

# Apuntes Computación

Rodrigo H. Avaria

# Capítulo 1 Introducción

Si mi única herramienta fuera un  
martillo, todos mis problemas parecían  
clavos

*Refrán popular*

**Hello world!** Vivimos en una sociedad con tradiciones y ritos, el rito manda que todo curso que implique código, comience con *Hello world!*.

Saludos y bienvenidas(personas), seguido del saludo de rigor, diré algo impopular: **Este curso no es necesario**. Si está dispuesta a poner de su parte y al trabajo disciplinado, este curso no es necesario, ya que usted puede aprender los contenidos de manera autodidacta, a su ritmo y según sus necesidades, este texto es un resumen de contenido tanto formal como informal existente en la red y en textos oficiales de programación que me he tomado la molestia de traducir y compilar para ustedes, dicho eso

## 1.0.1. Mindset

Al aprender a programar, usted aprenderá a estructurar el pensamiento cuando se trata de enfrentar un problema; así, frente a cada molestia, situación, ineficiencia, debemos pensar: *hay una solución para esto, sólo necesito encontrarla*, esto puede tomar unos pocos segundos, minutos, días, semanas o más, pero si le dedicamos el esfuerzo adecuado, por el tiempo suficiente, la solución aparecerá.

## GIYF\GIYBF (Google is you friend \best friend)

Mi lengua madre es el Español, pero tuve el privilegio de aprender tempranamente Inglés, con lo que mis opciones de búsqueda se multiplicaron, hágase un favor, abrace varios idiomas, aunque sea de forma técnica; de igual forma aprenda varios lenguajes de programación, cada uno tiene sus fortalezas y sus debilidades. Salvo casos muy particulares, no somos los primeros ni los últimos en enfrentarnos a los problemas y preguntas que nos ofrece la programación, así por cada error de código, por cada *"sería ideal si mis resultados tuvieran esta*

forma”, hay muchos desarrolladores en el mundo que se encontraron o se encontrarán en situaciones parecidas, aprender a usar google de forma eficiente **y** en varios idiomas le puede ahorrar tiempo y energía que podrá usar luego, en cosas que le sean más gratificantes. Tome en consideración que si bien es cierto todos podemos usar un motor de búsqueda, no todos creen que pueden hacerlo bien y no todos saben hacerlo bien. Usted **debe** aprender a buscar de manera **efectiva y eficiente**, así como a sentirse en comodidad haciéndolo, esta es una verdad fundamental, porque cuando estén trabajando (principalmente de noche, programadores be like trabajar de noche) y nada funciona, lo único que evita que renunciemos, es saber que existe una solución y que, si le dedicamos el esfuerzo adecuado, por el tiempo suficiente, la solución aparecerá en la pantalla ( por creación propia o por adaptación de fragmentos de código ajeno). Para esto usted debería ser capaz de segmentar su trabajo en tareas más pequeñas fácilmente realizables o de corta duración (*efecto tiktok* ) cuya consecución le dará la motivación para la llevar a cabo la siguiente.

---

### Aprender a solucionar problemas

Al final del día, el código es sólo una herramienta para solucionar problemas. La parte más dura es saber cómo resolver problemas. Es por esto que un aspecto fundamental de aprender a programar es mantenerse constantemente aprendiendo. Lo que usted aprenda en este curso no será ni por cerca el total de habilidades que usted necesitará en el futuro, principalmente porque:

- ❁ El material contenido en este curso y en las clases que lo acompañan están escrito en pasado, no en futuro.
- ❁ Algunos de los problemas a los que usted se enfrentará aun no se inventan.
- ❁ Algunas de las herramientas que usted usará para enfrentarse a los problemas de antes mencionados, aun no se inventan.
- ❁ La cantidad de cosas que ignoramos es inconmensurable, pero podemos aprender muchas de ellas con el esfuerzo adecuado, paciencia y tiempo

## 1.0.2. Aprender a escribir código

Aprenda varios lenguajes, **pero** recuerde siempre que *"quién mucho abarca, poco aprieta"*, aprenda al menos uno en profundidad, sepa los mensajes de error más comunes, hágase un repositorio de lugares fidedignos a los cuales recurrir para buscar ayuda. Haga de este lenguaje su caballito de batalla, su carta de presentación. En este curso veremos someramente 2, Python y R.

## El secreto para aprender a escribir código es

El secreto para aprender a escribir código es escribir código, *i.e* mientras más código escriba, mejor código escribiré, si constantemente se asocia a alguien que le de las respuestas ¿aprendió realmente a usar las herramientas que brinda el curso?. Para hacer de esto algo personal, comience un proyecto personal. Meterse en problemas le llevará a situaciones fuera de las tareas y evaluaciones de este curso, le dará experiencia que luego puede transferir. Para exponerse a otros habientes visite competencias\hackatons, esto le dará la oportunidad de desarrollar un proyecto y medir sus habilidades al tiempo que le permitirá, ganar experiencia, conocer gente afín, hacer *networking* y exponerse a la evaluación de clientes \empleadores

## Capítulo 2 Maquinas de Turing

Una computadora merecería ser  
llamada inteligente si pudiera engañar a  
un humano haciéndole creer que es  
humana

*Alan Turing*

### El computador es una máquina estúpida

El computador es una máquina elegante, inteligentemente pensada, pero estúpida que nunca se equivoca, ¿por qué? porque sólo hace lo que le mandan a hacer; por esto, recuerde las siguientes reglas:

1. El computador nunca se equivoca
2. Si el computador se equivoca, ver regla 1.

Esto debería ser un mensaje de humildad, si el código no entrega la solución que necesitamos es porque no le hemos dado las instrucciones correctas, deberá aprender a expresarse de manera sucinta y adecuada, de tal forma que el computador le entienda y le obedezca.

Más allá de la broma (que no es broma), un computador no sabe de infinito ni el axioma de completitud (a.k.a. hipótesis de continuidad) un computador es una máquina que sólo conoce un conjunto en el que sólo sabe realizar una operación (lo que se conoce como el grupo  $(\mathbb{Z}_2, +)$ ); así, el computador sólo sabe sumar números discretos (finitos). Todo lo que un computador hace, ya sea Whatsapp, ChatGTP, Dali, blockchain, o un app en el celular, es producto una abstracción llamada “máquinas de Turing ”

Las máquinas de Turing (MT) son un modelo teórico para un computador, o mejor dicho y en orden de temporal, un computador es una realización material de un abstracto teórico llamado máquinas de Turing; estas fueron propuestas por el matemático, lógico, criptoanalista filósofo, biólogo teórico y héroe de guerra británico Alan Turing en 1936 como una forma de formalizar y estudiar el concepto de algoritmo (ver definición 4.2.2).

Como lo muestra la figura 2.1, una MT consiste en una cinta infinita dividida en celdas,

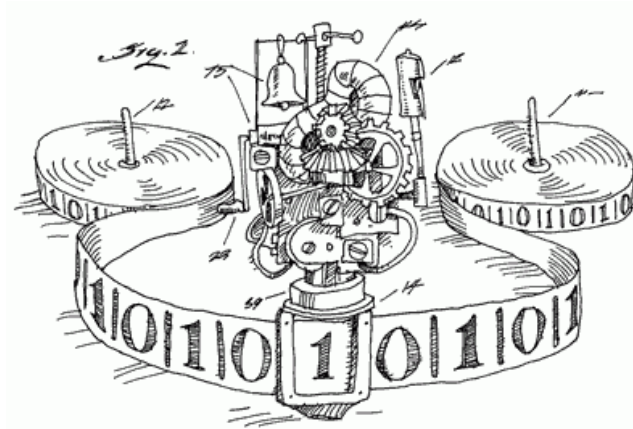


Figura 2.1. Representación caricaturizada de una maquina de Turing, tomada de ai-notes-mx

donde cada celda puede contener un símbolo de un alfabeto finito y un única cabezal de lectura/escritura que puede operar en una de las celdas de la cinta a la vez. La máquina tiene un conjunto finito de estados, y en cada estado puede realizar una de las siguientes operaciones: mover el cabezal hacia la izquierda o hacia la derecha en la cinta, leer el símbolo en la celda actual, escribir un nuevo símbolo en la celda actual, o cambiar al siguiente estado. Cada MT también tiene un estado de aceptación y un estado de rechazo[1].

La idea detrás de las máquinas de Turing es que cualquier problema que se pueda resolver mediante un algoritmo se puede resolver con una máquina de Turing. Es decir, cualquier función computable se puede implementar mediante una máquina de Turing. Con esta premisa las MT han sido utilizadas para demostrar la no computabilidad de ciertos problemas, así como para desarrollar y analizar algoritmos y lenguajes de programación.

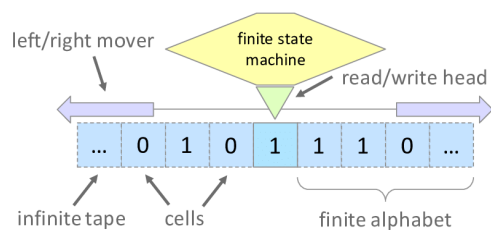


Figura 2.2. Representación y materialización de Maquinas de Turing, tomadas de researchgate.net y medium.com respectivamente

Ya que las máquinas de Turing una entidad son abstractas, no pueden ser implementadas directamente en una computadora física ¿Cómo pasamos entonces, desde el mundo de las ideas a un *smart phone*? A partir de las ideas de Turing, surgieron diferentes arquitecturas

de computadoras <sup>(1)</sup> que han evolucionado a lo largo del tiempo para satisfacer las necesidades de procesamiento de información de la sociedad llegando a lograr las máquinas que conocemos hoy. Así, hay varias formas de materializar un computador.

Aunque las arquitecturas de computadores modernos no son una realización directa de una MT, su diseño, funcionamiento y capacidad para realizar operaciones computacionales se inspiran en gran medida en la teoría de MT y en otros modelos teóricos de la computación. Ambas utilizan la manipulación de símbolos para realizar operaciones y se basan en la idea de tener un conjunto finito de operaciones básicas que se pueden utilizar para construir operaciones más complejas.

1. **Arquitectura Von Neumann:** esta es la arquitectura más común en las computadoras modernas: se basa en el concepto de tener una sola memoria para almacenar datos y programas, junto a un procesador que ejecuta instrucciones de forma secuencial. La arquitectura Von Neumann también incluye unidades de entrada/salida (E/S) para permitir que la computadora interactúe con el mundo exterior. La mayoría de los computadores personales modernos utilizan esta arquitectura. Por ejemplo, una CPU Intel Core i5 usada por los computadores de escritorio, utiliza que está diseñada según la arquitectura de von Neumann.
2. **Arquitectura Harvard:** esta arquitectura se diferencia de la arquitectura de Von Neumann en que tiene dos memorias separadas: una para almacenar datos y otra para almacenar programas. Esto permite que la computadora pueda acceder a la memoria de programa y a la memoria de datos simultáneamente, lo que puede mejorar el rendimiento. Esta arquitectura se utiliza a menudo en dispositivos incrustados, como microcontroladores. Un ejemplo de arquitectura de Harvard es la placa de desarrollo AVR de Atmel, que se utiliza comúnmente en proyectos de electrónica.
3. **Arquitectura RISC:** RISC (Reduced Instruction Set Computing) es una arquitectura que se enfoca en tener un conjunto de instrucciones reducido, lo que hace que la CPU sea más

---

<sup>(1)</sup> En el contexto de las ciencias de la computación e informática, la arquitectura de una computadora se refiere a la estructura fundamental, cómo se organizan y comunican entre sí los componentes de hardware y software de un computador para realizar tareas y procesar datos. La arquitectura de una computadora incluye elementos como el procesador, la memoria, el almacenamiento, los dispositivos de entrada y salida, el sistema operativo y los programas de aplicación. También puede incluir aspectos como la topología de red, la seguridad y la escalabilidad del sistema.

rápida y eficiente. La arquitectura RISC también utiliza un conjunto de registros para almacenar datos y una estructura de pipeline para mejorar el rendimiento. Muchos procesadores modernos utilizan la arquitectura RISC, como los procesadores ARM que se utilizan en dispositivos móviles y en sistemas integrados. Por ejemplo, el procesador ARM Cortex-A7 utilizado en la Raspberry Pi 2, (una placa de desarrollo de computadora de placa única).

4. Arquitectura CISC: CISC (Complex Instruction Set Computing) es una arquitectura que se enfoca en tener un conjunto de instrucciones complejas, lo que puede hacer que la programación sea más fácil. La arquitectura CISC también utiliza un conjunto de registros para almacenar datos y puede realizar varias operaciones complejas en una sola instrucción. Esta arquitectura se utiliza en muchas computadoras personales y servidores, como los procesadores Intel x86. Un ejemplo de una máquina que utiliza la arquitectura CISC es el procesador Intel Core i7.

Arquitectura paralela: las arquitecturas paralelas se enfocan en tener múltiples procesadores que trabajan juntos para ejecutar tareas. Esto puede mejorar significativamente el rendimiento de la computadora para tareas que pueden ser paralelizadas, como el procesamiento de imágenes y el análisis de datos. Las supercomputadoras y los clústeres de computadoras son ejemplos de sistemas que utilizan arquitecturas paralelas. Por ejemplo, el supercomputador Fugaku, utiliza una arquitectura paralela que se basa en procesadores ARM y que es capaz de realizar 442 cuatrillones de operaciones por segundo.

Es importante tener en cuenta que muchos sistemas de computadoras modernos utilizan una combinación de varias arquitecturas, ya que cada una de estas arquitecturas tiene sus propias características únicas y se adapta mejor a ciertos tipos de tareas; así, la elección de una arquitectura específica depende las necesidades y requisitos específicos de cada proyecto y de otros factores, como el costo y la eficiencia energética.

### Ejercicio 2.1

Estos no son las únicas arquitecturas disponibles, queda como ejercicio \tarea averiguar sobre otras arquitecturas como la computación neuromórfica y la computación cuántica.



---

### 2.0.1. Material complementario

1. Universidad de Cambridge: Turing Machines
2. Scholarpedia: Turing Machines
3. Up and Atom (Video) : How Computer Science Was Created By Accident
4. La máquina de Turing Explicada: Charla de Javier García en el Aula141 sobre la máquina de Turing. En esta charla se explica desde cero el funcionamiento con un ejemplo concreto.

---

### ? Referencias

- [1] Cristian Fernando Vilca Gutierrez. «Revista de Investigación Estudiantil Illuminate». En: *Revista de Investigación Estudiantil Illuminate* 18 (1 nov. de 2018), págs. 31-40. ISSN: 2415-2323. URL: [http://www.revistasbolivianas.ciencia.bo/scielo.php?script=sci\\_arttext&pid=&lng=pt&nrm=iso&tlng=%20http://www.revistasbolivianas.ciencia.bo/scielo.php?script=sci\\_abstract&pid=&lng=pt&nrm=iso&tlng=.](http://www.revistasbolivianas.ciencia.bo/scielo.php?script=sci_arttext&pid=&lng=pt&nrm=iso&tlng=%20http://www.revistasbolivianas.ciencia.bo/scielo.php?script=sci_abstract&pid=&lng=pt&nrm=iso&tlng=)

## Capítulo 3 Git: persona desagradable

"Soy un bastardo egoísta, y los nombres de todos mis proyectos se refieren a mi. Primero 'Linux', ahora 'git'

*Linus Torvalds*

La página del manual describe a Git como "el estúpido rastreador de contenido", mientras el archivo read-me del código fuente va más allá: *"git" puede significar cualquier cosa, dependiendo de tu estado de ánimo.*

- ✿ Combinación aleatoria de tres letras que se puede pronunciar y que ningún comando común de UNIX utiliza realmente. El hecho de que sea una mala pronunciación de "get" puede o no ser relevante.
- ✿ Estúpido. Insignificante y despreciable. Simple. Haga su su elección en un diccionario de argot.
- ✿ "Rastreador de información global" (en inglés *Global Information Tracker*): está de buen humor y realmente funciona para usted. Los ángeles cantan, y de repente una luz llena la habitación.
- ✿ "Maldito camión idiota de m\*erda" (en inglés: "Goddamn idiotic truckload of sh\*t"): cuando se rompe.

El código fuente de Git se refiere al programa como "el administrador de información del infierno".

### 3.1. GIT

GIT es un sistema de control de versiones, distribuido y gratuito, utilizado para rastrear los cambios en archivos de código fuente durante la gestión de proyectos de software. Es una herramienta esencial para cualquier equipo de desarrollo de software, ya que permite a programadores colaborar en el mismo código fuente y llevar un registro de los cambios realizados en el proyecto a lo largo del tiempo [2]. Como señala la documentación oficial de Git [3], este sistema es "rápido, escalable y distribuido" y se utiliza ampliamente en la

industria del software para el desarrollo de proyectos de software de cualquier tamaño.

