Lecturas de Cátedra

Pensamiento lógico y programación en Python

Martín Goin y Edith Lovos

EDITORIAL UNRN

Pensamiento lógico y programación en Python

Lecturas de Cátedra

Pensamiento lógico y programación en Python

Martín Goin y Edith Lovos

EDITORIAL UNRN

Índice

Agradecimientos 9
Prólogo 11

Capítulo 1. Introducción a la programación
1. 1. Hitos de la programación 15
1. 2. ¿Qué es programar? 16
1. 3. ¿Por qué aprender a programar? 17
1. 4. Algoritmo. Un primer acercamiento 18
1. 5. Cómo diseñar algoritmos 19
1. 6. Del algoritmo al programa 19
1. 7. Python: historia y características 20

Capítulo 2. Diagramación lógica y pseudocódigo
2. 1. Diagramación lógica 25
2. 2. ¿Qué son las variables? 27
2. 3. ¿Qué son las constantes? 30
2. 4. Estructuras de control 30
2. 5. Operadores relacionales 36
2. 6. Operadores lógicos 36
2. 7. Otros operadores matemáticos especiales 40
2. 8. Pseudocódigo 43
2. 9. Guía rápida del diagrama lógico a pseudocódigo 48
2. 10. Problemas: estructura de control de decisión 49

Capítulo 3. Estructuras de control de repetición e iteración
3. 1. Estructuras de control de repetición e iteración 53
3. 2. Estructura de control de repetición: sentencia «mientras» 55
3. 3. Variables contadoras y acumuladoras 57
3. 4. Problemas: estructura de control de repetición «mientras» 60
3. 5. Estructura de control de repetición e iteración: sentencia «para» 61
3. 6. Traza o prueba escritorio 66
3. 7. Máximos y mínimos 69
3. 8. Validación de datos de entrada 72
3. 9. Problemas: estructura de control de iteración «para» 73

Capítulo 4. Codificación
4. 1. Código Python 77
4. 2. Tipos de errores: sintácticos y semánticos 80
4. 3. Pasaje de pseudocódigo a código 81
4. 4. Tipos de datos numéricos 82

4. 5. Comentarios en Python 83
4. 6. Estructura de control de decisión «if», «else» y «elif» 83
4. 7. Problemas: estructura de control de decisión con Python 87
4. 8. Estructuras de control de repetición «for» 88
4. 9. Librerías 90
4. 10. Máximos y mínimos 92
4. 11. Estructura de control de repetición «while» 92
4. 12. Números aleatorios 96
4. 13. Problemas: estructura de control «for» y «while» 99

Capítulo 5. Arreglos: vectores y matrices
5. 1. Arreglos 103
5. 2. Vectores 103
5. 3. Problemas: arreglos y vectores 118
5. 4. Matrices 120
5. 5. Asignaciones condicionales 133
5. 6. Atributos de un arreglo 134
5. 7. Gráficas: Matplotlib 134
5. 8. Problemas: arreglos y matrices 139

Capítulo 6. Modularización: funciones y procedimientos
6. 1. Modularización 141
6. 2. Funciones 142
6. 3. Procedimientos 150
6. 4. Variables locales y globales 153
6. 5. Palabras reservadas 156
6. 6. Problemas 157

Autorías y colaboraciones 160

Agradecimientos

El libro no habría sido posible sin el apoyo de nuestra querida familia que acompaña en todo momento con paciencia en este largo camino.

Agradecemos a las y los docentes con quienes compartimos la tarea y el deseo de superarnos día a día, no solo en la enseñanza, sino también en la constante búsqueda de nuevas estrategias que nos permitan enriquecer nuestras prácticas didácticas.

También agradecemos a las y los estudiantes que nos recuerdan que enseñar también es aprender.

Prólogo

Martín Goin y Edith Lovos

En un mundo cada vez más digital, en el que la tecnología está en constante crecimiento y transformación, las habilidades de programación se han convertido en una práctica esencial. Muchas de las tecnologías digitales (productos y servicios) con las que interactuamos están naturalizadas o, en algunos casos, son imprescindibles en nuestra vida cotidiana.

El aprendizaje de la programación es un proceso en el que se ponen en juego habilidades técnicas y, también, otras consideradas *blandas*, como el trabajo en equipo, el autoaprendizaje, la capacidad de aprender del error, entre otras. A esto se suma que aprender a programar es una forma de desarrollar el pensamiento creativo que involucra la resolución de problemas usando algoritmos.

Este libro no está dirigido solo a quienes aspiran a convertirse en expertos desarrolladores, sino también para cualquier persona que desea entender cómo las aplicaciones de *software*, que utilizamos todos los días, cobran vida. Está diseñado como una guía que, paso a paso y progresivamente, permite comprender los conceptos fundamentales de la programación a través de Python, un lenguaje de uso muy extendido en el mundo profesional del desarrollo de *software*. Este surge como uno de los favoritos para la enseñanza y el aprendizaje, principalmente, porque es fácil de entender, versátil y, al mismo tiempo, poderoso.

Avanzar en este libro no solo permitirá el aprendizaje de la sintaxis y la semántica de Python a través de los diferentes ejemplos que se presentan, sino también el desarrollo del pensamiento lógico y creativo con la intención de resolver problemas. De este modo, se derrumba la idea de que la programación es compleja de aprender y de usar para la realización de ideas poniendo en evidencia que es accesible a cualquier persona, independientemente de su nivel de experiencia.

El libro funciona como un tutorial destinado a los primeros cursos de grado de carreras como licenciatura en Sistemas, licenciatura en Informática, Ingeniería Ambiental u otras que incluyen asignaturas vinculadas a la programación de computadoras. Está pensado para ayudar a desarrollar las habilidades necesarias para la apropiación del lenguaje Python como herramienta para codificar algoritmos. Para esto, están disponibles 65 ejemplos prácticos resueltos y 91 problemas estratégicamente pensados como desafíos que fortalecen la capacidad para resolverlos. Cada capítulo está diseñado con el fin de contribuir con el

anterior, de esta manera se asegura una comprensión sólida y accesible debido a las explicaciones detalladas y una batería de problemas que ayudan a aplicar lo aprendido.

El libro se divide en seis capítulos. Salvo el primero que expone los conceptos básicos de la programación y el lenguaje, el resto atiende y pondera aspectos prácticos para desarrollar algoritmos. Entonces, el capítulo 1, «Introducción a la programación», expone los hitos de la programación, define el concepto y la necesidad de su aprendizaje. Además, ofrece un acercamiento a los algoritmos, su diseño, el pasaje del algoritmo al programa, e incluye la historia y las características de Python.

Luego, el capítulo 2, que se titula «Diagramación lógica y pseudocódigo», profundiza en los siguientes temas: la diagramación lógica, las variables y constantes, el intercambio de variables, las estructuras de control (decisión simple, doble, anidadas, independientes), los operadores lógicos y relacionales, así como los operadores matemáticos especiales. Por último, se explica el pasaje del diagrama lógico al pseudocódigo.

En tanto, el capítulo 3, «Estructuras de control de repetición e iteración», se enfoca en las estructuras de repetición «mientras» y «para». También se abordan las variables contadoras y sumadoras (acumuladoras), la traza o prueba escritorio, los máximos y mínimos, las estructuras de decisión combinadas con bucles de repetición y la validación de datos de entrada.

La instalación y entorno Python se expone en el capítulo 4, que se titula «Codificación». Se explican temas como el modo *prompt* y la creación de archivo fuente .py, además de los errores sintácticos y semánticos que se pueden presentar. Asimismo, se enseña a convertir pseudocódigo a código, a utilizar las estructuras de decisión «if», «else», «elif», así como las de repetición «for», «while» y a encontrar los máximos y mínimos. Se detallan también los tipos de datos que Python soporta, además, cómo se incorporan los comentarios, las librerías y la generación de números aleatorios.

El capítulo 5, «Arreglos: vectores y matrices», aborda los arreglos unidimensionales y bidimensionales, los vectores y matrices, así como el uso de la librería NumPy, el ordenamiento y la obtención de los máximos y mínimos. También se explican los vectores paralelos y las operaciones aritméticas, las asignaciones condicionales, la copia y concatenación de arreglos, los filtros y las gráficas de funciones.

Por último, el capítulo 6, que se titula «Modularización: funciones y procedimientos», cubre temas como la modularización, los parámetros, las funciones y los procedimientos. También explica la combinación de funciones y procedimientos, desarrolla las variables locales y globales, y presenta las palabras reservadas.

Cabe aclarar que el libro no incluye todos los temas de un curso de programación, pues la propuesta es brindar los conocimientos básicos bajo el paradigma de la programación estructurada utilizando un lenguaje de alto nivel.

Como docentes de asignaturas sobre programación tanto en carreras de grado vinculadas a las ciencias informáticas como a otras, intentamos producir textos didácticos que acompañen a los estudiantes en sus primeros pasos. Con ese fin, recuperamos experiencias de la enseñanza y revisiones de otras obras que hemos producido como *Caminando junto al lenguaje C* y *Problemas y algoritmos. Un enfoque práctico*, ambas de la colección Lecturas de Cátedra de la Editorial UNRN de la Universidad Nacional de Río Negro. Estos materiales están disponibles en forma gratuita a través del sitio de la editorial: https://editorial.unrn.edu.ar/.

Para finalizar, invitamos a transitar el fascinante mundo de la programación y, sobre todo, a disfrutar cada momento del proceso en este recorrido con la certeza de que se generan conocimientos y desarrollan habilidades no solo para la resolución de problemas usando algoritmos, sino también para la aplicación de esta forma de pensamiento en otros ámbitos o situaciones de la vida cotidiana.

Autorías y filiaciones institucionales

Martín Mariano Julio Goin
Universidad Nacional de Río Negro, Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad Argentina. Río Negro, Argentina.

Edith Noemí Lovos
Universidad Nacional de Río Negro, Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad Argentina. Río Negro, Argentina.

Capítulo 1

Introducción a la programación

La mayor recompensa de mi trabajo no es lo que obtengo de él,
sino lo que me convierto mientras lo hago.
Margaret Hamilton

Python está diseñado para ser fácil de aprender y usar, lo
que permite a los principiantes comenzar rápidamente
sin verse abrumados por complejidades innecesarias.
Guido van Rossum

Temas:

Hitos de la programación. ¿Qué es programar? ¿Por qué aprender a programar? Algoritmo. Un primer acercamiento. Cómo diseñar algoritmos. Del algoritmo al programa. Python: historia y características.

1. 1. Hitos de la programación

Margaret Hamilton es científica, licenciada en Matemática y diplomada en Filosofía. Fue la directora de la división de ingeniería de la NASA que desarrolló el programa de la computadora a bordo del Apolo 11 (julio de 1969). Su aporte en el diseño de *software* fue extremadamente avanzado para la época, ya que se centró en la robustez y tolerancia de fallos. Podía detectar errores y corregirlos en el momento lo más rápido posible, priorizando la seguridad y la supervivencia de la tripulación del módulo lunar.

Hay que tener en cuenta que en esa década las computadoras todavía se componían de válvulas de vacío, es decir, que los ordenadores eran del tamaño de un aula. Por lo tanto, el desafío consistió en la reducción de los componentes para poder llevar adelante el programa aeroespacial. Los ingenieros de *hardware* lograron innovar, en 1966, una de las primeras computadoras de nombre Block II que usaba circuitos integrados (chips). El Block II tenía una velocidad de 2 MHz y una memoria de 125 kB, comparado a un teléfono celular es 16.000 veces más lento y con dos millones menos de capacidad de memoria.

Sin duda el Apolo Guidance Computer (AGC) fue una maravilla de la ingeniería y el *software* de Hamilton un éxito fundamental, sin ambos la misión habría sido imposible.

El trabajo de Hamilton en el programa Apolo 11 valió el título de ingeniera de *software* (carrera inexistente hasta el momento) y facilitó las bases para la industria moderna de desarrollo de este campo. Su innovación revolucionaria contribuyó a establecer a la carrera de Ingeniería de Software como una disciplina de estudio legítima.

Figura 1. 1. Hamilton con el código del programa de navegación del Apolo impreso en doce libros



Fuente: Draper Laboratory, 1969.

1. 2. ¿Qué es programar?

La acción de programar puede sintetizarse en saber usar un lenguaje de programación (sintaxis y semántica) para escribir (codificar) en el orden

adecuado las instrucciones que deseamos que ejecute una máquina (computadora, robot, etcétera). Así, antes de avanzar con el proceso de codificar, deberíamos tener en claro qué deseamos que haga la máquina. Ese propósito, que evidencia el término *qué*, está asociado a un problema que queremos resolver, por ejemplo, desde sumar dos números hasta simular el comportamiento de una bacteria en un determinado medio o aplicar un modelo matemático para analizar un conjunto de datos. Puesto de esta forma, programar implica resolver problemas construyendo soluciones (algoritmos) que pueden codificarse usando un lenguaje de programación (programa), y cuya ejecución en una computadora logra el resultado esperado. Entonces, antes de pasar a la codificación es necesario contar con el diseño del algoritmo.

Desde hace un tiempo es común escuchar que los algoritmos controlan o monitorean actividades de la vida cotidiana, desde la elección de una serie en una plataforma de *streaming* hasta el reconocimiento de individuos a partir de datos biométricos. También, surgen problemas relacionados con los algoritmos que implican el abordaje de los sesgos raciales, de género, etcétera, y la necesidad de considerar los efectos de un algoritmo mal diseñado o pensado solo para casos específicos. Estamos hablando de algoritmos de inteligencia artificial, es decir, que buscan simular el comportamiento humano y son entrenados con grandes conjuntos de datos que producimos los usuarios. Entonces, entender qué es un algoritmo, cómo funciona y cómo puede llegar a emular el comportamiento humano es parte de un saber necesario para participar en un mundo cada vez más digitalizado. Es un conocimiento que adquiere la misma importancia que saber leer y escribir.

1. 3. ¿Por qué aprender a programar?

La digitalización se integra en muchas de las actividades de la vida cotidiana debido a que están mediadas por las tecnologías:, el pago de una compra, el entretenimiento, el trabajo o la comunicación, solo por mencionar algunas. Muchas de las tecnologías digitales (productos y servicios) con las que interactuamos se han naturalizado o vuelto imprescindibles. El uso de dispositivos móviles (celular inteligente o computadora de bolsillo), el acceso a internet y la interacción con máquinas mediante lenguaje natural (como asistentes de voz del dispositivo móvil o *chatbots* de la plataforma de compras) abren un abanico de posibilidades (aprender, trabajar, socializar, entre otras). Sin embargo, también plantean importantes desafíos como las brechas de acceso, el impacto climático, social y económico de las tecnologías utilizadas, entre otras.

Sobre los desafíos, Serrano Bosquet y otros (2024), en una nota publicada en la revista *TecScience* del Instituto Tecnológico y de Estudios Superiores

de Monterrey, señalan que la principal necesidad es enfrentar las vulnerabilidades tecnológicas. Por ello, proponen la innovación social, es decir, poner los aportes de los avances tecnológicos al servicio de las personas. Para esto, es necesaria la participación activa de los ciudadanos en un entorno digitalizado y complejo que requiere de nuevas alfabetizaciones, como la digital, y habilidades tales como la capacidad de realizar abstracciones, de razonar críticamente, de poder establecer relaciones y de comunicarse en forma efectiva a través de diferentes medios y formatos, además de trabajar en equipos multidisciplinarios, entre otras (Cobo, 2019).

Ahora bien, una respuesta a la pregunta de por qué aprender a programar, más allá del desarrollo de las habilidades antes mencionadas, es que se trata de una oportunidad para el empoderamiento porque permite la posibilidad de pensar y diseñar soluciones a problemas existentes o descubrir nuevos desafíos cuyas soluciones pueden expresarse algorítmicamente.

El objetivo principal de este material es facilitar el aprendizaje usando Python, desde la comprensión de que la tecnología es fundamental para la intervención en el control del entorno digital (Fundación Sadosky, 2021) de una manera que resulte significativa.

¿Por qué enseñar Python? La razón principal es que se trata de un lenguaje ampliamente utilizado en el mundo laboral. Como se comentó anteriormente, en cada momento del día de una persona, el análisis de datos es relevante para casi cualquier actividad productiva dado que permite tomar decisiones informadas, optimizar operaciones y ofrecer mejores servicios a los usuarios. Python se destaca porque dispone de un conjunto de bibliotecas (NumPy, Panda, etcétera) diseñadas específicamente para el análisis y el procesamiento de datos que facilitan la programación. También posee herramientas específicas para la visualización de datos o la construcción de modelos de comportamiento y la automatización de tareas, sumado al soporte de una comunidad de desarrolladores global.

1. 4. Algoritmo. Un primer acercamiento

Un algoritmo es una secuencia de pasos finitos y bien definidos que permite alcanzar la solución a un problema (por ejemplo, preparar una taza de café u organizar una salida de *trekking*). Finito implica que el algoritmo inicia y termina, decir que es bien definido conlleva que los pasos son claros sin lugar a dobles interpretaciones para quien los debe seguir o ejecutar. Así, un algoritmo puede compararse, por ejemplo, al protocolo que lleva adelante un médico en una guardia para diagnosticar a un paciente. Cada paso del algoritmo es una acción que en algunos casos involucra la toma

de una decisión a partir de la evaluación de la información o la generación de nueva información o, inclusive, la repetición de acciones bajo ciertas condiciones. Desde el punto de vista de la ciencia informática, los pasos del algoritmo serán ejecutados por un autómata.

1. 5. Cómo diseñar algoritmos

A través de un algoritmo es posible procesar información (datos de entrada) para generar otra información (datos de salida). El avance en el diseño de un algoritmo implica el seguimiento de un proceso iterativo que puede resumirse en las siguientes etapas: análisis del problema, diseño del algoritmo y evaluación.

La etapa de análisis consiste en la comprensión profunda del problema a resolver. Esto implica la identificación de los datos de entrada y salida y las condiciones que deben cumplir (pre y postcondiciones). Un elemento central es la definición del conjunto de posibles datos de entrada y salida que permite validar el diseño. Este conjunto se conoce como *casos de prueba*. Es recomendable que su construcción se haga previa al diseño del algoritmo, como una forma de establecer cuáles son los resultados esperados. El caso debe tener en cuenta todos los posibles escenarios que puedan darse, y a partir de esto pensar en el diseño del algoritmo. Por otra parte, cuando el algoritmo se codifica (traduce) usando un lenguaje de programación –o sea, se convierte en un programa– los casos de prueba son una herramienta útil para depurar y mejorar el funcionamiento ante posibles errores.

1. 6. Del algoritmo al programa

Pasar del algoritmo al programa consiste en expresar la solución algorítmica siguiendo la sintaxis y semántica del lenguaje de programación elegido. Existen diferentes lenguajes, algunos visuales como Scratch, PilasBloques o App Inventor, por mencionar algunos. Estos permiten un primer acercamiento a la programación de una forma más amena, ya que la asociación entre bloques, colores y acciones facilita el aprendizaje en contraste con el modo texto. Es decir, es más simple construir una solución (algoritmo) usando elementos gráficos que escribiendo instrucciones (texto). En el caso de Python existe una plataforma de programación visual llamada EduBlocks que es útil para el primer acercamiento al lenguaje. En esta, mientras en una de las pantallas se va armando el programa con bloques, en la otra se puede visualizar la codificación en Python. De esta forma, colabora con la transición de la programación visual a la textual.

Además del lenguaje de programación, otro aspecto importante al momento de programar consiste en la elección del *entorno de desarrollo integrado* (IDE) que se va a utilizar para llevar adelante la tarea. Un IDE es una herramienta muy importante para el programador, ya que permite en una única herramienta integrar el editor de código, el compilador o intérprete que va a permitir ejecutar el programa junto con funcionalidades que van a facilitar la tarea de depuración para encontrar y corregir errores, entre otras (autocompletado, manejo de archivos, etcétera). El IDE oficial de Python se denomina *integrated development and learning environment* (IDLE). Su interfaz es muy simple (cuenta con lo básico para iniciarse en el desafío de escribir código) y fácil de usar. Este IDE viene por defecto con la instalación de Python.

1. 7. Python: historia y características

El lenguaje C fue creado a finales de la década de 1960 y principios de la de 1970 por el estadounidense Dennis M. Ritchie en los Laboratorios Bell. Veinte años después fue creado el lenguaje Python por el programador holandés Guido van Rossum en Stichting Mathematisch Centrum. Ambos desarrolladores coinciden en la intencionalidad de querer mejorar el lenguaje de programación que se estaba utilizando en su momento, en el primer caso la evolución del lenguaje B y en el segundo para suceder al lenguaje de programación ABC (desarrollado a principios de la década de 1980 como alternativa del BASIC). Además, existe una fuerte relación entre Python y el lenguaje C pues el primero ha implementado toda la librería estándar de C.

Figura 1.2. Guido van Rossum



Fuente: Hashemi, 2012.

La primera versión fue la 0.9.0 publicada en 1991, luego la 1.0 en 1994. Más tarde en el 2000, Python lanza la versión 2.0. Al siguiente año le fue otorgado a Van Rossum el Free Software Award (premio del *software* libre), en el 2005 lo contratan en Google y en el 2008 lanza la versión 3.0.[1]

Según los desarrolladores, el lenguaje Python se dirige a una subcultura dentro de la comunidad de programadores, caracterizada por sus propios métodos para escribir el código (Challenger Pérez, Díaz Ricardo y Becerra García, 2014). La filosofía de este lenguaje simplemente expone que mientras las ideas se mantengan e implementen de forma más sencilla y clara, mejores serán. Existe una lista denominada «el zen de Python» (Peters, 2004) que propone las características que debe cumplir la escritura del código en Python:

- Hermoso es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- El código legible cuenta.
- Casos especiales no son lo suficientemente especiales para romper las reglas.
- Casi siempre lo práctico vence a lo formal.
- Los errores no deben pasar nunca desapercibidos, a menos que se especifique este comportamiento.
- Ante una ambigüedad, descarte la tentación a adivinar.
- Debe haber una, y preferentemente una sola, manera obvia de lograr algo, aunque esta generalmente no está clara a primera vista a menos que seas un genio.
- Ahora es mejor que nunca, aunque a menudo es mejor nunca que justo ahora.
- Si la implementación es difícil de explicar, entonces es una mala idea.
- Si la implementación es fácil de explicar, entonces pudiera ser una buena idea.
- Los espacios de nombre son una buena idea, hagamos más de eso.

Respecto a las características técnicas es posible mencionar que Python es un lenguaje interpretado o de *script*, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos. Que es interpretado quiere

1 El curioso nombre que Van Rossum le otorgó al lenguaje proviene de un grupo de humoristas británicos de un programa de TV que emitía la BBC denominado Monty Python's Flying Circus (El Circo Ambulante de Monty Python), estrenado en octubre de 1969.

decir que no pasa por un lenguaje máquina como lo hacen los lenguajes C o C++, sino que se ejecuta directamente sin necesidad de compilación previa. La ventaja de los lenguajes compilados es que su ejecución es más rápida, pero los lenguajes interpretados son más flexibles y más portables.

Python es un lenguaje de tipado dinámico porque no requiere que las variables e identificadores se declaren, sino que los valores y las operaciones con las que se los puede manipular se determinan según el valor que se les ha asignado. La característica del tipado dinámico se refiere a que no necesitan ser declaradas las variables porque se determinan en el tiempo de ejecución. Sin embargo, si queremos cambiar el tipo de variable, es necesario realizar una conversión explícita previa al nuevo tipo, lo que determina que sea fuertemente tipado. El lenguaje es multiplataforma, es decir, puede ser utilizado en UNIX, Linux, DOS, Windows, OS/2, Mac OS, etcétera.

