otras características de los tipos de datos que hemos citado (otros tipos serán estudiados oportunamente).

En muchos lenguajes (tales como *C*, *C++* o *Java*) *no se puede* utilizar una variable en un programa si la misma no fue convenientemente *declarada* en ese programa. En esos lenguajes, la *declaración de una variable* se hace mediante la llamada *instrucción*

[6] La fuente de consulta general y más recomendable para el uso y aplicación del lenguaje Python, son sus propios manuales, tutoriales y documentación publicados en forma oficial por la *Python Software Foundation*. Esa documentación está designada con el número [5] en la bibliografía que aparece al final de esta Ficha.

declarativa, que implica indicar el nombre o identificador de la variable, e indicar a qué tipo de dato pertenece esa variable. Los lenguajes de programación que obligan a declarar una variable antes de poder usarla, comprueban que los valores asignados sean del tipo correcto y lanzan un error en caso de no serlo. En general, estos lenguajes se suelen denominar *lenguajes de tipado estático*.

Figura 8: Tabla de tipos de datos elementales en Python (sólo los más comunes).

Tipo (o Clase)	Descripción	Bytes por cada variable	Rango
bool	valores lógicos	1	[False, True]
int	números enteros	dinámico[7]	ilimitado[8]
float	números reales	8	hasta 15 decimales
str	cadenas de caracteres	2 * cantidad de caracteres	Unicode

Por su parte, *Python* y otros lenguajes (como *Perl* o *Lisp*) son *lenguajes de tipado dinámico*: esto significa que una misma variable puede recibir y almacenar valores de *tipos diferentes* durante la ejecución de un programa, y por lo tanto *las variables no deben declararse en base a un tipo específico prefijado antes de ser usadas*.

A diferencia de los lenguajes de tipado estático, una variable en Python se define (o sea, se aloja en memoria) *en el momento en que se le asigna un valor* (de cualquier tipo) por primera vez, y el tipo inicialmente asumido para esa variable es el que corresponda al valor que se le asignó. Por ejemplo, el siguiente esquema *define* cuatro variables en Python:

```
a = 14
b = 23.456
c = 'Python'
d = True
```

La variable *a* se define inicialmente como de tipo entero (*int*), y toma el valor 14. Respectivamente, las variables *b*, *c* y *d* se definen como flotante (*float*), cadena de caracteres (*str*) y lógica (*bool*). Note en los cuatro casos, el uso del *operador de asignación* (el *signo igual* =) para llevar a cabo la asignación de cada valor en la variable respectiva. Note también que en cada una de las cuatro instrucciones se usó directamente el operador de asignación y el lenguaje deduce el tipo y el tamaño en bytes de la variable de acuerdo al valor asignado. El *tipo es implícito* y en ningún caso se requiere usar el propio nombre de cada tipo (*int*, *float*, *str*, *bool* o el que fuese).

Si se trata de asignar un número entero, es suficiente con escribir ese número luego del operador de asignación, eventualmente precedido de un signo menos. Si el número es de coma flotante, recuerde usar *el punto* (y no *la coma*) como separador entre la parte entera y la parte decimal. Si quiere asignar una cadena de caracteres o un caracter único a una variable, encierre la cadena o el único caracter entre *comillas dobles* o entre *apóstrofos* (o *comillas simples)* como se hizo en el ejemplo. Como veremos, el uso de comillas dobles o simples es indistinto en *Python*, siempre que se mantenga consistente: si abrió con comillas dobles, debe cerrar con comillas dobles, pero si abrió con apóstrofos, debe cerrar con apóstrofos. Y finalmente, si desea asignar un valor lógico (de tipo *bool*) a una variable, debe

[7] Es decir, la cantidad de bytes que ocupa una variable de tipo int se calcula y asocia en el momento en que se necesita.

[8] En este contexto, "ilimitado" significa que no hay un *límite teórico* al rango de valores enteros que se pueden representar en Python, pero hay un *límite práctico* que es la *capacidad de memoria disponible*.

recordar que sólo existen dos posibles valores *bool*: el *True* y el *False*, sin comillas y escritos exactamente así: la *T* en mayúscula para el primero, y la *F* en mayúscula para el segundo.

Como *Python* es un lenguaje de *tipado dinámico*, entonces en Python una *variable puede cambiar de tipo* si se le asigna un valor de un tipo diferente al que tenía inicialmente, sin que esto provoque ninguna clase de error, y puede hacerse esto tantas veces como se quiera o necesite el programador. En la siguiente secuencia, la variable *n* se asocia inicialmente con el valor entero 20, se muestra en la consola de salida ese valor, y luego se cambia el valor y el tipo de *n* en las instrucciones sucesivas, para ilustrar la idea:

```
# asignación inicial de un valor int a la variable n…
n = 20
print('Valor de n: ', n)

# ahora el valor de n cambia a "Juan" (que es un str)…
n = 'Juan'
print('Nombre almacenado en n: ', n)

# y ahora se vuelve a cambiar el valor de n a un boolean…
n = False
print('Valor booleano almacenado en n: ', n)
```

Observe el uso del *signo numeral* (#) para introducir un *comentario de línea* que será ignorado por el intérprete Python. Lo que sea que se escriba a la derecha del signo # será tomado como un texto fuera del programa por Python, y su efecto dura hasta el final de la línea. Si se desea introducir un comentario que ocupe más de una línea, se recomienda simplemente escribir esas líneas iniciando a cada una con su correspondiente numeral (#), como en el ejemplo que sigue:

```
#
#  Un comentario de párrafo…
#  Asignación inicial de un valor int
#  a la variable n
#
n = 20
print('Valor de n: ', n)
```