GIT permite el registro de los cambios realizados en el código fuente del proyecto y su posterior distribución entre los miembros del equipo [2], esto ya que en la gestión de proyectos de software, es común trabajar en paralelo en diferentes partes del código. Por lo tanto, es necesario tener un control exhaustivo de las versiones del código fuente y su integración posterior. En este sentido, el uso de GIT se convierte en una herramienta esencial para asegurar la integridad del código y la coordinación entre los miembros del equipo [6].

El flujo de trabajo básico de Git consiste en tres etapas principales:

- a. la etapa de trabajo, donde quien desarrolla, trabaja en el código fuente de un proyecto.
- b. la etapa de preparación, donde quien desarrolla prepara los cambios que ha realizado en el código para su confirmación
- c. la etapa de confirmación, quien desarrolle, confirma los cambios en el repositorio de Git

Para utilizar GIT, es necesario, primero que todo, instalarlo y luego conocer algunas de las funciones básicas que ofrece.

### 3.1.1. Instalación y características de GIT

Para comenzar a usar Git en su proyecto, primero debe instalarlo. La forma más común de hacer esto es descargar e instalar Git en su sistema operativo desde el sitio web oficial ([4]). Una vez instalado, puede comenzar a usar Git desde la línea de comandos o mediante una interfaz gráfica de usuario. Algunas de las interfaces gráficas más populares incluyen GitHub Desktop ([5]) y Sourcetree ([1]) o el control de versiones integrado en VisualStudio Code.

En primer lugar, se debe inicializar un repositorio GIT en el directorio del proyecto. Esto se realiza con el comando "git init".

```
1 $ git init
```

A veces no es necesario crear un repositorio desde cero, podemos clonar un repositorio existente sobre el cual trabajaremos en grupo

```
1 $ git clone https://github.com/usuario/mi-repo.git
```

Una vez creado (o clonado) el repositorio, se pueden agregar los archivos a ser monitoreados por GIT mediante el comando "git add".

```
1 $ git add <nombre_del_archivo>
```

También se pueden agregar todos los archivos del directorio actual al repositorio utilizando el siguiente comando:

```
1 $ git add .
```

Una vez que se han agregado los archivos al repositorio, se deben confirmar los cambios realizando un "commit" para registrar los cambios. Según el libro de Ryan [8], el comando "commit" tiene la función de crear una instantánea del estado actual del proyecto y agregarla al historial de cambios o versiones del proyecto (esto es una de sus principales características).

```
1 $ git commit -m "Mensaje descriptivo de confirmacion"
```

Cada vez que hacemos un "commit", se realiza un cambio en el código fuente; GIT registra la modificación en los archivos del repositorio, agrega un mensaje descriptivo para que los otros miembros del equipo puedan entender qué se ha cambiado en el proyecto, lo que permite la comparación con versiones anteriores.

Una vez que se ha creado o modificado el historial de cambios en el repositorio, es posible compartirlo con otros miembros del equipo. Para ello, se utiliza el comando "git push", que permite enviar los cambios realizados a un repositorio remoto.

```
1 $ git push origin main
```

En este ejemplo, "origin" es el nombre del repositorio remoto y "main" es la rama del repositorio que queremos actualizar.

Por otro lado, si otras personas han hecho cambios en el repositorio remoto mientras trabajábamos localmente, podemos utilizar el comando "git pull" para obtener las últimas actualizaciones del repositorio remoto en nuestro repositorio local. Por ejemplo:

```
1 $ git push origin main
```

Según Chacon y Straub [2], estos comandos permiten la colaboración entre los miembros del equipo, así como la integración eficiente de los cambios realizados.

Es importante destacar que Git almacena los cambios en el repositorio de forma incremental, lo que significa que sólo se almacena la diferencia entre las versiones anteriores y la versión actual de los archivos. Esto reduce el tamaño del repositorio y hace que sea más rápido de descargar y de actualizar.

El uso de Git permite que cada usuario tenga una copia completa del historial de un proyecto, de tal forma que pueda trabajar de manera local, sin necesidad de estar conectado a una red, y sincronizar los cambios realizados posteriormente lo que aumenta la velocidad y la eficiencia del desarrollo colaborativo [2].

Otra función útil de GIT es la posibilidad de hacer "revert" o deshacer un commit anterior. Esto permite corregir errores y problemas en el código sin tener que rehacer todo el trabajo desde cero. Según lo explica el sitio oficial de GIT, "el comando revert deshace un commit específico creando un nuevo commit. Es una operación segura ya que no modifica la historia del repositorio. En vez de eso, crea un nuevo commit con el contenido revertido"[3].

Supongamos que tenemos un repositorio de Git con varios commits, y deseas revertir el commit más reciente porque contiene un error. Primero, necesitas obtener el hash del commit al que deseas revertir, que puedes obtener usando el comando git log. Luego, utiliza el comando git revert seguido del hash del commit:

```
1 $ git log
2 commit 785d83c7cf8b3f9c3bf4b16cfc979c4ec3b1a4a1 (HEAD -> main)
3 Author: Juan Perez <juan.perez@example.com>
4 Date: Mon Feb 28 10:00:00 2023 -0500
5
6 Add new feature
7
8 commit 9ac72e13f7b5faec99eb68f82a8b899cc0d69a95
9 Author: Juan Perez <juan.perez@example.com>
10 Date: Sat Feb 25 15:00:00 2023 -0500
11
```

```
$ git revert -m "Revert 'Add new feature' commit" 785d83c7cf8b3f9c3bf4b16...(1)
```

Esto creará un nuevo commit que deshace los cambios introducidos por el commit especificado. En este caso hipotético hemos agregado un mensaje al commit revertido utilizando la opción **-m**. Tenga en cuenta que la reversión no elimina el commit original del historial, sino que crea un nuevo commit que deshace los cambios originales. Por lo tanto, es una operación segura y no destructiva que no altera el historial existente del repositorio.

Otro concepto importante en Git es la rama (branch). Una rama es simplemente una versión paralela del repositorio. Esto permite a programadores trabajar en diferentes características o soluciones, en distintas versiones del mismo código sin interferir en el trabajo de los demás, y posteriormente fusionarlas (merged) en una única versión, cuando se considera que la rama está lista para ser integrada en la versión principal.

Según Chacon y Straub, "las ramas son una herramienta fundamental para colaborar en un proyecto de código abierto o incluso en un proyecto en solitario. Permite experimentar sin afectar la rama principal del proyecto, crear versiones de corrección de errores, probar nuevas características y más" [2]. Para Loeliger y McCullough [6], las ramas permiten la gestión eficiente de proyectos complejos, al tiempo que facilitan la resolución de conflictos al momento de integrar los cambios.

Para crear una nueva rama, se debe ejecutar el siguiente comando:

```
$ git branch <nombre_de_la_rama>
```

Para listar las ramas en Git desde la terminal, también podemos usar el comando `git branch`. Este comando lista todas las ramas en el repositorio local y resalta la rama actual con un asterisco. Para ver las ramas en el repositorio remoto, puedes usar el comando `git branch -r`. Y para ver todas las ramas (locales y remotas), puedes usar el comando `git branch -a`.

Para cambiar a una rama existente, se debe ejecutar el siguiente comando:

---

(1) Acá, los puntos suspensivos se han usado porque el ancho de la página no da para el identificador de el commit

```
1 $ git checkout <nombre_de_la_rama>
```

Cada persona puede trabajar en su propia rama, y posteriormente hacer una petición de "pull request" o solicitud de integración de su trabajo a la rama principal del proyecto. Según lo explica Pecinovsky, "la petición de pull request sirve como un registro de conversación alrededor del trabajo que se está haciendo. Una vez que se ha hecho la petición, los colaboradores pueden revisar el código, discutir cambios y aprobar o rechazar la solicitud" [7].

La capacidad de fusionar diferentes ramas es otra de las principales funciones de Git. Como señala el libro "Pro Git" de Scott Chacon y Ben Straub [2], la fusión "es el proceso de combinar dos o más ramas de desarrollo en una sola línea de desarrollo". Esto es útil porque permite a desarrolladores integrar diferentes soluciones y características en el código principal. Por ejemplo, si un equipo ha creado una rama para trabajar en una nueva característica, pueden fusionar esa rama con la rama principal una vez que la característica está completa.

En conclusión, GIT es una herramienta fundamental para el control de versiones de código en proyectos de programación. Permite la creación de ramas, la reversión de cambios, y la colaboración eficiente en equipos de trabajo. Su uso es ampliamente difundido en la comunidad de programadores, y es una habilidad esencial para cualquier persona que trabaje en desarrollo grupal de código. Conocer las funciones básicas de GIT es esencial para su correcto uso y para garantizar la integridad del código y la coordinación entre los miembros del equipo.

### 3.2. Actividad

Clonando el repositorio ComputaciónIDEUV y usando la terminal, podemos crear una nueva rama llamada "<rama-NombreEstudiante>" con el comando:

```
1 $ git checkout -b <rama-NombreEstudiante>
```

Esto creará una nueva rama llamada "<rama-NombreEstudiante>" y nos moverá a esa rama en el repositorio local. Luego podemos hacer cambios en los archivos en esta rama y hacer confirmaciones para registrar los cambios. Por ejemplo, si modificamos el archivo

"README.md" en la rama "<rama-NombreEstudiante>", podemos confirmar los cambios con los siguientes comandos:

```
1 $ git add README.md
2 $ git commit -m "Modificacion de README en <rama-NombreEstudiante>"
```

Estos comandos registran los cambios en el archivo "README.md" en la rama "<rama-NombreEstudiante>". Luego podemos cambiar de regreso a la rama principal (en este caso, la rama "main") con el comando:

```
1 $ git checkout main
```

Si revisamos el archivo "README.md" en esta rama, veremos que no contiene los cambios que hicimos en la rama "<rama-NombreEstudiante>". Podemos fusionar los cambios en la rama principal usando el comando:

```
1 $ git merge <rama-NombreEstudiante>
```

Esto fusionará los cambios que hicimos en la rama "<rama-NombreEstudiante>" con la rama principal. Ahora, si revisamos el archivo "README.md" en la rama "main", veremos los cambios que hicimos en la rama "<rama-NombreEstudiante>".

El permiso para hacer merge depende de la política de control de versiones que se haya establecido en el repositorio. En algunos casos, todos los usuarios pueden hacer merge, mientras que en otros casos solo algunos usuarios o un administrador del repositorio pueden hacerlo. Además, algunos repositorios pueden requerir que se realice una revisión de código antes de que se permita un merge.

Es importante tener en cuenta que cada rama tiene su propio historial de confirmaciones y cambios, y que la fusión de ramas puede ser un proceso complejo. Por lo tanto, es recomendable tener un buen entendimiento de Git y sus comandos antes de trabajar con ramas en un repositorio.

Para borrar una rama local en Git, se puede usar el comando

```
1 $ git branch -d <nombre de la rama>
```



Por ejemplo, si quieres borrar la rama llamada "<rama-NombreEstudiante>". Si la rama tiene cambios que aún no se han fusionado con otra rama, Git mostrará un mensaje de error y no permitirá borrar la rama hasta que los cambios se fusionen. En ese caso, puedes usar el comando `git merge` para fusionar los cambios o el comando `git stash` para guardar temporalmente los cambios en una pila y luego recuperarlos más tarde si es necesario.

También es posible forzar la eliminación de una rama, incluso si tiene cambios pendientes de fusión, usando el comando `git branch -D <nombre de la rama>`. Sin embargo, al usar este comando, los cambios no fusionados se perderán permanentemente.

---

## ? Referencias

- [1] Atlassian. *Sourcetree*. 2021. URL: <https://www.sourcetreeapp.com/>.
- [2] Scott Chacon y Ben Straub. *Pro Git*. New York: Apress, 2014.
- [3] The Git Community. *Git Documentation*. Accessed: 2023-02-28. GitHub. 2021. URL: <https://git-scm.com/docs>.
- [4] Git. *Git - Downloads*. 2021. URL: <https://git-scm.com/downloads>.
- [5] GitHub. *GitHub Desktop*. 2021. URL: <https://desktop.github.com/>.
- [6] Jon Loeliger y Matthew McCullough. «Version control with Git». En: *Linux Journal* 2012.219 (2012), págs. 1-16.
- [7] Aske Pecinovsky. *Git Version Control Cookbook*. Packt Publishing Ltd, 2016.
- [8] Michael Ryan. *Learning Git*. Packt Publishing Ltd, 2013.

## Capítulo 4 Introducción a la computación.

Primero resuelve el problema, luego  
escribe el código

John Johnson

Primero que todo, una distinción, aun cuando muchas veces puedan usarse como sinónimos, escribir código y programar no son lo mismo; están estrechamente ligados, sin embargo la programación es la manifestación de la lógica. Un programa es un conjunto de instrucciones que define el comportamiento de una aplicación/programa (software). Escribir código es algo mucho más pedestre, el código es la forma de implementar las instrucciones para la máquina [9]. Así programar requiere tener en claro mucho más que las instrucciones (código), programar requiere manejo de memoria y bases de datos, requiere tener bien claro el proyecto en general, los módulos en los que podemos separarlo y los pasos a seguir para unirlos coherentemente.

Durante la fase de diseño del desarrollo de un programa, cada tarea que debe realizar el programa se divide en una serie de pasos lógicos. Estos pasos se representan comúnmente mediante pseudocódigo, diagramas de flujo y/o tablas de decisión. Como preparar un postre: Las instrucciones paso a paso sobre cómo preparar el postre son un algoritmo de este ejemplo.

### 4.1. Diagramas de Flujo

Los diagramas de Flujo no son exclusivos de la programación y usted podrá encontrarlos en muchas situaciones en las que se esquematicen procesos. El primer método estructurado para documentar el flujo de procesos, el "diagrama de flujo de procesos", fue presentado por Frank y Lillian Gilbreth en la presentación "Gráficos de procesos: primeros pasos para encontrar la mejor manera de hacer el trabajo", a miembros de la Sociedad Estadounidense de Ingenieros Mecánicos (ASME) en 1921[3] Douglas Hartree en 1949 explicó que Herman Goldstine y John von Neumann<sup>(1)</sup> habían desarrollado un diagrama de flujo para planificar

<sup>(1)</sup> Si, ese von Neumann, el de la arquitectura von Neumann

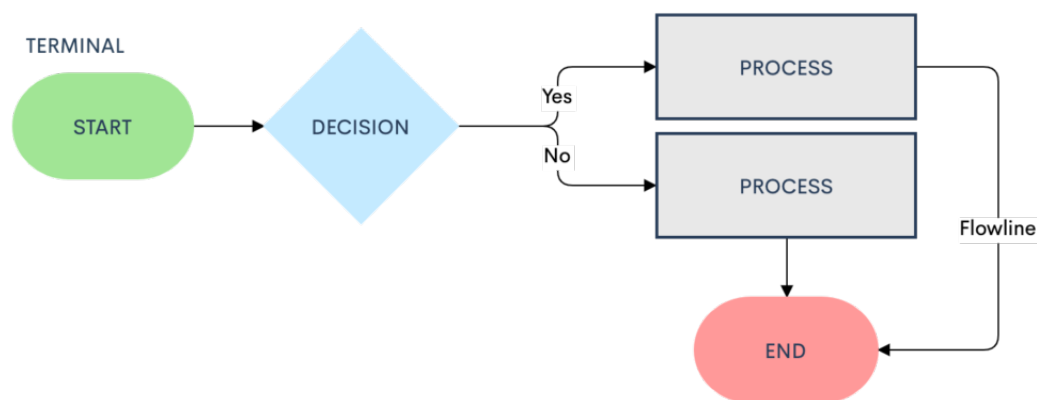


Figura 4.1. Un diagrama de flujo típico, sin propósito explícito, en el que se toma una decisión *yes/no*. Extraído de [zenflowchart.com](http://zenflowchart.com)

programas de computadora [4] Los diagramas de flujo de programación originales de Goldstine y von Neumann se pueden encontrar en su informe inédito, "Planificación y codificación de problemas para un instrumento informático electrónico, Parte II, Volumen 1"(1947), que se reproduce en las obras completas de von Neumann.[8]

El diagrama de flujo se convirtió en una herramienta popular para describir algoritmos, pero su popularidad disminuyó en la década de 1970, cuando las terminales informáticas interactivas se convirtieron en herramientas comunes para la programación informática, ya que los algoritmos se pueden expresar de manera más concisa como código fuente en lenguajes de alto nivel. A menudo se utiliza un pseudocódigo, que utiliza los modismos comunes de dichos lenguajes sin adherirse estrictamente a los detalles de uno en particular.

A principios del siglo XXI, los diagramas de flujo todavía se usaban para describir algoritmos informáticos.[9]

Un diagrama de flujo (flowchart) Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) y que tiene los pasos del algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se deben ejecutar.