Además, Python es considerado un lenguaje multiparadigma porque soporta varios estilos de programación. Entre ellos se encuentran los paradigmas orientado a objetos, estructurado, funcional y, en menor medida, lógico. Esto permite a los desarrolladores elegir la forma más adecuada de resolver un problema según el contexto, ya sea utilizando un solo paradigma o combinando varios. La sintaxis del lenguaje Python es muy simple y adecuada para comenzar a programar, pues se asemeja mucho al lenguaje natural. Por eso se dice que está más cerca del usuario y no tanto del procesador. Esta característica implica que no es el lenguaje más eficiente para la programación de bajo nivel, ya que se aleja de las instrucciones que recibe el procesador. Sin embargo, su claridad y facilidad de desarrollo lo han convertido en un recurso elegido por compañías como Google, Facebook, Yahoo y la NASA.

En cuanto a la licencia, Python es certificado por el movimiento Open Source compatible con la GPL (GNU Public Licence) de la Free Software Foundation. La principal diferencia con respecto a la GPL es que no impone una restricción. Esto significa que un desarrollador puede crear programas derivados sin la necesidad de entregar su código fuente y distribuirlo como *software* libre o privado.

En sus comienzos, uno de los propósitos que tuvo el lenguaje fue su facilidad en el aprendizaje. Rossum manifestó que llegará el día en el que la programación se convierta en una asignatura tan importante como la matemática y la física para los currículos de la enseñanza media (Van Rossum, 1999). En el ámbito universitario, tanto la comunidad docente como la científica consideran a Python una herramienta simple para dar los primeros pasos en programación y eficiente para resolver problemas en áreas de las ciencias aplicadas como estadística, matemática, biología, química, física y, claro está, computación. En la actualidad es el primer lenguaje de programación más utilizado en el mundo.

Lista de referencias

Lista de referencias de imágenes

Draper Laboratory. (1969, 1 de enero). *Margaret Hamilton. Restoration.jpg*. Wikimedia Commons. https://en.wikipedia.org/wiki/File:Margaret_Hamilton_-_restoration.jpg

Hashemi, Mahmoud. (2012, 11 de marzo). *Guido van Rossum* (cropped).jpg. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Guido_van_Rossum_(6984267183)_(cropped).jpg

Lista de referencias bibliográficas

Challenger Pérez, Ivet, Yanet Díaz Ricardo y Antonio Becerra García. (2014). El lenguaje de programación Python. *Ciencias Holguín*, *20*(2), pp. 1-13.

Cobo, Cristobal. (2019). Ciudadanía digital y educación: nuevas ciudadanías para nuevos entornos. *Revista Mexicana de Bachillerato a Distancia*, *11*(21).

Fundación Sadosky. (2021). *Program.AR: Pensamiento computacional en la escuela*. https://www.fundacionsadosky.org.ar/proyectos/programar/

Peters, Tim. (19 de agosto de 2004). *The Zen of Python*. https://peps.python.org/pep-0020/

Rossum van, Guido. (1999). *Computer programming for everybody. A scouting expedition for the programmers of tomorrow*. The Python Software Foundation. https://www.python.org/doc/essays/cp4e/

Serrano Bosquet, Javier Francisco, María Lina Carreño Correa y Emanuele Giorgi. (2024, 8 de agosto). *Vulnerabilidad tecnológica: qué es y cómo detenerla*. TecScience. https://tecscience.tec.mx/es/divulgacion-ciencia/vulnerabilidad-tecnologica/

Capítulo 2

Diagramación lógica y pseudocódigo

En el futuro es posible que las computadoras
no pesen más de 1,5 toneladas.
Mecánica popular*, 1949

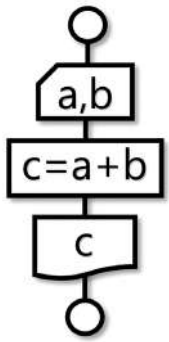
Temas:

Diagramación lógica. Las variables y constantes. Intercambio de variables. Estructuras de control (decisión simple, doble, anidadas, independientes). Operadores lógicos y relacionales. Operadores matemáticos especiales. Pasaje del diagrama lógico al pseudocódigo. Problemas.

2. 1. Diagramación lógica

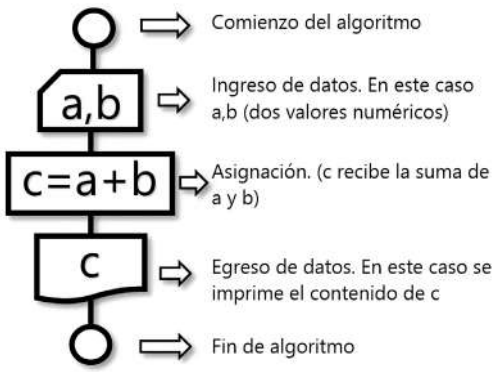
Una forma de acercarse y comprender el proceso de resolución de problemas usando algoritmos es a través del uso de gráficos. En este caso vemos la *diagramación lógica,* que es una forma de expresar gráficamente una solución algorítmica a través de un conjunto de figuras que tiene significado específico. Un ejemplo muy sencillo muestra el tema.

Ejemplo 1: Diseñar un algoritmo que permita el ingreso de dos números para mostrar su suma.



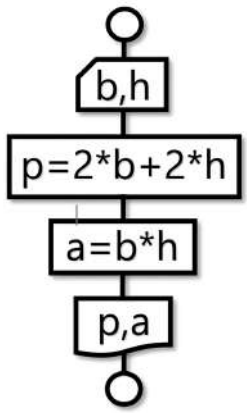
* Mecánica popular es una revista estadounidense dedicada a temas de ciencia y tecnología cuya primera publicación se efectuó en 1902. Pertenece a Hearst Communications. Desde 1947 comenzó a editarse para Latinoamérica.

Observamos símbolos que están conectados secuencialmente desde arriba hacia abajo y cada uno representa una instrucción. Vemos a continuación el significado de cada símbolo.



El flujo de datos está representado por las líneas verticales que unen los símbolos del algoritmo.

Ejemplo 2: Hallar el perímetro y el área de un rectángulo ingresando la base **b** y la altura **h**.



Es posible observar que a **p** se le asigna el cálculo del perímetro del rectángulo y el área en **a**.

Nota

Por ejemplo, si ingresamos **b** y **h** como 3 y 5 respectivamente, entonces la salida imprime lo siguiente: 16 15. Más adelante se mejora la salida agregando carteles (textos).

Desafío
Realizar un algoritmo para ingresar 4 números y que muestre el promedio.

Las operaciones aritméticas en los algoritmos deben expresarse siempre en una sola línea. Por ejemplo, si queremos realizar la siguiente operación:

$$c = \frac{-b + 9}{2a}$$

Entonces, debemos escribirlo de la siguiente manera:

$$c = (-b + 9) / (2 * a)$$

El asterisco (*) representa la multiplicación y la barra inclinada a derecha (/) es la división.

2. 2. ¿Qué son las variables?

La variable se puede entender como un contenedor de la información con la que trabaja un algoritmo. Este contenedor puede tener elementos de un determinado tipo o categoría. Pensemos como ejemplo una receta de cocina: tenemos una jarra con leche, un paquete de harina, un recipiente con capacidad para trabajar todos los ingredientes, molde para hornear, plato para servir, etcétera.

Receta casera para hacer una torta

1. Inicio.
2. Agregar la harina en un recipiente.
3. Agregar polvo para hornear.
4. Batir las claras de manera independiente.
5. Agregar las claras batidas en la mezcla.
6. Agregar leche y unas gotitas de esencia de vainilla, luego batir.
7. Batir la mantequilla con el azúcar de manera independiente.
8. Agregar la mantequilla batida a la mezcla principal.
9. Batir todos los ingredientes hasta que estén bien mezclados.
10. Engrasar y poner harina en el molde donde irá la mezcla.
11. Hornear a 350° por una hora.
12. Sacar del horno.
13. Dejar enfriar y voltear.
14. Servir en el plato.
15. Fin.

Así podemos observar que a los ingredientes los ubicamos en determinados tipos de contenedores, por ejemplo, de acuerdo a si son líquidos, sólidos, etcétera.

En términos más formales, una *variable* es un espacio en la memoria de la computadora que permite almacenar temporalmente información (dato) durante la ejecución del algoritmo (programa), y cuyo contenido puede cambiar durante esa ejecución. A su vez una variable se asocia a un tipo de información que se desea representar.

Un ejemplo es cuando se desea generar una solución algorítmica a partir de la edad, la altura y el peso de una persona para determinar, a partir de la información estandarizada, si tiene o no sobrepeso. En este problema identificamos tres datos de entrada: edad, altura y peso. Si bien son datos numéricos, pertenecen a diferentes conjuntos. La edad puede pensarse como número natural, la altura y el peso como valores fraccionarios.

Entonces, para identificar una variable es necesario darle un nombre o etiqueta, y además indicar el tipo de dato que representa. Los nombres o etiquetas de las variables siempre deben empezar con una letra y no pueden contener espacios en blanco. Si usamos más de un carácter para su identificación empezamos con una letra y luego podemos seguir con números o letras. Está permitido usar el guion bajo (_) entre medio.

Ejemplos válidos:
A, a, b1, a20, aa1, aA, b2c, promedio, sumatoria, a_1, b1_2

Ejemplos no válidos:
1b, 2c (comienzan con un número)
_S , A¿, h-y (contienen símbolos no válidos)
la temperatura (tiene espacio en blanco)
código (tiene tilde)

Se recomienda que el nombre de una variable sea representativo de la información, que sea corto y escrito en minúscula. Vemos algunos ejemplos:
Para una sumatoria → **suma, sum**
Para más de una sumatoria → **suma1, suma2, s1, s2**
Para la sumatoria de edades → **sumEdades, sum_edade**s
Para contar → **contador, cont, c1, c2, cantidad, cont_mujeres**
Para un promedio → **prom, prome, prom_edad, pro2, pTemp** (promedio de temperaturas)

Atención
En un algoritmo no debe existir más de una variable con el mismo nombre (identificación), aun cuando tengan distinta finalidad.

Las variables pueden contener valores numéricos y alfanuméricos, es decir, letras y símbolos, siempre y cuando estos últimos sean expresados entre comillas.

Ejemplos: **a1="20"** **nombre="Juan"** **simbolo="&"**

En cambio, las variables que contienen números no necesitan las comillas.

Ejemplos: **g1=20** **precio=129.85** **temperatura= -0.3**

Nota
El tipo de una variable permite determinar el conjunto de valores que puede tomar y las operaciones que se pueden realizar. En el caso de las variables numéricas es posible realizar operaciones aritméticas.

Importante
Una cuestión problemática con las variables es su valor o contenido inicial. En algunos lenguajes de programación, si una variable se usa sin antes haberle asignado un valor, tendrá un valor por defecto, mientras que en otros lenguajes eso podría generar un error durante la ejecución del programa. Entonces, para evitar estas situaciones, siempre que usamos una variable debemos asegurarnos de darle un valor inicial. Este valor dependerá del tipo de la variable y de la situación que se desea modelar.

2. 2. 1. Intercambio de variables

El intercambio (*swap*) de los valores de dos variables no es un procedimiento que pueda hacerse en forma directa, es decir, por ejemplo: si **a=8** y **b=2** entonces si **a=b** (le asignamos a **a** el valor de **b**, se pierde el valor original de **a**) y si luego hacemos **b=a** (le asignamos a **b** el valor de **a**), finalmente los valores de **a** y **b** serán el mismo (en este caso 2). Para solucionar esta situación debemos usar una variable auxiliar.
Para el caso anterior es: **c=a**, **a=b** y **b=c**. Entonces ahora tenemos **a=2** y **b=8**.

Atención
Recordar que solo podemos intercambiar variables que tengan el mismo tipo de contenido.

Importante
El tipo de una variable especifica el conjunto de valores que puede tomar y las operaciones que pueden hacerse.

2. 3. ¿Qué son las constantes?

La diferencia con las variables es que en el momento de la creación de una variable el valor del objeto es desconocido, mientras que para una constante no solo es conocido, sino que permanece inalterado durante la ejecución del procedimiento resolvente.

Las similitudes están en la forma de etiquetar (darles nombres), pero generalmente las letras se expresan en mayúsculas, por ejemplo: **PI, R1, EXPONENCIAL**.

2. 4. Estructuras de control

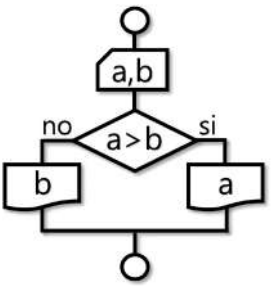
Las estructuras de control permiten alterar el flujo de ejecución secuencial de las instrucciones del algoritmo. El concepto de *flujo de control* se refiere al orden en que se ejecutan las sentencias o acciones (instrucciones) de un programa.

En los ejemplos anteriores se ha trabajado con un flujo lineal o secuencial.

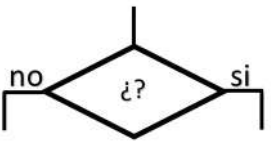
2. 4. 1. Estructura de control de decisión

La *estructura de control de decisión* permite bifurcar el flujo de ejecución del programa en función del resultado de la evaluación de una expresión lógica. Pensemos en un caso de la vida real, por ejemplo que estamos en una heladería y deseamos tomar un helado de menta granizada, pero si este sabor no está disponible no tomaremos nada. Aquí podemos observar que la acción de tomar el helado depende de la existencia del sabor menta granizada.
Para ver cómo se traduce este concepto de decisión al código, presentamos el siguiente caso.

Ejemplo 3: Crear un algoritmo que reciba dos números y, luego, muestre por mensaje cuál es el mayor.



Por ejemplo, probemos ingresar los números 3 y 7 (a **a** le corresponde 3 y a **b** le corresponde 7), entonces el camino que toma el algoritmo será el **no**, al ser falsa la condición **a>b**, en consecuencia su salida muestra el número 7.



Nota
El símbolo del rombo acostado representa la decisión según el resultado de la consulta (**¿?**) para saber su salida. Siempre toma uno de los dos caminos (por el **si** o por el **no**), pero nunca por ambos al mismo tiempo, ni tampoco por ninguno.

En el ejemplo 3 la condición es una relación de desigualdad, el signo es **>,** que significa *mayor que*.

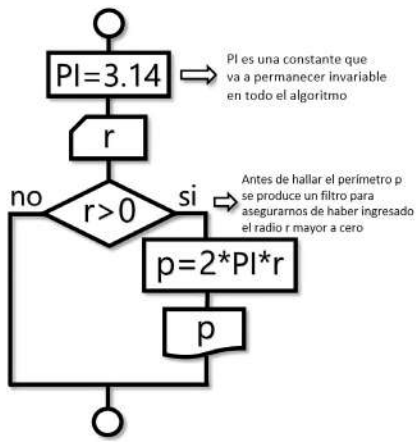
Recordemos
Un objeto es una variable cuando su valor puede modificarse y además posee un nombre que lo identifica y un tipo que describe su uso.
Un objeto es una constante cuando su valor no puede modificarse y además posee un nombre que lo identifica y un tipo que describe su uso.

Las estructuras de control de decisión se clasifican en simple, dobles, anidadas e independientes.

2. 4. 2. Estructura de control de decisión simple

Seguidamente, vemos un ejemplo de un algoritmo que utiliza una decisión simple.

> **Ejemplo 4:** Calcular y mostrar el perímetro de una circunferencia, siempre y cuando el radio que se ingresa sea mayor a 0. En caso que el valor radio no cumpla las condiciones, el algoritmo termina.



En este caso, si al ejecutar el algoritmo se ingresa un radio **r** de valor 0 o negativo, el programa simplemente termina, ya que solo funciona si la condición radio **> 0** (expresión lógica) es verdadera.

Cuando se cumple la condición dentro del rombo, el flujo de datos va para el lado del **si**, caso contrario se dirige hacia el **no**.

Importante
La estructura de control de decisión debe tener como mínimo una acción a ejecutar en caso que se cumpla la condición, es decir, en el camino del **si**.

Atención
En el ejemplo 4 usamos una constante **PI**. Generalmente, se usan nombres con mayúsculas a diferencia de las variables.

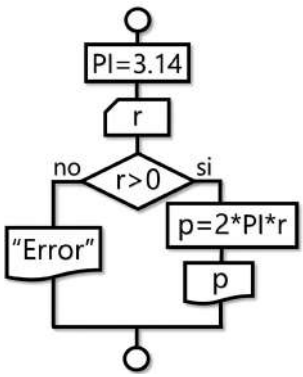
---

2. 4. 3. Estructura de control de decisión doble

La *estructura de control de decisión doble* es similar a la anterior con la salvedad de que se indican acciones no solo para la rama verdadera, sino también

para la falsa; es decir, en caso de que la expresión lógica sea verdadera se ejecuta un conjunto de acciones y en caso de que sea falsa se ejecuta otro grupo de acciones.

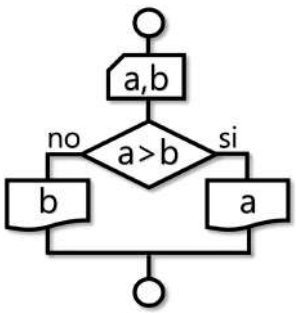
Ejemplo 5: Al igual que el ejemplo anterior pero en el caso de ingresar un radio que no cumpla con la condición de ser un valor mayor que cero, el algoritmo muestra un mensaje con la palabra **"Error"**.



Cuando la evaluación de la expresión lógica (radio > 0) es falsa, se ejecuta el camino del **no**. En otro caso, toma la rama del **si**.

Nota
Los mensajes en la salida siempre van entre comillas, en este caso "Error".

Volvemos a ver el ejemplo 3.



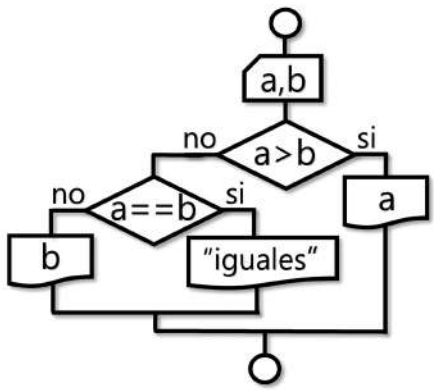
Nota
Este algoritmo también es de decisión doble. ¿Qué sucede si ingresamos dos números iguales? ¿Cuál sería el resultado?

La solución del problema anterior se realiza usando decisiones anidadas.

2. 4. 4. Estructura de control de decisión anidada

Existe la posibilidad de tener una decisión dentro de otra, que se llama *decisión anidada*, y se usa cuando tenemos más de una alternativa. Indicamos la solución del problema anterior (posible ingreso de dos números iguales) en el ejemplo 6.

Ejemplo 6: Mostrar por pantalla el número más grande (entre dos) ingresado por teclado. Si los dos números son iguales, debe mostrar el cartel "**iguales**".



Nota
Para detectar la igualdad en la decisión se usa el doble igual (**==**).
Dentro de las salidas se pueden imprimir mensajes (textos) siempre entre comillas: "**iguales**".

Desafío
Y si la primera condición **a>b** la cambiamos por **a==b**. ¿Cómo se modifica el algoritmo para que siga cumpliendo el objetivo?
Importante
En el ejemplo anterior existen tres posibles resultados: que **a** sea el mayor, que **a** y **b** sean iguales o que **b** sea el mayor. Es suficiente resolver el problema usando dos estructuras de decisión ya que una se toma por descarte como en este caso con **b** mayor. Siempre en la anidación se usan N-1 decisiones, siendo **N** la cantidad de casos en total.

A continuación, ofrecemos problemas para resolver. En estos casos, se deben identificar los datos de entrada y salida, sus tipos y las restricciones que están asociadas, antes de plantear la solución algorítmica:

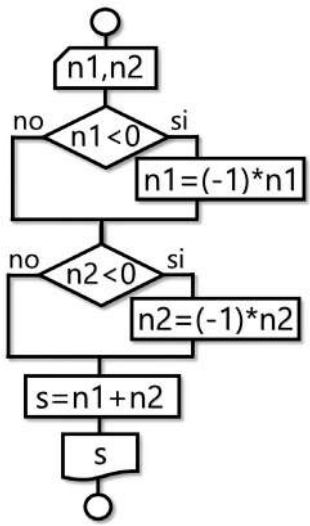
a. Crear un algoritmo que permita leer tres números enteros distintos y muestre el mayor.

b. Crear un algoritmo que permita el ingreso de tres números distintos y muestre el menor.
c. Crear un algoritmo que permita el ingreso de tres números distintos, y que luego determine y muestre el mayor, el del medio y el menor. Por ejemplo, si lee 3, -1, 6, la salida esperada sería 6, 3, -1.

2. 4. 5. Estructura de control de decisiones independientes

Las *decisiones independientes* son aquellas que se establecen sin depender una de otras, es decir, sin estar relacionadas. Simplemente se ubican una debajo de la otra y en cualquier orden, ya que el orden no afecta su ejecución. Vemos el ejemplo 7.

Ejemplo 7: Ingresar dos números por teclado y calcular la suma. En caso que los números sean negativos, previo a la suma se debe cambiar su signo.



Nota
En este caso tenemos dos decisiones simples que son independientes (podrían estar en cualquier orden). La operación **N=(-1)*N** hace cambiar el signo de negativo a positivo. La variable que se encuentra a la izquierda del signo igual (=) va a tener un nuevo valor (comúnmente decimos que estamos pisando el valor anterior).

2. 5. Operadores relacionales

La tabla muestra los distintos operadores de relación entre dos números.

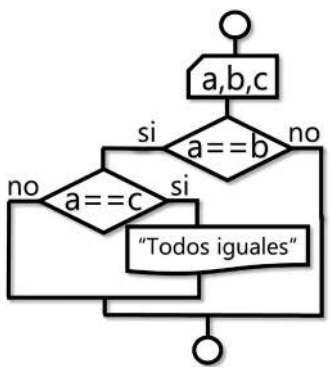
Operador	Significado	Equivalente en matemática
>	Mayor que	>
<	Menor que	<
>=	Mayor o igual que	≥
<=	Menor o igual que	≤
= =	Igual	=
!=	Distinto	≠

Nota
Es importante el orden lineal en los símbolos cuando interviene el igual: **>=**, **<=**, **!=**.

Estos operadores nos permiten establecer comparaciones entre variables. Por ejemplo, en el siguiente ejercicio se solicita establecer una comparación entre dos variables numéricas. Si **N1=8** y **N2=3** entonces decimos que la expresión **N1>N2** es verdadera y la expresión **N1<N2** es falsa. De igual manera **N1>=N2** también es verdadera, mientras que **N1<=N2** sigue siendo falsa.

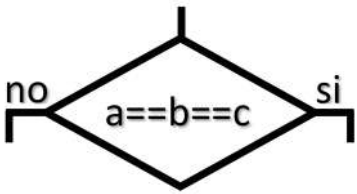
2. 6. Operadores lógicos

Ejemplo 8: Queremos averiguar si tres números son iguales utilizando decisiones. Una solución posible es:



En este caso, una decisión depende de la anterior.

Nota
La ubicación de los laterales de la decisión del **si** y el **no** es indistinta, lo importante es tomar en cuenta la lógica.