Todo el bloque escrito en rojo en el ejemplo anterior, será ignorado por Python como si simplemente no existiese. Existe un tipo de comentario de párrafo designado como *comentario de documentación* o *cadena de documentación* (conocido como *docstring* en la terminología propia de Python) que consiste en usar el signo *triple apóstrofo* (' ' ') o *triple comilla* (" " ") para indicar el inicio del texto comentado, y nuevamente el *triple apóstrofo* o la *triple comilla* para cerrar su efecto. Sin embargo, los *docstrings* son tipos especiales de comentarios, que se usan con el objetivo de documentar un sistema y no sólo para intercalar texto libre en un programa. Volveremos en una ficha posterior sobre los *docstrings*.

Los dos ejemplos anteriores nos muestran además (muy básicamente) la forma de usar la función predefinida *print()* para mostrar un mensaje y/o el valor de una variable en la consola estándar de salida [6].

Note también que *Python* es *case sensitive* (o sea, es sensible a la diferencia entre mayúsculas y minúsculas) tanto para palabras e identificadores reservados, como para identificadores del programador. Así, por caso, el nombre de la función es *print* y no *Print*, y como vimos, las constantes *True* y *False* deben escribirse con la primera letra en mayúscula y el resto en minúscula pues de otro modo se producirá un error.

El tamaño en bytes de una variable es automáticamente ajustado por Python de acuerdo al valor que se asigna en la misma, sin que ese detalle deba preocupar al programador. Así, si se trabaja con números enteros y se asignan a una variable valores como 4, 2345 o 54763, Python adecuará el tamaño de la variable usada a uno, dos o cuatro bytes según sea el caso, y así en forma similar con otros valores que requieran mayor espacio o con otros tipos de datos.

Un hecho importante derivado de la forma de trabajo de la instrucción de asignación, es que cuando se asigna un valor a una variable, esta asume el nuevo valor, y *pierde cualquier valor anterior* que hubiera contenido. Por ejemplo, considérese el siguiente segmento de programa:

```
a = 2
a = 4
```

En este caso, se comienza asignando el valor 2 a la variable *a*, pero inmediatamente se asigna el 4 en la misma variable. El valor final de *a* luego de estas dos instrucciones, es 4, y el 2 originalmente asignado se pierde. Lo mismo ocurre con cualquier otra variable, sea cual sea el tipo de la misma.

En Python una variable puede cambiar dinámicamente de tipo, pero lo que *no* se puede hacer es intentar usar una variable sin haberle asignado previamente un valor inicial alguna vez. El siguiente ejemplo provocará un error (suponiendo que la variable *b* no haya sido usada nunca antes):

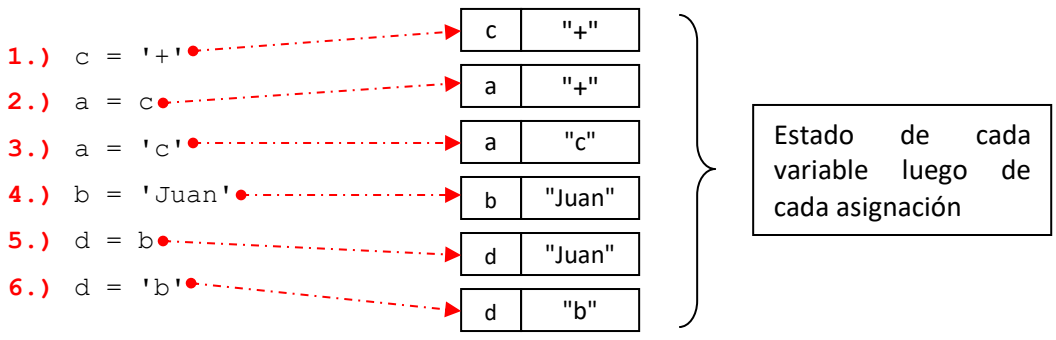
```
a = 34
print('Valor de a: ', a)

# esto provocará un error: b no está previamente asignada…
print("Valor de b: ", b)
```

En algunas situaciones, puede esperarse que una variable sea asignada con el valor *None* (que implícitamente equivale a un *False*), indicando que la variable *es válida* pero carece de valor en ese momento:

```
r = None
print('Valor actual de r: ',  r)
```

Los siguientes ejemplos son válidos para ilustrar algunos elementos básicos en cuando a gestión de variables y las posibilidades de la instrucción de asignación (los números a la izquierda de cada línea se usan para poder hacer referencia a cada ejemplo, y no forman parte de un programa en Python):

Figura 9: Ejemplos de asignaciones de variables en Python.