Los símbolos estándar normalizados por ANSI (abreviatura de American National Standards Institute) son muy variados, aquí se presentan algunos:

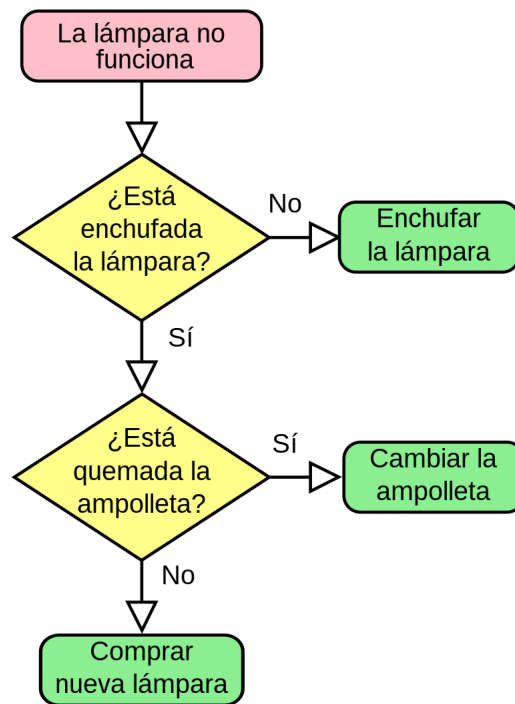


Figura 4.2. Un diagrama de flujo mostrando el proceso para lidiar con una lámpara que no funciona. Extraído de wikipedia.org

## 4.2. Programación estructurada

### Definición 4.1

Programación Estructurada: Es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de código. Esta metodología se basa en el uso de subrutinas y 3 estructuras de control de flujo de programa: secuencias, condicionales, bucles para crear programas más fáciles de leer, entender y mantener.

A partir de la definición, podemos decir que es una forma de escribir código que hace uso extensivo de las construcciones de flujo de control estructurado de selección (*if/then/else*) y repetición (*while* y *for*), estructuras de bloque y subrutinas que se muestran en la figura 4.3.

En su paper de 1968 Dijkstra [1], publicó una teoría vinculada a la programación estructurada, la cual indicaba que al diseñar cualquier programa es conveniente tomar en cuenta los siguientes fundamentos recogidos desde [7]:

- ✿ – El teorema estructural, que expresa que se puede compilar cualquier programa utilizando solo tres estructuras de control esenciales: estructura secuencial, estructura de alternativas y estructura repetitiva.

- ✿ – Al delinear los programas se exhorta a aplicar la técnica descendente, llamada también de arriba hacia abajo.
- ✿ – Deben limitarse los rangos de validez y visibilidad de las variables y las estructuras de datos.

Como sugiere la palabra, se puede definir como un enfoque de código en el que las ordenes o pasos se realizan como una estructura única y secuencial. Significa que el código ejecutará instrucción por instrucción, una tras otra. No admite la posibilidad de saltar de una instrucción a otra (de una a línea a otra como sucedía antes con la ayuda de una declaración como *GOTO*). Por lo tanto, las instrucciones en este enfoque se ejecutarán de manera estructurada y en serie [6, 8].

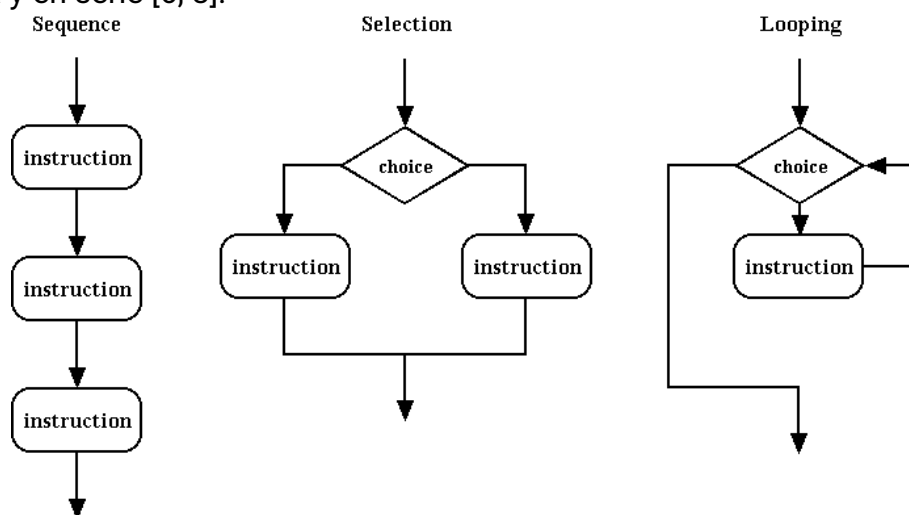


Figura 4.3. Secuencias de control en la programación estructurada. Tomada desde thecomputerstudents.com

### 4.2.1. Estructuras de control

El teorema estructural indica que cualquier algoritmo con un único punto de inicio y de culminación se puede constituir como una composición de tres estructuras de control [7].

- ✿ Estructura de secuencia o lineal: Esta estructura es simplemente la secuencia o sucesión de dos o más operaciones o comandos.
- ✿ Estructura de decisión o de alternativas: Es la selección de un comando entre dos o más posibles alternativas.
- ✿ Estructura de ciclo o repetitiva con una pregunta inicial: Se repiten ciertos comandos siempre que se cumpla una determinada condición. También el ciclo se puede realizar con un

contador.

Al programar de forma lógica y clara estas estructuras de control admitidas, la programación estructurada permite un enfoque eficiente de las funciones con cualquier grado de dificultad.

La entrada y salida en un programa estructurado son eventos únicos. Significa que el programa utiliza elementos de entrada única y salida única. Por lo tanto, un programa estructurado debería ser un programa bien mantenido, ordenado y limpio. Esta es la razón por la que el enfoque de programación estructurada es bien aceptado en el mundo de la programación [8] .

Ventajas del enfoque de programación estructurada:

- ✿ Fácil de leer, de entender y de usar.
- ✿ Más fácil de depurar y de mantener.
- ✿ Principalmente basado en problemas (en lugar de basado en máquinas)
- ✿ El desarrollo es "fácil" ya que requiere "poco esfuerzo" y tiempo.
- ✿ Independiente de la máquina, en su mayoría.

Desventajas del enfoque de programación estructurada:

- ✿ Dado que es independiente de la máquina, lleva tiempo convertirlo en código de máquina.
- ✿ El programa depende de factores modificables como los tipos de datos. Por lo tanto, debe actualizarse, según la necesidad, sobre la marcha.

Como una contra parte a la programación estructurada hay varios paradigmas de programación, cada uno con sus propios principios y prácticas que, aun cuando están fuera del alcance de este curso, les invito a explorarla y a implementarla en algún futuro. Algunos de los paradigmas más comunes son:

- ✿ Programación orientada a Objetos: Es un paradigma de programación que usa **objetos** con datos y comportamiento, en sus interacciones a través de métodos y mensajes pa-

ra diseñar aplicaciones y programas. Usa varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

#### Ejemplo 4.1

Algunos lenguajes utilizados cuando trabajamos orientados a objetos: Java, Python, Ruby, C++, C#, Objective-C, Swift.

#### Comentarios

Un objeto es un miembro de una clase, mientras que una clase es un elemento abstracto que define las variables y los métodos de miembros que se desprenden de ella e.g. ambos, Ferrari y Twingo son objetos ( *también llamados instancias*), miembros la clase de los Automóviles, por lo que heredan algunas características comunes, como ser vehículos terrestres a motor, la necesidad de ruedas, asientos, y a la fecha de este apunte, usar combustible fósil en un motor de combustión. De igual forma Casio y Rolex pertenecen a la clase de los relojes ¿ que elementos comparten ?

- ❁ Programación funcional (PF): este paradigma enfatiza el uso de funciones, que toman entrada y producen salida sin ningún efecto secundario. La PF fomenta la inmutabilidad y evita el estado mutable y los datos compartidos.

#### Ejemplo 4.2

Algunos lenguajes utilizados cuando trabajamos orientados a funciones: Haskell, Lisp, ML, F#, Scala, Erlang.

- ❁ Programación dirigida por eventos (PDE): este paradigma gira en torno al concepto de eventos, que son desencadenados por acciones del usuario o cambios en el sistema. La PDE se utiliza a menudo en la programación de GUI, donde la interfaz de usuario responde a la entrada del usuario.

#### Ejemplo 4.3

Algunos lenguajes utilizados cuando trabajamos orientados por eventos: JavaScript, Visual Basic, C#, Java, Python.

- ❁ Programación concurrente: este paradigma implica la ejecución simultánea de múltiples

hilos o procesos para mejorar el rendimiento y la capacidad de respuesta. La programación concurrente requiere una sincronización cuidadosa para evitar condiciones de carrera y otros problemas.

### Ejemplo 4.4

Algunos lenguajes utilizados cuando trabajamos con orientación concurrente: Java, Python, Erlang, Go, Rust, Clojure.

- ✿ Programación declarativa: este paradigma se enfoca en expresar lo que se debe hacer en lugar de cómo hacerlo. La programación declarativa a menudo implica especificar reglas o restricciones, que se utilizan para generar automáticamente una solución.

### Ejemplo 4.5

Algunos lenguajes utilizados cuando trabajamos con orientación declarativa: SQL, Prolog, XSLT, Haskell, Erlang.

Es importante tener en cuenta que muchos lenguajes de programación pueden admitir múltiples paradigmas, lo que significa que pueden permitir a los programadores elegir el paradigma que mejor se adapte a la tarea en cuestión. Además, algunos lenguajes de programación han sido creados específicamente para enfatizar un paradigma en particular. Por ejemplo, Haskell se creó principalmente para la programación funcional, mientras que Erlang se creó principalmente para la programación concurrente y distribuida.

También hay muchos otros paradigmas de programación, como la programación lógica, la programación por restricciones y la programación orientada a aspectos. Cada paradigma tiene sus propias fortalezas y debilidades, y elegir el paradigma adecuado para una tarea en particular puede marcar una gran diferencia en la calidad y mantenibilidad del código.

### 4.2.2. Importancia de la programación estructurada

La importancia de la programación estructurada radica en que permite a los programadores escribir código fuente más modular, eficiente y confiable. Al organizar el código en bloques lógicos bien definidos, los programadores pueden dividir tareas complejas en tareas más simples y manejables, lo que facilita la comprensión del código y reduce la cantidad de



errores introducidos durante la codificación.

La programación estructurada se considera un pilar fundamental en la enseñanza de la programación debido a su simplicidad y claridad, lo que permite a los estudiantes comprender mejor los conceptos de programación y desarrollar habilidades de codificación sólidas. También ha sido esencial en el desarrollo de lenguajes de programación modernos y herramientas de desarrollo que se utilizan en la actualidad.

Este paradigma de programación nos permite pensar en términos de flujo discretizado en pasos, usando tan sólo los 3 elementos antes mencionados: secuencias, elecciones, bucles. Para esto introduciremos el concepto de pseudocódigo (independientemente del lenguaje de la máquina, cercano al natural, convertible en cualquier lenguaje de programación.)

#### **Definición 4.2**

Pseudocódigo: literalmente "código falso". Es una descripción de alto nivel de un algoritmo, escrito de una manera que se asemeja a un lenguaje de programación pero no es una sintaxis formal. El propósito del pseudocódigo es proporcionar una representación simplificada y legible por humanos de un algoritmo que pueda ser entendido fácilmente por personas que no están familiarizadas con el lenguaje de programación utilizado para implementarlo.

Es un término que se usa a menudo en programación; una metodología que permite al programador representar la implementación de un algoritmo de tal forma que se puede interpretar sin importar la experiencia o conocimiento de programación. El pseudocódigo, como sugiere su nombre, es un código falso o una representación de código que puede ser entendido incluso por alguien con algún conocimiento de programación a nivel escolar [5].

Cuando escribimos código en un lenguaje de programación, estamos sujetos a una sintaxis estricta y patrones rígidos. Dado que el pseudocódigo es un método informal de diseño de programas, no tiene que obedecer ninguna regla establecida, sólo ser consistente. El pseudocódigo actúa como el puente entre su cerebro y el ejecutor de código del computador. Permite planificar instrucciones que siguen un patrón lógico, sin incluir todos los detalles técnicos.

#### Ejemplo 4.6

- ✿ Si calificación de estudiante es mayor o igual a 4,0:
  - ✿ Estudiante aprueba
- ✿ De lo contrario:
  - ✿ Estudiante reprueba

#### Ejemplo 4.7

- ✿ inicializar *aprobado* = cero
- ✿ inicializar *reprobado* = cero
- ✿ inicializar *estudiantes* = uno
- ✿ WHILE *estudiantes* sea menor o igual a diez:
  - ✿ ingrese el resultado del próximo examen:
  - ✿ IF la calificación del examen es mayor o igual a 4,0:
    - ❖ Estudiante aprueba
    - ❖ Agregar uno a los *aprobado*
  - ✿ ELSE:
    - ❖ Estudiante reprueba
    - ❖ Agregar uno a los *reprobado*
  - ✿ Agregar uno al contador de estudiantes
- ✿ Imprime el numero de *aprobado*
- ✿ Imprime el numero de *reprobado*
- ✿ IF *aprobado/estudiantes* es mayor que 0,6:

✿ Imprimir “aumentar la matrícula”

✿ ELSE:

✿ Imprimir “revisar contenido”

El pseudocódigo puede ser muy simple como el ejemplo 4.2.2 o más complejo como el ejemplo 4.2.2

## Cómo resolver problemas usando pseudocódigo

Estos son pasos sugeridos para resolver problemas con pseudocódigo:

### 1. Paso: Asegúrese de entender la pregunta:

- ✿ Debe leer y comprender la pregunta correctamente. Este es posiblemente el paso más importante en el proceso.
- ✿ Si no comprende correctamente la pregunta, no podrá resolver el problema y descubrir los posibles pasos a seguir. Una vez que identifique el problema principal a resolver, estará listo para abordarlo.

### 2. Paso: Entender lo que hace o debe hacer un programa:

- ✿ Comprender que todo lo que hace un programa es (opcionalmente) aceptar datos como entrada, trabajar en los datos poco a poco y finalmente devolver una salida. El cuerpo del programa es lo que realmente resuelve el problema y lo hace línea por línea.

### 3. Paso: Desglose el problema:

- ✿ Ahora necesita dividir el problema en partes más pequeñas y subproblemas. Con cada problema más pequeño que resuelvas, se acercará más a la solución del problema principal.
- ✿ Representar estos pasos de resolución de problemas de la manera más clara y comprensible posible. **Esto es el pseudocódigo.** Sugerencias:

- i. Comandos clave en mayúscula (WHILE, IF, ELSE, FOR, ) e.g. WHILE estudiantes<=10

- ii. Escriba una expresión por línea
- iii. Use indentación
- iv. Sea específico
- v. KISS (*Keep it Simple, Stupid*)

### Algoritmo, pseudocódigo, programa

Discutiremos el malentendido más común: un algoritmo y un pseudocódigo son una misma cosa. ¡No, ellos no son! Echemos un vistazo a las definiciones primero.

#### Definición 4.3 Algoritmo

Procedimiento bien definido, con enfoque lógico sistemático, un procedimiento paso a paso bien definido que permite resolver un problema en un tiempo finito. Un algoritmo consta de un conjunto de instrucciones, llamados pasos, ejecutados en un orden específico para lograr el resultado deseado. El propósito de un algoritmo es tomar entradas y producir salidas mientras resuelve un problema particular de manera sistemática y eficiente.

Un algoritmo se utiliza para proporcionar una solución a un problema particular en forma de pasos bien definidos. Siempre que use una computadora para resolver un problema en particular, los pasos que conducen a la solución deben instruirse adecuadamente a la máquina. Al ejecutar un algoritmo en un computador, se combinan varias operaciones sencillas para realizar operaciones matemáticas más complejas. Los algoritmos se pueden expresar usando lenguaje natural, diagramas de flujo, etc. Vea un ejemplo para una mejor comprensión. Como programadores, todos conocemos el programa de búsqueda lineal. (Búsqueda lineal)

#### Ejemplo 4.8

Algoritmo de búsqueda lineal de un término  $x$  en un array  $arr[]$ :

1. Comience desde el elemento más a la izquierda de  $arr[]$  y uno por uno compare  $x$  con cada elemento de  $arr[]$ .

2. Si  $x$  coincide con un elemento, devuelve el índice.
3. Si  $x$  no coincide con ninguno de los elementos, devuelve -1.

Para la definición de pseudocódigo ver 4.2.2, pero diremos que es una forma informal y simple de escribir programas en los que se representa la secuencia de acciones e instrucciones en una forma que los humanos pueden entender fácilmente [9]; usando frases cortas para escribir el código de un programa sin implementarlo en un lenguaje de programación específico.

**Ejemplo 4.9**

Pseudocódigo para búsqueda lineal de un término  $x$  en una lista:

```
FUNCTION linearSearch(list, x):  
  FOR index FROM 0 -> length(list):  
    IF list[index] == x THEN  
      RETURN index  
    ENDIF  
  ENDLLOOP  
  RETURN -1  
END FUNCTION
```

Aquí, no hemos usado ningún lenguaje de programación específico, pero escribimos los pasos de una búsqueda lineal en una forma más simple que luego puede modificarse aún más al llevarlo a un programa.