Atención
Esto es incorrecto. Las desigualdades o igualdades siempre deben relacionarse de a pares.

Para solucionar situaciones como estas, es necesario trabajar con los operadores lógicos «and», «or» y «not».

2. 6. 1. Conjunción lógica «and»

El denominado *producto lógico* contiene el operador «and» que es la evaluación simultánea del estado de verdad de las proposiciones lógicas involucradas. Así, por ejemplo, la expresión lógica: **a and b** será verdadera únicamente si **a** y **b** lo son. Cualquier otro estado dará como resultado el valor falso.

A continuación, se presenta la tabla de verdad del operador lógico «and».

Expresión Lógica		Resultado
a	b	a and b
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

2. 6. 2. Disyunción lógica inclusiva «or»

La denominada *suma lógica* contiene el operador lógico «or» que es la evaluación no simultánea del estado de verdad de las variables lógicas involucradas. Esto implica que al tener el estado verdadero por lo menos en una de las variables afectadas, la operación dará un resultado verdadero.

La expresión **a or b** será falsa únicamente cuando el estado de ambas variables sea falso. A continuación, se muestra la tabla de verdad del operador lógico «or».

Expresión Lógica		Resultado
a	b	a or b
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

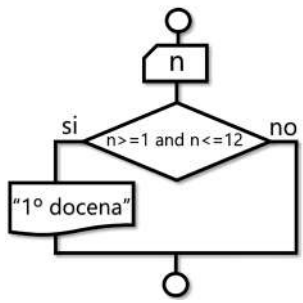
2. 6. 3. Negación o complemento lógico «not»

El efecto del operador «not» es negar el valor de la expresión lógica como se indica en la tabla.

Expresión Lógica	Resultado
a	not a
verdadero	falso
falso	verdadero

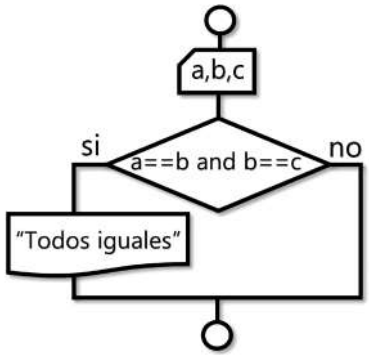
Vemos algunos ejemplos para poder comprender un poco más el concepto de operadores lógicos.

Ejemplo 9: Diseñar un algoritmo que solicite al usuario que ingrese un número natural y verifique que el número ingresado se encuentre dentro de la primera docena de números naturales, es decir, entre el 1 y el 12.



Nota
No es posible indicarlo como: **1<=n<=12**. Esto generaría un error porque no sería una expresión lógica.

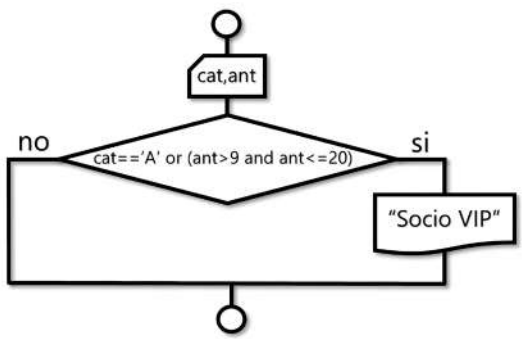
Volvamos al ejemplo 8, el que detecta si los tres números son iguales. Ahora se puede resolver usando el operador lógico «and» de la siguiente manera:



Nota
Si al evaluar **a==b** se obtiene un resultado falso, entonces no es necesario evaluar la expresión **a==c**. Esto se denomina *evaluación de circuito corto*. Si la primera expresión es falsa, como el operador que las vincula es un **and,** aunque la segunda sea verdadera, el resultado será falso, de ahí que no tenga sentido realizar la evaluación.

Una condición puede expresarse en forma lógica usando más de un operador (or, and, not). Presentamos el ejemplo 10.

Ejemplo 10: Se desea evaluar a un socio de un club deportivo, de acuerdo a su categoría (**A, B, C**) o su antigüedad en el club. Si el socio tiene categoría **A** o su antigüedad se encuentra entre los 10 y 20 años (incluidos), se debe informar la condición de "**Socio VIP**".



La expresión lógica **cat='A' or (ant>9 and ant<=20)** permite asegurar que el socio cumple con las condiciones impuestas.

Atención
El algoritmo funciona para categorías escritas en mayúsculas: ¿Qué pasaría si se ingresa '**a**' (minúscula)? ¿Nuestro socio será VIP? Si no funciona, ¿cómo podríamos solucionarlo?

Es común cometer errores usando operadores lógicos. Por ejemplo, en estos casos no se pueden sostener las relaciones que están entre paréntesis al mismo tiempo:

```
(a>b) and (a==b)
(n<10) and (n>20)
```

2. 7. Otros operadores matemáticos especiales

Además de los operadores básicos matemáticos como suma (+), resta (-), división (/) y producto (*), existen otros operadores que se irán trabajando a medida que avanzamos.

La tabla refleja algunos de los operadores matemáticos especiales para **n, n1, n2** variables numéricas.

Operación	Funcionalidad	Sintaxis
Potencia	pow	pow(n1,n2)
Raíz cuadrada	sqrt	sqrt(n)
Valor absoluto	abs	abs(n)
Seno	sin	sin(n)
Coseno	cos	cos(n)
Tangente	tan	tan(n)
Logaritmo natural	log	log(n)
Logaritmo base 10	log10	log10(n)
Exponencial e	exp	exp(n)
Módulo	%	n1%n2
Redondear	round	round(n1,n2)
Truncar	trunc	trunc(n)

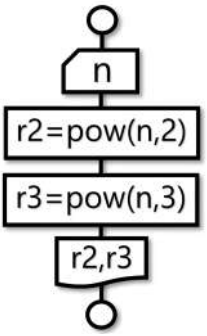
Nota
Recordar que las operaciones aritméticas deben generarse en una línea, característica propia de los lenguajes de programación.

El siguiente ejercicio tiene como consigna: Escribir en una línea cada una de las siguientes ecuaciones aritméticas utilizando operadores especiales matemáticos:

$$1) x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad 2) y = \sqrt[5]{\sqrt{-x+z}} \qquad 3)\ w = \frac{(x+y)2}{\left(m^2+1\right)^3}$$

A continuación, se muestran ejemplos con los operadores analizados.

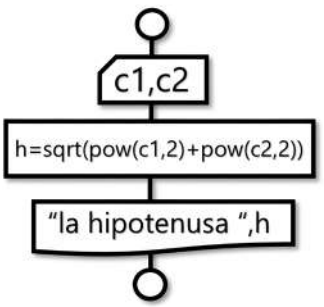
Ejemplo 11: Ingresar un valor numérico entero y que muestre el valor del mismo elevado al cuadrado y al cubo. Es decir, si **n** es el valor ingresado, la salida esperada será **n²** y **n³** en las variables **r2** y **r3**, respectivamente.



Importante

No es posible utilizar como identificadores de variables las palabras «cos», «sin», «tan», «abs», «pow», y otras por ser palabras reservadas del pseudocódigo. En otros capítulos aparecen más palabras reservadas.

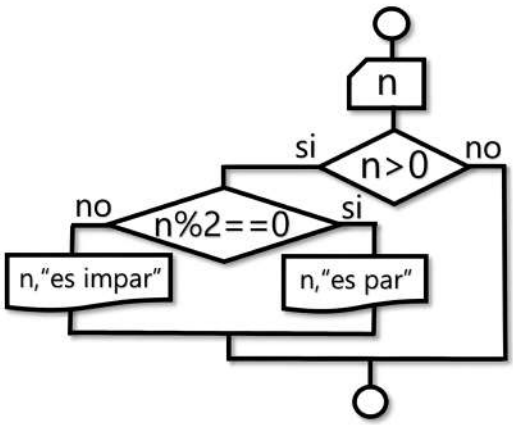
Ejemplo 12: Se ingresan por teclado los catetos **c1** y **c2** de un triángulo rectángulo, luego hallar y mostrar su hipotenusa **h**.



Nota
Las funciones pueden estar dentro de otras funciones como en el caso anterior en el que la suma de dos potencias **pow** está dentro de la raíz cuadrada **sqrt**. Se debe prestar mucha atención a los paréntesis.

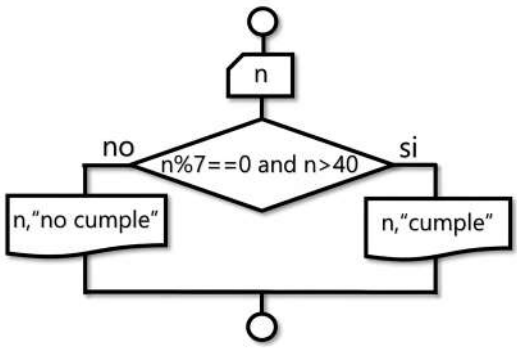
Atención
Otra forma de expresar la potencia es usando un doble asterisco (**). Por ejemplo: **m=n³** se puede escribir como **m=n**3** o utilizando la función **pow** antes vista **m=pow(n,3)**.

Ejemplo 13: Ingresar un número natural por teclado y determinar si el mismo es par o impar dando un mensaje por pantalla. Tener en cuenta que el número leído debe ser mayor a 0.



Nota
Primero, el algoritmo detecta que el valor del número ingresado es positivo y no nulo, en cuyo caso procede a validar si es par o impar. El operador **%** (módulo) da como resultado el resto de la división entera entre números enteros. Para este caso, cualquier número natural dividido 2 da como resto de la división un 1 o un 0 (cero), entonces los números pares tendrán como resultado un 0, mientras que los impares un 1.

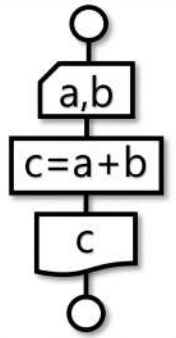
Ejemplo 14: Ingresar un número entero y determinar si cumple con ser divisible por 7 y mayor a 40. En cualquier caso, emitir el mensaje "**cumple**" o "**no cumple**", concatenado con el número **n**.



2. 8. Pseudocódigo

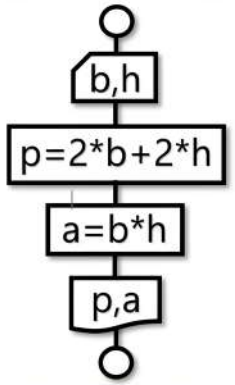
Otra forma de expresar los algoritmos es a través de texto con lenguaje coloquial (expresión humana), aunque más cercano al código. Con el propósito de mostrar cómo se realiza la escritura en este lenguaje recurrimos a los ejemplos de diagrama lógico de las secciones anteriores y realizamos la transformación en pseudocódigo. Este lenguaje está provisto de un conjunto de símbolos que conforman la sintaxis.

Retomamos el ejemplo 1 en el que se solicita la creación de un algoritmo que permita el ingreso de dos números para mostrar su suma.

Diagrama Lógico	Pseudocódigo
	ingresar(a) ingresar(b) c = a + b imprimir(c)

Nota
A la derecha de cada símbolo figura el pseudocódigo. Tanto **ingresar** como **imprimir** son palabras claves que determinan la acción correspondiente. Todas las palabras claves van en minúscula.

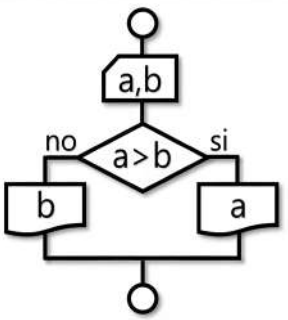
Luego se muestra la trasformación en el ejemplo 2 cuya consigna dice: Hallar el perímetro y el área de un rectángulo ingresando la base **b** y la altura **h**.

Diagrama Lógico	Pseudocódigo
	ingresar(b) ingresar(h) p = 2*b+2*h a = b*h imprimir(p,a)

Nota
En el pseudocódigo el operador de asignación (=) se mantiene igual que en el diagrama lógico.

Mostramos otro ejercicio con el ejemplo 3 en el que se ingresan dos números y, luego, se pide mostrar por mensaje cuál es el mayor.

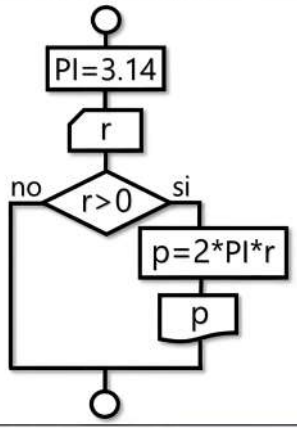
Diagrama Lógico	Pseudocódigo

	ingresar(a)
	ingresar(b)
	si(a>b):
	  imprimir(a)
	sino:
	  imprimir(b)

Nota
En este ejemplo aparece la decisión que se traduce en **si** y en **sino**. Es muy importante saber que ambas terminan con dos puntos (**:**) y que están indentadas (con una sangría o tabulación a la derecha).

Importante
La *indentación* es un tipo de notación o técnica que se utiliza para mejorar la legibilidad del código fuente. Sirve para separar bloques de código, lo que en algunos pseudocódigos requiere la declaración de inicio y fin del bloque, pero en nuestro caso lo hacemos sencillamente ubicando una sangría a la derecha (espacios en blanco).

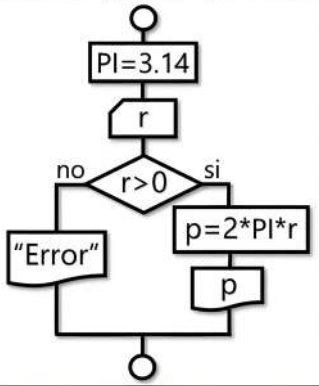
La consigna del ejercicio del ejemplo 4 dice: Mostrar el perímetro de una circunferencia, siempre y cuando el radio que se ingresa sea mayor a 0 (controlar dicho ingreso).

Diagrama Lógico	Pseudocódigo
	PI=3.14 ingresar(r) si(r>0): p=2*PI*r imprimir(p)

Nota
En este caso tenemos un bloque indentado conformado por dos sentencias, la asignación **p=2*PI*r** y el **imprimir(p)**. Quiere decir que estas dos sentencias que conforman un bloque están bajo el dominio del camino del **si** de la estructura de decisión **si(r>0)**:

En tanto el ejemplo 5 pide: Del mismo modo que en el ejemplo 4, pero en el caso de ingresar un radio erróneo (valor 0 o negativo) mostrar el cartel "**Error**".

Diagrama Lógico	Pseudocódigo

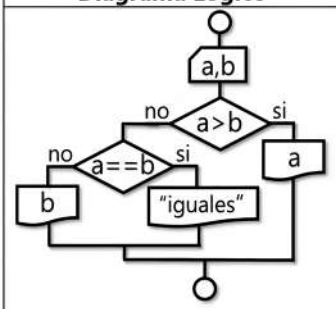
	PI=3.14
	ingresar(r)
	si(r>0):
	  p=2*PI*r
	  imprimir(p)
	sino:
	  imprimir("Error")

Nota
En realidad el pseudocódigo se escribe sin dejar líneas (renglones) vacías. En adelante se muestra de esa forma.

A continuación, se presenta la transformación a pseudocódigo en los ejemplos 6, 7, 8, 12 y 13 de las secciones anteriores.

El ejemplo 6 pide mostrar el número más grande (entre dos) ingresado por teclado. Si los dos números son iguales mostrar el cartel "**iguales**".

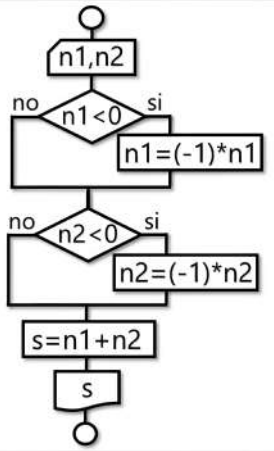
Diagrama Lógico	Pseudocódigo

	ingresar(a)
	ingresar(b)
	si(a>b):
	  imprimir(a)
	sino:
	  si(a==b):
	  imprimir("iguales")
	  sino:
	  imprimir(b)

Nota
En estos casos aparecen los **si** anidados. Observar cuidadosamente la indentación.

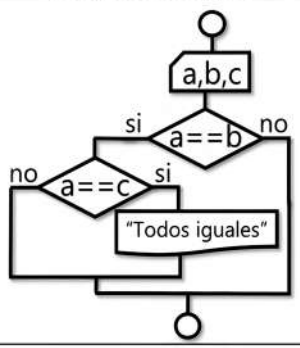
La consigna del ejemplo 7 solicita ingresar dos números por teclado y sumarlos. En caso que los números sean negativos, previo a la suma se debe cambiar su signo.

Diagrama Lógico	Pseudocódigo

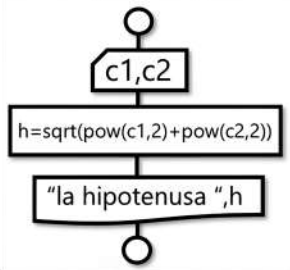
	ingresar(n1)
	ingresar(n2)
	si(n1<0):
	  n1=(-1)*n1
	si(n2<0):
	  n2=(-1)*n2
	s=n1+n2
	imprimir(s)

En tanto el ejemplo 8 dice: Supongamos que quisiéramos saber si tres números son iguales utilizando decisiones. Una solución posible sería:

Diagrama Lógico	Pseudocódigo

	ingresar(a)
	ingresar(b)
	ingresar(c)
	si(a==b):
	  si(a==c):
	  imprimir("Todos iguales")

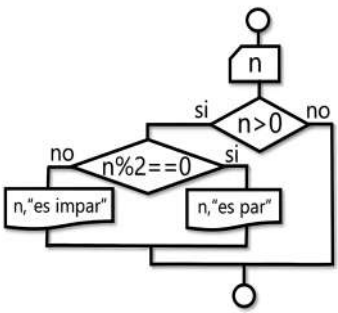
El siguiente ejercicio es el ejemplo 12: Ingresar por teclado los catetos **c1** y **c2** de un triángulo rectángulo para hallar y mostrar su hipotenusa **h**.

Diagrama Lógico	Pseudocódigo
	ingresar(c1) ingresar(c2) h=sqrt(pow(c1,2)+pow(c2,2)) imprimir("la hipotenusa ",h)

Nota
En este ejercicio se observa que los operadores aritméticos se mantienen y también la combinación de texto y variable en la función **imprimir**.

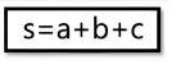
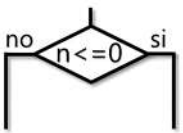
Por último, mostramos la escritura con el ejemplo 13 que pide ingresar un número natural por teclado para saber y mostrar si es par o impar. Tener en cuenta que además debe ser mayor a 0.

Diagrama Lógico	Pseudocódigo

	ingresar(n)
	si(n>0):
	  si(n%2==0):
	  imprimir(n,"es par")
	  sino:
	  imprimir(n,"es impar")

Desafío
Transformar los diagramas lógicos de los ejemplos 9 y 14 de las secciones anteriores en pseudocódigo.

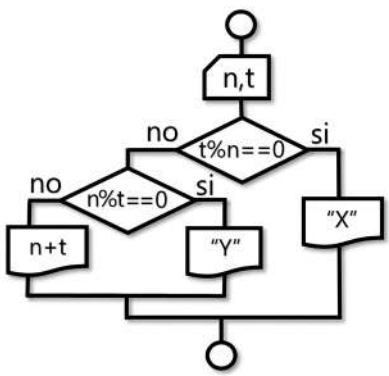
2. 9. Guía rápida del diagrama lógico a pseudocódigo

Diagrama Lógico	Pseudocódigo
	Conectores de inicio y fin de algoritmo No se manifiestan en el pseudocódigo
	Ingreso de datos: **ingresar(n1)**
	Asignación: **s=a+b+c**
	Estructura de decisión: **si(n<=0):** *sentencias* **sino:** *sentencias*
	Egreso de datos: **imprimir(s)**

2. 10. Problemas: estructura de control de decisión

A continuación, proponemos los siguientes ejercicios para practicar los temas abordados. Se pueden resolver primero recurriendo al diagrama lógico y, luego, pasar a pseudocódigo, o bien directamente en pseudocódigo.

1. Diseñar un algoritmo que, para cualquier número entero de entrada, imprima su antecesor y sucesor.
2. Realizar un algoritmo que solicite dos datos: país y capital. Y luego imprima la capital del país. El cartel debe ser como lo indica el siguiente ejemplo: "**Katmandú es la capital de Nepal**".
3. Diseñar un algoritmo que muestre por pantalla el doble y el triple de un número ingresado por teclado.
4. Diseñar un algoritmo que indique si el número ingresado es par y además muestre los dos números pares consecutivos. Por ejemplo si se ingresa el 10 imprime "**Par 12 y 14**". Pero si se ingresa un impar solo muestre "**Es impar**".
5. Diseñar un algoritmo que calcule y muestre el 30 % del número ingresado por teclado.
6. Diseñar un algoritmo que lea el precio de un artículo electrónico, luego calcule y muestre el resultado del aumento del 25 % y el valor del descuento del 15 %.
7. Diseñar un algoritmo que, dados tres números enteros, calcule e imprima el promedio de los mismos.
8. Transformar el siguiente diagrama lógico en pseudocódigo. Luego responder cuál es el valor de salida si se siguen los pasos del algoritmo para los siguientes valores de **n** y **t**. a) n=7, t=21; b)n=6, t=2; c)n=8, t=3.



9. Diseñar un algoritmo que solicite al usuario si desea calcular el área o el perímetro de una circunferencia a partir de leer el valor del radio de la misma. Pero si comete una equivocación en la solicitud que el programa muestre "**Error**", en otro caso mostrar los valores.

10. Realizar un algoritmo que calcule el área de un trapecio, siendo la fórmula:

$$A = \frac{(B+b)h}{2}$$

11. Crear un algoritmo que convierta y muestre en yardas, metros, pies y pulgadas un valor ingresado en centímetros.
12. Diseñar un algoritmo que convierta y muestre la temperatura en Fahrenheit ingresándola en Celsius.
13. Diseñar un algoritmo que calcule el volumen de un cilindro dado su radio y altura (primero, el programa deberá verificar si son positivas).
14. Diseñar un algoritmo que permita el ingreso de tres números (distintos entre sí) para mostrar el menor, el del medio y el mayor (en ese orden). Por ejemplo si a=4, b=8, c=1 el resultado será: 1, 4, 8.
15. Diseñar un algoritmo que lea 2 números enteros **n1** y **n2** y determine si **n1** es divisible por **n2**, en cuyo caso muestre un cartel que diga "**es divisible**" o "**no es divisible**".
16. Diseñar un algoritmo para calcular el porcentaje de varones y de mujeres que hay en un grupo, ingresando previamente la cantidad de varones y de mujeres.
17. Diseñar un algoritmo que indique con carteles si un número ingresado por teclado es negativo, positivo o es 0.
18. Partiendo del problema anterior, si además el número es positivo, debe mostrar si es par o impar.
19. Diseñar un algoritmo que determine y muestre si un triángulo es "**Escaleno**", "**Equilátero**" o "**Isósceles**" a partir de leer los lados del mismo. Recuerde utilizar los operadores lógicos. Además responder: ¿cuántas formas se encuentran de resolver este problema?
20. Diseñar un algoritmo que imprima con un cartel el número de docena («**primera**», «**segunda**» o «**tercera**») ingresando un número entero (del 1 al 36). Utilizar operadores lógicos. Puede ocurrir que el usuario cometa un error en el ingreso, en esos casos imprimir un cartel que diga "**fuera de rango**".
21. Diseñar un algoritmo que imprima con un cartel **"Correcto"** según el siguiente caso: si el número **n** es múltiplo de 5, positivo y se encuentra entre los 25 primeros números.
22. Diseñar un algoritmo que muestre las soluciones de una ecuación cuadrática (aplicando Bhaskara).