En la línea 1.) se asigna el caracter "+" a la variable *c*, y luego en la línea 2.) se asigna la variable *c* en la variable *a*. Esto significa que la variable *a* queda valiendo *lo mismo* que la variable *c* (es decir, ambas contienen ahora el signo "+")[9]. Observar la diferencia con lo hecho en la línea 3.), en donde se asigna a la variable *a* el caracter "c" (y **no** el *contenido de la variable c*). En la línea 4.) se asigna la cadena "Juan" a la variable *b*, y en la línea 5.) se asigna el contenido de la variable *b* en la variable *d* (de nuevo, ambas quedan valiendo lo mismo: la cadena "Juan"). En la línea 6.) se está asignando literalmente la cadena "b" a la variable *d*, por lo que las líneas 5.) y 6.) no son equivalentes…

Para finalizar esta sección, recordemos que el nombre o identificador de una variable (y en general, el identificador de cualquier elemento creado por el programador) es elegido por el propio programador, pero para ello deben seguirse ciertas reglas básicas, que indicamos a continuación [6]:

- El nombre o identificador de una variable en Python, sólo puede contener letras del alfabeto inglés (mayúsculas y/o minúsculas, o también dígitos (0 al 9), o también el guion bajo **(_)** (también llamado guion de subrayado).
- El nombre de una variable *no debe comenzar* con un dígito.
- Las palabras reservadas del lenguaje Python no pueden usarse como nombres de variables.
- El nombre de una variable puede contener cualquier cantidad de caracteres de longitud.
- Recordar que Python es *case sensitive*: Python hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador *sueldo* no es igual al identificador *Sueldo* y Python tomará a ambos como dos variables *diferentes*.

Hacemos notar que estas reglas son las más elementales, y son las que fija la versión Python 2. Pero en la versión Python 3 se permite utilizar letras acentuadas, letras que no sean del alfabeto inglés, letras de idiomas no occidentales, etc.[10] En la medida de lo posible, sin embargo, trataremos de ajustarnos a las cinco reglas que dimos más arriba para evitar confusiones. Por ejemplo, los siguientes identificadores de variables son válidos en la versión Python 3:

Figura 10: Ejemplos de identificadores de variables *válidos* en Python 3.

Identificador	Observaciones
n1	Válido en Python 2 y Python 3
nombre_2	Válido en Python 2 y Python 3
sueldo_anterior	Válido en Python 2 y Python 3
x123	Válido en Python 2 y Python 3
año	Válido en Python 3 – No válido en Python 2 por el uso de la **ñ** (no inglesa)
número	Válido en Python 3 – No válido en Python 2 por el uso de la **ú** (no valen acentos en inglés)

[9] Cuando se asigna un caracter o una cadena de caracteres a una variable, las comillas (simples o dobles) usadas para delimitar al caracter o a la cadena, no quedan almacenados en la variable. En la gráfica se muestran las comillas como si estuviesen contenidas en las variables, pero sólo por razones de claridad.

[10] En definitiva, en Python 2 un identificador se forma usando caracteres *ASCII* que representen letras, guiones bajos o dígitos, pero en Python 3 se aceptan también caracteres Unicode (*no – ASCII*), para dar cabida a símbolos propios de otros idiomas diferentes del inglés.

Pero los siguientes son incorrectos, por los motivos que se indican:

Figura 11: Ejemplos de identificadores de variables *no válidos* en Python 3.

Identificador	Incorrecto por el siguiente motivo:
3xy	Comienza con un dígito.
dir ant	Usa un espacio en blanco como separador (caracter no válido).
nombre-2	Usa el guion alto y no el bajo como separador
else	Es palabra reservada del lenguaje.

5.] Visualización por consola estándar y carga por teclado.

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. Como ya adelantáramos, el lenguaje Python (versión Python 3) provee la función predefinida *print()* para hacerlo[11]. Esta función permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres, lo cual es útil para lograr salidas de pantalla amigables para quien use nuestros programas [6]. La forma de usar la función se muestra en los siguientes ejemplos:

```
a = 3
b = a + 1
print(b)
```

Aquí, la instrucción de la última línea muestra en pantalla el valor contenido en la variable *b*, o sea, el número 4. Sin embargo, una salida más elegante sería acompañar al valor en pantalla con un mensaje aclaratorio, lo cual puede hacerse en forma similar a lo que sigue:

```
print('El resultado es:  ', b)
```

Observar que ahora no sólo aparecerá el valor contenido en *b*, sino también la cadena "*El resultado es:* " precediendo al valor. En general, si lo que se desea mostrar es una combinación entre cadenas de caracteres y valores de variables, entonces cada elemento se escribe entre los paréntesis, usando una coma a modo de separador, en el orden en que se espera que aparezcan visualizados,

Notar también que para mostrar el valor de una variable, el nombre de la misma *no lleva comillas ni apóstrofos*, pues en ese caso se tomaría al nombre en forma literal. La instrucción que sigue, muestra literalmente la letra "*b*" en pantalla:

```
print('b')
```

Así como Python 3 provee la función *print()* para la visualización de mensajes y/o valores en consola estándar, también provee la función *input()* que permite realizar con comodidad operaciones de carga de datos desde el teclado mientras el programa se está ejecutando.

La función *input()* permite tomar como parámetro una cadena de caracteres (es decir, escribir esa cadena entre los paréntesis de la función), que acompañará en la consola al *prompt* o *cursor* que indique que se está esperando la carga de datos. Lo que sea que el

[11] Aquí corresponde una aclaración: en la versión Python 2, *print* no es una función sino una *instrucción* del núcleo del lenguaje, y por lo tanto se usa sin los paréntesis: en Python 2, la forma de mostrar el valor de la variable *b* sería *print b*, mientras que en Python 3 (donde *print* es una función) se usan los paréntesis: *print(b)*.

usuario cargue, será retornado por la función *input()* en forma de cadenas de caracteres (con el salto de línea final removido) [6]. Así, una instrucción de la forma:

```
nom = input('Ingrese su nombre: ')
```

provocará que se muestre en consola el mensaje "*Ingrese su nombre:* ", con el cursor del *prompt* parpadeando a continuación y dejando el programa en estado de espera hasta que se el usuario ingrese datos y presione <*Enter*>. Si al ejecutar esta instrucción, por ejemplo, el usuario escribe la palabra "*Ana*" (sin las comillas) y presiona <*Enter*>, la variable *nom* quedará valiendo la cadena "*Ana*" (de nuevo, sin las comillas).