### Definición 4.4 Programa

Una implementación ejecutable de un algoritmo. Es el código exacto escrito para el problema siguiendo todas las reglas de un lenguaje de programación específico, siguiendo la sintaxis de ese lenguaje. El propósito de un programa es tomar entradas y producir salidas de acuerdo con los pasos especificados en el algoritmo. Los programas son ejecutados por una computadora y pueden realizar cálculos y operaciones complejas para resolver problemas.

Un programa es un conjunto de instrucciones que debe seguir la máquina. Esta no puede leer un programa directamente, porque solo entiende el código de máquina. Pero puede escribir cosas en un lenguaje de computadora, y luego un compilador o intérprete puede hacerlo comprensible para la ella.

### Ejemplo 4.10

Veamos el código para una función en Python3, para la búsqueda lineal

```
def search( arr, n, x):  
    for i in range(n):  
        if (arr[i] == x):  
            return i  
    return -1
```

En resumen, un algoritmo es un procedimiento para resolver un problema, el pseudocódigo es una descripción de alto nivel del algoritmo y un programa es una implementación específica del algoritmo en un lenguaje de programación.

### Ejercicio 4.1

Reescriba el ejemplo 4.2.2(1.2) usando las sugerencias, de tal forma que se vea como el ejemplo 4.2.2 (1.4)

### 4.2.3. Complejidad Algorítmica

A menudo llamamos a la complejidad de un algoritmo "tiempo de ejecución"; así a la hora de medir el tiempo, siempre lo haremos en función del número de operaciones elementales (en adelante *OE*) que realiza el algoritmo, entendiendo por operaciones elementales aquellas que se realizan en tiempo acotado por una constante. Así, consideraremos *OE* las operaciones aritméticas básicas, asignaciones a variables, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como *1OE* (por favor mantenga esto en su mente).

### Principio de Invarianza

Dado un algoritmo y dos de sus implementaciones  $I_1$  e  $I_2$ , suponga que tardan  $T_1(n)$  y  $T_2(n)$  segundos respectivamente, el Principio de Invarianza afirma que existe una constante real  $c > 0$  y un número natural  $n_0$  tales que para todo  $n \geq n_0$  se verifica que  $T_1(n) \leq c \cdot T_2(n)$ .

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado, no va a diferir más que en una constante multiplicativa [2]

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo del orden de  $T(n)$  si existen una constante real  $c > 0$  y una implementación  $I$  del algoritmo que tarda menos que  $c \cdot T(n)$ , para toda entrada de tamaño  $n$ . Dos factores a tener muy en cuenta son: la constante multiplicativa y el  $n_0$  para los que se verifican las condiciones; pues si bien, a priori, un algoritmo de orden cuadrático es mejor que uno de orden cúbico, en el caso de tener dos algoritmos cuyos tiempos de ejecución son  $106n^2$  y  $5n^3$  el primero sólo será mejor que el segundo para tamaños de la entrada superiores a 200,000.

**Ejemplo 4.11**

Para determinar el tiempo de ejecución, dado el siguiente algoritmo:

```

CONST n = ...; (* num. maximo de elementos de un vector *);
ARRAY vector = [1...n] OF INTEGER;

PROCEDURE Buscar(a:vector;c:INTEGER):CARDINAL;
VAR j:CARDINAL;
BEGIN
1  j:=1;
2  WHILE (a[j]<c) AND (j<n) DO
3      j:=j+1
4  ENDLOOP;
5  IF a[j]=c THEN
6      RETURN j
7  ELSE RETURN -1
8  ENDIF
END Buscar

```

Calcularemos primero el número *OE* que se realizan:

1. asignación: *1OE*.
2. bucle, 2 comparaciones, acceso al vector, y AND: *4OE*.
3. un incremento y una asignación (*2OE*).
4. ENDLOOP *0OE*
5. una condición y un acceso al vector (*2OE*).
6. un RETURN (*1OE*) si la condición se cumple.
7. un RETURN (*1OE*), cuando la condición del IF anterior es falsa

Con esto, si vemos el ejemplo 4.2.3 1.6:



- ✿ En el mejor caso el algoritmo efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone **2OE** (suponemos que las expresiones se evalúan de izquierda a derecha, y con “cortocircuito”, es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia,  $T(n) = 1 + 2 + 3 = 6$ .
- ✿ En el caso peor, se efectúa la línea (1), el bucle se repite  $n-1$  veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. Por tanto:

$$\begin{aligned}
 T(n) &= 1 + \left( \left( \sum_{i=1}^{n-1} (4 + 2) \right) + 4 \right) + 2 + 1 \\
 &= 1 + ((n-1) \cdot (4 + 2) + 4) + 2 + 1 \\
 &= ((n-1) \cdot 6) + 8 \\
 &= 6n + 2
 \end{aligned} \tag{4.1}$$

- ✿ En el caso medio, el bucle se ejecutará un número de veces entre **0** y  $n-1$ , y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder. Como existen  $n$  posibilidades (puede que el número buscado no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de  $\frac{1}{n}$ . Con esto, el número medio de veces que se efectuará el bucle corresponde a la esperanza es la distribución uniforme discreta entre **0** y  $n$

$$\sum_{i=1}^{n-1} i \frac{1}{n} = \frac{n-1}{2} \tag{4.2}$$

Por lo que, siguiendo la lógica expuesta en la ecuación 4.1, pero usando la esperanza

calculada en la ecuación 4.2 se tiene que

$$T(n) = 1 + \left( \left( \sum_{i=1}^{\frac{n-1}{2}} (4 + 2) \right) + 4 \right) + 2 + 1 \quad (4.3)$$

$$= 3n + 3$$

### Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

- ✿ Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante  $c$  que menciona el *Principio de Invarianza* dependerá de la implementación particular, pero nosotros supondremos que  $c = 1$
- ✿ El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.
- ✿ El tiempo de ejecución de la expresión

“CASE C OF  $v1:S1 | v2:S2 | \dots | vn:Sn$  END;”

es  $T = T(C) + \max\{T(S1), T(S2), \dots, T(Sn)\}$ . Obsérvese que  $T(C)$  incluye el tiempo de comparación entre  $C$  y todos los valores en  $\{v1, v2, \dots, vn\}$ .

- ✿ El tiempo de ejecución de la expresión

“IF C THEN S1 ELSE S2 END;”

es  $T = T(C) + \max\{T(S1), T(S2)\}$ .

- ✿ El tiempo de ejecución de un bucle/loop de sentencias

“WHILE C DO S END;”

es  $T = T(C) + (n^\circ \text{iteraciones}) \cdot (T(S) + T(C))$ . Obsérvese que tanto  $T(C)$  como  $T(S)$  pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo

- ❁ Para calcular el tiempo de ejecución del resto de sentencias iterativas (FOR, REPEAT, LOOP) basta expresarlas como un bucle WHILE. A modo de ejemplo, considere que el tiempo de ejecución del bucle:

```
FOR i:=1, INC(i) TO n DO
    S
END;
```

puede ser calculado a partir del bucle equivalente:

```
i:=1;
WHILE i<=n DO
    S;
    i+=INC(i) ;
END;
```

- ❁ El tiempo de ejecución de una función  $F(P_1, P_2, \dots, P_n)$  es 1 (por la llamada), más el tiempo de evaluación de los parámetros  $\{P_1, P_2, \dots, P_n\}$  más el tiempo que tarde en ejecutarse  $F$ , esto es,  $T = 1 + \sum_{i=1}^n T(P_i) + T(F)$ . No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas  $OE$  como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- ❁ El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia como las vistas en el ejemplo 4.2.3 1.6

#### Definición 4.5

La complejidad algorítmica se refiere a la medida de la cantidad de recursos (como tiempo, memoria o esfuerzo computacional) requeridos por un algoritmo para resolver un problema. Se utiliza para analizar y comparar la eficiencia de los algoritmos y para determinar la idoneidad de un algoritmo para una tarea determinada, proporciona

una forma de describir el crecimiento del uso de recursos del algoritmo a medida que aumenta el tamaño de la entrada.

La complejidad algorítmica cae dentro de una rama de la informática teórica llamada *teoría de la complejidad computacional* y su análisis es útil cuando se comparan algoritmos o se buscan mejoras, esto ya que el término se usa para como medida de "cuánto tiempo" tardaría un algoritmo en completarse dada una entrada de tamaño  $n$ .

Es importante tener en cuenta que hemos usado comillas ("cuanto tiempo") pues lo que nos preocupa es el número de pasos, o más bien, el orden (aproximación) del conteo de operaciones de un algoritmo, en lugar de contar los pasos exactos. No es el tiempo de ejecución real en términos de milisegundos, ya que esto puede variar dependiendo del *hardware*; e.g. al contrastar la complejidad de 2 algoritmos uno de 10 pasos vs uno de 30 pasos, el resultado para ambos podría ser el misma.

### Cotas de complejidad. Medidas asintóticas

Si un algoritmo se tiene que escalar, debe calcular el resultado dentro de un límite de tiempo finito y práctico incluso para valores grandes de  $n$ . Por esta razón, la complejidad se calcula asintóticamente cuando  $n$  tiende a infinito. Si bien la complejidad generalmente se expresa en términos de tiempo, a veces la complejidad también se analiza en términos de espacio, lo que se traduce en los requisitos de memoria del algoritmo.

#### Comentarios Cota Superior

Dado  $g, f : \mathbb{R}_+ \rightarrow \mathbb{R}$ , decimos que  $g(n) = O(f(n)) \iff \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N}$  tal que  $g(n) \leq kf(n) \quad \forall n \geq n_0$

Intuitivamente,  $g \in O(f)$  indica que  $g$  está acotada superiormente por algún múltiplo de  $f$ . Normalmente estaremos interesados en la menor función  $f$  tal que  $g$  pertenezca a  $O(f)$  [2].

**Comentarios** Cota inferior

Dado  $g, f : \mathbb{R}_+ \rightarrow \mathbb{R}$ , decimos que  $g(n) = \Omega(f(n)) \iff \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N}$  tal que  $g(n) \geq kf(n) \quad \forall n \geq n_0$

Intuitivamente,  $g \in \Omega(f)$  indica que  $g$  está acotada inferiormente por algún múltiplo de  $f$ . Normalmente estaremos interesados en la mayor función  $f$  tal que  $g$  pertenezca a  $\Omega(f)$ , a la que denominaremos su cota inferior [2].

**Comentarios** Asintóticamente equivalente

Dado  $g, f : \mathbb{R}_+ \rightarrow \mathbb{R}$ , decimos que  $g(n) = \Theta(f(n)) \iff \exists k_1, k_2 \in \mathbb{R}_+, n_0 \in \mathbb{N}$  tal que  $k_1 f(n) \leq g(n) \leq k_2 f(n) \quad \forall n \geq n_0$

Intuitivamente,  $g \in \Theta(f)$  indica que  $g$  está acotada tanto superior como inferiormente por múltiplos de  $f$ , es decir, que  $g$  y  $f$  crecen de la misma forma [2]

Podemos hacer una analogía entre estas notaciones y comparaciones entre números:

$$f(n) = \Theta(g(n)) \approx f = g.$$

$$f(n) = O(g(n)) \approx f \leq g.$$

$$f(n) = o(g(n)) \approx f < g.$$

$$f(n) = \Omega(g(n)) \approx f \geq g.$$

$$f(n) = \omega(g(n)) \approx f > g.$$

La notación  $O(f(n))$  también se denomina notación asintótica o notación "Big O", es usada para representar la complejidad. La complejidad del cálculo asintótico  $O(f(n))$  determina en qué orden los recursos como el tiempo de CPU, la memoria, etc. son consumidos por el algoritmo que se articula en función del tamaño de los datos de entrada.

La complejidad se puede encontrar en cualquier forma: como constante  $O(1)$ ; el crecimiento logarítmico es  $O(\log(n))$ ; el crecimiento lineal es  $O(n)$ ; el crecimiento logarítmico lineal es  $O(n \cdot \log(n))$ ; el crecimiento cuadrático es  $O(n^2)$ ; el crecimiento exponencial es  $O(2^n)$ ; el crecimiento factorial es  $O(n!)$ . Sus órdenes de crecimiento también se pueden

comparar de mejor a peor:

$$O(1) < O(\log(n)) < O(\sqrt{n}) < O(n) < O(n \cdot \log(n)) < O(n^2) < O(n^3) < O(2^n) < O(10^n) < O(n!)$$

Esto nos da una relación de equivalencia sobre el conjunto de los algoritmos

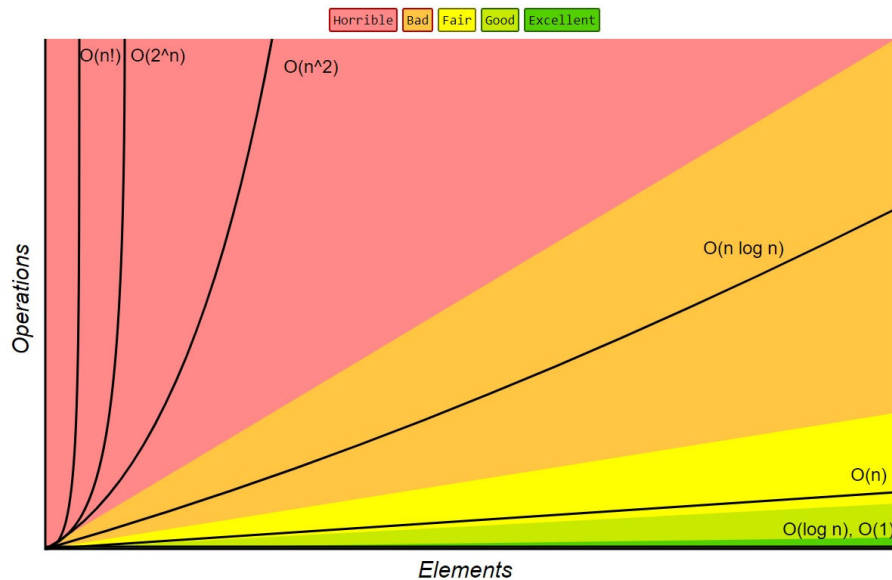


Figura 4.4. Orden de crecimiento de los algoritmos de acuerdo a la notación Big-O. Tomada desde bigocheatsheet.com



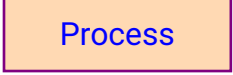
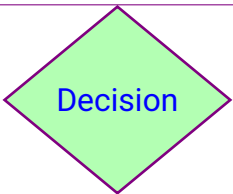
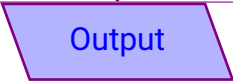
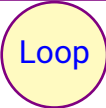
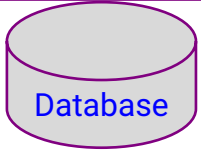


### Ejercicio 4.2

Demuestre que la existencia de la cota superior determina una relación de equivalencia. ¿Es esta relación una relación de orden parcial o total? Justifique enunciando la o las propiedades que son satisfechas y la (o las) que no.

### ? Referencias

- [1] Edgar Dijkstra. «Edgar Dijkstra: Go To Statement Considered Harmful». En: *Communication of the ACM* 11 (3 mar. de 1968), págs. 147-148.
- [2] Rosa Guerequeta García y Antonio Vallecillo Moreno. «Capítulo 1 LA COMPLEJIDAD DE LOS ALGORITMOS 1.1 INTRODUCCIÓN». En: 1998. Cap. 1. ISBN: 84-7496-666-3. URL: <http://www.lcc.uma.es/~av/Libro/>.
- [3] Frank Gilberth y Lilliam Gilberth. «Process Chart». En: *The American Society of Mechanical Engineers* (dic. de 1921).

- [4] Douglas Hartree. *Calculating Instruments and Machines*. Urbana. University of Illinois, mayo de 1949. ISBN: 9781107630659.
- [5] *How to write a Pseudo Code?* - GeeksforGeeks. Ago. de 2021. URL: <https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/>.
- [6] Karl P. Hunt. «An introduction to structured programming». En: *Behavior Research Methods & Instrumentation* 11 (2 mar. de 1979), págs. 229-233. ISSN: 1554351X. DOI: [10.3758/BF03205654](https://doi.org/10.3758/BF03205654)/METRICS. URL: <https://link.springer.com/article/10.3758/BF03205654>.
- [7] Lifeder. *Programación estructurada: características, ejemplos, ventajas, aplicaciones*. Mar. de 2020. URL: <https://www.lifeder.com/programacion-estructurada/>.
- [8] Prabhu Rishabh. *Structured Programming Approach with Advantages and Disadvantages* - GeeksforGeeks. URL: <https://www.geeksforgeeks.org/structured-programming-approach-with-advantages-and-disadvantages/>.
- [9] Kingsley Ubah. *What is Pseudocode? How to Use Pseudocode to Solve Coding Problems*. Mayo de 2021. URL: <https://www.freecodecamp.org/news/what-is-pseudocode-in-programming/>.