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

Recordar que las soluciones posibles en una ecuación de segundo grado son:
La solución única real.

- Dos valores reales.
- No tiene solución real (mostrar un cartel **"No tiene solución en los reales"**).

Además, tener en cuenta que son tres algoritmos por cada ecuación.

$a) P(x) = x^2 - 3x + 2$;

$b) P(x) = 2x^2 + 4x + 2$;

$c) P(x) = 3x^2 + 2$

Capítulo 3

Estructuras de control de repetición e iteración

Las computadoras son buenas siguiendo instrucciones,
no leyendo tu mente.
Donald Knuth*

Temas:

Estructuras de repetición e iteración: «mientras» y «para». Variables contadoras y sumadoras (acumuladoras). Traza o prueba escritorio. Máximos y mínimos. Estructuras de decisión combinadas con bucles de repetición e iteración. Validación de datos de entrada. Problemas.

3.1. Estructuras de control de repetición e iteración

Si bien los ejemplos presentados en el capítulo 2 no requieren el uso de estructuras de repetición o iteración, la resolución de problemas en programación con frecuencia demanda ejecutar varias veces un conjunto de instrucciones.

El siguiente ejercicio muestra la necesidad de procesos donde una misma serie de instrucciones debe ejecutarse múltiples veces. La consigna solicita determinar el promedio de las edades de 8 personas. Una solución posible podría ser ingresar 8 edades (una variable por cada edad **e1, e2, e3, e4, e5, e6, e7, e8**) y luego calcular su promedio. Ahora bien, ¿qué sucede si en lugar de 8 personas son 100 o más? En ese caso, estamos frente a una verdadera dificultad ya que tenemos que hacer uso de una cantidad extrema de variables. Está claro que no es lo mismo hallar el promedio de 8 números que el promedio de 100 o más. Veamos el modo de resolver el problema propuesto con las herramientas trabajadas hasta el momento para tener una mejor idea.

Para esto volvemos al ejemplo propuesto.

* Donald Ervin Knuth es un reconocido experto en ciencias de la computación estadounidense y matemático, famoso por su fructífera investigación dentro del análisis de algoritmos y compiladores. Es profesor emérito de la Universidad de Stanford.

Ejemplo 15: Ingresar por teclado 8 edades, luego hallar y mostrar su promedio.

```
ingresar(e1)
ingresar(e2)
ingresar(e3)
ingresar(e4)
ingresar(e5)
ingresar(e6)
ingresar(e7)
ingresar(e8)
sum = e1+e2+e3+e4+e5+e6+e7+e8
prom = sum/8
imprimir("El promedio de edades es",prom)
```

Pero también se puede resolver de la siguiente manera:

```
ingresar(e)
sum = e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
ingresar(e)
sum = sum+e
prom = sum/8
imprimir("El promedio de edades es",prom)
```

La primera solución resuelve el problema en pocas líneas pero utiliza una gran cantidad de variables. La segunda solución utiliza pocas variables pero muchas líneas de comando que se repiten de a pares **ingresar(e)** y **sum=sum+e**. Basta imaginar si el algoritmo en vez de resolver el promedio con 8 números debe resolverlo, por ejemplo, con 100. Necesitamos para la primera solución 100 variables y para la segunda solución más de 200 líneas de trabajo.

Para poder tratar con este tipo de situaciones de manera más efectiva, existen las estructuras de control de repetición e iteración: «mientras» y «para».

3. 2. Estructura de control de repetición: sentencia «mientras»

La estructura de control «mientras» permite repetir una instrucción o grupo de instrucciones en tanto la expresión lógica sea verdadera. De esta forma, la cantidad de veces que se reiteran las instrucciones no necesita ser conocida por anticipado, sino que depende de una condición.

Lo primero que hace esta sentencia es evaluar si se cumple o no la condición. En caso que se cumpla se ejecuta el bucle (ciclo), pero si no se cumple entonces no se ejecuta ninguna acción. En otras palabras, el bucle sigue iterando en presencia de «mientras» y con la condición que se cumple, por eso es importante no caer en ciclos de repetición infinitos o que nunca pueda entrar al ciclo.

A continuación vemos cómo se puede solucionar el ejemplo 15:

```
i=0 ; sum=0
mientras(i<8):
    ingresar(e)
    sum=sum+e
    i=i+1
prom = sum/8
imprimir("El promedio de edades es",prom)
```

De esta manera todo lo que encierra el comando **mientras** es lo que se encuentra con una indentación (sangría de línea o tabulación) que se refleja con las flechas. Es decir, las instrucciones como **ingresar(e)**; **sum=sum+e** y **i=i+1** se repiten tantas veces mientras la variable numérica **i** sea menor a 8. Al inicio se asigna a **i** el valor 0 y en cada vuelta se incrementa en 1.

La variable **sum** funciona como sumador o acumulador de las edades **e**. Fuera del alcance del **mientras** se calcula el promedio **prom** para luego mostrarlo. Todo contador y sumador/acumulador debe inicializar en 0 antes del ciclo de repetición.

Atención

Para ahorrar espacio, la primera línea presenta más de una asignación **i = 0**; **sum = 0**. Esto se puede hacer en tanto estén separados por punto y coma (**;**).

Nota
¿Qué deberíamos cambiar del algoritmo si le asignamos de entrada a la variable **i** un 1, manteniendo la cantidad de 8 edades para hallar el promedio?

Ejemplo 16: Calcular la suma de números ingresados por teclado hasta que se ingrese 0.

```
s=0
ingresar(n)
mientras(n!=0):
        s=s+n
        ingresar(n)
imprimir("La suma acumulada es",s)
```

En este caso, la condición que se evalúa en cada vuelta va a depender del ingreso de la variable **n**. Es necesario ingresar **n** antes de arrancar con **mientras**, porque se debe evaluar la condición antes de entrar al ciclo de repetición. Luego, es importante no olvidar de ingresar **n** dentro del bucle, porque si no lo hacemos caemos en un ciclo de repetición infinito.

Atención
En el ejemplo anterior: ¿cuál es el resultado si arrancamos ingresando el valor 0?

Dentro de la condición del «mientras» se pueden incorporar operadores lógicos y relacionales. A continuación, mostramos un ejemplo.

Ejemplo 17: Hallar el promedio de números ingresados por teclado hasta que aparezca uno que sea par y mayor o igual a 20.

```
c=0 ; s=0
ingresar(n)
mientras(n%2==0 and n>=20):
        c=c+1
        s=s+n
        ingresar(n)
imprimir("El promedio es",s/c)
```

¿Qué pasa si de entrada ingresamos un número que no cumple la condición del **mientras**? En este caso, y a diferencia del ejemplo 15, salta un error en **imprimir("El promedio es ", s/c)** ya que se está intentando hacer una división por el valor 0, porque la variable **c** nunca pudo incrementarse. Por ende, podemos corregir el error haciendo una consulta previa al cálculo de la siguiente forma:

```
c=0 ; s=0
ingresar(n)
mientras(n%2==0 and n>=20):
        c=c+1
        s=s+n
        ingresar(n)
si(c>0):
        imprimir("El promedio es",s/c)
sino:
        imprimir("No existen datos")
```

Nota
Cuando culmina el ciclo **mientras** el algoritmo pregunta **si(c>0)** para asegurar que efectivamente se puede dividir **s/c**, caso contrario en **sino:** solo se imprime el mensaje **"No existen datos"**. A esto lo llamamos *validar datos*.

Atención
Observar que las estructuras **mientras** y **si** finalizan siempre con dos puntos (**:**).

3. 3. Variables contadoras y acumuladoras

3. 3. 1. Variable contadora

La *variable contadora* se encuentra en ambos miembros de una asignación a la que se le suma un valor entero constante (por lo general la constante es un 1). Los contadores se utilizan con la finalidad de contar sucesos, acciones o iteraciones internas en un bucle, proceso, subrutina o donde se requiera cuantificar. Deben necesariamente ser inicializados antes del comienzo de un ciclo de repetición. En el ejemplo 15 tenemos la variable contadora **i** que arranca con **i=0** como valor inicial y, dentro del ciclo, la

asignación **i=i+1** incrementa su valor para efectuar el conteo.

La inicialización consiste en asignar a una variable un valor inicial. En este caso, al contador se le da el número desde el cual necesitamos se inicie el conteo (por lo general siempre comienzan en 0). Para decrementar cambiamos el signo **+** por **-**, por ejemplo: **c=c-1**.

3. 3. 2. Variable acumuladora (sumadora)

La *variable acumuladora* se encuentra en ambos miembros de una asignación a la que se le suma un valor constante o variable (según el caso). Los sumadores pueden ser de tipo entero o con decimales.

En el ejemplo 15 utilizamos **sum** (variable acumuladora), que realiza la acción de sumar sobre sí misma un conjunto de valores numéricos (en este caso las edades). Al finalizar el ciclo de repetición la sumatoria tendrá acumulado todos esos valores. Es necesario haber inicializado su valor antes del comienzo de un ciclo de repetición, en el caso del ejemplo fue con **sum=0**.

La inicialización consiste en asignar al sumador un valor inicial, es decir, el número desde el cual necesitamos se inicie la sumatoria (por lo general siempre comienzan en 0).

El equivalente del ejercicio 15 en lenguaje matemático es:

$$S = \sum_{i=0}^{8} E$$

Ejemplo 18: Contar la cantidad de veces que se ingresan números pares y la cantidad de números impares hasta que se ingrese un número negativo. Recordar que el 0 no se toma como número par.

```
cp=0 ; ci=0
ingresar(n)
mientras(n>=0):
        si(n>0):
            si(N%2==0):
                cp=cp+1
            sino:
                ci=ci+1
        ingresar(n)
imprimir("La cantidad de números pares es ", cp)
imprimir("La cantidad de números impares es ", ci)
```

Mientras la condición del ciclo de repetición cumpla con **n** mayor o igual a 0, dentro del ciclo se pregunta primero si dicho número es mayor a 0 y luego si es par o impar para incrementar las variables contadoras **cp** o **ci**, respectivamente.

Nota
El valor de **n** debe ingresarse antes del ciclo y dentro del mismo. Además, es necesario inicializar las variables contadoras o sumadoras.

Importante
La diferencia entre un contador y un acumulador es que mientras el primero va aumentando en una cantidad fija preestablecida, el segundo va aumentando en una cantidad o valor variable. Y la similitud está en que siempre deben tener un valor inicial antes del bucle de repetición.

Ejemplo 19: A la salida de un cine se hace una serie de preguntas a cada espectador para saber finalmente lo siguiente:

a) Porcentaje de mujeres.
b) Promedio de calificación de la película. Cada espectador califica de 1 al 10.
c) Cantidad de personas mayores a 60 años de edad.

La cantidad de personas no se sabe, por lo tanto, se termina la carga de registros respondiendo con la letra **N** a la pregunta: ¿Desea ingresar datos S/N?

```
cm=0 ; ct=0 ; scal=0 ; c60=0
imprimir("¿Desea ingresar datos S/N? ")
ingresar(seguir)
mientras(seguir=='S'):
        ingresar(genero)
        ingresar(edad)
        ingresar(calificacion)
        ct=ct+1
        si(genero=='Mujer'):
            cm=cm+1
        si(edad>60):
            c60=c60+1
        scal=scal+calificacion
        imprimir("¿Desea ingresar datos S/N? ")
        ingresar(seguir)
imprimir("El % de mujeres es", 100*cm/ct)
imprimir("El promedio de calificación es", scal/ct)
imprimir("La cantidad de personas mayores a 60 es", c60)
```

En este caso, la salida o culminación del bucle **mientras** lo determina la variable alfanumérica **seguir** con la respuesta **N**, entretanto el usuario ingresa **S** para continuar cargando. Esta pregunta se debe hacer antes de ingresar al bucle y dentro del mismo. Si se omite al comienzo, nunca ingresaremos. En tanto, si no lo hacemos dentro del bucle, nunca finalizaremos pues se generan infinitas iteraciones.

La variable alfanumérica **genero** se utiliza para ingresar 'Mujer', 'Varón' o 'No binario', según el caso. Va para el camino del **si** cuando es **'Mujer'**.

La variable **ct** cuenta la cantidad de personas en total, **cm** la cantidad de mujeres, **c60** la cantidad de personas mayores a 60 años de edad y **scal** efectúa la sumatoria de calificaciones para proceder al promedio.

Atención

¿Qué sucede si no fueron mujeres al cine? Si surge un inconveniente: ¿cómo se puede resolver?

---

3. 4. Problemas: estructura de control de repetición «mientras»

23. Crear un algoritmo que muestre solo los números impares hasta encontrar uno nulo. El usuario ingresa los números.
24. Crear un algoritmo que muestre 10 números múltiplos de 5. Estos son ingresados por el usuario.
25. Crear un algoritmo que pida que el usuario ingrese 10 valores numéricos enteros como máximo. Se desea verificar si esos valores están en orden estrictamente creciente. Si lo están, debe mostrar al final del proceso la frase: **"Elementos ordenados»**. Si no lo están, cuando se detecta el primer valor que altera el orden, se debe finalizar el proceso y mostrar por pantalla el número que lo interrumpió.
26. Para un curso de alumnos y alumnas se necesita lo siguiente:
    a. Cantidad de personas que tienen mascotas.
    b. Promedio de edad de los varones.
    c. Cantidad de mujeres que no tienen mascotas.

    Para cada registro se necesita preguntar: ¿Desea seguir cargando S/N?
27. Se ingresan números de la ruleta (0, 1, 2,…,36), para hallar y mostrar lo siguiente:
    a. Porcentaje de números pares y números impares.
    b. Cantidad de números que se encuentran en la 1º y 3º docena (juntos).
    c. Promedio de los números que se encuentran en la 2º docena.

    El algoritmo termina cuando se ingresa un número que no forma parte de la ruleta.

28. Del siguiente algoritmo se desea saber con qué datos culminan las variables **a**, **b**, **c**, **sn** sabiendo que la secuencia de ingreso en **n** es la siguiente: 3, 4, 1, 0, 6.

```
a=10 ; b=0 ; sn=0 ; c=0
mientras(a>b):
        ingresar(n)
        b=b+1
        a=a-1
        si(n==0):
            c=c+2
        sino:
            si(n%2!=0):
                sn=sn+b
            sino:
                sn=sn+a
c=c*3
imprimir("El resultado 1 es", sn)
imprimir("El resultado 2 es", c)
```

29. Se ingresan números N hasta que la suma alcance los 1000. Hallar:
    a. La cantidad de números negativos.
    b. La suma de los números que se encuentran entre el 1 y el 10 (incluidos).
    c. El promedio de los mayores a 10.

3. 5. Estructura de control de iteración: sentencia «para»

Cuando se desea ejecutar un conjunto de acciones en un determinado número de veces, usamos la sentencia «para».

Estos casos requieren el conocimiento anticipado del número de iteraciones. Para efectuar el algoritmo del ejemplo 15 lo hacemos de la siguiente manera:

```
sum=0
para i en rango(8):
    ingresar(e)
    sum=sum+e
prom=sum/8
imprimir("El promedio de edades es",prom)
```

Todo lo que encierra el comando **para** es lo que se encuentra con una indentación (sangría de línea o tabulación) que se refleja con las flechas, es decir, las instrucciones **ingresar(e)** y **sum=sum+e** se repiten tantas veces como lo indica el rango. En este caso, el rango es 8 y la variable **i** indica el número de vuelta o iteración, comenzando por el 0 hasta el 7.

La variable **sum** funciona como sumador o acumulador de las edades **e**. Fuera del alcance del **para** se calcula el promedio **prom** para luego mostrarlo. Recordar que todo contador y sumador/acumulador debe inicializar en el valor 0 antes del ciclo de iteración **para**. En este caso, y a diferencia del **mientras**, no hace falta inicializar **i**, tampoco incrementar **i=i+1** ya que implícitamente lo determina el rango.

Nota

El **rango(n)** indica la cantidad de vueltas, siempre será desde el valor 0 hasta n-1 y la variable **i,** se la llama contadora de vuelta, irá tomando el valor entero por cada una (0, 1, 2,..., n-1).

Atención

Es muy importante tener en cuenta que los promedios siempre deben ser calculados fuera del bucle. ¿Qué sucedería si se calculan dentro del ciclo?

Ejemplo 20: Mostrar la tabla de multiplicación del 6.

```
para i en rango(1,11):
        imprimir( " 6 x ", i , " = " , 6*i )
```

En el código modificamos lo que está dentro del paréntesis del rango para que inicie las vueltas desde 1 hasta 10 (recuerde que siempre culmina con n-1).

El **imprimir** se encuentra dentro del dominio del **para** por su indentación, por lo tanto, se ejecutará 10 veces. Dentro del **imprimir** hay una combinación de datos numéricos y alfanuméricos (texto) que obtiene el siguiente resultado por pantalla:

```
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 56
6 x 10 = 60
```

Los datos numéricos están conformados por la variable contadora de vuelta **i** y el producto **6*i**, mientras que el texto es **"6 x " , " = "**que se mantiene igual por cada vuelta.

Ejemplo 21: Mostrar la tabla de multiplicación de un número que ingresamos.

```
ingresar(num)
para i en rango(1,11):
        imprimir(num,"x",i,"=",num*i)
```

La única diferencia respecto al ejemplo 20 es que podemos mostrar la tabla de multiplicación de cualquier número entero **num** que ingresamos previo al ciclo de iteración. En la función **imprimir** se observa una combinación de texto **"x ", "= "** y las variables numéricas **num**, **i**, **num*i**.

Nota

En estos casos usamos la variable **i** como contadora de vuelta, pero se podría etiquetar de otra manera. Lo importante es que no existan ambigüedades, es decir que no usemos una misma variable para propósitos distintos. En esos casos, se deben diferenciar.

Ejemplo 22: Se ingresan por teclado 100 números enteros para saber cuántos son divisibles por 3. Recordar que hay que descartar previamente el 0.

```
contar=0
para p en rango(100):
        ingresar(n)
        si(n!=0 and n%3==0):
                contar=contar+1
imprimir("La cantidad de n° divisibles a 3 es",contar)
```

Sentencias bajo el dominio del para

La variable **contar** está reservada para contabilizar la cantidad de números que son divisibles por 3, por lo tanto, se lo inicializa en 0 (antes del **para**). En este caso, la variable **p** es la contadora de vueltas, según el rango serán 100. Se presenta en este ejemplo una combinación de estructuras de control (iteración y decisión). Muy importante es ver la indentación, es decir, lo que está bajo el dominio del ciclo de iteración y el **contar=contar+1** bajo el dominio del **si**. Claramente, el comando **imprimir** se encuentra fuera de cualquier dominio. ¿Qué sucedería si el comando **imprimir** está indentado (tabulado a la derecha) dentro del dominio del **para**?

Atención
Observar que la estructura **para** también finaliza con dos puntos (**:**).

Ejemplo 23: En una veterinaria se desea saber el promedio de edad de gatos y perros (por separado) que fueron asistidos durante un mes. En total se registraron 30 animales y la veterinaria solo atiende gatos y perros.

```
cg=0 ; cp=0 ; seg=0 ; sep=0
para i en rango(30):
        ingresar(animal)
        ingresar(edad)
        si(animal=='gato'):
                cg=cg+1
                seg=seg+edad
        sino:
                cp=cp+1
                sep=sep+edad
imprimir("El promedio de edad de los gatos es", seg/cg)
imprimir("El promedio de edad de los perros es", sep/cp)
```

En este caso son 4 inicializaciones en 0 para contadores **cp** y **cg** y para sumadores (acumuladores) **sep** y **seg**. La variable alfanumérica **animal** se utiliza para ingresar el tipo de mascota (**'gato'** o **'perro'**). Según el caso, va para el camino del **si** o del **sino**. Para calcular el promedio será necesario un contador y un sumador por cada animal.

Nota
Recordar que en la estructura **si,** al tratarse de una consulta sobre una variable alfanumérica, el dato va entre apóstrofes **si(animal=='gato')** a diferencia de un dato de tipo numérico.

Atención
¿Qué sucede si en un mes la veterinaria no recibe ningún perro? ¿Cuál sería el resultado de **sep/cp**?

Vamos a ampliar las consignas del ejemplo 23 diciendo que el veterinario nos informa la cantidad de animales que ingresan y que, además de gatos y perros, atiende canarios. Entonces las nuevas consignas que necesita el programa son promedio de edad de gatos, promedio de edad de perros y solo la cantidad de canarios.

```
cg=0 ; cp=0 ; seg=0 ; sep=0 ; cc=0
para i en rango(30):
        ingresar(animal)
        si(animal=='gato'):
            cg=cg+1
            seg=seg+edad
        sino:
            si(animal=='perro'):
                cp=cp+1
                sep=sep+edad
            sino:
                cc=cc+1
imprimir("El promedio de edad de los gatos es", seg/cg)
imprimir("El promedio de edad de los perros es", sep/cp)
imprimir("La cantidad de canarios es",cc)
```

Se agregó una variable contadora **cc** para los canarios. Dentro del primer **sino** se vuelve a preguntar por los perros ya que en este caso luego de negar el tipo de mascota **'gato'** queda la posibilidad de ser un perro o un canario. Es decir que por descarte otro tipo de animal cae en el último **sino** de la decisión, por lo tanto, incrementa en 1 la variable **cc**.

Ejemplo 24: Ingresar un número para mostrar de modo descendente y ordenado sus consecutivos hasta culminar con el 1. Por ejemplo: si **n** es 9 entonces debe mostrar el 8, 7, 6,..., 2, 1, 0.

```
ingresar(n)
para i en rango(n):
        imprimir(n-i)
```

Arranca la impresión desde **n-0**, luego imprime **n-1**, y así hasta llegar al imprimir un 1.

3. 6. Traza o prueba escritorio

La *traza o prueba escritorio* sirve para revisar el comportamiento, el funcionamiento y la verificación de un algoritmo. Por lo general se hace con una tabla encabezada por todas las variables que se involucran en el programa. A medida que se produce la secuencia de instrucciones, las filas reproducen los cambios que sufren durante la ejecución del algoritmo.

Hagamos la prueba con el ejemplo 23. Su algoritmo era el siguiente:

```
cg=0 ; cp=0 ; seg=0 ; sep=0
para i en rango(30):
        ingresar(animal)
        ingresar(edad)
        si(animal=='gato'):
            cg=cg+1
            seg=seg+edad
        sino:
            cp=cp+1
            sep=sep+edad
imprimir("El promedio de edad de los gatos es", seg/cg)
imprimir("El promedio de edad de los perros es", sep/cp)
```

Esta vez vamos a probar con 7 ingresos (tipo de animal y edad) para simplificar la verificación.

i	cg	cp	seg	sep	animal	edad
	0	0	0	0		
0	1		3		gato	3
1		1		5	perro	5
2		2		9	perro	4
3	2		5		gato	2
4		3		13	perro	4
5	3		8		gato	3
6	4		11		gato	3

En la primera fila aparecen ceros en las variables contadoras y sumadoras (acumuladoras), luego la doble línea indica el comienzo del bucle de iteración donde **i** parte en 0. En cada vuelta se ingresa (gato/perro y su edad), dependiendo de la estructura de control de decisión, si el animal es gato se incrementa en 1 el contador **cg** y se acumula en **seg** su edad y si el animal es perro se incrementa en 1 el contador **cp** y se acumula en **sep** su edad. En este ejemplo en particular, el promedio de edades de gatos será 11/4 (2,75) mientras que para los perros será 13/3 (4,33).