Si se quiere cargar por teclado un número entero o un número en coma flotante, deben usarse funciones predefinidas del lenguaje que hagan la conversión de cadena a número. Para eso, Python provee al menos dos de ellas:

- *int(cadena)*: retorna el número entero representado por la cadena tomada como parámetro (escrita entre los paréntesis).
- *float(cadena)*: retorna el número en coma flotante representado por la cadena tomada como parámetro.

En ambos casos, si la cadena tomada como parámetro no representa un número válido que pueda convertirse, se producirá un error en tiempo de ejecución y el programa se interrumpirá.

Por lo tanto, las siguientes expresiones permiten en Python cargar por teclado un número entero en la variable *n* (la primera instrucción) y un número en coma flotante en la variable *x* (la segunda):

```
n =  int(input('Ingrese un valor entero: '))
x =  float(input('Ingrese un valor en coma flotante: '))
```

Note que en ambas líneas de este ejemplo, se han usado las comillas simples o apóstrofos para encerrar las cadenas de caracteres. Recuerde que en Python es indistinto el uso de las comillas dobles o las simples para hacer esto, siempre que se mantenga consistente (abrir y cerrar con el mismo tipo de comilla). Las mismas dos instrucciones podrían haber sido escritas así:

```
n =  int(input("Ingrese un valor entero: ") )
x =  float(input("Ingrese un valor en coma flotante: "))
```

O incluso combinando así:

```
n =  int(input("Ingrese un valor entero: ") )
x =  float(input('Ingrese un valor en coma flotante: '))
```

Tenga en cuenta que en todo lenguaje existen convenciones de trabajo y consejos de buenas prácticas, y Python no es la excepción. En ese sentido, indicamos que lo más común es que se usen *comillas simples* para delimitar caracteres o cadenas de caracteres, y esa es la convención que hemos tratado de mantener a lo largo de esta ficha de estudio.

Puede verse que la posibilidad de hacer carga por teclado mientras el programa se ejecuta, permite mayor generalidad en el planteo de un programa [4]. A modo de ejemplo, suponga que se quiere desarrollar un pequeño programa en Python para sumar los números contenidos en dos variables. Un primer intento podría ser el siguiente:

```
a = 5
```

```
b = 3
c  =  a  +  b
print('La suma es:  ', c)
```

El pequeño programa anterior funciona… pero podemos darnos cuenta rápidamente que así planteado es muy poco útil, pues indefectiblemente el resultado mostrado en pantalla será 8... El programa tiene muy poca *flexibilidad* debido a que el valor inicial de la variable *a* es siempre 5 y el de *b* es siempre 3. Lo ideal sería que *mientras el programa se ejecuta*, pueda *pedir* que el usuario ingrese por teclado un valor para la variable *a*, luego otro para *b*, y que luego se haga la suma (en forma similar a como permite hacerlo una calculadora...) Y eso es justamente lo que permite hacer la función *input()*. El siguiente esquema muestra la forma correcta de hacerlo:

```
a = int(input('Primer valor: '))
b = int(input('Segundo valor: '))
c  =  a  +  b
print('La suma es:  ', c)
```

Observar que de esta forma, cada vez que se ejecuta el programa se puede cargar un valor distinto en cada variable, y obtener diferentes resultados sin tener que modificar y volver a ejecutar el programa.

6.] Operadores aritméticos en Python.

Hemos visto que siempre se puede asignar en una variable el *resultado de una expresión*. Una *expresión* es una fórmula en la cual se usan *operadores* (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de *operandos* de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice *expresión aritmética*. El siguiente es un ejemplo de una *expresión aritmética* en Python:

```
num1 = 10
num2 = 7
suma = num1 + num2
```

En la última línea se está asignando en la variable *suma* el resultado de la expresión *num1 + num2* y obviamente la variable *suma* quedará valiendo 17. Note que en una asignación *primero* se evalúa cualquier expresión que se encuentre a la derecha del signo =, y *luego* se asigna el resultado obtenido en la variable que esté a la izquierda del signo =. La siguiente tabla muestra los *principales operadores aritméticos* del lenguaje Python (volveremos más adelante con un estudio más detallado sobre la aplicación de estos operadores) [6]:

Figura 12: Tabla de operadores aritméticos básicos en Python.