Símbolo	Significado
	Este símbolo representa el inicio del diagrama, usualmente no recibe inputs.
	Este símbolo representa el ingreso de <i>inputs</i> o información en el diagrama.
	Este símbolo representa un proceso o acción que necesita ser realizado para continuar con el flujo. Puede ser un cálculo, una operación o cualquier otro tipo de tarea que necesite ser ejecutada.
	Este símbolo se usa cuando hay múltiples salidas basadas en condiciones, casos o criterios. Por lo general se usa un booleano (Si/No; <i>True/False</i> ) que luego lleva a diferentes caminos en el diagrama.
	Este símbolo representa la salida o <i>output</i> de los datos/información desde el diagrama.
	Este símbolo se utiliza para representar un bucle o ciclo en el flujo del programa. El bucle permite que se repita una serie de acciones o procesos varias veces en función de una que condición sea satisfecha o un contador.
	Este símbolo representa una base de datos, que se utiliza para almacenar y recuperar información de manera estructurada y eficiente.
	Este símbolo representa un documento, que puede ser cualquier tipo de archivo digital, como un archivo de texto, una imagen, un video, etc.
	Este es el punto final del diagrama.

Cuadro 4.1. tabla con algunos, no todos, de los símbolos estandarizados usados en diagramas de flujo



# Capítulo 5 Variables y Tipos de Datos en Python

## 5.1. Variables

En matemáticas estamos acostumbrados a las variables; ustedes ¿están acostumbrados a las variables? ¿qué es una variable?

$$A = P \cdot \left(1 + \frac{r}{100}\right)^n \quad (5.1)$$

La ecuación anterior no entrega una cantidad (*Amount*) y hemos de considerar que: *P* es una cantidad primaria y tiene un valor de 100 (*Primary = 100*), *r = 5* y *n = 7*. Esta forma es completamente convencional en ciencias, así cuando usamos una calculadora, ingresamos números, y operamos entre números. Cuando programamos, hay una capa extra de abstracción que a veces puede producir errores.

Pensemos en el siguiente ejemplo

```
1 print(100*(1 + 5.0/100)**7)
```

```
1 primary = 100
2 r = 5.0
3 n = 7
4 amount = primary * (1+r/100)**n
5 print(amount)
```

Ambos textos desarrollan la misma operatoria. Una ventaja de usar variables, es que; con buenos nombres de variables, las operaciones se vuelven más intuitivas, fáciles de leer e interpretar para los humanos. Otra ventaja, quizás más importante es que no debemos reescribir todo un código cada vez que queremos hacer una variación. En la medida que nuestro código se vuelva más complejo, será evidente que: la probabilidad de error aumenta cercana a 1 cuando tenemos que cambiar un mismo dato varias veces, en varias líneas de código. Es por esto, que si un mismo dato se repite varias veces (más de una vez), el dato debe ser almacenado en una variable [1]

Es común tener una declaración por línea, aunque es posible poner varias declaraciones en una línea, separadas por punto y coma (`primario = 100; r = 5,0; n=7`).

En la ecuación (5.1), primero introdujimos la fórmula en sí y luego definimos y explicamos las variables utilizadas en la fórmula (*P, r, nyA*) en la siguiente línea. Este enfoque es completamente estándar en matemáticas, pero no tiene sentido en programación.

Las instrucciones en un código se llaman **declaraciones**, y se leen una a una, de arriba a abajo, **NO siempre de izquierda a derecha**, en la medida que el código es ejecutado. Esto puede sonar trivial, pero tiene implicancias para la lógica del programa, todas las variables deben definirse arriba de la línea donde se utilizan. por ejemplo:

```
1 >>> variable = 0
2 >>> variable = 2022
3 >>> variable = variable + 1
4 >>> print(variable)
5 2023
```

En el ejemplo anterior ¿qué sentido tiene decir que `variable = 0`? si `variable=0` ¿qué sentido tiene decir que `variable = 2022`? claramente  $0 \neq 2022$  ¿estamos de acuerdo? Lo que ha sucedido en las 2 primeras líneas son asignaciones, a `variable` le hemos asignado 2 valores distintos y el que será usado en el próximo paso será la última asignación válida. ¿qué sentido tiene ahora decir `variable = variable + 1`? ¿que clase de dios permite que un dato sea igual a su sucesor? En matemáticas y ciencias afines, esto no tiene sentido, pero en cuanto a código si. El lado derecho de la igualdad se evalúa primero, usando el valor previamente asignado en la línea 2 (i.e. 2022) y luego la `variable` es actualizada cuando se le asigna el un nuevo valor. En este sentido `=` se le conoce como operador de asignación y en sentido estricto, no es lo mismo que `=` en matemáticas. Es lo mismo, pero no es igual.

Para programar en Python es muy útil aprender inglés, para luego olvidar todo el inglés que no es Python. Así los nombres de las variables, quedan a criterio de cada quien, pero es importante conservar un balance entre sucinto y explicativo.

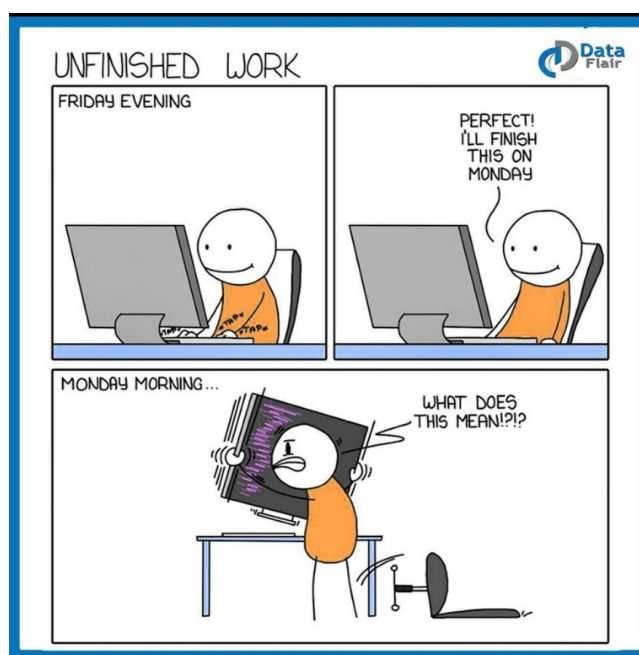


Figura 5.1. Parece broma, pero es anécdota. Es muy usual no entender el trabajo de otra persona, incluso cuando otra persona es usted del pasado

## 5.2. Comentarios

Los comentarios son útiles para explicar el proceso de pensamiento en los programas. Es posible combinar los puntos fuertes de los dos programas anteriores y tener nombres de variables compactos y una descripción más detallada de lo que significa cada variable. Esto se puede hacer usando comentarios, como se ilustra en el siguiente ejemplo:

```

1  # Programa que calcula el crecimiento de un deposito bancario
2  primary = 100 # cantidad inicial
3  r = 5.0 # tasa de interez en %
4  n = 7 # cantidad de años
5  amount = primary * (1+r/100)**n
6  print(amount)

```

Python es un lenguaje de programación interpretado de alto nivel que soporta varios tipos de datos. Los tipos de datos son importantes en Python ya que permiten a los programadores manejar diferentes tipos de información. Según la documentación oficial de Python [PythonSoftwareFoundation], los principales tipos de datos en Python incluyen números, cadenas, listas, tuplas, conjuntos y diccionarios.

Principales tipos de datos:

❁ **Números:** En Python, los números se pueden representar como enteros (int), números de coma flotante<sup>(1)</sup> (float) o números complejos (complex). Los números enteros son números sin decimales, mientras que los números de coma flotante son números con decimales. Los números complejos se utilizan para representar números con una parte real y una parte imaginaria. En cuanto a números, python funciona tan normalmente como su calculadora científica. Para los números, además de las operaciones iniciales, se pueden utilizar los métodos int(), float() y complex() para convertir entre diferentes tipos de números.

```
1  # Ejemplo de números
2  a = 5      # Entero
3  b = 3.14   # Coma flotante
4  c = 2 + 3j # Complejo
5
6  >>> 2*a
7  10
8  >>> a+b
9  8.14
10 >>> 11*c
11 22+33j
```

Notará que en el caso anterior, para el cuerpo de los números complejos **C** Python hace uso de la letra j, esto es porque es usual que la letra *i* se use para los ciclos iterativos 6.3

❁ **Cadenas de texto:** Las cadenas de texto (str o strings) son una secuencia de caracteres. En Python, las cadenas se pueden definir utilizando comillas simples o dobles. Las cadenas de texto **son inmutables**, lo que significa que **no se pueden cambiar después de ser creadas**. Para las cadenas, se pueden utilizar los métodos len() para obtener la longitud de una cadena y split() para dividir una cadena en subcadenas.

```
1  # Ejemplo de cadenas de texto
2  >>> cadena = "Hola, mundo"
3  >>> print(cadena.upper())
4
5
```

<sup>(1)</sup> Si, es raro, pero es parte de la jerga. El término se utilizó por primera vez en los primeros días de la computación cuando los informáticos estaban desarrollando formas de representar números reales (es decir, números con partes fraccionarias) en forma binaria.

El término "punto flotante" se refiere al hecho de que el punto decimal puede "flotar" o moverse hacia la izquierda o hacia la derecha según la magnitud del número que se representa. En otras palabras, el punto decimal no está fijo en su lugar, como lo está en una representación de punto fijo.

```
4     HOLA, MUNDO
6     >>> print(cadena.split())
7     ['Hola,', 'mundo']
8     >>> nueva_cadena = "-".join(["Hola", "mundo"])
9     >>> print(nueva_cadena)
10    'Hola-mundo'
```

❁ Listas: Las listas son una estructura de datos que permite almacenar una **colección ordenada y mutable de elementos**. Los elementos en una lista pueden ser de cualquier tipo de dato. Se definen con corchetes y los elementos se separan con comas. A diferencia de las tuplas, las listas son mutables, lo que significa que se pueden agregar, eliminar o modificar elementos. Para las listas, se pueden utilizar los métodos `append()` para agregar elementos al final de la lista, `remove()` para eliminar un elemento específico y `sort()` para ordenar los elementos en orden ascendente.

```
1     #Ejemplo de listas
2     >>> lista = [1]
3     >>> lista.append(2)
4     >>> print(lista)
5     [1,2]
6     >>> lista.remove(1)
7     >>> print(lista)
8     [2]
9     >>> lista2 = [3, 1, 4, 2]
10    >>> lista2.sort
11    >>> print(lista2)
12    [1, 2, 3, 4]
13    >>> lista[0]=10
14    >>> print(lista2)
15    [10,2,3,4]
16    >>> lista2[-1]=6.28
17    >>> print(lista2)
18    [10,2,3,6.28]
```

❁ Tuplas: Las tuplas son similares a las listas, pero son inmutables, lo que significa que una vez definidos, no se pueden modificar. Se definen con paréntesis y los elementos se

separan con comas. En las tuplas, se pueden acceder a los elementos utilizando el índice y no hay métodos para modificar los elementos de una tupla.

```
1 # Ejemplo de tuplas
2 >>> tupla = (1, 2, 3)
3 >>> print(tupla[0])      # Imprime 1
4 1
5 >>> tupla[0] = 0 # no funciona porque tuplas son inmutables
```

❁ Conjuntos: Los conjuntos son una colección **no ordenada de elementos únicos**. Se definen con llaves y los elementos se separan con comas. Para los conjuntos, se pueden utilizar los métodos `add()` para agregar elementos al conjunto y `remove()` para eliminar elementos del conjunto.

```
1 # Ejemplo de conjuntos
2 >>> conjunto = {1, 2, 3}
3 >>> conjunto.add(4)
4 >>> print(conjunto)
5 {1, 2, 3, 4}
6 >>> conjunto.remove(2)
7 >>> print(conjunto)
8 {1, 3, 4}
```

❁ Diccionarios: Los diccionarios son una colección no ordenada y mutable de pares clave:valor. Se definen con llaves y los pares clave:valor se separan con comas. Los diccionarios se utilizan comúnmente para almacenar información estructurada. Para los diccionarios, se pueden acceder a los elementos utilizando la clave y los métodos `keys()` y `values()` para obtener las claves y los valores, respectivamente.

```
1 # Ejemplo de diccionarios
2 >>> diccionario = {"clave1": "valor1", "clave2": "valor2"}
3 >>> print(diccionario["clave1"])
4 "valor1"
5 >>> print(diccionario.keys())
6 ["clave1", "clave2"]
7 >>> print(diccionario.values())
8 ["valor1", "valor2"]
```

## Capítulo 6 Estructuras de Control (Python)

Una persona que programa quedó atrapada en la ducha, las instrucciones en la botella de shampoo decían: Hacer espuma, enjuagar, repetir.

Anónimo

### 6.1. ESTRUCTURA DE CONTROL CONDICIONAL: IF-ELSE

Si las variables le dan "memoria" al programa, los condicionales le otorgan "inteligencia". Gracias a los condicionales, el computador puede "tomar una decisión", dadas las condiciones, respecto de si ejecutar o no un conjunto de instrucciones. La forma más simple de un condicional es así:

Si se cumple una condición lógica, ejecute el conjunto 1 de instrucciones Sino se cumplen, ejecuta el conjunto 2 de instrucciones

#### Ejemplo 6.1

**Si (el pronóstico para hoy es soleado)**<sup>(condición)</sup>

1- me alegro(instrucción 1 del Si)

2- me visto con ropa ligera(instrucción 2 del Si)

**Si no**<sup>(Nota que aquí no hay condición explícita)</sup>

1- me desmotiva(instrucción 1 del Sino)

2- me visto con ropa abrigada(instrucción 2 del Sino)

Nota que al decir "Si no", no necesito establecer una condición para que se ejecute el conjunto de instrucciones. Por lógica, "Si no", significa "Si no se cumple la condición anteriormente establecida".

Existen diferentes tipos de estructuras de control, pero todas ellas tienen en común el uso de condiciones lógicas para determinar si se ejecutan o no las instrucciones asociadas a ellas.

Normalmente, las condiciones lógicas permiten evaluar una (o más) preguntas sobre

el valor de las variables. Por ejemplo, la condición lógica (`tmp > 32`) es **VERDADERA** si la variable `tmp` tiene almacenado un valor mayor a `32` y **FALSA** si tiene almacenado un valor \_mayor o igual a `32`. También podemos emplear condiciones compuestas usando de los operadores lógicos **AND**, **OR**. Por ejemplo, la condición lógica (`tmp > 32 AND nivel_agua < 20`) es **VERDADERA** si la variable `tmp` tiene almacenado un valor mayor a `32` y si la variable `nivel_agua` es menor a `20`. De lo contrario, la condición lógica es **FALSA**.

La sintaxis de la estructura condicional estándar en un condicional en Python es la siguiente:

```
1  if ( <condición(es) una o más> ) :
2      // instrucciones del bloque if
3  else:
4      // instrucciones del bloque else
```

La estructura general se compone de la siguiente forma<sup>(1)</sup>: El `if` tiene asociada una condición lógica, que puede ser **VERDADERA** o **FALSA**. Si la condición es **VERDADERA**, entonces el control del programa ejecutará el bloque de instrucciones `if`. En cambio, si la condición es **FALSA**, el control del programa ejecutará el bloque de instrucciones `else`.

Note que no existe una condición explícita para el `else`. Sólo el `if` tiene asociada una condición lógica explícita. La condición del `else` es implícita (es la negación de la condición del `if`). Es decir, el bloque de instrucciones `else` se ejecuta si no se cumple la condición explícita asociada al `if`.

Notar también que cuando el control del programa escoge un camino, no puede regresar a ejecutar las instrucciones del bloque que no eligió. Por ello la persona que programa debe definir correctamente el curso de acción que seguirá su programa cuando se encuentre en ejecución.

Aparte de la estructura estándar `if...else...`, existen varias maneras de usar la estructura condicional. Las describiremos a continuación.

---

<sup>(1)</sup>Supón que este código está contenido en un programa.



### 6.1.1. if...

Aunque una manera de referirse a los condicionales es llamándolos `if-else`, no es obligación usar un `else` por cada `if` del programa. Cuando es necesario actuar (ejecutar instrucciones) en caso que se cumpla una condición (pero no actuar en caso que no se cumpla), se usa sólo `if`.

#### Ejemplo 6.2

se solicita al usuario que ingrese nombre, apellido y edad. Si tiene menos de 18 años, se despliega un mensaje determinado. De lo contrario, no hay acciones que ejecutar.

```
1 print("Ingresa por favor tu nombre, apellido y edad" )
2 nombre = input()
3 apellido = input()
4 edad = int( input() )
5 if ( edad<18 ):
6     print( nombre, " ,recuerda pedir el código de autorización
7         de tu tutor" )
```

Notar que, si la persona tiene 18 años o más, no desplegaremos mensaje alguno. Por lo tanto, en este caso el `else` no es necesario.