Importante
Se utiliza para depurar un algoritmo pero no es una técnica que permita demostrar la correctitud del algoritmo.
Solo permite asegurar que el algoritmo es correcto para los valores de los datos de entrada usados.

Nota
Se recomienda hacer la prueba escritorio (traza) en algoritmos que tienen una cierta complejidad para evacuar dudas. Los ingresos deben ser variados y contemplar todos los casos que posibilita el programa.

A modo de ejercitación, realizar la traza del siguiente algoritmo.

```
cg=0 ; cp=0 ; cc=0
para i en rango(10):
        ingresar(animal)
        si(animal=='gato'):
            cg=cg+1
        sino:
            si(animal=='perro'):
                cp=cp+1
            sino:
                cc=cc+1
imprimir("El % de gatos es", cg/10)
imprimir("El % de perros es", cp/10)
imprimir("El % de canarios es", cc/10)
```

Ejemplo 25: El factorial de un número se denota de la siguiente manera: **n!** y su resultado es **n!=n*(n - 1)*(n - 2)*...*1**. Por ejemplo: 5!=5*4*3*2*1 siendo el resultado 120.

```
factorial = 1
ingresar(n)
para i en rango(n):
        factorial=factorial*(n-i)
imprimir("El factorial de",N,"es",factorial)
```

El valor de entrada factorial debe ser 1, la variable contadora de vueltas **i** juega un papel muy importante porque le va restando a **n** que multiplica a factorial. Por ejemplo, si **n** es igual a 5 entonces la secuencia de **factorial=factorial*(n - i)** es la siguiente:

Fórmula	Resultado
factorial=1*(5 – 0)	factorial = 5
factorial=5*(5 – 1)	factorial= 20
factorial=20*(5 – 2)	factorial= 60
factorial=60*(5 – 3)	factorial= 120
factorial=120*(5 – 4)	factorial= 120

En la última vuelta el valor de **factorial** se mantiene ya que es multiplicado por el 1.

Nota
Si a la variable **factorial** se le hubiese asignado de entrada el valor 0, ¿qué pasaría?

3. 7. Máximos y mínimos

Para hallar el valor máximo y mínimo de una lista de valores numéricos es necesario utilizar una variable adicional.

Ejemplo 26: Hallar la persona de mayor altura, sabiendo que se toman 20 muestras. La unidad de medida será en metros. Por ejemplo: 1.67 (un metro con 67 centímetros).

```
max_alt=0
para i en  rango(20):
        ingresar(altura)
        si(altura>max_alt):
              max_alt=altura
imprimir( "La mayor altura registrada es " ,max_alt)
```

El valor inicial de la variable **max_alt** (de mayor altura) es un número claramente absurdo por ser 0. Cuando se ingresa el primer valor altura dentro del bucle, la condición del control de la decisión se efectúa al ser verdadera. Entonces, el valor de **max_alt** cambia por el valor de altura. A medida que se ingresan las otras alturas, la sentencia evalúa si aparece un valor mayor o no a **max_alt**. En caso afirmativo se le asigna a este el nuevo valor de altura.

Se debe tener en cuenta que si se produce el caso de empate la condición es falsa, por lo tanto, no sufre efecto.

Nota
Desarrollar el ejemplo 26 usando la estructura «mientras».

Por el contrario, si el ejercicio trata de hallar la mínima altura entre las 20 personas, entonces es necesario cambiar el nombre de la variable **max_alt** por **min_alt** y asignarle un valor extremadamente grande de entrada (por ejemplo **min_alt=1000**). La condición de la decisión es en este caso **altura<min_alt**.

A continuación, se muestra el ejemplo 27 pero para hallar el mínimo.

```
min_alt=1000
para i en  rango(20):
        ingresar(altura)
        si(altura<min_alt):
              min_alt=altura
imprimir( "La menor altura registrada es " ,min_alt)
```

Nota
¿Qué alternativa existe para evitar asignar a **max_alt** y a **min_alt** un valor absurdo?

Responder qué sucede si además de registrar la máxima altura o mínima altura de la persona, se desea conocer la ubicación de la persona en una escala de altura. A continuación, se presenta un ejemplo relacionado.

Ejemplo 28: Mostrar la mayor altura registrada de un grupo de 100 personas y además en la ubicación en que se encuentra.

```
max_alt=0
para i en  rango(100):
        ingresar(altura)
        si(altura>max_alt):
              max_alt=altura
              pos_max=i+1
imprimir( "La mayor altura registrada es " , max_alt)
imprimir( "y se encuentra en la ubicación ", pos_max)
```

Incorporamos una nueva variable llamada **pos_max** que va reservando la ubicación del mayor al pasar afirmativamente por la decisión **si**.

Atención
¡Ojo que en el rango la lista comienza en 0! En estos casos se necesita sumarle 1 a la posición **i**.

Nota
Desarrollar el ejemplo 28 usando la estructura «mientras».

Se puede dar el caso en que se combinan ciclos de repetición e iteración, por ejemplo, un «mientras» dentro de otro, un «para» dentro de otro, un

«para» dentro de un «mientras», un «mientras» dentro de un «para». Este último caso lo vamos a ver con el siguiente ejemplo.

Ejemplo 29: El usuario debe ingresar 30 números enteros positivos **n>0**, para que el algoritmo cuente la cantidad de números múltiplos de 5. Pero si el usuario se equivoca al ingresar un número nulo o negativo el programa debe permitir volver a ingresar hasta que no cometa el error (ese error de ingreso no se contabiliza).

```
cm5=0
para i en rango(30):
        ingresar(n)
        mientras(n<=0):
                ingresar(n)
        si(n%5==0):
                cm5=cm5+1
imprimir("La cantidad de nº divisibles a 5 es",contar)
```

No sale del bucle hasta que se ingrese un n>0

Nota
El **mientras** cumple la función de filtro o limpieza porque se asegura que **n** sea positivo antes de detectar si es múltiplo de 5 o no. La indentación es un factor importante ya que determina qué instrucciones están bajo su dominio. En este caso el segundo **ingresar(n)** en la quinta línea del algoritmo está dominado solo por el **mientras**.

De esta manera se garantiza el ingreso y el chequeo de 30 números efectivamente mayores a 0.

Con la estructura «mientras» también podemos averiguar valores máximos y mínimos. Para mostrar esto retomamos el ejemplo 28 que dice: Mostrar la mayor altura registrada de un grupo de 100 personas y además en qué ubicación se encuentra. Pero en este caso no decimos que son 100 personas (no sabemos cuántas son).

```
max_alt=0 ; i=1
ingresar(altura)
mientras(altura!=0):
        si(altura>max_alt):
            max_alt=altura
            pos_max=i
        i=i+1  ⟹ i actúa como contador de vuelta
        ingresar(altura)
imprimir("La mayor altura registrada es",max_alt)
imprimir("y se encuentra en la ubicación",pos_max)
```

El ciclo **mientras** va a culminar cuando el usuario ingrese un 0, es decir, un número que no cumple con la altura de una persona. La variable **i** se va incrementando a medida que suceden las vueltas y en vez de iniciar con 0 le damos el valor 1 por tratarse de la primera posición (ubicación de la persona).

3. 8. Validación de datos de entrada

La *validación de datos de entrada* es una parte fundamental en el desarrollo de un algoritmo porque sirve para asegurar que los datos que provee el usuario sean correctos. Se puede efectuar de distintas maneras.

En el ejemplo 28 donde se pedía la altura de cada persona se pueden filtrar aquellos errores de ingreso como alturas negativas, nulas o superiores a 3 metros. Entonces tenemos que agregar en el algoritmo una condición para cada ingreso.

```
max_alt=0
para i en rango(100):
        ingresar(altura)
        si(altura>0 and altura <3):
            si(altura>max_alt):
                max_alt=altura
                pos_max=i+1
        sino:
            imprimir("Ingreso incorrecto")
imprimir("La mayor altura registrada es",max_alt)
imprimir("y se encuentra en la ubicación",pos_max)
```

Nota

Lo que está en negrita en el ejemplo anterior es agregado al algoritmo original. En este caso se asegura el ingreso correcto de cada altura mayor a 0 y menor a 3, pero el inconveniente es que al filtrar alturas erróneas se pierde el dato. Por ejemplo, si filtramos tres ingresos falsos, tenemos en total 97 alturas registradas y no 100.

Para evitar el inconveniente anterior, agregamos la estructura **mientras**.

```
max_alt=0
para i en rango(100):
        ingresar(altura)
        mientras(altura<=0 or altura >=3):
            imprimir("Vuelva a ingresar")
            ingresar(altura)
        si(altura>max_alt):
            max_alt=altura
            pos_max=i+1
imprimir("La mayor altura registrada es",max_alt)
imprimir("y se encuentra en la ubicación",pos_max)
```

En este caso, nos permite realizar varios intentos hasta que la altura esté dentro de lo esperado y de esta manera nos aseguramos que efectivamente serán 100 los ingresos. Lo mismo sucede si cambiamos la condición dentro del **mientras** de la siguiente forma:

```
max_alt=0
para i en rango(100):
        ingresar(altura)
        mientras(not(altura>0 and altura <3)):
            imprimir("Vuelva a ingresar")
            ingresar(altura)
        si(altura>max_alt):
            max_alt=altura
            pos_max=i+1
imprimir("La mayor altura registrada es",max_alt)
imprimir("y se encuentra en la ubicación",pos_max)
```

Nota
En estos casos, la disyunción **or** es equivalente a la negación de la conjunción **and**.

3. 9. Problemas: estructura de control de iteración «para»

30. Imprimir los números pares del 30 al 2.
31. Imprimir las tres primeras potencias de los números 1 al 5.
32. Dado un número, imprimir su tabla de multiplicar (del 0 a 10).

33. Imprimir la cantidad de personas que se encuentren entre los 18 y 25 años de edad de un total de **N** personas. La cantidad total de personas debe ser ingresada por el usuario.
34. Imprimir la cantidad de mujeres y varones (**M**/**V**) de un total de 70 personas. Mostrar también el porcentaje de cada uno respecto del total.
35. De un total de **N** turistas que visitan Bariloche en agosto, se desea saber lo siguiente:
    a. Cantidad de turistas extranjeros mayores de edad.
    b. Promedio de edad de los turistas nacionales.
    c. Cantidad de turistas que visitan la ciudad por primera vez.
    d. Se recomienda evaluar primero qué datos se les pide a cada turista y saber en qué momento. Se pide además el uso de la traza.
36. Interpretar el siguiente algoritmo en pseudocódigo si la secuencia del ingreso de la variable **a** es la siguiente: 40, 20, 29, 30, 29, 28. ¿Con qué valores finalizan las variables **m**, **x1** y **x2** ?
    En este caso, se recomienda el uso de traza.

```
x1=1 ; x2=30
para i en rango(6):
        ingresar(a)
        si(a>x2):
            x1=x1+1
            m=x1*2
        sino:
            si(x1<=3):
                x2=x2-i
            sino:
                m=x2-8
```

37. Se desea conocer el promedio de peso de 40 personas y, además, el mayor peso y el menor peso registrados.
38. Crear un algoritmo que muestre 6 números múltiplos de 7. Estos son ingresados por el usuario, pero puede equivocarse, entonces debe volver a ingresar hasta que sea correctamente múltiplo de 7.
    Tener en cuenta el ejemplo 29 usando «mientras».
39. Crear un algoritmo que permita el ingreso de 50 números que genera una ruleta (0 al 36) para mostrar lo siguiente:
    a. Porcentaje de números pares.
    b. Porcentaje de números impares.
    c. Porcentaje de ceros.

    Recordar que el 0 no se considera par. Se recomienda la traza.

40. Crear un algoritmo para resolver la siguiente función:

$$\frac{n!}{(n-k)!k!}$$

El usuario debe ingresar los valores enteros de **n** y **k**.

Nota
Recordar que **n! k! (n-k)!** son factoriales, un tema que vimos en el ejemplo 25.

---

41. Ingresar 60 temperaturas, mostrar la mayor y en qué ubicación se encuentra. La primera temperatura se encuentra en la ubicación 1 y no en 0.
42. Se ingresan 10 pares de temperaturas **t1** y **t2**. Hallar el promedio de las temperaturas **t1** y el promedio de las temperaturas **t2**. Y al finalizar mostrar un cartel que indique cuál fue el promedio más grande (**"Temperaturas 1"** o **"Temperaturas 2"** o **"Son iguales"**).

Capítulo 4

Codificación

> Los programas deben ser escritos para que los lean las personas,
> y solo accidentalmente para que lo ejecuten las máquinas.
> Abelson y Sussman*

Temas:

Instalación y entorno Python. Modo *prompt* y creación de archivo fuente .py. Errores sintácticos y semánticos. Pasaje de pseudo a código. Estructura de decisión «if», «else», «elif». Estructuras de repetición «for», «while». Máximos y mínimos. Tipo de datos. Comentarios. Librerías. Números aleatorios. Problemas.

4. 1. Código Python

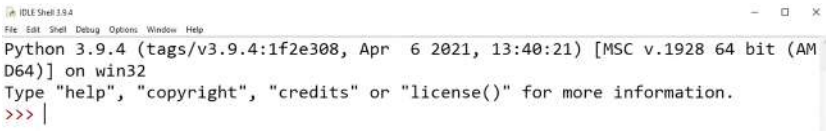
Hasta el momento hemos desarrollado algoritmos con pseudocódigo resolviendo distintos problemas. Para poder ejecutar los algoritmos en la computadora es necesario codificarlos en un lenguaje que pueda ser comprendido por la máquina. Esto puede ser entendido como un proceso de traducción del pseudocódigo al lenguaje de programación que se desea utilizar, por lo que resulta muy importante comprender la sintaxis y la semántica del lenguaje en cuestión.

Al algoritmo escrito en un lenguaje de programación se lo considera programa fuente y se guarda con la extensión propia del lenguaje. En el caso de Python, los archivos tienen la extensión **.py**. En este capítulo comenzamos a conocer y utilizar el lenguaje de programación de alto nivel Python.

Antes de avanzar, es importante tener descargada la versión 3 de Python. Para ello, se accede al sitio oficial https://www.python.org. Una vez que finalice la instalación, al efectuar la ejecución nos aparece una pantalla como la siguiente:

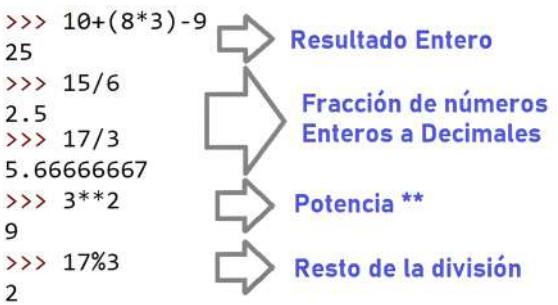
---

* Harold Abelson es profesor de Ingeniería Eléctrica y Ciencias de la Computación del Instituto Tecnológico de Massachusetts (MIT), miembro de la Institute of Electrical and Electronics Engineers (IEEE), cofundador junto a Lawrence Lessig de Creative Commons, forma parte de la Free Software. Gerald Jay Sussman es catedrático de Ingeniería Eléctrica del MIT. Se licenció y doctoró en Matemática en el MIT en 1968 y 1973, respectivamente. Ha participado en la investigación sobre inteligencia artificial en el MIT desde 1964.

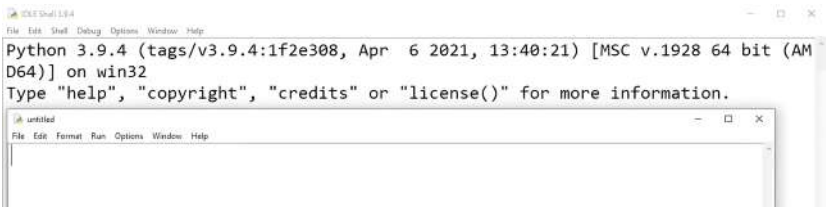


El entorno integrado de desarrollo con el que se trabaja se denomina *integrated development and learning environment* (IDLE). Es un editor de código que provee funcionalidades que nos permiten crear, modificar, guardar y ejecutar nuestros programas. IDLE se puede instalar en los sistemas operativos Windows, macOS y Linux/Unix.

Puede que el texto del encabezado no sea exactamente igual a la imagen, depende de la versión y el tipo de computadora que utilizamos. El *prompt* nos permite empezar a trabajar; está representado por tres símbolos **>>>**. Por ejemplo, si tecleamos 4+5 aparece el resultado 9 y abajo nuevamente el *prompt* esperando otra instrucción. Este modo se lo denomina de *comando* o *interactivo*. Tiene la ventaja de ejecutar la orden en tiempo real y es muy útil cuando se desea ejecutar comandos básicos o secuencias de comandos reducidas. Por ejemplo, podemos usarlo como una calculadora.



El *prompt* nos facilita realizar pruebas de instrucciones inmediatas pero también podemos crear archivos. Para esto es necesario ingresar a **File → New File** y nos aparece una nueva ventana dentro de la anterior.

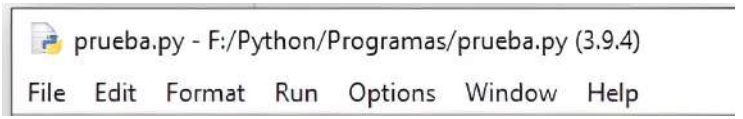


En esta ventana ya no aparece el símbolo *prompt* porque es simplemente un editor de texto. Por eso, se visualiza que los menús de esta ventana no son iguales que los de la ventana principal de IDLE. Además, no permite efectuar pruebas como en el caso anterior en el que simplemente ingresamos una operación aritmética 4+5 para ver su resultado.

En la parte superior izquierda de la ventana aparece la palabra *untitled* que significa sin título, es decir, sin nombre de archivo. Por lo tanto, se recomienda darle uno antes de iniciar el programa. Para eso ingresamos a **File → Save** y le damos un nombre de archivo y la ubicación.

Nota

Si el usuario no escribe la extensión **.py**, IDLE la agrega automáticamente.

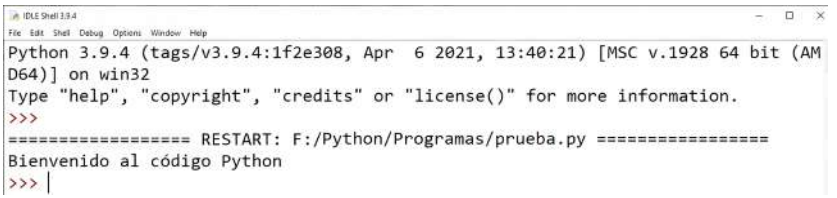


Hacemos la prueba para que simplemente imprima el mensaje **"Bienvenido al código"** escribiendo lo siguiente:

print(«Bienvenido al código Python»)

Luego lo ejecutamos mediante la opción del menú **Run → Run module** (también puede utilizar la tecla F5). Nos vuelve a preguntar si lo queremos guardar.

El resultado se ve reflejado en la ventana principal del IDLE.



El procedimiento de trabajo es siempre el mismo: escribir o modificar el programa en la ventana secundaria, guardar y ejecutarlo. La salida del programa se muestra en la ventana principal.

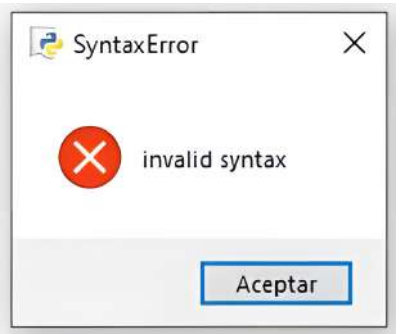
También se pueden ejecutar en el IDLE los programas creados previamente. Los programas se abren de dos formas: desde el explorador con doble clic o desde IDLE accediendo al menú y luego **File → Open**.

Nota
El IDLE colorea el texto de acuerdo con su sintaxis. Los colores ayudan a identificar los distintos tipos de elementos y a localizar errores. El programa permite modificar los colores y las fuentes ingresando al menú y luego **Options → Configure** IDLE.

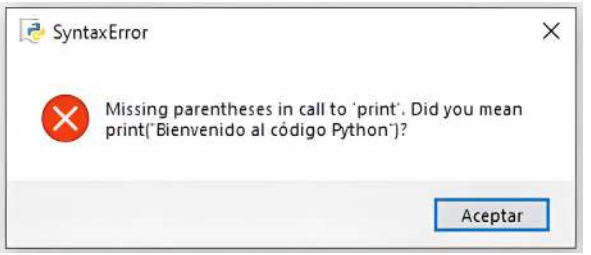
4. 2. Tipos de errores: sintácticos y semánticos

Los *errores sintácticos* ocurren cuando se produce o se encuentra algún error de sintaxis al ejecutar una instrucción o un programa. Python genera un mensaje de error en inglés que indica dónde se ha producido y una descripción del tipo de error. A veces el mensaje de error se muestra en la ventana principal de IDLE y otras como ventana emergente.

Un ejemplo es el siguiente: **prin(«Bienvenido al código Python»)**. El error lo cometemos en la instrucción **prin** (sin la **t** final). Entonces el resultado es **"NameError: name ‹prin› is not defined"**. Otro error puede ser: **print(Bienvenido al código Python»)** pues faltan las primeras comillas del mensaje, entonces aparece la siguiente ventana emergente:



Otro error puede ser que olvidemos los paréntesis **print "Bienvenido al código Python»**, entonces surge la siguiente ventana:



Atención
Es muy común cometer errores sintácticos al iniciarse en el mundo de la programación, a veces por tipeo, otras por distracción o simplemente por no recordar una expresión. Esto se modifica a medida que practicamos. Los errores forman parte fundamental del proceso de aprendizaje. No hay que desesperarse; cada error es una oportunidad para mejorar y avanzar.

En tanto, los *errores semánticos* suceden cuando tenemos una falla lógica en el algoritmo. Se consideran los peores ya que son difíciles de detectar porque el programa se ejecuta sin ningún mensaje de error, por lo tanto, obtenemos un resultado que puede pasar por verdadero.

Por ejemplo, si en alguna línea del algoritmo pensamos calcular el cuadrado de un número **b** para asignarlo a la variable **a** y escribimos **a=b*2** en vez de **a=b**2**, para **b** igual a 2 esto funciona ya que la respuesta es 4 (la misma en ambas ecuaciones), pero no sucede lo mismo con otros números.

4. 3. Pasaje de pseudocódigo a código

Ahora que conocemos el editor IDLE trabajamos siempre en la ventana secundaria en la que podemos guardar nuestros programas. Se aconseja hacerlo con nombres representativos y no extensos, por ejemplo **Ejerc_1.py**.

Comenzamos a transformar el primer ejercicio (ejemplo 1) que vimos en pseudocódigo del capítulo 2. Este ejemplo tiene como consigna: Diseñar un algoritmo que permita el ingreso de dos números para mostrar su suma.