Operador	Significado	Ejemplo de uso
+	suma	a = b + c
-	resta	a = b – c
*	producto	a = b * c
/	división de coma flotante	a = b / c
//	división entera	a = b // c
%	resto de una división	a = b % c
**	potencia	a = b ** c

Los operadores de *suma*, *resta* y *producto* funcionan tal como se esperaría, sin consideraciones especiales. A modo de ejemplo, la siguiente secuencia de instrucciones calcula y muestra la suma, la resta y el producto entre los valores de dos variables *a* y *b*:

```
a = 5
b = 2

suma = a + b
resta = a - b
producto = a * b

print('Suma: ', suma)
print('Resta: ', resta)
print('Producto: ', producto)
```

Por otra parte, Python provee dos operadores diferentes para el cálculo de la *división*: el primero (/) calcula el cociente entre dos números, obteniendo siempre un resultado de coma flotante, sin truncamiento de decimales (lo que se conoce como *división real*). El segundo (//) calcula el *cociente entero* entre dos números, lo que significa que los decimales se truncan al hacer el cálculo, y siempre se obtiene sólo la parte entera de la división. El siguiente modelo aplica ambos operadores:

```
a = 7
b = 3

cr = a / b
ce = a // b

# la siguiente muestra el valor 2.3333333333333335
print('Division real: ', cr)

# la siguiente muestra el valor 2
print('Division entera: ', ce)
```

Un operador muy útil es el que permite calcular el *resto* o *módulo* de una división (%). En general, aplica sobre operandos de tipo entero (aunque puede aplicarse también sobre valores *float*). El uso de operador resto permite valerse sin problemas de diversas características de la llamada *aritmética modular* (que es justamente la parte de la aritmética que estudia las propiedades de los conjuntos de números que tienen el mismo resto al dividir por el mismo número). Por ahora, sólo nos interesa mostrar algunos ejemplos de aplicación:

```
a = 7
b = 3

r = a % b

# la siguiente muestra el valor 1
print('Division real: ', r)
```

En el esquema anterior, la variable *r* queda valiendo 1, ya que 1 es el resto de dividir 7 por 3 (el cociente entero es 2, quedando un resto de 1). En general, el operador *resto* es muy valioso en casos en que se quiere aplicar conceptos de *divisibilidad*: como la expresión *r = x % y* calcula el resto de dividir en forma entera a *x* por *y*, y asigna ese resto en la variable *r*, entonces si ese resto *r* fuese 0, implicaría que *x* es divisible por *y*, o lo que es lo mismo, que *x* es múltiplo de *y*.

Note que para usar y aplicar el operador *resto* no es necesario usar antes el operador *división*. Ambos operadores se pueden aplicar sin tener que usar el otro. Si usted sólo desea calcular el resto de una división, simplemente use el operador *%* para obtenerlo y no use el operador división.

El último de los operadores aritméticos básicos de Python es el operador potencia (**) que permite en forma muy simple calcular el valor de x^y, para cualquier valor (entero o flotante) de *x* e *y*. El siguiente esquema muestra la forma básica de usarlo:

```
a = 2
b = 3

p = a ** b

# la siguiente muestra el valor 8 (o sea, 2 al cubo)
print('Potencia: ', p)
```

Veremos más adelante, a lo largo del desarrollo del curso, aplicaciones más relevantes de todos los temas presentados en esta ficha de introducción.

7.] Uso del *shell* de Python.

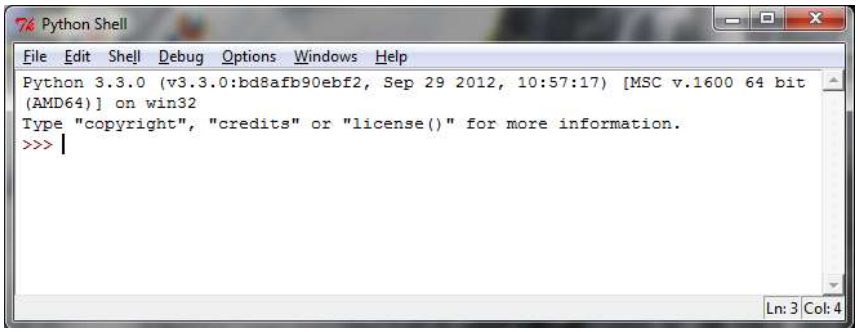
Si se ha descargado e instalado el *SDK* de *Python*, entonces (y sólo con eso) se cuenta ya con la posibilidad de escribir, testear y ejecutar en forma básica sus primeros programas en Python a través del *editor del shell de comandos* de Python. Saber cómo hacer esto será efectivamente muy útil para comprender mejor las secciones que siguen, por lo cual se hace aquí una pequeña introducción (aunque más pronto que tarde pasaremos a usar un *IDE* (*Integrated Development Environment* o *Entorno Integrado de Desarrollo*: un programa que permite editar, depurar, ejecutar y testear programas, mucho más sofisticado que el simple *editor del shell* que viene con el SDK).

En general, este *editor del shell* forma parte de una sencilla aplicación designada como el *Python IDLE GUI* (Python Integrated DeveLopment Environment – Graphic User Interface) y se accede desde el grupo de programas Python haciendo click sobre el ícono correspondiente (y es útil también tener esta aplicación a mano a través de un acceso directo en su escritorio):

Figura 13: Ícono de acceso al Python IDLE GUI (o Shell de Python).

Al abrir el *IDLE GUI*, se muestra una simple ventana de edición de textos, pero integrada con el intérprete de Python, como la que se ve a continuación:

Figura 14: El shell de Python.



Esa ventana muestra el símbolo ">>>" que es el *prompt* (o *símbolo del sistema*) del shell de Python, indicando que el citado shell está a la espera de recibir instrucciones. Como Python es *interpretado*, en este momento se puede escribir cualquier instrucción de Python y presionar *<Enter>* para ejecutarla de inmediato (si está correctamente escrita) [7].