### if...else.. (Sin anidación)

Cuando hay dos bloques de acciones distintas que deben realizarse dependiendo del cumplimiento (o no) la condición lógica asociada al `if`, entonces se usa la estructura `if...else....` En esta sección veremos la estructura simple (un `if` seguido de un `else`). Esto ocurre cuando sólo se debe tomar una única decisión.

#### Ejemplo 6.3

Se solicita al usuario que ingrese un número y se despliega por pantalla si acaso el número es o no múltiplo de 3.

```
1 print( "Ingresa un número por favor" )
2 num = int( input() )
```

```

3     if ( num%3 == 0 ):
4         print( "El número ingresado es múltiplo de 3" )
5     else:
6         print( "El número ingresado no es múltiplo de 3" )
7

```

### if...else..(ANIDADOS)

Cuando es necesario volver a tomar una decisión que depende del resultado de una decisión anterior, se usan estructuras condicionales anidadas (una dentro de otra).

#### Ejemplo 6.4

En un restaurante hay 2 menús: El menú 1 es cazuela y el Menú 2 es pescado frito (elegir entre el Menú 1 y el Menú 2 corresponde a la primera decisión). Si eliges el Menú 1, debes luego tomar una segunda decisión: elegir si quieres cazuela de vacuno o de ave (esta segunda decisión solo aparece si la primera decisión fue Menú 1). En cambio, si eliges el Menú 2, debes luego decidir si quieres congrio o reineta.

Este tipo de caso se puede visualizar como un árbol de decisión, como se muestra en la Figura 6.1. Un posible código para el problema anterior sería el siguiente:

```

1  print( "Elige tu menú: 1. Cazuela 2. Pescado frito" )
2  opcion = int( input() )
3  if ( opcion == 1 ):
4      print( "Ingresa 1 si quieres cazuela de vacuno y 2 si
5      quieres cazuela de ave")
6      cazuela = int( input() )
7      if ( cazuela == 1 ):
8          print( "Preparando cazuela de vacuno" )
9      else:
10         print( "Preparando cazuela de ave" )
11 else:
12     print( "Ingresa 1 si quieres congrio y 2 si quieres
13     reineta" )

```

```

14 pescado = int( input() )
15 if ( pescado == 1 ):
16     print( "Preparando congrio frito" )
17 else:
18     print( "Preparando reineta frita" )

```

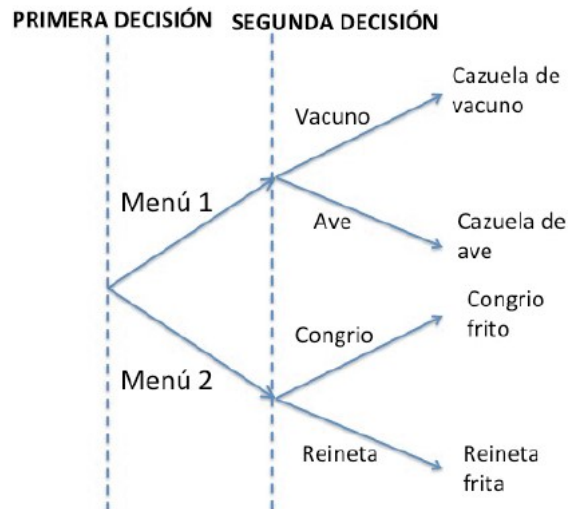


Figura 6.1. Ejemplo de un árbol de decisión

Se usa el adjetivo "anidado" porque hay una estructura `if...else...` dentro (anidada) de otra. No es necesario que haya exactamente una estructura `if...else...` dentro del `if` y del `else`. Se pueden usar combinaciones distintas.

#### Ejemplo 6.5

```

1 if ( <condición_1> )
2     if ( <condición_2> ) :
3         //entra solo si condición 1 Y condición 2 son verdaderas
4     else:
5         //entra solo si condición 1 es verdadera Y 2
6         es FALSA
7 else:
8     // entra si la condición 1 es FALSA. Nunca evalúa
9     la condición 2

```

Note que no todas las decisiones múltiples corresponden a `if...else...` anidados. Si una se-

gunda decisión no depende del resultado de la primera, no se trata de estructuras anidadas, else secuenciales (una después de la otra) como veremos a continuación.

### VARIOS if... (SIN else)

Esta estructura se usa generalmente para construir un menú donde el usuario debe elegir una de las opciones. Si son pocas opciones (2 – 4), se suele usar varios if.

#### Ejemplo 6.6

Pedirle al usuario que seleccione su rango de edad y desplegar un mensaje distinto dependiendo de la elección del usuario.

```

1  print( "Seleccione su rango de edad" )
2  print( "1. Menor de 18 años" )
3  print( "2. Entre 18 y 84 años, ambos inclusive" )
4  print( "3. Mayor de 85 años" )
5  respuesta = int( input() )
6  if (respuesta == 1):
7      print("Lo siento, eres menor de edad." )
8
9  if (respuesta == 2):
10     print("Bienvenido!!" )
11
12  if (respuesta == 3):
13     print("Volumen no apto para sus oídos" )

```

Notar como esta estructura obliga al computador a evaluar todas las opciones, aún cuando el usuario haya elegido la primera, algunos prefieren de todos modos usar una estructura if...else... en estos casos:

```

1  print( "Seleccione su rango de edad" )
2  print( "1. Menor de 18 años" )
3  print( "2. Entre 18 y 84 años, ambos inclusive" )
4  print( "3. Mayor de 85 años" )
5  respuesta = int( input() )
6  if (respuesta == 1):

```

```

7     print( "Lo siento, eres menor de edad." )
8 else:
9     if (respuesta == 2):
10        print( "Bienvenido!!" )
11    else:
12        print( "Volumen no apto para sus oídos" )

```

Ya que así, si el usuario elige la respuesta 1, el resto del código no se ejecuta. En este curso, cualquiera de las 2 opciones se considera correcta (aun cuando la segunda es más rápida de ejecutar, pero en un programa tan pequeño como este la diferencia es imperceptible).

### 6.1.2. PROBLEMAS PROPUESTOS

1. Para el siguiente código en Python:

```

1     \#Algoritmo Acceso
2     print("Ingresa tu edad:")
3     edad = int(input())
4     if (edad<18):
5         print("Acceso denegado")
6     else:
7         if (edad<25) :
8             print("Pase por caja 1")
9         else:
10            if (edad<30) :
11                print("Pase por caja 2")
12            else:
13                print("Pase por caja 3")
14

```

- I. Escribe lo que se despliega en pantalla al ejecutar código cuando el usuario ingresa el valor 25.
- II. ¿Qué conjunto de valores hace que el programa despliegue el mensaje “Pase por caja 3”?

2. Escriba el código de un algoritmo que solicite las notas de dos pruebas, con ellas calcule el promedio y luego despliegue 3 mensajes distintos dependiendo del promedio:

- a. "Felicitaciones, vas camino a aprobar" (si el promedio es mayor o igual a 4.0);
  - b. "Atención, vas camino a reprobar" (promedio mayor o igual a 3.0 pero menor a 4.0);
  - c. "Pocas posibilidades de aprobar" (para promedio menor a 3.0);
3. Escribe el código de un algoritmo que solicite por pantalla el número entero A, y luego el número entero B. Luego, debe verificar si  $\frac{A}{B}$  produce una división entera (es decir, el resto de la división es 0). Si este es el caso, deberá mostrar por pantalla "B divide exactamente a A". En caso contrario "B no divide a A en forma".
4. Escribe el código de un algoritmo que solicita los valores a,b y c de una ecuación cuadrática de la forma:

$$y = ax^2 + bx + c$$

y luego despliegue por pantalla los siguientes mensajes:

- ❁ "No hay soluciones reales" (si el discriminante es negativo)
- ❁ "Hay una solución real y es igual a " y desplegar el valor (cuando el discriminante es igual a 0)
- ❁ "Hay dos soluciones reales iguales a " y desplegar sus valores (cuando el discriminante es positivo).

Recuerda que las soluciones se calculan con la siguiente fórmula.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

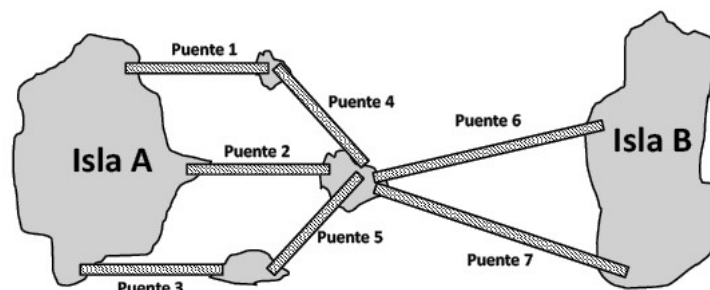


Figura 6.2

5. La Figura 6.2 muestra un sistema de puentes que conecta las islas A y B. Los puentes son de tipo mecano y el clima en esta área geográfica es muy hostil. Por lo tanto, es común que los puentes no estén operativos. Escribe un programa en Python que

solicite el estado de cada uno de los puentes (0: no operativo, 1: operativo) e informe si es posible viajar desde la isla A hasta la isla B (es decir, si existe al menos un camino de puentes en buen estado que conecte ambas islas).

6. Escribe un programa en Python que determine si una fecha (se ingresa día, mes y año) ocurre antes o después que otra.

## 6.2. Estructuras de Control Iterativa

Una estructura de control iterativa ó ciclo, permite realizar varias veces el mismo conjunto de operaciones. El ciclo se repite mientras se cumple alguna condición lógica.

En Python existen dos tipos de estructuras de repetición (ó iterativas): `while` y `for`.

### 6.2.1. Ciclos con estructura de control `While`

La estructura `while` evalúa una condición lógica antes de comenzar cada iteración. Mientras la condición es `verdad`, entonces se ejecutan las instrucciones que están dentro de la estructura `while`. En caso contrario, el ciclo termina y se sigue con la ejecución de las instrucciones escritas fuera del ciclo. Las instrucciones que se consideran dentro del ciclo son aquellas que están tabuladas de tal forma que se encuentran a la derecha de la palabra reservada `while`.

La sintaxis de la estructura `while` en Python es la siguiente:

```
1 while (condición):
2     <instrucciones del while>
3     <primera instrucción fuera del while>
```

Es decir, el conjunto de instrucciones se repetirá mientras la condición sea verdadera. Si la condición es falsa ya desde un principio, el conjunto de instrucciones no se ejecuta nunca.

#### Ejemplo 6.7

El siguiente código informa si el número ingresado por el usuario es positivo ó negativo, y se detiene cuando el usuario ingresa el número 0:

```
1 numeroUsuario=int(input("Ingresa un número, si quieres
2     terminar ingresa el 0: "))
```

```

3 while (numeroUsuario!=0):
4     if (numeroUsuario>0):
5         print("Es positivo")
6     else:
7         print("Es negativo")
8     numeroUsuario=int(input("Ingresa otro número (0 para
9         terminar): "))
10 print("Fin del ciclo while")

```

En este ejemplo, si se ingresa 0 la primera vez, la condición `(numero!=0)` será falsa. Por lo tanto, ni siquiera entra al bloque del `While`, saliendo del ciclo y continuando con la siguiente instrucción que es:

```
print("Fin del ciclo while");
```

### 6.2.2. Repetición con instrucción `for`

Una estructura de repetición `for`, permite iterar una ó más instrucciones. El número de veces que se repiten las instrucciones está definido por una variable de control (normalmente llamada contador). La sintaxis es la siguiente:

```

1 for nombreVariable in range(valInicial,valFinal,Paso):
2     <instrucciones>

```

Donde `nombreVariable` es el nombre de la variable de control (normalmente se le llama contador), `range(valInicial,valFinal, Paso)` genera los números sobre los cuales iterará la variable `nombreVariable`. Los números generados por la función `range` van desde `valInicial` hasta `valFinal-1`, con un incremento ó decremento de `Paso`.

Por ejemplo, para la siguiente estructura `for`:

```

1 for contador in range(1,6,1):
2     print("\#")

```

La función `range(1,6,1)` genera los números **1, 2, 3, 4, 5**. Por lo cual, contador tomará cada uno de esos valores. La siguiente tabla muestra cómo va cambiando el valor de la variable contador (columna izquierda) junto con lo que va apareciendo en pantalla (columna



derecha, entre paréntesis se muestran los símbolos que fueron impresos por pantalla en iteraciones anteriores):

Contador	Despliegue por Pantalla
1	#
2	(#)#
3	(##)#
4	(###)#
5	(####)#

Al finalizar la ejecución del ciclo, en pantalla quedan desplegados 5 símbolos #.

Notar que se podría realizar lo mismo, con un ciclo `while` de la siguiente manera:

```

1 Contador=1
2 for contador in range(1,6,1)
3     while (contador<6):
4         print("\#")
5         Contador=contador+1

```

Sin embargo, el uso de `for` es más compacto (2 líneas en vez de 4 líneas para este ejemplo de ciclo `while`).

### 6.3. Ciclos anidados

En su versión más simple, un ciclo anidado es un ciclo (ciclo interno) dentro de otro ciclo (ciclo externo), como muñecas rusas. La forma de llevar el tiempo es un ejemplo de ciclos anidados fácil de entender:

- ✿ el segundero del reloj es un ciclo anidado dentro del ciclo del minuterero: por cada vuelta completa que da el segundero (ciclo interno), el minuterero ejecuta una iteración (avanza un minuto, ciclo externo).
- ✿ Cada 30 días (ciclo “interno” de 30 ciclos), se avanza 1 mes (ciclo “externo”). Tres ciclos “externos (3 meses) corresponden a 90 días: 30 días en la primera iteración del ciclo externo (mes 1), 30 días en la segunda iteración del ciclo externo (mes 2) y 30 días en la tercera iteración del ciclo externo (mes 3).
- ✿ Cada vez que hay una repetición dentro de otra, se usan ciclos. Por ejemplo, para dibujar el siguiente patrón por pantalla:

#####

#####

#####

Basta con reconocer que hay 3 repeticiones de la misma línea: #####

Pero esta línea es, a su vez, otra repetición: se repite 5 veces el símbolo #. Por lo tanto, aquí se puede reconocer un ciclo anidado: el ciclo externo (línea) se repite 3 veces (3 líneas), mientras que el ciclo interno (símbolo) se repite 5 veces. Un ejemplo de código que implementa este patrón por pantalla es el siguiente:

```
1 for i in range(1,4,1):
2     for j in range(1,6,1):
3         print("#",end=" ")
4     print("")
```

Atención con la 3ra línea de código (`print("#",end=)`). Esta línea, permite mostrar por pantalla el carácter # y no saltar a la siguiente línea (`end=`). Por este motivo, la 6ta línea de código es: `print()`, lo que permite saltar una línea.

---

## 6.4. Hombros de Gigantes

*Si he llegado a ver tan lejos que otros es porque me subí a hombros de gigantes.*

Este capítulo está enteramente basado en el trabajo del profesor Miguel Carrasco PhD.

*Rendre à César ce qui est à César*

---

### 6.4.1. PROBLEMAS PROPUESTOS

1. Escribe un algoritmo que resuelva el siguiente problema: mostrar por pantalla los primeros 10 números pares. Resuelva este problema de 3 maneras distintas usando ciclo `while` y ciclo `for`.
2. Escribe un algoritmo que resuelva el siguiente problema: mostrar por pantalla las primeras 4 potencias de 2. Resuelve este problema de 3 maneras distintas: usando ciclo `while`, ciclo `for`.
3. Escribe un algoritmo que resuelva el siguiente problema: mostrar por pantalla los primeros

N números pares. N es un valor ingresado por el usuario. Resuelve este problema de 2 maneras distintas: usando ciclo `while` y ciclo `for`.

4. Escribe un algoritmo que resuelva el siguiente problema: desplegar por pantalla todos los números enteros que existen entre los números A y B (ambos inclusive) que ingresa el usuario. Nota que si A es menor que B, se despliegan los números en orden creciente. Pero si A es mayor que B, los números se despliegan en orden decreciente.
5. Escribe un algoritmo en Python que emule la solicitud de clave en una cajero automático: si el usuario equivoca la clave 3 veces consecutivas, el cajero muestra el mensaje “Tarjeta retenida”, se “traga” la tarjeta (esta última acción no la puedes emular en POython ya que la instrucción “Tragar tarjeta” no existe) y termina la ejecución del algoritmo. Si el usuario logra ingresar la clave válida en uno de los 3 primeros intentos, entonces aparece el mensaje “Bienvenid@ al servicio”. Supón que la clave válida se encuentra almacenada en una variable.
6. Escribe un algoritmo en Python que solicite números enteros al usuario, y vaya calculando la suma de todos ellos. La solicitud de números al usuario termina cuando se ingresa un -1, en cuyo caso se muestra por pantalla el resultado de la suma. Está permitido que el usuario ingrese como primer valor un -1, en cuyo caso el algoritmo termina inmediatamente,
7. Escribe el algoritmo que permite desplegar por pantalla los números pares que se encuentran entre dos números ingresados. El menor valor se almacena en la variable numInferior y el mayor en la variable numSuperior.
8. Escribe en código un algoritmo que solicite al usuario el número de empleados de una empresa y luego calcule el salario a pagar a cada uno de ellos. Para ello, para cada empleado, debe solicitar los siguientes datos: nombre, número de horas semanales trabajadas por el empleador (NHT), valor de la hora trabajada (VHC). Luego, el salario se calcula como  $NHT * VHC$ .
9. Escribe el código de un algoritmo que calcule el factorial de un número ingresado. El factorial de un número  $n$  (denominado  $n!$ ) se calcula como:  $n * (n-1) * (n-2) * (n-3) * \dots * 1$ .
10. Escribe el pseudo-código de un algoritmo que solicita, de uno en uno, los datos de género de un grupo de personas y determina el porcentaje de hombres, mujeres y personas que no

desean entregar esta información. Para género masculino el usuario debe ingresar 'm', 'f' para el femenino y 'n' si no desea entregar esta información. El programa termina cuando se ingresa el caracter 'x'.