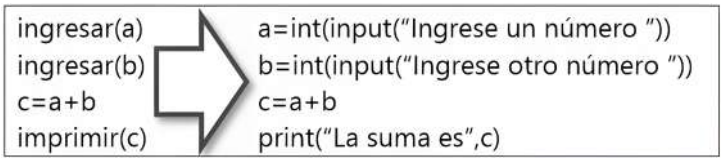
```
ingresar(a)
ingresar(b)
c=a+b
imprimir(c)
```

Los ingresos en Python se efectúan con la instrucción **input** pero su transformación al Python es: **variable=tipo_de_variable(input("Cartel"))**. En este caso, el tipo de variable entera se la conoce como **int** que es la abreviatura de **integer** y el cartel es para informar al usuario que debe ingresar los datos.

Para el ejemplo anterior, la primera línea será **a=int(input("Ingrese un número ")**, siendo la segunda similar **b=int(input("Ingrese otro número ").**

Las asignaciones no sufren cambios y el imprimir será **print("Cartel", variable)** en nuestro caso: **print("La suma es ",c)**.

Entonces finalmente tendremos:



En realidad la codificación exacta de **ingresar(a)** es **a=int(input())**, es decir que **input** no contiene ningún cartel, pero para mejorar la interface agregamos un mensaje como: **"ingrese un número "**.

Nota

Los operadores de **Entrada/Salida** son:

- **input:** devuelve el valor ingresado por teclado tal como se lo digita.
- **print:** escribe el valor de los argumentos que se le pasan. Las cadenas de texto son impresas sin comillas y un espacio en blanco es insertado entre los elementos.

---

4. 4. Tipos de datos numéricos

Python soporta cuatro tipos de datos numéricos diferentes:

- int (entero con signo).
- long (entero largo).
- float (flotantes): representan números reales y se escriben con un punto decimal dividiendo la parte entera y fraccional.
- complex (números complejos).

Los tipos de datos numéricos en un programa pueden ir cambiando. Si en una operación aritmética hay datos de distinto tipo, el resultado de la operación será la del tipo más general. Por ejemplo: **>>>4*2.6** el resultado será 10.4. El conjunto de números flotantes abarca a los enteros.

Hagamos una prueba con el programa del ejemplo 1, pero cambiando el ingreso de **b** como número flotante:

```
a=int(input("Ingrese un número "))
b=float(input("Ingrese otro número "))
c=a+b
print("La suma es",c)
```

Si probamos ingresar, por ejemplo, 4 y luego 5.7, el resultado será 9.7. Los enteros suelen ocupar menos memoria que los flotantes. Un número flotante consta de dos partes: mantisa y exponente. El exponente se separa de la mantisa con la letra **E**, por ejemplo: El número 2E3 significa que tiene mantisa 2 y exponente 3, y representa el número $2*10^3$, es decir, 2000. El número 2.3E-3 significa que tiene mantisa 2.3 y exponente -3 y representa el número $2{,}3*10^{-3}=0{,}0023$. Recomendamos ver la norma internacional IEEE Standart 754.

Nota
Crear el programa del ejemplo 2 del capítulo 2 siendo **h** y **r** números flotantes.

4. 5. Comentarios en Python

Python, como la mayoría de los lenguajes, permite al programador introducir comentarios en el código fuente, pero que el compilador ignora siempre. Existen dos formas de comentar. Una es cuando lo hacemos para una línea. En este caso, se declara el comentario cuando comienza con el símbolo #. Entonces, el intérprete de Python encuentra # en el código e ignora todo lo que hay después de ese símbolo y no produce ningún error. La otra forma es usando triple apóstrofes (**'''**) al inicio y al final del bloque de código para comentar más de una línea.

Vemos el ejemplo 1 que contiene los dos tipos de comentarios:

```
# Ejemplo 1
a=int(input("Ingrese un número ")) # Se ingresa un número entero
b=float(input("Ingrese otro número ")) # Se ingresa un número flotante
c=a+b
print("La suma es",c)   '''El resultado de c será un número flotante
porque abarca a los enteros'''
```

En el ejemplo tenemos tres comentarios de línea con **#** y al final un bloque de comentarios entre apóstrofes.

4. 6. Estructura de control de decisión «if», «else» y «elif»

Retomando el ejemplo 3 del capítulo 2 que pide ingresar dos números para mostrar por mensaje cuál es el mayor, el pasaje de pseudo a código será el que se muestra en la tabla.

Pseudocódigo	Código

ingresar(a)
ingresar(b)
si(a>b):
  imprimir(a)
sino:
  imprimir(b)
	a=int(input("Ingrese un número "))
	b=int(input("Ingrese otro número "))
	if(a>b):
	  print("El mayor es",a)
	else:
	  print("El mayor es",b)

Notamos que los pasajes son directos: el **si** por el **if** y el **sino** por el **else**. Lo mismo sucede con la indentación.

Atención
Para mejorar la comunicación con el usuario en la codificación agregamos **"carteles"** en las entradas y salidas **input/print** a diferencia de los ejemplos vistos en pseudocódigo.

El ejemplo 4 del capítulo 2 pedía mostrar el perímetro de una circunferencia, siempre y cuando el radio que se ingresa sea mayor a 0 (controlar dicho ingreso).

```
# Ejemplo 4
PI=3.1416
r=float(input("Ingrese el radio de la circunferencia "))
if(r>0):
   perim=2*PI*r
   print("El perímetro de la circunferencia es",perim)
```

Nota
Los operadores relacionales **>, <, >=, <=, = =, !=** se mantienen como en el capítulo 2. Lo mismo sucede con la asignación de las constantes como es el caso de **PI=3.1416**.

Atención
Podríamos evitar el uso de la variable **perim** (perímetro) generando el cálculo dentro del **print** de la siguiente manera:
print(«El perímetro de la circunferencia es «,2*PI*r).

Desafío
Realizar la codificación del ejemplo 5 del capítulo 2.

El ejemplo 6 del capítulo 2 tiene el caso de la estructura de control anidada, cuyo enunciado pide mostrar el número más grande entre dos ingresados por teclado, además de si los dos números son iguales mostrar el cartel **"Son iguales"**.

Pseudocódigo	Código

ingresar(a)
ingresar(b)
si(a>b):
  imprimir("El mayor es",a)
sino:
  si(a==b):
  imprimir("Son iguales")
  sino:
  imprimir("El mayor es",b)
	a=int(input("Ingrese un número "))
	b=int(input("Ingrese un número "))
	if(a>b):
	  print("El mayor es",a)
	else:
	  if(a==b):
	  print("Son iguales")
	  else:
	  print("El mayor es",b)

Atención
Se recomienda ir probando los programas en el IDLE a medida que vamos avanzando.

Desafío
Crear un programa comenzando con la decisión **a** igual a **b**, y otro comenzando con **b** mayor a **a**.

La solución anterior podría tener una modificación en la que «else» se fusiona con el «if» formando el «elif», como a continuación:

La ventaja es una reducción en las líneas de comando y en la indentación. De esta forma, muchas veces se entiende mejor el algoritmo. A continuación, vemos otro ejemplo que contiene el **elif**.

Ejemplo 30: Se lee desde el teclado la edad de una persona, y se desea mostrar por pantalla el grupo etario al que pertenece según la siguiente clasificación: grupo A (menores a 16 años), grupo B (entre 16 y 19 años), grupo C (entre 20 y 34 años) y grupo D (mayores a 34 años).

```
# Ejemplo 30
edad=int(input("Ingrese la edad "))
if(edad<16):
   print("Se encuentra en el grupo A")
elif(edad>=16 and edad<20):
   print("Se encuentra en el grupo B")
elif(edad>=20 and edad<35):
   print("Se encuentra en el grupo C")
else:
   print("Se encuentra en el grupo D")
```

Nota
Este tipo de estructura también se conoce como *selección de opción múltiple*.

Vemos a continuación el ejemplo 7 del capítulo 2 que presenta estructuras de decisiones independientes.

Pseudocódigo	Código

ingresar(n1)
ingresar(n2)
si(n1<0):
  n1=(-1)*n1
si(n2<0):
  n2=(-1)*n2
s=n1+n2
imprimir("La suma es",s)
	n1=int(input("Ingrese un número"))
	n2=int(input("Ingrese un número"))
	if(n1<0):
	  n1=(-1)*n1
	if(n2<0):
	  n2=(-1)*n2
	s=n1+n2
	print("La suma es",s)

Nota
Para mejorar la prolijidad y entendimiento de los **input** que contienen una leyenda, se recomienda agregar un espacio en blanco al final del mensaje para separarlo del ingreso. Por ejemplo, si fuera:
int(input("Ingrese un número")) se supone que el usuario ingresa el número 23, en la pantalla muestra **"Ingrese un número23"**.

El ejemplo 9 del capítulo 2 pedía lo siguiente: Diseñar un algoritmo que solicite al usuario un número natural y verifique que el número ingresado se encuentre dentro de la primera docena de números naturales, es decir, entre el 1 y el 12.

Pseudocódigo	Código

ingresar(n)
si(n>=1 and n<=12):
  imprimir("Está en la 1º docena")
	n=int(input("Ingrese un número "))
	if(n>=1 and n<=12):
	  print("Está en la 1º docena")

Al igual que los operadores relacionales, los operadores lógicos «and», «or» y «not» se mantienen iguales que en el pseudocódigo.

Nota
Desarrollar en Python el ejemplo 10 del capítulo 2 usando operadores lógicos.

Con los operadores matemáticos especiales sucede lo mismo, se mantienen como en el pseudocódigo.

Vemos el ejemplo 11 del capítulo 2 en el que se ingresa un valor numérico entero para mostrarlo elevado al cuadrado y a cubo, $\mathbf{n^2}$ y $\mathbf{n^3}$.

```
# Ejemplo 11
n=int(input("Ingrese un número "))
r2=pow(n,2)  # Obtiene el cuadrado del número n ingresado
r3=pow(n,3)  # Obtiene el cubo del número n ingresado
print(n," a la 2 es",r2)
print(n," a la 3 es",r3)
```

Atención
El **print** permite mostrar más de una variable, en este caso para el primer **print n** y **r2** y para el segundo **n** y **r3** alternado por carteles y separados por comas. Recordar que se pudo evitar el uso de las variables directamente haciendo el cálculo en el **print** de la siguiente forma:

```
print(n," a la 2 es",pow(n,2))
print(n," a la 3 es",pow(n,3))
```

Nota
Desarrollar en Python el ejemplo 12, 13 y 14 del capítulo 2 usando operadores matemáticos y operadores lógicos.

4. 7. Problemas: estructura de control de decisión con Python

Programar en Python los problemas del 1 al 22 del capítulo 2. Se recomienda leer el enunciado de cada problema para el desarrollo en código, tratando de no ver los resueltos en pseudocódigo.

4. 8. Estructuras de control de repetición «for»

La estructura de repetición «para» del pseudocódigo se indica con la palabra reservada «for» en Python. A continuación, se muestra su especificación:

Pseudocódigo	Python
para i en rango(N):	for i in range(N):

Siendo **i** el índice del ciclo y **N** el número de vueltas.

Tomemos el ejemplo 15 del capítulo 3 donde nos pedían hallar el promedio de las edades de 8 personas.

Pseudocódigo	Código

sum=0
para i en rango(8):
  ingresar(e)
  sum=sum+e
prom=sum/8
imprimir("El promedio es",prom)
	sum=0
	for i in range(8):
	  e=int(input("Ingrese la edad "))
	  sum=sum+e
	prom=sum/8
	print("El promedio es",prom)

Vemos ahora el ejemplo 20 del capítulo 3 en el que se pedía mostrar la tabla de multiplicación del número 6.

Pseudocódigo	Código

para i en rango(1,11):
  imprimir("6 x ",i,"=", 6 * i)
	for i in range(1,11):
	  print("6 x ",i,"=", 6 * i)

Nota

En algunos casos el valor que contiene el **range** puede ser ingresado previamente por el usuario.

A continuación, se presenta el ejemplo.

```
'''Ejemplo 20 (pero el usuario decide la
cantidad de productos de la tabla del 6'''
n=int(input("Ingrese un número "))
for i in range(1,n):
    print("6 x ",i,"=", 6 * i)
```

Nota
Recordar que los comentarios en el código que superan más de una línea (renglón) utilizan al comienzo y al final los tres apóstrofes (''').

El tratamiento en código con las variables sumadoras (acumuladoras) y contadoras no cambia. Vemos el ejemplo 22 que pedía ingresar 100 números enteros para saber cuántos son divisibles por 3. Se debe recordar que hay que descartar previamente el 0.

```
#Ejemplo 22
contar=0
for p in range(100):
    n=int(input("Ingrese un número "))
    if(n!=0 and n%3==0):
    contar=contar+1
print("La cantidad de números divisibles a 3 es",contar)
```

Nota
Para poner a prueba el programa anterior es conveniente cambiar el valor 100 por otro más pequeño, por ejemplo 10.

Los operadores matemáticos que se utilizan para hacer cálculos especiales son los siguientes:

Operación	Símbolo	Sintaxis
Potencia	pow	pow(N1,N2)
Raíz cuadrada	sqrt	sqrt(N)
Valor absoluto	abs	abs(N)
Seno	sin	sin(N)
Coseno	cos	cos(N)
Tangente	tan	tan(N)
Logaritmo natural	log	log(N)
Logaritmo base 10	log10	log10(N)
Exponencial e	exp	exp(N)
Módulo	%	N1%N2
Redondear	round	round(N1,N2)
Truncar	trunc	trunc(N)

La sintaxis se mantiene pero debemos agregar la librería para que funcionen.

4. 9. Librerías

Las librerías (o *bibliotecas*, por su traducción al español) de Python representan un conjunto de funcionalidades que facilitan el trabajo del programador. Cada una de las librerías de Python dispone de diferentes módulos con funciones específicas, por ejemplo, librerías matemáticas, de juegos, entre otras categorías. Para incluir y poder utilizar las funcionalidades de una librería en los programas, es necesario usar el comando **import** y el nombre de la librería a importar. Ejemplo en el caso de las operaciones trigonométricas: **import math**.

Ejemplo 31: Ingresar en radianes el ángulo de una circunferencia para hallar y mostrar la tangente, el coseno y el seno.

```
import math
radianes=float(input("Ingrese el ángulo en radianes "))
seno = math.sin(radianes)
coseno = math.cos(radianes)
tangente = math.tan(radianes)
print(seno)
print(coseno)
print(tangente)
```

En este caso cada vez que usemos una función trigonométrica debemos agregar delante **math.** También es posible en nuestro programa renombrar las librerías dandoles otro nombre que funcione como una especie de sinónimo, por ejemplo: **import math as M** entonces en lugar de escribir **math** escribimos **M**.

```
import math as M
radianes=float(input("Ingrese el ángulo en radianes "))
seno = M.sin(radianes)
coseno = M.cos(radianes)
tangente = M.tan(radianes)
print(seno)
print(coseno)
print(tangente)
```

Ejemplo 32: Leer desde el teclado el valor en grados del ángulo de una circunferencia. Luego calcular y mostrar la tangente, el coseno y el seno. En este caso, necesitamos π **(pi)** para convertir grados a radianes.

```
import math as M
grados=float(input("Ingrese el ángulo en grados "))
radianes = (grados*M.pi)/180
seno = M.sin(radianes)
coseno = M.cos(radianes)
tangente = M.tan(radianes)
print(seno)
print(coseno)
print(tangente)
```

Para usar π también hace falta la librería Math que en este caso la llamamos **M**.

Seguramente, los resultados que se obtienen en estas pruebas arrojan muchos decimales después de la coma. Para redondear lo hacemos con el comando **round**, por ejemplo, **seno=round(M.sin(radianes),2)** entonces tendrá dos decimales. Probar con el cálculo del coseno y la tangente.

Importante

Los operadores (**pow**, **abs**, **%** y **round**) no necesitan la librería Math, el resto de la tabla sí.

En los capítulos se hace uso de diversas librerías.

Nota

Realizar la prueba de codificar en Python el ejemplo 28 del capítulo 3, pero los promedios deben ser redondeados a 2 decimales.

En el ejemplo 24 del capítulo 3 se pedía el ingreso de un número entero para mostrar de modo descendente y ordenado sus consecutivos hasta culminar con el 1. Por ejemplo: si **N** es 8 entonces debe mostrar el 8, 7, 6,..., 2, 1.

```
#Ejemplo 24
n=int(input("Ingrese un número "))
for i in range(n):
    print(n-i)
```

El ejemplo 24 también se puede desarrollar utilizando la estructura de control **mientras** (como veremos más adelante).

Python a diferencia de otros lenguajes permite invertir el rango usando la funcionalidad **reversed** y obtener un orden descendente: N, N-1, N-2,..., 2, 1.

```
#Ejemplo 24 (usando funcionalidad reversed
n=int(input("Ingrese un número "))
for i in reversed(range(n)):
    print(i)
```

Nota
Realizar en Python el ejemplo 25 del capítulo 3.

4.10. Máximos y mínimos

Recurrimos al ejemplo 26 del capítulo 3 para hallar la persona de mayor altura sabiendo que se toman 20 muestras. La unidad de medida es en metros, por ejemplo: 1.67 (un metro con 67 centímetros).

```
max_alt=0
for i in range(20):
   altura=float(input("Ingrese la altura "))
   if(altura>max_alt):
      max_alt=altura
print("La mayor altura registrada es",max_alt)
```

Nota
Recordar que al ingresar números decimales se debe usar el **float** para el **input**.

Realizar en Python el ejemplo 27, para hallar la menor altura, y el ejemplo 28 que pide la altura máxima y su ubicación (ambos del capítulo 3).

4.11. Estructura de control de repetición «while»

La estructura de repetición «mientras» del pseudocódigo se indica con la palabra reservada «while» en Python. A continuación, se muestra su especificación:

Pseudocódigo	Python
mientras(¿condición?):	while(¿condición?):

En este caso, la condición es la prueba verdadera para la iteración.

Vemos directamente el ejemplo 16 del capítulo 3 que pedía calcular la suma de números ingresados por teclado hasta que se ingrese 0.

```
#Ejemplo 16
s=0
n=int(input("Ingrese un número "))
while(n!=0):
    s=s+n
    n=int(input("Ingrese un número "))
print("La suma acumulada es",s)
```

Nota

Simplemente el «mientras» del pseudocódigo pasa a ser el «while» del código.

A continuación, vemos cómo codificar el ejemplo 17 que pedía hallar el promedio de números **n** ingresados por teclado hasta que aparezca uno que sea par y mayor o igual a 20.

```
#Ejemplo 17
c=0 ; s=0
n=int(input("Ingrese un número "))
while(n%2==0 and n>=20):
        c=c+1
        s=s+n
        n=int(input("Ingrese un número "))
print("El promedio es",round(s/c,2))
```

Nota

Las asignaciones pueden escribirlas en una misma línea separadas por punto y coma como en el ejemplo: **c=0**; **s=0** y además para reducir los decimales del promedio usamos la función **round**.

Si de entrada ingresamos un número que no cumple la condición del **while** entonces podemos cometer el error de efectuar en el **print** una división por 0. Al hacer la prueba aparece por pantalla el siguiente mensaje:

```
Traceback (most recent call last):
  File "F:\UNRN\Python\Programas\prueba11.py", line 7, in <module>
    print("El promedio es",round(S/C,2))
ZeroDivisionError: division by zero
```

Para eso debemos validar el dato antes de cometer el error.

Nota

Editar el ejemplo anterior solucionando el inconveniente. Luego, codificar y ejecutar el ejemplo 18 del capítulo 3.

Codificamos el ejemplo 19 que pide: A la salida de un cine se hace una serie de preguntas a cada espectador para saber finalmente lo siguiente:

a. Porcentaje de mujeres.
b. Promedio de calificación de la película. Cada espectador califica del 1 al 10.
c. Cantidad de personas mayores a 60 años de edad.

La cantidad de personas no se sabe por lo tanto se termina la carga de datos respondiendo con **N** a la pregunta: ¿Desea ingresar datos S/N?

```
#Ejemplo 19
cm=0 ; ct=0 ; scal=0 ; c60=0
seguir=str(input("¿Desea ingresa  datos S/N? "))  ⇨ imprimir("¿Desea ingresar datos S/N? ")
while(seguir=='S'):                                 ingresar(seguir)
   genero=str(input("Ingrese el género "))
   edad=int(input("Ingrese la edad "))
   calificacion=int(input("Ingrese la calificación "))
   ct=ct+1
   if(genero=='Mujer'):
      cm=cm+1
   if(edad>60):
      c60=c60+1
   scal=scal+calificacion
   seguir=str(input("¿Desea ingresar datos S/N? "))  ⇨ imprimir("¿Desea ingresar datos S/N? ")
print("El % de mujeres es",round(100*cm/ct,2))          ingresar(seguir)
print("El promedio de calificación de los espectadores es",round(scal/ct,2))
print("La cantidad de personas mayores a 60 es",c60)
```

Al usar el pseudocódigo para el ingreso en la variable «seguir» se necesitaban dos líneas de comando «imprimir» e «ingresar», pero con el Python simplemente se funde en una línea que hace las dos cosas. Además, al ser un dato alfanumérico se debe anteponer al **input** la función **str** que significa *string* (cadena de caracteres), lo mismo sucede con el ingreso del género de la siguiente forma: **genero = str(input(«Ingrese el género «))**.

Nota

Cuando ingresamos datos alfanuméricos que serán utilizados en estructuras de control (decisión/repetición) es conveniente convertirlos a mayúsculas. No es lo mismo ingresar **'s'** que **'S'** ya que en el **while(seguir=='S')** se notaría la diferencia. Lo mismo sucede con el género, por ejemplo, si ingresamos **'mujer'**, el algoritmo no lo toma en

cuenta ya que está esperando que sea **'Mujer'** (con la M mayúscula). Para solucionar este inconveniente se debe convertir el contenido de las variables alfanuméricas agregando la función **upper()** de la siguiente forma: **if(seguir.upper()=='S')** y **if(genero.upper()=='MUJER')**.

```
#Ejemplo 19  (con el agregado de la función upper)
cm=0 ; ct=0 ; scal=0 ; c60=0
seguir=str(input("¿Desea ingresar datos S/N? "))
while(seguir.upper()=='S'):
   genero=str(input("Ingrese el género "))
   edad=int(input("Ingrese la edad "))
   calificacion=int(input("Ingrese la calificación "))
   ct=ct+1
   if(genero.upper()=='MUJER'):
      cm=cm+1
   if(edad>60):
      c60=c60+1
   scal=scal+calificacion
   seguir=str(input("¿Desea ingresar datos S/N? "))
print("El % de mujeres es",round(100*cm/ct,2))
print("El promedio de calificación de los espectadores es",round(scal/ct,2))
print("La cantidad de personas mayores a 60 es",c60)
```

Atención
A este ejemplo le falta validar. Realizar la validación.

Volviendo al ejemplo 24 para mostrar números enteros ordenados de modo descendente, también es posible realizarlo usando el **while**:

```
#Ejemplo 24
n=int(input("Ingrese un número "))
i=n
while(i>0):
   print(i)
   i=i-1
```

En este caso, sin usar el comando **reversed**, se le asigna de entrada a **i** el valor de **n** para ir mostrando a medida que se decrementa con **i=i-1** dentro del dominio del **while**, mientras **i** sea mayor a 0.

4.12. Números aleatorios

Para generar números aleatorios en Python, usamos la librería Random. Este módulo ofrece una serie de funciones que generan números o elementos aleatorios. A continuación, vemos las funciones «randint», «uniform» y «choice» que consideramos más importantes.