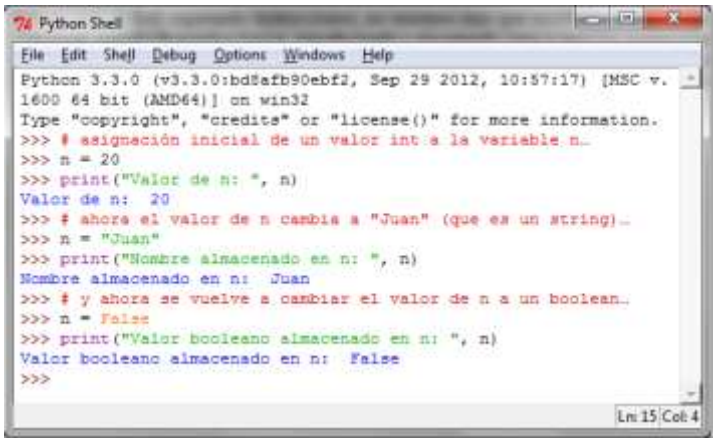
Como veremos oportunamente, en Python no es necesario escribir un programa sujeto a una estructura declarativa rígida: en principio, y sobre todo si se está trabajando directamente con el editor del shell, es suficiente con escribir una instrucción debajo de la otra y pedir la ejecución de cada una. En ese sentido, un lote de instrucciones Python correctamente escritas y listas para ser ejecutadas se conoce como un *script* Python (dejando la palabra "programa" para describir a una secuencia de instrucciones que responde a una estructura más elaborada… cosa que veremos).

Y bien: con el shell esperando instrucciones, no tenemos más que escribirlas y ejecutarlas. Puede comenzar, a modo de práctica básica y para asegurarse que todo está en orden, introduciendo y ejecutando (una a una) las instrucciones de la siguiente secuencia, que explicaremos más adelante. Solo asegúrese de no copiar y pegar el bloque completo en el editor del shell, ya que el intérprete espera ejecutar una por una las instrucciones (de lo contrario, obtendrá un mensaje de error):

```
n = 20
print('Valor de n: ', n)

n = 'Juan'
print('Nombre almacenado en n: ', n)

n = False
print('Valor booleano almacenado en n: ', n)
```

Si todo anduvo bien, al terminar de editar y ejecutar estas instrucciones la ventana del shell debería mostrar algo como lo se muestra en la *Figura 15* (no verá las líneas escritas en rojo).

Figura 15: Ejemplo de uso del shell de Python.



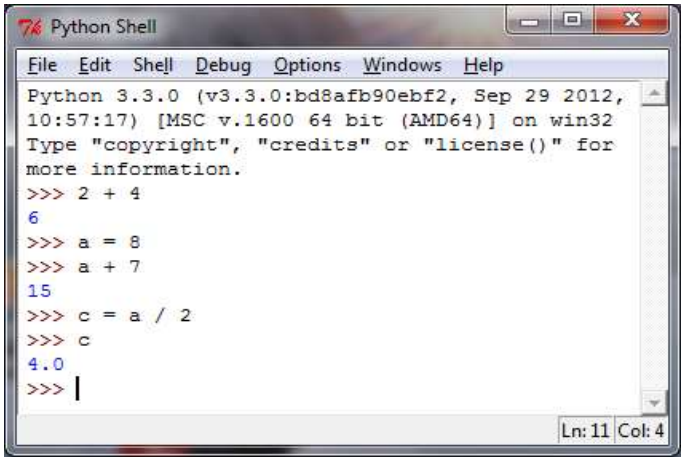
Es interesante notar que el intérprete del shell ejecuta cualquier expresión Python válida, por lo cual se puede usar a modo de "calculadora" introduciendo expresiones (aritméticas, lógicas, etc.) directas, como se muestra en la *Figura 16* de la *página 24*.

Observe además que si está trabajando con el editor del shell, no necesita usar la función *print()* para mostrar el valor de una variable o de una expresión: basta con escribir esa variable o expresión en el prompt, y simplemente el intérprete la reemplazará por su valor.

Obviamente, un script o cualquier conjunto de código fuente en Python, se puede almacenar en un archivo externo para poder recuperarlo, re-editarlo y volver a ejecutarlo cuando sea

necesario. En este sentido, la convención del lenguaje es que un archivo que contenga código fuente Python puede tener *cualquier nombre* que el programador desee, seguido de la extensión *.py.* La opción *File* del menú superior del shell de Python contiene a su vez las opciones que permiten grabar o recuperar un script. Dejamos para el estudiante la tarea de explorar el resto las opciones de la barra de menú del IDLE, y las aplicaciones que pudieran tener.

Figura 16: Uso del shell de Python a modo de calculadora.



Para cerrar esta sección, digamos que si un programa o script Python está ya desarrollado y almacenado en un archivo externo, entonces se puede ejecutar ese programa *directamente desde la línea de órdenes del sistema operativo huésped*. Los detalles de cómo hacer esto están resumidos en esta misma sección de *Temas Avanzados*, sección *b.) Ejecución por línea de órdenes en Python*, en *página 24*. Dejamos para el alumno el estudio y aplicación de estos temas.

8.] Ejecución por línea de órdenes en Python.

Cuando se escribe un programa o script Python, se grabará en un archivo que puede tener el nombre que prefiera el programador, pero con extensión *.py* (que como ya indicamos en la Ficha, es la extensión típica de un archivo que contiene código fuente en Python). Un archivo fuente normalmente se crea a través del editor de textos del *IDE* que esté usando el programador (que en nuestro caso será *PyCharm*) y el mismo *IDE* se encargará de la extensión del archivo. Pero si no dispone de ningún *IDE*, o está practicando la forma de trabajo directa con el *SDK* (como ahora…) obviamente puede crear el fuente *con cualquier editor de textos* (por ejemplo, con el *Notepad*, con *Wordpad*, o con el que prefiera) siempre y cuando el programador recuerde colocar la extensión *.py* al archivo que cree.