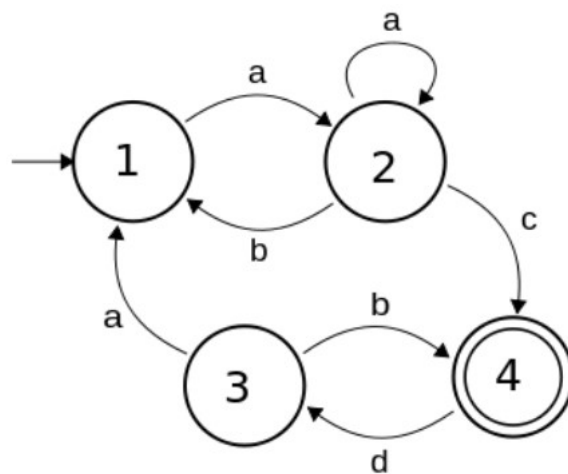


Figura 6.3

11. La figura 6.3 de abajo representa las diferentes posiciones (identificadas con números dentro de círculos) por las que atraviesa una máquina de un casino muy especial, con la que se puede jugar con fichas del tipo a, b, c y d. Escribe el pseudo código de un algoritmo que, comenzando por la posición inicial 1, solicite una ficha a jugar en cada instante. Dependiendo del valor de la ficha, se va avanzando en las posiciones hasta intentar llegar a la posición final 4. La siguiente posición de la máquina se determina a partir de la posición actual y la ficha que se inserta, y así sucesivamente.

Por ejemplo, si se está en la posición 1 y se ingresa una ficha a, se avanza a la posición 2 (se queda en 1 en otro caso).

El juego termina cuando el usuario presiona n. En ese caso, si la máquina se encuentra en la posición 4, el usuario gana el juego. Si se encuentra en cualquier otra posición, pierde.

12. Escribe el código que calcula el resto de la división entre dos enteros positivos A y B, para A y B indicados por el usuario.
13. Escribe el código que permite determinar si un número entero positivo P es primo. El valor de P se solicita al usuario.
14. Escribe el código que permite resolver la siguiente serie:  $1! + 2! + 3! + \dots + N!$ . El valor de N se solicita al usuario.
15. Realice un programa que determine si un número es palíndromo. Un número palíndromo

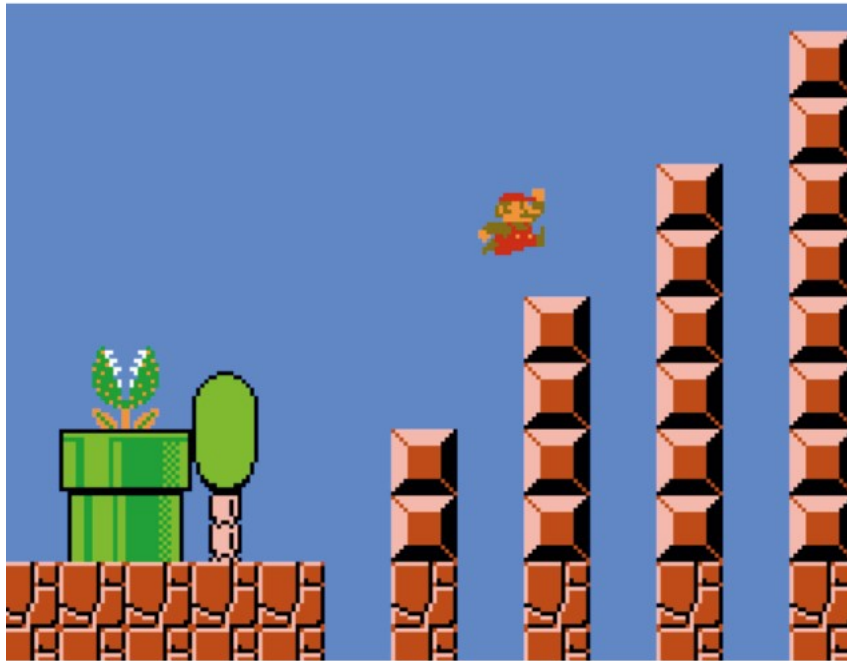


Figura 6.4

es aquel que se lee de la misma manera de izquierda a derecha, como de derecha a izquierda, por ejemplo: 131 y 7887.

16. Realice un programa que solicite números al usuario hasta que ingrese un 0 y devuelva el menor de todos los números ingresados que sean distintos a 0.
17. Escribe un programa que dibuje una matriz de símbolos #. El número de filas y el número de columnas de la matriz debe ser ingresado por el usuario.
18. En el mundo 8-1, del juego Super Mario Bros de Nintendo, el personaje debe superar unos obstáculos de escaleras y precipicios. A continuación se muestra una imagen con el obstáculo (una escalera): Si uno voltea la escalera en 180 grados y representa cada escalón con el símbolo #, el programa en Python que imprime la escalera desplegaría los siguientes caracteres por pantalla:

```
#####
```

```
#####
```

```
###
```

```
###
```

```
##  
##  
#  
#
```

Escriba un programa en Python que solicite al usuario ingresar el número de escalones, que despliegue además en pantalla la escalera correspondiente con #. En el caso del ejemplo anterior, el número de escalones es 4.

19. Para el mismo escenario de la pregunta anterior: Si uno voltea la escalera en 90 grados a la derecha, y representa cada escalón con el símbolo \* (asterisco), el programa en Python que imprime la escalera desplegaría los siguientes caracteres por pantalla:

```
* *  
* * * *  
* * * * * *  
* * * * * * * *
```

Escribe un programa en Python que pida al usuario ingresar el número de escalones y despliegue en pantalla la escalera correspondiente con asteriscos. En el caso del ejemplo anterior, el número de escalones es 4.

20. Para el mismo escenario de la pregunta anterior: Si uno no voltea la escalera, y representa cada escalón con el símbolo \* (asterisco), el programa en Python que imprime la escalera desplegaría los siguientes caracteres por pantalla:

```

          * *
        * * * *
      * * * * *
    * * * * * *
  
```

Escribe un programa en Python que pida al usuario ingresar el número de escalones y despliegue en pantalla la escalera correspondiente con asteriscos. En el caso del ejemplo anterior, el número de escalones es 4.

Notar que: No se preocupa si no le queda una escalera perfecta, esto es porque el ancho del carácter " " es más pequeño, que el ancho del carácter "\*"

21. Escriba un programa que despliegue en pantalla los valores de la función  $f(x, y)$  para el rango de  $x$  que indique el usuario:

$$f(x, y) = \sum_{k=x}^y k^2$$

El programa debe solicitar al usuario el valor menor y mayor del rango de  $[x, y]$ . Un ejemplo de la ejecución del programa es la siguiente:

```

1  Ingrese valor de x: 5
2  Ingrese valor de y: 55
3  La suma cuadratica del intervalo [x,y] es: 56950
  
```

22. Escriba un programa que despliegue en pantalla los valores de la función  $f(x)$  para el rango de  $x$  que indique el usuario:

$$f(x) = \sum_{k=0}^x k^2$$

El programa debe solicitar al usuario el valor menor y mayor del rango de  $x$ . Un ejemplo de la ejecución del programa es la siguiente:

```

1  Ingrese valor inferior de x: 2
  
```

```
2  Ingrese valor superior de x: 5
3  x          f(x)
4  =====
5  2          5
6  3          14
7  4          30
8  5          55
```

- 23.** Escribir el código que permite resolver la siguiente serie:  $1! + 2! + 3! + \dots + N!$ . El valor de  $N$  se solicita al usuario.
- 24.** Crear un programa que, para un número de filas ingresadas por el usuario, despliegue el siguiente patrón de números:

```
1
12
123
1234
12345
```

- 25.** Crear un programa que, para un número de filas ingresadas por el usuario, despliegue el siguiente patrón de números:

```
1
23
456
78910
```



26. Escriba el código que escriba los  $N$  primeros números primos. El valor de  $N$  se solicita al usuario.
27. Escriba un código que le permita conocer los  $N$  primeros términos de la serie de *Fibonacci*

# Capítulo 7 Funciones (Python)

Para entender la recursividad primero  
hay que entender la recursividad

Refrán popular

## 7.1. Implementación básica de funciones

Este capítulo introduce un concepto fundamental de programación: funciones. Estamos acostumbrados a las funciones de las matemáticas, donde normalmente definimos una función  $f(x)$  como una expresión matemática de  $x$ , y luego podemos evaluar  $f(x)$  para diferentes valores de  $x$ , trazar la curva  $y = f(x)$ , resolver ecuaciones del tipo  $f(x) = 0$ ; o dada otra función  $g(x)$  componer  $g(f(x))$ , entre otras cosas. Existe un concepto de función similar en la programación, donde una función es una pieza de código que toma una o más variables como entrada/*input*, lleva a cabo algunas operaciones usando estas variables y produce una salida a cambio. El concepto de función en programación es más general que en matemáticas, y no se restringe a números o expresiones matemáticas, pero la idea general es exactamente la misma[3].

### 7.1.1. Programando con funciones

Una función es un conjunto de instrucciones que se ejecutan cada vez que se "llama"/"invoca" a la función desde el programa. Ya hemos usado muchas funciones de Python en los capítulos anteriores. Quizás la función más usada en Python es la función `print()`, que permite desplegar información por pantalla. La segunda función más usada, podríamos decir que es `input()`, que permite capturar información desde la pantalla (y también desplegar una instrucción cuando se escribe un texto entre comillas como argumentos, entre los paréntesis redondos).

Durante el desarrollo de un código se puede llamar a funciones ya creadas por otros (como `print()`) o funciones creadas por nosotros.

En el primer caso (llamar a funciones ya creadas por otros), es necesario incluir la bi-

función	resultado
<code>ceil(x):</code>	retorna el entero más pequeño que sea igual o mayor que x
<code>exp(x):</code>	retorna el valor de $e^x$
<code>fabs(x):</code>	retorna el valor absoluto de x
<code>factorial(x):</code>	retorna el factorial de x
<code>floor(x):</code>	retorna el entero más grande que sea igual o menor que x
<code>log(x, baseLog=10):</code>	retorna el logaritmo de x en base baseLog
<code>pow(x,y):</code>	retorna x elevado y
<code>sqrt(x):</code>	retorna la raíz cuadrada de x

Cuadro 7.1. Algunas funciones del módulo **math**. Para listar todas las funciones de un módulo importado, escribe `help(nombreModulo)` e.g. `help(math)`

biblioteca (también llamado módulo y a veces librería) en la que esta función se encuentra declarada. Para esto, usamos el comando `import`. Por ejemplo, la función `print()` se encuentra declarada en el módulo **builtins** (`import builtins`). Sin embargo, no necesitamos escribir `import builtins` porque esa opción viene por defecto. Para llamar funciones de cualquier otra biblioteca, se debe escribir primero el nombre del módulo, seguido de un punto y luego el nombre de la función.

Por ejemplo:

```
1 >>> builtins.print("hola mundo")
```

Una de las bibliotecas más comunes en Python es **math**. Dicha biblioteca da acceso a múltiples funciones matemáticas. Las funciones matemáticas del módulo **math** son esencialmente las mismas a las que estamos acostumbrados con las matemáticas o al presionar botones en una calculadora:

Por ejemplo:

```
1 from math import *
2 x= 42
3 y=sin(x)*log(x)
```

Adicionalmente hemos trabajado con funciones no-matemáticas como `len` y `range`:

```
1 n = len(somelist)
2 for i in range(5, n, 2):
3     ...
```

y también usamos funciones que estaban vinculadas a objetos específicos a las que se

accedía con la sintaxis de puntos, por ejemplo, `append` para agregar elementos a una lista:

```
1 C = [5, 10, 40, 45]
2 C.append(50)
```

Este último tipo de función es bastante especial, ya que está vinculada a un objeto y opera directamente sobre ese objeto (`C.append` cambia a `C`). Estas funciones también se conocen como *métodos* y se considerarán con más detalle cuando hablemos de *Clases*, *Objetos* y *Métodos* (fuera de las pretensiones de este curso). En el presente capítulo consideraremos principalmente funciones regulares, no métodos. En Python, las funciones proporcionan un fácil acceso al código ya existente escrito por otros (por ejemplo, `sin(x)`). Hay mucho código de este tipo en Python, y casi todos los programas implican la importación de uno o más módulos y el uso de funciones predefinidas de ellos. Una ventaja de las funciones es que podemos usarlas sin saber nada de cómo están implementadas implementado. Todo lo que necesitamos saber es qué entra y qué sale, y la función se puede utilizar como una caja negra [3].

Las funciones también proporcionan una forma de reutilizar el código que hemos escrito, ya sea en proyectos anteriores o como parte del proyecto actual, y este es el enfoque principal de este capítulo. Las funciones nos permiten delegar responsabilidades y dividir un programa en tareas más pequeñas, lo cual es esencial para resolver todos los problemas de cierta complejidad. Como veremos más adelante en este capítulo, dividir un programa en funciones más pequeñas también es conveniente para probar y verificar que un programa funciona como debería. Podemos escribir pequeños fragmentos de código que prueben funciones individuales y garantizar que funcionen correctamente antes de poner las funciones, todas juntas en un programa completo. Si tales pruebas se realizan correctamente, podemos tener cierta confianza en que nuestro programa principal funciona como se espera [3].

Entonces, ¿cómo escribimos una función en Python? Comenzando con un ejemplo simple, considere la función matemática

$$A(n) = P \cdot (1 + r/100)^n$$

En el segundo caso que nombramos (para llamar a funciones hechas en casa), es ne-

cesario definir la función. Esto es: - definir el nombre de la función - definir sus eventuales argumentos de entrada - escribir el código que debe ejecutarse cuando se invoca a la función - definir el retornar de la función

Para valores dados  $P = 100$  y  $r = 5,0$ , podemos implementar esto en Python de la siguiente manera:

```
1  def cantidad(n):  
2      P = 100  
3      r = 5.0  
4      return P*(1+r/100)**n
```

Estas dos líneas de código son muy similares a los ejemplos del Capítulo anterior, pero contienen algunos conceptos nuevos que vale la pena mencionar. Comenzando con la primera línea, `def cantidad(n):` se llama el "encabezado de la función" y define la interfaz de la función. Todas las definiciones de funciones en Python comienzan con la palabra `def`, que es, simplemente, cómo le decimos a Python que el siguiente código define una función. Después de `def` viene el nombre de la función, seguido de paréntesis que contienen los argumentos de la función (a veces llamados parámetros). Nuestra función toma un solo argumento ( $n$ ), pero podemos definir funciones que toman múltiples argumentos, separándolos con comas. Los paréntesis deben estar allí, incluso si no queremos que la función tome ningún argumento, en cuyo caso dejaríamos los paréntesis vacíos [3].