Partimos de un ejemplo muy sencillo que pide simular el azar de una moneda para saber si es cara o seca. Establecemos la suposición de que cara es 0 y seca es 1, entonces:

```
import random
x=random.randint(0,1)
print(x)
```

Siempre que usamos librerías deben importarse al inicio del algoritmo. La función **randint** genera al azar números enteros (desde, hasta).

Nota
Es importante recordar que podemos sustituir el nombre de la librería por otro nombre como en el siguiente algoritmo.

```
import random as rd
x=rd.randint(0,1)
print(x)
```

Ejemplo 33: Crear un programa que genere 15 números enteros aleatorios del 0 al 10.

```
import random as rd
for i in range(15):
    x=rd.randint(0,10)
    print(x)
```

Nota
Podríamos evitar usar la variable **x** directamente imprimiendo **print(rd.randint(0,10))**.

Si queremos obtener números aleatorio flotantes tenemos que usar la función **uniform.** Vemos el ejemplo anterior pero con números flotantes de hasta dos decimales.

```
import random as rd
for i in range(15):
    x=rd.uniform(0,10)
    print(round(x,2))
```

Nota
Recordar que **round** no necesita librería.

Ejemplo 34: Crear un programa que elija colores al azar de la siguiente lista: blanco, amarillo, rojo, rosa y azul hasta que salga el rojo. Obviamente pueden repetirse.

```
import random as rd
colores=('blanco','amarillo','rojo','rosa','azul')
x='activar'
while(x!='rojo'):
    x=rd.choice(colores)
    print(x)
```

Nota
Para que pueda entrar al bucle **while** tuvimos que asignarle un valor a **x** (distinto a **'rojo'**). La función **choice** sirve para elegir al azar algún elemento del rango **colores**. Los elementos deben expresarse entre apóstrofes y separados por comas.

Ejemplo 35: Crear un programa que genere y muestre 50 números al azar que correspondan a las caras de un dado y hallar el porcentaje de pares e impares obtenidos.

```
import random as rd
cpares=0 ; cimpares=0
for i in range(50):
    x=rd.randint(1,6)
    print(x)
    if(x%2==0):
        cpares=cpares+1
    else:
        cimpares=cimpares+1
print(100*cpares/50)
print(100*cimpares/50)
```

Ejemplo 36: Vamos a jugar un poco. Crear un programa que simule arrojar dados entre dos jugadores **J1** y **J2**. En cada tirada se debe comparar los valores de los dados y el jugador que saque la mayor cantidad suma un punto, pero en caso de empate se reparte un punto a cada uno. En total arrojan cinco veces y al final el algoritmo debe informar quien es el ganador. Puede que exista empate y lo debe informar también.

```
import random as rd
pa=0;pb=0
for i in range(5):
    a=rd.randint(1,6)
    b=rd.randint(1,6)
    print("Dados: J1->",a,"vs J2->",b)
    if(a>b):
        pa=pa+1
    else:
        if(a<b):
            pb=pb+1
        else:
            pa=pa+1
            pb=pb+1
    print("Puntaje J1=",pa,"y J2=",pb)
    x=input("presione una tecla para seguir")
if(pa>pb):
    print("Felicitaciones: Ganador J1")
else:
    if(pa<pb):
        print("Felicitaciones: Ganador J2")
    else:
        print("Empate !!!")
```

Las variables **pa** y **pb** se encargan de mantener la contabilidad a medida que se conoce quien gana o empata (lógicamente comienzan siempre con el valor 0). Mientras que **a** y **b** se encargan de obtener el valor de cada dado que es inmediatamente informado con el primer **print**, una vez hecha la comparación con los **if** anidados se muestra el puntaje que van obteniendo ambos jugadores.

Al finalizar el juego (fuera del ciclo **for**) se comparan los puntajes **pa** y **pb** para saber quién ganó (puede haber empate).

Nota
En este caso el **x=input("presione una tecla para seguir")** sirve para hacer una pausa en cada partida y así poder ir controlando el juego. El contenido de la variable **x** no tiene sentido ni efecto en el programa.

Ejemplo 37: Ingresar los dos catetos de un triángulo rectángulo para hallar la superficie. Los datos ingresados deben ser validados, es decir, ambos deben ser positivos. Agregar comentarios de línea con el símbolo **#** y de bloque con el símbolo **'''**.

```
# Ejemplo 37 Capítulo 4
a=int(input("Ingrese cateto a "))
b=int(input("ingrese cateto b "))
while(a<=0):
    a=int(input("Vuelva a ingresar cateto a "))
while(b<=0):
    b=int(input("Vuelva a ingresar cateto b "))
print("\n La superficie del triángulo es",(a*b)/2)

'''En el print "\n" sirve para generar un ENTER'''
```

En el ejemplo tenemos tres comentarios de línea con # y al final un bloque de comentarios entre apóstrofes.

Nota
Muchas veces el uso de comentarios, ya sea por línea o bloques, tiene otro propósito sobre todo en programas extensos o modularizados para ir probando de a sectores.

4. 13. Problemas: estructura de control «for» y «while»

43. Imprimir los números impares del 99 al 3.
44. Dado un número, imprimir su tabla de multiplicar (del 0 a 10).
45. Crear un programa que simule las caras de un dado mostrando 15 números aleatorios.

Nota
Es necesario usar la librería Random para efectuar operaciones aleatorias.

46. Crear un programa que simule el juego de la ruleta y muestre los números que van saliendo hasta que aparezca el 0. Los números de la ruleta van desde el 0 hasta el 36.
47. Hallar e imprimir la cantidad y el promedio de edad de personas que residen en Argentina hace más de 5 años. El total de personas es cargado por el usuario previamente.
48. Hallar e imprimir el porcentaje de mujeres y varones. La carga se termina tecleando una **X**.
49. De un total de 40 turistas que visitan la ciudad de Bariloche en mayo, se desea saber lo siguiente:
    a. Cantidad de turistas argentinos menores de edad.
    b. Promedio de edad de los turistas extranjeros.
    c. Cantidad de turistas que visitan la ciudad por primera vez.

    Se recomienda evaluar primero qué datos se pide a cada turista y en qué momento.
50. Se desea conocer el promedio de peso de pacientes de una guardia del hospital y además el mayor y el menor peso registrado. No se sabe la cantidad de personas.
51. Completar e imprimir una lista que contenga los valores trigonométricos (coseno, seno y tangente) a medida que van ingresando los siguientes valores en grados: 0, 30, 45, 60, 90, 120, 180, 270 y 360. Los resultados deben tener 4 decimales.

    Por cada línea que genera el **print** debe aparecer el grado, el coseno, el seno y la tangente.

    Recordar que los grados deben pasarse a radianes e importar la librería Math.
52. A la salida de un museo a cada visitante se le hace una serie de preguntas para saber y mostrar al final del día lo siguiente:
    a. La cantidad de extranjeros.
    b. El promedio de edad de argentinos.
    c. Porcentaje de argentinos que tienen entre 18 y 22 años de edad (por sobre el total de personas).
    d. La persona de menor edad (mostrar la edad y si es extranjero o nacional).

    Agregar como comentario al principio del algoritmo el enunciado del problema.
53. Se sabe que el volumen de un cono es: $V = \frac{\pi hr^2}{3}$ siendo r (radio) y h (altura).

    Tenemos un registro de 60 conos (solo el radio y su altura) y se desea saber:
    a. La altura más baja de los conos.
    b. Mostrar para cada cono su volumen.

En este caso, el usuario puede cometer errores en el ingreso con el radio y la altura (contemplar ese inconveniente).

54. Crear un algoritmo que muestre 10 números múltiplos de 5. Estos son ingresados por el usuario, pero puede equivocarse. Entonces debe volver a ingresar hasta que sea correctamente múltiplo de 5.
55. Crear un programa que muestre 30 números que salen de la ruleta para saber finalmente lo siguiente:
    a. El número más alto.
    b. La cantidad de números que salieron y son de la segunda docena.
    c. El promedio de números impares.
    d. El porcentaje de ceros obtenidos.

    Agregar como comentario al principio del algoritmo el enunciado del problema.
56. Crear una lista de 100 elementos al azar que contenga las vocales en mayúscula.

Capítulo 5

Arreglos: vectores y matrices

> Programar sin una arquitectura o diseño en mente es como explorar una gruta solo con una linterna: no sabes dónde estás, dónde has estado ni hacia dónde vas.
> Danny Thorpe*

Temas:

Arreglos unidimensionales y bidimensionales. Vectores y matrices. Uso de la librería Numpy. Ordenamiento. Máximos y mínimos. Vectores paralelos y operaciones aritméticas. Asignaciones condicionales. Copia y concatenación de arreglos. Filtros. Gráficas de funciones. Problemas.

5.1. Arreglos

Un *arreglo* es una estructura de datos que permite almacenar una colección de datos del mismo tipo (por ejemplo: los apellidos de un grupo de estudiantes de un curso, los salarios de los empleados de una empresa). Se puede asociar a cada elemento un número de orden en la colección (por ejemplo: nº de legajo). Esta última característica permite el acceso directo y el recorrido secuencial de sus elementos.

Los arreglos se pueden clasificar de acuerdo a la cantidad de dimensiones en:

- Vector: arreglo de 1 dimensión (unidimensional).
- Matriz: arreglo de 2 dimensiones.
- Multidimensionales (> 2 dimensiones).

5.2. Vectores

Los vectores se utilizan en distintos ámbitos. Por ejemplo, en el área de la física se usan para magnitudes escalares, en biología como agente orgánico, en geometría como segmentos de recta dirigidos. Pero en informática el *vector* denomina una zona de almacenamiento continuo que alberga un

* Danny Thorpe fue un programador estadounidense conocido principalmente por su trabajo en el lenguaje de programación Delphi, además fue jefe de herramientas de Windows y net en Borland Corporation. Falleció en octubre de 2021.

conjunto de datos del mismo tipo, es decir, homogéneos. El número de elementos es finito y cada uno está ubicado en una posición determinada.

Vemos la representación gráfica de un vector:

Fisher	Karpov	Petrosian	Carlsen	Capablanca	Kasparov	Najdorf

Se trata de una lista de históricos maestros del ajedrez. El conjunto es finito, su tamaño es fijo porque contiene siete elementos y es homogéneo porque son del mismo tipo (*string*) que están almacenados en un vector de posiciones contiguas. El vector debe tener un único nombre de variable: le ponemos **ajedrez** (relacionado a los elementos). Cada casillero tiene asignado un índice, en este caso es del 0 al 6.

El acceso de modo directo se realiza con el nombre de la variable y el índice, por ejemplo: **ajedrez[2]**. En este caso, su contenido es Petrosian.

ajedrez[0] → Fisher
ajedrez[1] → Karpov
ajedrez[2] → Petrosian
ajedrez[3] → Carlsen
ajedrez[4] → Capablanca
ajedrez[5] → Kasparov
ajedrez[6] → Najdorf

Nota

¿Por qué es importante conocer la cantidad de dimensiones de un arreglo? Porque las mismas determinan la cantidad de índices necesarios para recorrer la estructura.

Vemos otro ejemplo:

Se desea almacenar las temperaturas mínimas registradas en los últimos siete días de julio. Una vez finalizado el proceso se debe identificar e informar el día más cálido.

En este problema podemos utilizar un vector de tamaño 7, en el que en cada posición se registra un valor de temperatura (día 1, día 2,...., día 7). A este vector lo vamos a llamar **vecTemp**.

vecTemp	4	0	-1	9	11	12	8
	día 1	día 2	día 3	día 4	día 5	día 6	día 7

5. 2. 1. Arreglos en Python: NumPy

Python dispone de diferentes tipos de datos estructurados: listas, tuplas, cadenas de caracteres (*strings*), diccionarios y conjuntos. Todos permiten almacenar elementos no necesariamente del mismo tipo. Para trabajar con la idea de arreglo presentada anteriormente, Python provee la librería NumPy que proporciona funcionalidades para trabajar con arreglos de números. Esta es muy usada en la computación científica (ciencia de datos, ingeniería, etcétera). El principal beneficio es que permite una generación y manejo de datos extremadamente rápido.

El sitio oficial es https://numpy.org/

El primer paso para utilizar la librería NumPy es descargarla e instalarla. Solo requiere tener instalado Python. Una vez instalada, en los programas necesitamos incluir la librería importándola de esta manera: **import numpy as np**.

Volviendo a los ejemplos de los ajedrecistas y las temperaturas, a continuación presentamos cómo se crean los vectores con Python.

```
import numpy as np
ajedrez=np.array(["Fisher","Karpov","Petrosian","Carlsen","Capablanca","Kasparov","Najdorf"])
```

Para acceder al contenido de un arreglo necesitamos conocer el nombre del arreglo y la posición a la cual queremos acceder. Se debe tener en cuenta que la posición debe ser un valor válido. Y si buscamos imprimir el tercer ajedrecista, agregamos en Python lo siguiente: **print(ajedrez[2])**.

Nota

Recordar que el primer índice del vector es 0 (cero).

La posición a la que se intenta acceder debe respetar el tamaño del vector, el que se indicó al momento de crearlo. Observemos el resultado de ejecutar **print(ajedrez[7])**.

```
Traceback (most recent call last):
  File "F:/UNRN/Python/Programas/numpy3.py", line 4, in <module>
    print(ajedrez[7])
IndexError: index 7 is out of bounds for axis 0 with size 7
```

Nota

Esto sucede por querer mostrar algo que es inexistente. La ubicación 7 está fuera de rango.

Para el caso de las temperaturas, creamos el vector **vecTemp** y mostramos el dato del día 5.

```
import numpy as np
vecTemp=np.array([4,0,-1,9,11,12,8])
print("La temperatura del día 5 es:", vecTemp[4],"grados")
```

Nota
La temperatura del día 5 se ubica en el índice 4.

Cuando se trata de arreglos con contenido numérico se pueden realizar operaciones aritméticas, por ejemplo si queremos hallar el promedio de las temperaturas de los días 1, 3 y 7.

```
import numpy as np
vecTemp=np.array([4,0,-1,9,11,12,8])
suma=vecTemp[0]+vecTemp[2]+vecTemp[6]
print("El promedio es", round(suma/3,2),"grados")
```

Nota
Podríamos haber evitado la variable **suma** y directamente generar la operación dentro del **print** de la siguiente manera:
print("El promedio es", round((vecTemp[0]+vecTemp[2]+vecTemp[6])/3,2),"grados").

Si el arreglo a recorrer es un vector, entonces al contener una dimensión necesitamos solo un índice. Supongamos que deseamos recorrer el vector **vecTemp** y mostrar su contenido.

```
import numpy as np
vecTemp=np.array([4,0,-1,9,11,12,8])
# Recorrido completo vector
for indice in range(0,7,1):
    print(vecTemp[indice])
```

Nota
En este caso, hemos realizado un recorrido completo del vector. Recordar que en Python los arreglos comienzan a indexarse desde 0, por eso en la estructura de control **for** el índice comienza en 0.

Supongamos que queremos hallar el promedio de todas las temperaturas.

```
import numpy as np
vecTemp=np.array([4,0,-1,9,11,12,8])
# Promedio de temperaturas
suma=0
for indice in range(0,7,1):
    suma=suma+vecTemp[indice]
print("El promedio de las temperaturas es", round(suma/7,2),"grados")
```

La librería NumPy presenta muchos métodos incorporados que son funciones ventajosas que permiten reducir las líneas de código como si fueran atajos. Por ejemplo, si queremos crear un vector de tamaño 5 que contenga solo valores 0. Para esto usamos la funcionalidad **zeros** como se muestra en el ejemplo a continuación.

```
import numpy as np
vecEnteros=np.zeros(5)  # Genera un vector con 5 ceros
print(vecEnteros)       # Muestra todos sus elementos
```

Nota

La función **zeros** recibe como parámetro el tamaño del arreglo a crear, en este caso el 5. Este parámetro debe ser siempre un valor mayor a 0. Además no hace falta recorrer con un bucle el vector para imprimir todos sus elementos, basta con hacerlo con la función **print**. También se pueden generar unos usando la función **ones**, por ejemplo: **unos=np.ones(10)**.

Si buscamos repetir un elemento, ya sea numérico o alfanumérico en un vector, podemos usar la función **full()**.

```
import numpy as np
vec1=np.full(7,'a') # Repite 7 veces 'a'
vec2=np.full(5,23) # Repite 5 veces 23
print(vec1)
print(vec2)
```

El resultado es:

```
['a' 'a' 'a' 'a' 'a' 'a' 'a']
[23 23 23 23 23]
```

Nota

La función **full** tiene dos parámetros, el primero es la dimensión del arreglo y el segundo es el elemento que se repite.

Para generar vectores que tengan valores equidistantes dentro de un rango podemos usar la función de NumPy llamada **linspace()**.

```
import numpy as np
vec1=np.linspace(0, 10, 5) # 5 valores equidistantes
vec2=np.linspace(1,10,15) # 15 valores equidistantes
print(vec1)
print(np.round(vec2,2))
```

```
[ 0.   2.5  5.   7.5 10. ]
[ 1.    1.64  2.29  2.93  3.57  4.21  4.86  5.5
  6.14  6.79  7.43  8.07  8.71  9.36 10.  ]
```

Nota

El primer vector **vec1** contiene 5 números que distan 2,5 entre sí desde el 0 hasta el 10 y el segundo contiene 15 números que distan 1,64285714 entre sí. En estos casos mostramos el vector con redondeo de dos decimales.

Aclaración

Cuando utilizamos el **round()** en un **print** lo hacemos solo a efectos de la comprensión visual, sin cambiar su valor. Pero si queremos efectivamente cambiar sus valores, debemos hacerlo antes del **print**. Por ejemplo, luego de generar el vector **vec2** con **linspace()** aplicamos **vec2=np.round(vec2,2)**.

Usamos la función **arange()** en el caso que quisiéramos generar arreglos con valores equidistantes dentro de un rango definido. Por lo general, requiere tres parámetros: el valor inicial, el valor final y el salto (diferencia entre cada valor).

```
import numpy as np
vec1=np.arange(1, 10, 1) # con salto 1
vec2=np.arange(1, 5, 0.5) # con salto 0,5
print("vector 1",vec1)
print("vector 2",vec2)
```

El resultado es:

```
vector 1 [1 2 3 4 5 6 7 8 9]
vector 2 [1.  1.5 2.  2.5 3.  3.5 4.  4.5]
```

Nota

El primer vector contiene números naturales con salto 1 y el segundo vector con salto decimal de 0,5.

Ejemplo 38: Crear un vector de nombre **fibo** con dimensión 10 que contenga ceros y luego ingrese al primer y segundo índice un 1 para generar la sucesión de Fibonacci, recordando que se obtiene el siguiente elemento siempre sumando los dos anteriores, de manera que el resultado sea 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

```
import numpy as np
fibo=np.zeros(10) # Genera un vector con 10 ceros
fibo[0]=1         # Agrega en los dos primeros lugares un 1
fibo[1]=1
for i in range(2,10,1):         # Parte del 3º lugar
    fibo[i]=fibo[i-2]+fibo[i-1] # Suma los dos anteriores
print(fibo)
```

Nota
En este caso inevitablemente se debe usar un bucle **for** para ir agregando elementos nuevos.

Ejemplo 39: Crear un vector de nombre **rule** que contenga 20 números aleatorios de una ruleta. Los números van del 0 al 36.

```
import numpy as np
rule = np.random.randint(0, 36, size=20)
print(rule)
```

Nota
La función de **random.randint** genera números aleatorios enteros entre 0 y 36 y la instrucción **size** confirma la dimensión del vector **rule**.

Ejemplo 40: Recorrer el vector **rule** (generado en el ejemplo anterior) hasta encontrar el primer número 0.

```
import numpy as np
rule = np.random.randint(0, 36, size=20)
print(rule) # Mostramos el vector de numeros aleatorios
indice=0
while (indice<20) and (rule[indice]!=0):
    indice=indice+1
print("El primer cero sale en la posición", indice)
```

El resultado será:

```
[20 32 34 17 29 21 31  8 35 12 17  0 23 33  0 31  8  9  4 32]
El primer cero sale en la posición 11
```

Nota

No fue necesario recorrer todo el arreglo. El **while** termina por dos razones: la primera, recorre la totalidad del vector y no encuentra un 0, la otra opción es que encuentra un 0. Mientras tanto vamos incrementando el índice para finalmente obtener la posición del primer 0. En el ejemplo, el resultado nos dice que fue en la posición 11. Hay que tener en cuenta que la primera posición del vector es el 0.

Atención

¿En qué cambia la solución si ahora se desea mostrar por pantalla todas las posiciones donde aparece el 0? Para buscar es necesario recorrer el vector, y de acuerdo a las condiciones/requerimientos de búsqueda es necesario hacer un recorrido completo, lo que determina la estructura de control adecuada para realizar el recorrido.

5. 2. 2. Máximos y mínimos

Ejemplo 41: Hallar el día más cálido y su temperatura. El total de datos es de 12 y son los siguientes: 4.8, 5.7, 11.0, -9.7, 6.1, 7.0, 13.5, -4.8, 3.3, 13.5, 5.3, -1.1.

```
import numpy as np
vTemp=np.array([4.8,5.7,11.0,-9.7,6.1,7.0,13.5,-4.8,3.3,13.5,5.3,-1.1])
print(vTemp) # Mostramos las temperaturas
print("La temperatura máxima registrada es",np.max(vTemp))
print("y el día es el",np.argmax(vTemp)+1)
```

El resultado será:

```
[4.8, 5.7, 11.0, -9.7, 6.1, 7.0, 13.5, -4.8, 3.3, 13.5, 5.3, -1.1]
La temperatura máxima registrada es 13.5
y el día es el 7
```

Nota

La función **np.max()** permite hallar el máximo y **np.argmax()** su posición. Al tratarse de días le sumamos 1 para evitar el primer día como 0. Tener en cuenta que solo sirve para hallar el primer máximo.

Si por el contrario nuestra búsqueda es el mínimo, las funciones serán **np.min()** y **np.argmin()**, respectivamente.

Ejemplo 42: Del mismo modo que el ejemplo 41, pero puede que sean más de un día los de mayor temperatura. En nuestra lista figuran dos días con la misma temperatura, 13.5 grados.

```
import numpy as np
vTemp=np.array([4.8,5.7,11.0,-9.7,6.1,7.0,13.5,-4.8,3.3,13.5,5.3,-1.1])
print(vTemp) # Mostramos las temperaturas
Tempe_maxima=np.max(vTemp)
for i in range(1,12,1):
    if Tempe_maxima==vTemp[i]:
        print("La máxima es",vTemp[i],"el día Nº",i+1)
```

El resultado será:

```
[4.8, 5.7, 11.0, -9.7, 6.1, 7.0, 13.5, -4.8, 3.3, 13.5, 5.3, -1.1]
La máxima es 13.5 el día Nº 7
La máxima es 13.5 el día Nº 10
```

Nota

Primero en la variable **Tempe_maxima** guardamos el valor más alto de temperatura y luego dentro del bucle **for** detectamos los días que tienen ese máximo con el índice **i+1**.

Importante

Tenemos la opción de pausar un algoritmo utilizando la función **input**, dentro de un ciclo de repetición, de manera que el programa espere a que el usuario presione una tecla (o escriba algo) antes de continuar con la siguiente iteración. Esto se hace simplemente agregando **input("Presione una tecla para continuar...")**.

En el ejemplo 42, la interrupción podría producirse después de cada **print("La máxima es")** para ver detenidamente cada resultado.