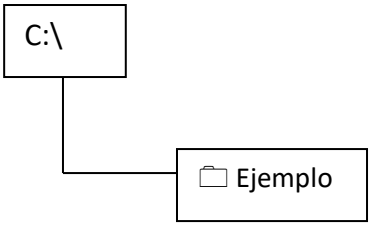
Una vez creado el fuente, el programador querrá verificar si tiene errores de sintaxis, depurar esos errores y finalmente ejecutarlo. Ese proceso en Python se lleva a cabo mediante el programa que en la plataforma Windows se llama *python.exe* del *SDK* de Python. Este programa es el *intérprete de Python*: toma un código fuente e intenta ejecutar una a una las instrucciones del mismo, a menos que alguna contenga un error de sintaxis (en cuyo caso, interrumpe la ejecución y avisa del error con un mensaje en la consola de salida). En caso de lanzarse un error de compilación, el programador debe volver al código fuente con el editor de textos, corregir el error, volver a grabar el fuente y finalmente intentar ejecutar otra vez el programa.

Aun cuando en la práctica un programador usará un *IDE* sofisticado para facilitar su trabajo, es importante que ese programador entienda a la perfección el proceso de interpretación y ejecución,

ya sea para dominar todos los aspectos técnicos y poder comprender los problemas que podrían surgir, como para poder salir delante de todos modos si le toca trabajar en un contexto donde no tenga acceso a un *IDE*. En este último caso, deberá poder hacer todo el proceso usando *directamente* los programas del SDK.

Mostraremos entonces aquí el proceso completo que debe llevarse a cabo para ejecutar un programa o script Python desde la *línea de órdenes*. Los pasos generales son los siguientes:

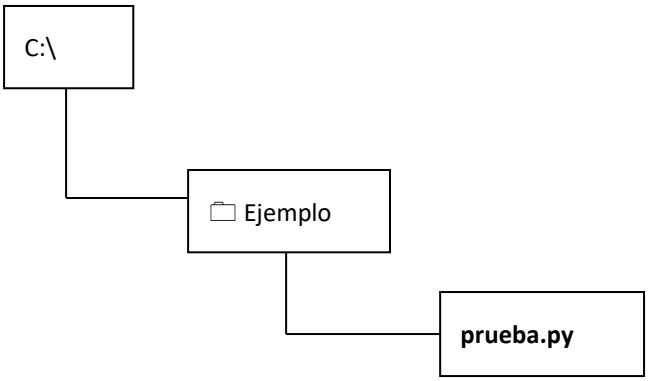
i. Es conveniente, a los efectos de simplificar el proceso, que se cree primero una carpeta o directorio para contener todos los archivos asociados al programa que se está desarrollando. Esa carpeta en general se conoce como *carpeta del proyecto* y un *IDE* la crea en forma automática. En este caso, para mayor simplicidad, se puede pensar en crear esa carpeta a nivel del *directorio raíz* del disco local principal. En este ejemplo, crearemos una carpeta con el nombre *Ejemplo* en el directorio raíz del disco C, quedando así:



ii. Ahora debe crear y editar un archivo fuente Python sencillo, usando cualquier editor de textos (el Block de Notas, el Wordpad, o cualquiera que tenga a mano). En este caso, plantearemos el típico ejemplo de un script sencillísimo que sólo muestra en pantalla el mensaje "Hola Mundo". El código fuente podría verse así (asumiendo que el archivo se llamará *prueba.py*)

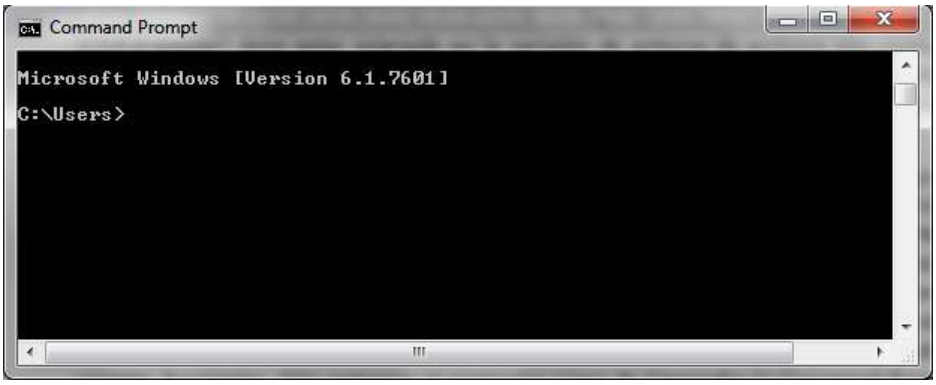
```
# Ejemplo de script muy simple en Python
print('Hola Mundo!!!...')
```

iii. Cuando grabe el fuente que acaba de editar, recuerde hacerlo dentro de la carpeta *C:\Ejemplo* que ha creado antes (en el punto i.), y también recuerde grabar el fuente con el nombre *prueba.py*. Si está usando el Block de Notas, recuerde poner el nombre del archivo entre comillas cuando grabe, para evitar que se agregue la extensión ".txt" a su archivo. En nuestro caso, así debería verse todo en el explorador de Windows al terminar este paso:



iv. Salga ahora a la ventana de la línea de órdenes del sistema operativo. Si está trabajando con Windows 10, una forma de hacerlo es mediante el botón *Inicio* del escritorio de Windows. Busque allí el ítem *Windows System* y elija entonces el programa *Línea de Comandos* (o