Las líneas que siguen al encabezado de la función son el cuerpo de la función, que debe *indentarse* (un aforismo para decir que se les aplica sangría). La sangría tiene el mismo propósito que en los bucles vistos previamente; especificar qué líneas de código pertenecen dentro de la función o al cuerpo de la función. Las dos primeras líneas del cuerpo de la función son asignaciones regulares, pero dado que ocurren dentro de una función, definen las **variables locales**  $P$  y  $r$ . Las variables locales y el argumento  $n$ , se usan dentro de la función al igual que las variables regulares. Volveremos a este tema con más detalle más adelante. La última línea del cuerpo de la función comienza con la palabra clave `return`, que también es nueva en este capítulo y se usa para especificar la salida devuelta por la función. Es importante no confundir esta declaración de devolución con las declaraciones de impresión que usamos anteriormente [3]. El uso de `print` simplemente mostrará algo en la pantalla,

		Funciones que retornan valores	
		Si	No
Funciones que reciben parámetros	Si	<p><b>Descripción genérica:</b></p> <pre>def &lt;nombre&gt;(&lt;parametros&gt;):     &lt;instrucciones&gt;     return &lt;valor/variable&gt;</pre> <p><b>Ejemplo de declaración:</b></p> <pre>def promedio(a,b):     return (a+b)/2</pre> <p>Llamada desde programa:</p> <pre>prom = promedio(val1,val2)</pre>	<p><b>Descripción genérica:</b></p> <pre>def &lt;nombre&gt;(&lt;parametros&gt;):     &lt;instrucciones&gt;     return</pre> <p><b>Ejemplo de declaración:</b></p> <pre>def promedio(a,b):     print((a+b)/2)     return</pre> <p>Llamada desde programa:</p> <pre>promedio(val1,val2)</pre>
	No	<p><b>Descripción genérica:</b></p> <pre>def &lt;nombre&gt;():     &lt;instrucciones&gt;     return &lt;valor/variable&gt;</pre> <p><b>Ejemplo de declaración:</b></p> <pre>def promedio():     a = int(input("ingrese 1ra     nota"))     b = int(input("ingrese 2da     nota"))     return (a+b)/2</pre> <p>Llamada desde programa:</p> <pre>prom = promedio( )</pre>	<p><b>Descripción genérica:</b></p> <pre>def &lt;nombre&gt;():     &lt;instrucciones&gt;     return</pre> <p><b>Ejemplo de declaración:</b></p> <pre>def promedio():     a = int(input("ingrese 1ra     nota"))     b = int(input("ingrese 2da     nota"))     print((a+b)/2)     return</pre> <p>Llamada desde programa:</p> <pre>promedio( )</pre>

Figura 7.1. Tabla con estructura de funciones

mientras que `return` hace que la función proporcione una salida, que se puede considerar como una variable que se devuelve al código que llamó a la función. Considere, por ejemplo,

```
1 n = len(alguna lista)
```

donde `len` devuelve un número entero que se asignó a una variable `n`.

Otra cosa importante a tener en cuenta sobre el código anterior es que no hace mucho; de hecho, la definición de una función, por si sola, esencialmente hace nada antes de ser llamada<sup>(1)</sup>. El análogo a la definición de función en matemáticas es simplemente escribir una función  $f(x)$  como una expresión matemática. Esto define la función, pero no hay salida hasta que comenzamos a evaluar la función para algunos valores específicos de  $x$ . En programación, decimos que llamamos a la función cuando la usamos [3].

Al programar con funciones, es común referirse al *programa principal* al "tejido" que usamos para relacionar variables y otras funciones; básicamente a cada línea de código que no está dentro de una función. Al ejecutar el programa, solo se ejecutan las declaraciones en

<sup>(1)</sup> Esto no es del todo cierto, ya que definir la función crea un objeto *función*. Esto lo podemos verificar definiendo una función ficticia en el shell de Python y luego llamando a `dir()` para obtener una lista de variables definidas. Sin embargo, no se produce ningún resultado visible hasta que llamamos a la función, y olvidarse de llamar a la función es un error común al comenzar a programar con funciones.

el programa principal; el código dentro de las definiciones de funciones no se ejecuta hasta que incluimos una llamada a una función específica en el programa principal. Ya hemos llamado a funciones predefinidas como `sen`, `len`, etc., en capítulos anteriores, y una función que hemos escrito nosotros mismos se llama exactamente de la misma manera:

```
1  def cantidad(n):
2      P = 100
3      r = 5.0
4      return P*(1+r/100)**n
5  year1 = 10
6  a1 = cantidad(year1) # call
7  a2 = cantidad(5) # call
8
9  print(a1, a2)
10 print(cantidad(6)) # call
11 a_list = [cantidad(year) for year in range(11)] #multiple calls
```

La llamada `cantidad(n)` para algún argumento `n` devuelve un objeto **flotante (1)** o un *float*, lo que esencialmente significa que `cantidad(n)` es reemplazada por este objeto flotante. Por lo tanto, podemos llamar `cantidad(n)` en cualquier lugar donde se pueda usar un *float*. Tenga en cuenta que, a diferencia de muchos otros lenguajes de programación, Python no requiere que se especifique el tipo de argumentos de función. A juzgar solo por el encabezado de la función, el argumento de `cantidad(n)` podría ser cualquier tipo de variable. Sin embargo, al observar cómo se usa `n` dentro de la función, podemos decir que debe ser un número (entero o flotante). Si escribimos funciones más complejas donde los tipos de argumentos no son obvios, podemos insertar un comentario inmediatamente después del encabezado, lo que llamaremos **documentación**, para decirles a los usuarios cuáles deberían ser los argumentos. Volveremos al tema de la documentación más adelante [3].

## 7.2. Argumentos y variables locales

Al igual que en Matemáticas, podemos definir en Python, funciones con más de un argumento. En la función que hemos estado usando ¿qué pasa si queremos variar, no sólo `n`, si no que también `P` y `r`?

```
1 def cantidad(P, r, n):
2     return P*(1+r/100.0)**n
3
4 # sample calls:
5 a1 = cantidad(100, 5.0, 10)
6 a2 = cantidad(10, r= 3.0, n=6)
7 a3 = cantidad(r= 4, n = 2, P=100)
```

Dentro de una función, no hay distinción entre dichas **variables locales** y los argumentos entregados a la función. Los argumentos también se convierten en variables locales y se usan exactamente de la misma manera que cualquier variable que definamos dentro de la función. Sin embargo, existe una distinción importante entre variables locales y **variables globales**. Las variables definidas en el programa principal se convierten en variables globales, mientras que las variables definidas dentro de las funciones son locales. Las variables locales solo están definidas y disponibles dentro de una función, mientras que las variables globales se pueden usar en cualquier parte de un programa. Si tratamos de acceder a *P*, *r*, o *n* (por ejemplo, por `print(P)`) desde fuera de la función, simplemente obtendremos un mensaje de error indicando que la variable no está definida.

Los argumentos sin nombre se llaman **argumentos posicionales**, mientras que los argumentos con nombre se llaman **argumentos de palabra-clave** (en inglés *keyword arguments*). Los argumentos palabra-clave deben coincidir con la definición de la función, e.g. llamar `cantidad(100, 5.0, año=5)` no tiene sentido ya que `año` no está definido dentro de la función. Además, existe un orden de trabajo, un *keyword* no puede ir antes de un argumento posicional, `cantidad(100, 5.0, n=5)` está bien, pero `cantidad(P=100, 5.0, 5)` no y el programa enviará un mensaje de error.

La diferencia entre variables locales y globales radica en que los argumentos pasados a una función y las variables definidas dentro de ella se convierten en variables locales que solo son visibles dentro de la función. Sin embargo, las variables globales son accesibles tanto dentro como fuera de la función.

```
1 P = 100
2 r = 5.0
3 def cantidad(n):
```



```

4     r = 4.0
5     return P*(1+r/100)**n
6     print(cantidad(n=6))
7     print(r)

```

### Actividad 7.1

¿qué pasa si, dentro de la función, usamos el comando `global r` antes de la declaración de `r` (línea 4)?

La palabra clave `global` le dice a Python que queremos cambiar una variable global y no definir una nueva local. **Se debe minimizar el uso de variables globales dentro de las funciones y, en su lugar, definir todas las variables utilizadas dentro de una función como variables locales o como argumentos pasados a la función.** De manera similar, si queremos que la función cambie una variable global, resulta mejor hacer que la función devuelva esta variable, en lugar de usar la palabra `global`. Es difícil pensar en un solo ejemplo en el que usar `global` sea la mejor solución y, **en la práctica, nunca debería usarse.**

#### 7.2.1. Los valores de retorno múltiples se devuelven como una tupla

Para un ejemplo más relevante, supongamos que queremos implementar una función matemática para que se devuelvan tanto el valor de la función como su derivada. Considere, por ejemplo, la fórmula física simple que describe la altura alcanzada por un objeto en movimiento vertical:

$$y(t) = v_0(t) + \frac{1}{2} \cdot g \cdot t^2 \quad (7.1)$$

$$y'(t) = v_0 - g \cdot t \quad (7.2)$$

donde  $v_0$  es la velocidad inicial,  $g$  es la constante gravitatoria y  $t$  es el tiempo. La derivada de la función es la aceleración, dada por  $y'$ , y podemos implementar una función Python que devuelve tanto el valor de la función como la derivada:

```

1 def yfunc(t, v0):

```

```

2   g = 9,81
3   y = v0*t - 0,5*g*t**2
4   dydt = v0 - g*t
5   return y, dydt
6 ##
7 posicion, velocidad = yfunc(0.6, 3)

```

Notar que la antes de la llamada, hemos expresado que buscamos 2 elementos de salida, separados por una coma.

### Actividad 7.2

¿Qué sucede si en lugar de llamar 2 elementos de salida, llamamos sólo uno?

```

1   pos_vel = yfunc(0.6,3)
2   print(pos_vel)
3   print(type(pos_vel))

```

Una función puede entregar tantas salidas como sean necesarias, puede incluso no entregarlas en tupla, si no en una lista o algún otro formato que consideremos necesario.

### Ejemplo 7.1

Para un ejemplo de función más relevante, que surgirá con frecuencia en nuestra vida, considere una aproximación para  $-\ln(1-x)$  cuando  $n$  es finito y  $|x| < 1$ :

$$L(x; n) = \sum_{i=1}^n \frac{x^i}{i}$$

Para muchos propósitos, sería útil que, además del valor de la suma, la función devolviera el error de la aproximación, es decir,  $-\ln(1-x) - L(x; n)$ :

La función correspondiente de Python para  $L(x; n)$  sería algo como:

```

1 from math import log
2
3 def L(x,n):
4     s = 0
5     for i in range(1, n + 1):
6         s += x**i/i

```

```

7     value_of_sum=s
8
9     error = -log(1-x) - value_of_sum
10    return value_of_sum, error
11 ##### uso de la función
12 x = 0.5
13
14 print(L(x, 3))
15 print(L(x, 10))
16 print(L(x, 15))
17 print(-log(1-x))

```

El resultado de la declaración de impresión indica que la aproximación mejora a medida que aumenta el número de términos  $n$ , como es habitual en este tipo de series aproximadas.

### 7.3. Funciones Lambda

Las funciones lambda (expresiones lambda o funciones **anónimas**) son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño [2]. Resulta complicado explicar las diferencias, la documentación oficial nos dice: *las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una definición*

Hemos destacado la palabra anónimas ya que no es necesario proporcionar un nombre para las funciones lambda [1]

La sintaxis general para definir expresiones lambda

```

1 lambda parametro(s):return valores

```

e.g. `lambda x: x*x` entrega un número al cuadrado, mientras que `lambda x,y: np.sqrt(x*x + y*y)` entrega la distancia Euclidiana entre 2 números

Las funciones lambda pueden ser más complejas e.g. si usamos listas por comprensión; no conforme con eso, podemos definir las y evaluarlas en un sólo paso: recordemos nuestra

aproximación a  $\log\left(\frac{1}{1-x}\right)$

```
1 (lambda x, n: sum([(x**i)/i for i in range(1,n+1)]))(0.5,10)
```

Pero claro que siempre podemos nombrar nuestras expresiones lambda y sacarlas del anonimato

```
1 log_x= lambda x, n: sum([(x**i)/i for i in range(1,n+1)])
2 ##### uso de la función
3 x = 0.5
4
5 print(log_x(x, 3))
6 print(log_x(x, 10))
7 print(log_x(x, 15)-L(x,15)[0])
8 print(-log(1-x)-log_x(x, 15))
```

Las funciones lambda también aceptan `*args` y `**kwargs` y podemos alimentar funciones con expresiones lambda.

Algunos casos de uso son :

- ❁ Cuando tiene una función cuya expresión de retorno es una sola línea de código y no necesita hacer referencia a la función en otro lugar del mismo módulo, puede usar funciones lambda.
- ❁ Puede usar funciones lambda cuando use funciones nativas, como `map()`, `filter()` y `reduce()`.
- ❁ Las funciones Lambda pueden ser útiles para clasificar estructuras de datos de Python, como listas y diccionarios.

## 7.4. Recursividad

### Definición 7.1

La recursividad es una estrategia para solucionar problemas llamando a una función dentro de si misma.

La recursividad es un concepto difícil de entender en principio, pero luego de analizar diferentes problemas aparecen puntos comunes. En python, las funciones pueden llamarse

a sí mismas, cuando esto suceda decimos que la función es recursiva.

¿Cuándo usar recursividad?

- ✿ Cuando un problema se puede dividir en subproblemas idénticos, pero más pequeños.
- ✿ Para explorar el espacio en un problema de búsqueda.

## 7.5. Problemas Propuestos

1. El siguiente código pide al usuario su año de nacimiento y le indica cuántos años cumple este año (2023).

```
1 def edad():
2     print("En qué año naciste?")
3     anio=int(input())
4     print("Este año cumples ",2023-anio," años")
5     return
6
7 edad()
```

Transforma el código para que mantenga la misma funcionalidad (solicitar al usuario su año de nacimiento e indicarle cuántos años cumple el año 2023) pero con una nueva función edad que:

- a. recibe como argumento de entrada el año de nacimiento del usuario. La función no retorna valor alguno.
  - b. recibe como argumento de entrada el año de nacimiento del usuario y retorna el número de años que cumple el 2023.
2. Para el siguiente programa, escriba la función pedirDato() que solicita al usuario un número entero entre 1 y 100, ambos inclusive. La función debe validar que el número ingresado se encuentre en el rango. Si está en rango, retorna el valor del número ingresado. Sino, debe mantenerse pidiendo un número hasta que el usuario ingrese un valor válido.
  3. Escribe el código de la función tipoTriangulo() que recibe como argumentos de entrada las longitudes enteras de los 3 lados de un triángulo y retorna un 0 si el triangulo es escaleno, un 1 si es isósceles y un 2 si es equilátero.

4. Escriba un programa que le pida al usuario que ingrese la longitud de los lados de tantos triángulos como desee. El usuario debe indicar que ha finalizado el ingreso de datos ingresando un cero (0) en cualquiera de los campos. Al finalizar el ingreso de datos, el programa debe indicar el total de triángulos ingresados y cuántos triángulos de cada tipo fueron ingresados.
5. Escriba una función que reciba como parámetro dos números (num1 y num2) y muestre por pantalla todos los números entre num1 y num2 que son divisibles por 7, pero no por 5.
6. Escriba una función que reciba como parámetro dos números (num1 y num2) y muestre por pantalla una matriz de dos dimensiones, correspondiente a la multiplicación de los números.
7. Escriba la función `esPrimo` que recibe un número y retornará 1 si el número es primo y 0 en caso contrario.
8. Escriba un programa que le solicite un número al usuario e imprima por pantalla todos los números primos que dividen al número ingresado. Este programa deberá hacer uso de la función definida en el punto anterior.
9. Escriba una función que reciba dos puntos del espacio y retorne la distancia euclideana entre los puntos. Hint, utilice las funciones de la librería `math`
  - ✿ `pow(x, y)`: retorna x elevado a y
  - ✿ `sqrt(x)`: retorna la raíz cuadrada de x
10. Escriba una función que reciba dos números enteros y genere un número aleatorio entero entre estos dos números. Para ello importe la librería `random` y utilice la función `random()` que devuelve un número aleatorio entre 0 y 1.
11. Cree una función que reciba un número entero y solicite al usuario adivinar el número ingresado. Para ello, la función deberá solicitar por teclado un número al usuario. En caso que el número haya sido adivinado, entonces se desplegará por pantalla, que el número fue adivinado y retornará el valor 1. Caso contrario, la función retornará 0 y desplegará por pantalla la información, “el número ingresado es mayor que el número a adivinar” ó “el número ingresado es menor que el número a adivinar”, según sea el caso.

12. Cree un juego donde tenga que adivinar un número en 5 oportunidades, utilizando las funciones creadas en los 2 ejercicios anteriores. En caso que lo adivine se desplegará **felicitaciones**, caso contrario se desplegará el número que debía de adivinar.
13. Implemente de un método recursivo que reciba un parámetro de tipo entero y luego llame en forma recursiva con el valor del parámetro menos 1, hasta llegar a cero.
14. Escriba una función que reciba 1 número enteros y calcule el factorial del número ingresado
15. Implemente un método recursivo para calcular el factorial de un número entero y contaste la eficiencia con la función del paso anterior
16. Escriba una función que reciba una lista numérica y ordene los elementos de la lista, usando recursividad.

---

## ? Referencias

- [1] Sebastian Bustamante. *Expresiones Lambda en Python*. Dic. de 2020. URL: <https://www.freecodecamp.org/espanol/news/expresiones-lambda-en-python/>.
- [2] *Funciones Lambda | El Libro De Python*. 2023. URL: <https://ellibrodepython.com/lambda-python>.
- [3] Joakim Sundnes. «Functions and Branching». En: ed. por Aslak Tveito et al. Springer, Cham, 2020, págs. 35-56. ISBN: 978-3-030-50356-7. DOI: [10.1007/978-3-030-50356-7\\_4](https://doi.org/10.1007/978-3-030-50356-7_4). URL: [https://link.springer.com/chapter/10.1007/978-3-030-50356-7\\_4](https://link.springer.com/chapter/10.1007/978-3-030-50356-7_4).