Command Prompt). Con eso se abrirá una pequeña ventana de fondo negro: esa es la ventana de *línea de órdenes* de Windows, y su aspecto es similar al de la siguiente captura de pantalla:



v. Es posible que los nombres de los directorios o carpetas mostrados no coincidan necesariamente con los que verá usted. En el caso del ejemplo, la última línea que se muestra en la ventana (que se designa con el nombre general de *prompt*) tiene este aspecto:

```
C:\Users>_
```

vi. El prompt le está indicando en qué directorio o carpeta está ubicado el sistema de línea de órdenes de Windows en ese momento. En el estado indicado antes, se podrá ejecutar cualquier archivo con extensión *.exe* que se encuentre dentro de la carpeta *Users* del disco *C*, pero no se podrán ejecutar archivos *.exe* de otras carpetas, a menos que se ajuste el valor de la variable de entorno llamada *PATH*, de forma que contenga los nombres de los directorios que contienen a los *.exe* que se desea ejecutar. Como nos interesa ejecutar el programa *python.exe* (o sea, el intérprete del SDK), debemos probar si ese programa está *visible* para la línea de órdenes. Simplemente pruebe a escribir lo siguiente en la línea del prompt (sólo la palabra *python* que se remarca en azul en el ejemplo: el resto es el contenido del prompt que ya está visible):

```
C:\Users>python
```

vii. Si al presionar *<Enter>* aparece un mensaje de error similar a este:

```
"python" is not recognized as an internal or external command
```

entonces el intérprete no es *visible* para la línea de órdenes (recuerde que el programa *python.exe* se encuentra en el directorio del SDK, posiblemente en la carpeta llamada *C:\Program Files\Python 3.8* o similar. Verifique el nombre del directorio del SDK en su equipo). Debemos ajustar el valor de la variable *PATH* del sistema operativo, lo cual puede hacerse así (recuerde: los nombres de los directorios pueden ser diferentes... usted debe usar la ruta de la carpeta que contenga al SDK de Python EN ***SU*** COMPUTADORA PERSONAL):

```
C:\Users>set PATH=C:\Program Files\Python 3.8
```

viii. Lo anterior debe dejar visible el intérprete para la línea de órdenes. Si ahora vuelve a probar con:

```
C:\Users>python
```

debería ver algunas líneas de ayuda para usar el intérprete, y no un mensaje de error. El siguiente paso es cambiar el directorio activo para entrar al *directorio del proyecto donde*

está el archivo prueba.py que se quiere interpretar y ejecutar. En nuestro caso puede hacerlo con el comando *cd* (por "*change directory*") en forma similar a esta secuencia:

```
C:\Users>cd \       [al presionar <Enter> cambia al directorio raíz]
C:\>cd Ejemplo      [al presionar <Enter> cambia al directorio Ejemplo]
C:\Ejemplo>_
```

ix. Con el *prompt* apuntando al directorio *C:\Ejemplo*, puede ya probar a ejecutar con el siguiente comando (sólo escriba lo que figura en azul… el resto es el *prompt*):

```
C:\Ejemplo>python prueba.py
```

x. El proceso puede demorar un par de segundos. Si aparecen mensajes de error, entonces deberá volver al editor de textos, abrir nuevamente el archivo fuente, corregir el error que pudiera tener, grabar los cambios, y luego volver a ejecutar hasta que logre que el programa se ejecute sin errores. Si todo anduvo bien y sin errores, el programa será ejecutado y en este caso debería aparecer a renglón seguido el mensaje *Hola Mundo!!!...* como se ve a continuación (el *prompt* volverá a aparecer inmediatamente debajo del mensaje):

```
C:\Users>python prueba.py
Hola Mundo!!!...
C:\Users>_
```

Con esto el proceso se completa. Sugerimos encarecidamente al estudiante que intente aplicar estos pasos hasta lograr hacerlo en forma correcta, y asegurarse de poder entenderlos. Intente cometer algún error a propósito en el script contenido en el archivo *prueba.py* para comenzar a familiarizarse con el proceso de corrección de errores y el uso del intérprete. Y no se desespere: tiene que saber manejarse con el SDK en forma directa, pero cuando haya aprendido a hacerlo pasará a usar un IDE profesional (como *PyCharm*) y muchos de estos detalles quedarán automatizados.

Bibliografía

[1] G. Beekman, Introducción a la Informática, Madrid: Pearson Education, 2006.

[2] C. Stephenson and J. N. Petterson Hume, Introduction to Programming in Java, Toronto: Holt Software Associates Inc., 2000.

[3] R. Kurzweil, La Era de las Máquinas Espitiruales, Buenos Aires: Editorial Planeta Argentina, 2000.

[4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[5] S. Lipschutz, Matemáticas para Computación, México: McGraw-Hill / Interamericana, 1993.

[6] Python Software Foundation, "Python Documentation," 2015. [Online]. Available: https://docs.python.org/3/. [Accessed 24 February 2015].

[7] M. Pilgrim, Dive Into Python - Python from novice to pro, Nueva York: Apress, 2004.