5. 2. 3. Ordenar arreglos

Para ordenar un vector solo hace falta utilizar la función **sort()** del Numpy como en el siguiente ejemplo:

```
import numpy as np
vec1=np.random.randint(-10, 10, 12)
print("vector 1",vec1)
vec2=np.sort(vec1) # generar vec2 con vec1 ordenado
print("vector 2",vec2)
```

El resultado es:

```
vector 1 [ -4   2  -4  -7   9  -3  -1   6 -10  -4  -1   3]
vector 2 [-10  -7  -4  -4  -4  -3  -1  -1   2   3   6   9]
```

Nota

Generamos primero un vector con 12 números enteros aleatorios entre -10 y 10 y luego lo asignamos al **vec2**, pero ordenado de modo ascendente.

Para ordenar de modo descendente:

```
import numpy as np
vec1=np.random.randint(-10, 10, 12)
print("vector 1",vec1)
vec2=np.sort(vec1)[::-1] # genera orden descendente
print("vector 2",vec2)
```

El resultado es:

```
vector 1 [ 8  4  2  0  3 -6 -7  6 -3 -5  4  0]
vector 2 [ 8  6  4  4  3  2  0  0 -3 -5 -6 -7]
```

Nota

La instrucción **[:: -1]** indica que el orden del vector será descendente. Recordar, en ambos ejemplos, que **vec1** permanece con los datos generados originalmente. Los cambios se reflejan en **vec2**.

5. 2. 4. Copia de arreglos

Existen distintas formas de copiar arreglos: por referencia y por arreglo.

Presentamos el modo por referencia:

```
import numpy as np
vec1=np.array([6,2,9,8,1])
print(vec1) # Mostrar vector 1
vec2=vec1
print(vec2) # Mostrar vector 2
```

El resultado es:

```
[6 2 9 8 1]
[6 2 9 8 1]
```

Nota

La copia por referencia se realiza asignando la variable de un arreglo a otro. Después de la asignación ambas variables comparten los mismos datos en la misma área de memoria, es decir, que si se produce un cambio en la original, lo mismo sucede en la copia.

Probemos con el siguiente ejemplo:

```
import numpy as np
vec1=np.array([6,2,9,8,1])
print(vec1) # Mostrar vector 1
vec2=vec1
vec1[2]=3 # Efectuamos una modificacion en vec1
print(vec2) # Mostrar vector 2
```

El resultado es:

```
[6 2 9 8 1]
[6 2 3 8 1]
```

Nota
Al compartir la misma área de memoria, lo que cambia en **vec1** lo mismo sucede en **vec2**, es decir que la copia es siempre dependiente del original.

Se muestra la forma por valor:

```
import numpy as np
vec1=np.array([6,2,9,8,1])
print(vec1) # Mostrar vector 1
vec2=vec1.copy()
vec1[2]=3
print(vec2) # Mostrar vector 2
```

El resultado es:

```
[6 2 9 8 1]
[6 2 9 8 1]
```

Nota
En este caso cada vector tiene su espacio de memoria, entonces son independientes. La función **copy()** permite esta independencia.

5. 2. 5. Concatenar arreglos

Concatenar significa ampliar, en el caso de arreglos si aplicamos la función **concatenate()** de NumPy, su resultado es un vector ampliado. En el siguiente ejemplo se concatenan los vectores **vec1** y **vec2** (de dimensiones 5 y 3, respectivamente), entonces su resultado es el **vec3** de dimensión 8.

```
import numpy as np
vec1=np.array([6,2,10,8,1])
vec2=np.array([2,5,7])
vec3=np.concatenate((vec1,vec2))
print(vec3) # Mostrar vector 3
```

La respuesta es:

```
[ 6  2 10  8  1  2  5  7]
```

También se puede hacer con la función **append()**.

```
import numpy as np
vec1 = np.array([6,2,10,8,1])
vec2 = np.append(vec1,[2,5,7])
print(vec2) # Mostrar vector 2
```

El resultado es:

```
[ 6  2 10  8  1  2  5  7]
```

Nota

Si no es necesario generar nuevos vectores, se puede concatenar simplemente haciendo la asignación **vec1=np.append(vec1,[2,5,7])**.

---

5. 2. 6. Filtros

El filtrado en arreglos es esencial en la manipulación de datos y es una habilidad para trabajar fundamentalmente en el área de las matemáticas, en probabilidad y estadística.

Podemos aplicar filtros sin la necesidad de recorrerlos con un bucle. Por ejemplo, si queremos obtener los números pares de un vector y menores a 15:

```
import numpy as np
vec1=np.arange(1, 20, 1)
vec2=vec1[(vec1 % 2 == 0) & (vec1<15)] # Aplicamos filtro
print("vector 2",vec2)
```

El resultado es:

```
vector 2 [ 2  4  6  8 10 12 14]
```

Nota

En este caso, el filtro que aplicamos en **vec1** queda en **vec2** evitando modificar el vector origen, pero si queremos mostrar el resultado sin la necesidad de generar un nuevo vector podemos directamente usar: **print(«vector 1»,vec1[(vec1%2 == 0)&(vec1<15)])**.

Importante
Los operadores lógicos dentro de los filtros cambian respecto a los que veníamos usando en el código. Pasan de ser «and», «or», «not» por **&**, **|**, **~**, respectivamente.

Los filtros también nos permiten efectuar cálculos para sumatorias o contadores. Por ejemplo, si necesitamos hallar y mostrar la cantidad de ceros y unos de un arreglo con números binarios:

```
import numpy as np
vecta=np.random.randint(0,2,15)
print(vecta)
ceros=len(vecta[vecta==0])
unos=len(vecta[vecta==1])
print("La cantidad de ceros es",ceros,"y unos es",unos)
```

Nota
La función **len()** devuelve la cantidad dependiendo de la condición que contiene el filtro asignándolo a una variable entera: ceros o unos, según el caso. Otra alternativa para obtener este resultado es con la función **sum()**, ejemplo: **unos=sum([vecta==1])**.

Otra opción es el siguiente caso donde se efectúa la sumatoria de números mayores a 7 de un vector:

```
import numpy as np
vecta=np.random.randint(0,10,12)
print(vecta)
sumatoria=sum(vecta[vecta>7])
print("La sumatoria de nº mayores a 7 es",sumatoria)
```

Nota
En este caso, la función que ejerce la sumatoria se llama **sum()** y se asigna a la variable entera **sumatoria**.

En realidad todas las funciones del NumPy pueden ser utilizadas en los filtros. A continuación, listamos algunas que pueden ser de gran utilidad y que no fueron mencionadas hasta el momento:

Sintaxis	Funcionalidad	Tipo
np.cbrt(a)	Raíz cúbica	Aritmético
np.multiply(a,b)	Multiplica elemento por elemento	Aritmético
np.divide(a,b)	Divide elemento por elemento	Aritmético
np.deg2rad(x)	Convierte grados a radianes	Trigonometría
np.rad2deg(x)	Convierte radianes a grados	Trigonometría
np.arcsin(x)	Inversa de seno	Trigonometría
np.arccos(x)	Inversa de coseno	Trigonometría
np.exp(x)	Exponencial	Exponenciales y logarítmicas
np.log(x)	Logaritmo natural	Exponenciales y logarítmicas
np.log10(x)	Logaritmo en base 10	Exponenciales y logarítmicas
np.mean(a)	Promedio de los elementos	Estadística
np.median(a)	Mediana de los elementos	Estadística
np.std(a)	Desviación estándar de los elementos	Estadística
np.var(a)	Varianza de los elementos	Estadística
np.equal(a,b)	Compara si los elementos son iguales	Comparación
np.not_equal(a,b)	Compara si son elementos distintos	Comparación

5. 2. 7. Operaciones aritméticas con arreglos

Una ventaja importante que presenta Python son las operaciones aritméticas que podemos efectuar en los arreglos, por ejemplo, el siguiente algoritmo realiza la suma de dos vectores.

```
import numpy as np
vec1=np.array([6,2,10,8,1])
vec2=np.array([2,5,7,4,2])
vec3=vec1+vec2  # suma de vectores
print(vec3) # Mostrar vector 3
```

El resultado es:

```
[ 8  7 17 12  3]
```

Nota
Los vectores deben tener las mismas dimensiones, de lo contrario salta un error. Además de la suma se pueden efectuar las otras operaciones resta (**-**), producto (*) y división (**/**).

En el siguiente ejemplo le asignamos a **vec3** la división de dos vectores.

```
import numpy as np
vec1=np.array([6,2,10,8,1])
vec2=np.array([2,5,7,2,6])
vec3=vec1/vec2  # division de vectores
print(np.round(vec3,2)) # Mostrar vector 3
```

La respuesta es:

```
[3.   0.4  1.43 4.   0.17]
```

Nota

Además aplicamos el redondeo con **round()** usando dos decimales para cada elemento del **vec3** (No modifica los valores solo los muestra con dos decimales).

Para modificar sus valores redondeando, tenemos que volver a asignar al **vec3** su mismo vector aplicando la función **round()** antes de imprimir **vec3=np.round(vec3,2).**

Son posibles otras operaciones aritméticas como sumar un número al vector de la siguiente forma:

```
import numpy as np
vec1=np.array([1,2,3,4,5])
print(vec1) # Mostrar vector 1
numero=int(input("Ingrese un número entero "))
vec2=numero+vec1
print(vec2) # Mostrar vector 2
```

La respuesta es:

```
[1 2 3 4 5]
Ingrese un número entero 5
[ 6  7  8  9 10]
```

Nota

A cada elemento del vector **vec1** se le suma un número entero ingresado por el usuario. En este caso, se hizo la prueba con un 5.

También funciona con la resta, el producto y la división. Probar con estas operaciones.

Ejemplo 43: Crear dos vectores **v1** y **v2** de dimensión 8 con números entre 1 y 10 flotantes obtenidos aleatoriamente y luego efectuar la siguiente operación:

$$v3 = \sqrt{3} \cdot (v1 - v2) + v1^2$$

```
import numpy as np
import math
v1=np.random.uniform(1,10,size=8)
v1=np.round(v1,1)
v2=np.random.uniform(1,10,size=8)
v2=np.round(v2,1)
print(v1) # Mostrar vector 1
print(v2) # Mostrar vector 2
v3=math.sqrt(3)*(v1-v2)+pow(v1,2)
print(np.round(v3,1))
```

El resultado es:

```
[6.4 2.2 9.2 5.8 3.5 6.3 6.2 1.4]
[9.6 2.3 2.9 1.6 3.4 1.3 2.1 5.5]
[35.4  4.7 95.6 40.9 12.4 48.4 45.5 -5.1]
```

Nota

Incorporamos la librería Math para poder efectuar operaciones de potencia y raíz. Se demuestra que cualquier operación aritmética es posible, incluso usando vectores.

5. 3. Problemas: arreglos y vectores

57. Crear un arreglo unidimensional (vector) de números enteros aleatorios donde se indique la dimensión por teclado y el rango de valores que puede generar, de nombre **h1,** y crear otro vector **h2** que tenga los elementos del **h1** pero ordenado de modo ascendente. Mostrar ambos.
58. Crear un vector **v1** de números enteros aleatorios (entre el 1 y el 30) donde se indique la dimensión por teclado. Luego crear otro vector **v2** pero que contenga solo los múltiplos de 3. Por último, mostrar ambos vectores pero el **v1** ordenado de modo descendente.
59. Juego de adivinanza: Crear un vector de dimensión 10 con los números binarios (aleatorios), luego ingresar un número por teclado para adivinar la cantidad de unos que tiene el vector.
    a. Si resulta ser la misma cantidad, mostrar el siguiente cartel **"Magnífico"**.
    b. Si solo erra por uno, mostrar **"Muy Bueno"**.
    c. El resto será el cartel **"Lo lamento"**.

    Claro está que se debe mostrar el vector inmediatamente después del resultado.

60. Crear el mismo juego que el anterior, pero que se pueda efectuar en el mismo algoritmo tres veces y que ahora se obtenga un puntaje al finalizar las partidas.
    a. Si resulta ser la misma cantidad, obtiene 5 puntos.
    b. Si solo se erra por uno, consigue 2 puntos.
    c. El resto no suma puntos.

    Al finalizar las tres partidas el algoritmo debe mostrar el vector e informar el puntaje logrado.
61. Crear un vector de 20 números aleatorios de la ruleta [0..., 36], luego mostrar el vector y el siguiente informe:
    a. Cantidad de pares e impares obtenidos.
    b. Cantidad de números de la primera, segunda y tercera docena.
    c. El promedio.

    Tener en cuenta que para **a** y **b** se debe excluir el 0 en la contabilidad.
62. Crear un vector de 25 números que contenga aleatoriamente tres números (1, 2, 3). Mostrar el vector. Luego hallar y mostrar el % de cada número.
63. Crear los siguientes vectores con 10 letras: **f1** (a,a,b,a,b,a,b,a,a,b) y **f2** (a,b,b,a,a,b,b,b,a,b).

    Crear otro de nombre **f3** que contenga 10 ceros y luego modificar en cada posición de acuerdo al siguiente criterio:
    a. Si tienen la misma letra se coloca un 1.
    b. Si son distintos un -1.

    Por último, mostrar **f3** y hallar el % de negativos y positivos del vector resultante.
64. Vectores paralelos: Crear tres vectores (**x**, **senx** y **cosx**) con números flotantes.

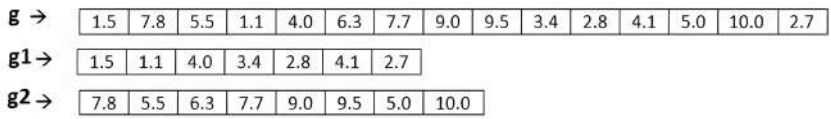
    Se trata de almacenar los valores trigonométricos en dos vectores tomando como base el vector **x**. Entonces se necesita:
    a. Crear el vector **x** de valores secuenciales desde el -5 hasta el 5, con un salto de 0,5 (usar una función de NumPy).
    b. El vector **senx** debe contener los valores de la función seno tomando los valores de la variable independiente del vector **x**.
    c. El vector **cosx** debe contener los valores de la función coseno tomando los valores de la variable independiente del vector **x**.
    d. Mostrar los tres vectores y los valores máximos y mínimos de los vectores **senx** y **cosx**.
65. Finalizan 20 atletas una maratón registrando los siguientes datos: tiempo establecido (en minutos), el número del competidor y su edad.

Crear tres vectores **tmara**, **nmara** y **emara** para cargar en cada uno de modo aleatorio los datos mencionados. Luego mostrar el podio del 1º, 2º y 3º con sus datos, por ejemplo:

a. El primero es el nº 18 con una marca de 130 minutos y su edad es 30 años.
b. El segundo es el nº 40 con una marca de 133 minutos y su edad es 28 años.
c. El tercero es el nº 13 con una marca de 137 minutos y su edad es 31 años.

66. Usar dos vectores que contengan los valores de los ejes cartesianos **x** e **y**. Tener en cuenta la función cuadrática $\mathbf{f(x)=x^2-3x+2,5}$ con intervalos de 0,5. El dominio de la función va de -5 a 5, tanto **x** como **y** deben estar almacenadas en los vectores **vx** y **vy**, respectivamente (usar dos decimales).
67. Crear un vector **g** de dimensión 15 con números aleatorios flotantes entre 1 y 10 con un decimal. Mostrarlo y luego generar dos vectores **gmenor** y **gmayor**, en el primero debe contener los números menores a 5 y el segundo los mayores o iguales a 5 obtenidos del vector **g**.

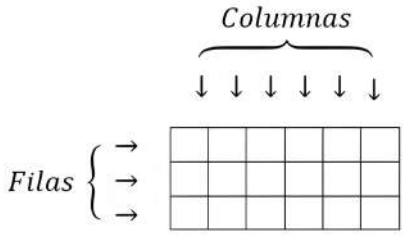
    Por ejemplo:



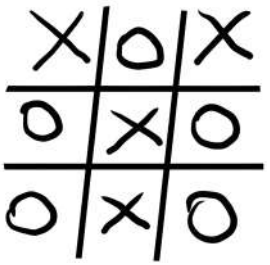
    Es decir que la suma de las dos dimensiones **g1** y **g2** debe ser siempre 15.

5. 4. Matrices

Muchas veces los vectores son mencionados como matrices. Las *matrices* son una estructura de datos bidimensional donde los elementos se organizan en filas y columnas.



A continuación planteamos el siguiente ejercicio en el que se supone que tenemos una partida de tatetí, representando una matriz de 3 filas x 3 columnas, que contiene elementos de tipo carácter. Para recorrer la matriz necesitamos 2 índices: uno para las filas y otro para las columnas (en ese orden).



En Python, la carga de la matriz tatetí es:

```
import numpy as np
tateti=np.array([[1,0,1],[1,1,0],[0,1,0]])
print(tateti)
```

El resultado es:

```
[[1 0 1]
 [1 1 0]
 [0 1 0]]
```

Nota
Tener en consideración que los elementos son numéricos (el 1 para la cruz y el 0 para el círculo). En todo caso, se puede generar el tatetí pero con los caracteres **X** y **O**.

Al igual que en vectores podemos crear matrices usando solamente el cero.

```
import numpy as np
mat=np.zeros((3,5))
print(mat)
```

El resultado es:

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Nota
En este caso la matriz **mat** contiene 3 filas y 5 columnas. Para llenar usando el 1 se lo hace con **mat=np.ones((3,5))**.

En el caso de querer repetir el mismo carácter o el mismo número en la matriz usamos la función **full()** de NumPy.

```
import numpy as np
mat1=np.full((3,5),'h')
mat2=np.full((3,5),'12')
print(mat1)
print('\n')
print(mat2)
```

El resultado es:

```
[['h' 'h' 'h' 'h' 'h']
 ['h' 'h' 'h' 'h' 'h']
 ['h' 'h' 'h' 'h' 'h']]

[['12' '12' '12' '12' '12']
 ['12' '12' '12' '12' '12']
 ['12' '12' '12' '12' '12']]
```

Nota
El **print('\n')** sirve para generar una línea en blanco entre las dos matrices (solo a efectos de mejorar la visual).

Para generar matrices aleatoriamente podemos usar la función **rand()** de NumPy.

```
import numpy as np
mat2x3=np.random.rand(2,3)
print(mat2x3)
```

El resultado es:

```
[[0.77265706 0.28996307 0.11986147]
 [0.13865621 0.70618365 0.14479633]]
```

Nota
En este caso creamos una matriz **mat** de dimensión 2x3 con los números al azar por defecto (valores que oscilan entre el 0 y el 1).

Pero si queremos que los elementos aleatorios de la matriz sean valores enteros tenemos que utilizar la función **randint()** y además el rango de valores (inicio, final). El siguiente ejemplo muestra la generación de enteros que oscilan entre 1 y 10.

```
import numpy as np
mat2x3 =np.random.randint(1,10,size=(2,3))
print(mat2x3)
```

El resultado es:

```
[[4 9 4]
 [9 1 7]]
```

Si queremos hacerlo con números flotantes entre 1 y 10 usamos la función **uniform()**.

```
import numpy as np
mat2x3=np.round(np.random.uniform(1,10,size=(2,3)),2)
print(mat2x3)
```

El resultado es:

```
[[9.11 6.01 6.43]
 [3.28 6.27 7.92]]
```

Nota

En este caso agregamos la función **round()** para dejar cada número de la matriz con dos decimales. Recordar que si hacemos esto solo a efectos visuales, y no en la asignación, cada elemento flotante tendrá 8 decimales.

Al igual que en los vectores también podemos concatenar con matrices. El siguiente ejemplo muestra cómo armar una matriz a partir de vectores **vec1** y **vec2**.

```
import numpy as np
vec1 = np.array([1, 2, 3, 4])
vec2 = np.array([5, 6, 7, 8])
matriz=np.concatenate(([vec1], [vec2]))
print(matriz)
```

El resultado es:

```
[[1 2 3 4]
 [5 6 7 8]]
```

También se puede concatenar entre matrices para formar una nueva matriz.

```
import numpy as np
mat1=np.array([[1,4,6],[4,7,1]])
mat2=np.array([[2,2,9],[3,3,5]])
matriz=np.concatenate((mat1,mat2))
print(matriz)
```

El resultado es:

```
[[1 4 6]
 [4 7 1]
 [2 2 9]
 [3 3 5]]
```

Se puede hacer entre vectores y matrices:

```
import numpy as np
vec1 = np.array([1, 2, 3, 4])
mat1 = np.array([[5, 6, 7, 8],[9,10,11,12]])
matriz=np.concatenate(([vec1], mat1))
print(matriz)
```

El resultado es:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

5. 4. 1. Acceso al contenido específico de la matriz

Al igual que en los vectores se puede acceder a un elemento ubicado según sus coordenadas, por ejemplo **A[1][3]**, para nosotros será (2,4).

```
import numpy as np
A=np.array([[3,6,7,1],[5,1,0,9],[3,3,4,6]])
print(A)
print("El elemento de la matriz en (2,4) es el",A[1][3])
```

El resultado es:

```
[[3 6 7 1]
 [5 1 0 9]
 [3 3 4 6]]
El elemento de la matriz en (2,4) es el 9
```

Ejemplo 44: Crear y mostrar la siguiente matriz de nombre **A5x5** que contenga números enteros aleatorios cuyo rango de valores oscilan entre el 1 y el 10. Luego que efectúe la sumatoria total y lo muestre.

```
import numpy as np
A5x5=np.random.randint(1,10, size=(5,5))
print(A5x5)
suma=0
for i in range(5):
    for j in range(5):
        suma=suma+A5x5[i][j]
print("La sumatoria total es",suma)
```

El resultado es:

```
[[1 1 1 9 4]
 [3 5 6 8 3]
 [5 5 7 4 7]
 [1 7 6 3 6]
 [9 2 4 8 1]]
La sumatoria total es 116
```

Nota

Con un doble **for** cuyos índices **i** para recorrer las filas y **j** para recorrer las columnas se puede acceder a cada uno de los elementos por medio de **A5x5[i][j]** para ir sumándolos en la variable suma. Pero NumPy recurre a una función más directa **sum()**. Por lo tanto, podemos ahorrar líneas de comandos y variables simplemente usando: **print(«La sumatoria total es»,np.sum(A5x5))**.

Ejemplo 45: Obtener la cantidad de valores pares de una matriz de 4x4 que contenga números enteros aleatorios cuyo rango de valores oscila entre el 1 y el 10.

```
import numpy as np
A4x4=np.random.randint(1,10, size=(4,4))
print(A4x4)
soloPares=A4x4 % 2==0
print("La cantidad de nº pares es",np.sum(soloPares))
```

El resultado es:

```
[[7 9 1 2]
 [5 5 9 7]
 [4 6 4 3]
 [2 9 1 1]]
La cantidad de nº pares es 5
```

Nota

Evitamos usar el recorrido con doble **for** simplemente filtrando la matriz con la creación de un nuevo vector **soloPares** y luego aplicamos la función **sum().**

Supongamos que nos piden mostrar la lista de números pares de la matriz **A4x4**.

```
import numpy as np
A4x4=np.random.randint(1,10, size=(4,4))
print(A4x4)
soloPares=A4x4 % 2==0
print("Los pares son:",A4x4[soloPares])
```

El resultado es:

```
[[3 2 7 2]
 [9 4 1 8]
 [1 8 9 6]
 [7 3 9 8]]
Los pares son: [2 2 4 8 8 6 8]
```

Nota

El **print** permite mostrar el filtro ejecutando **matriz[filtro]**, en nuestro caso **A4x4[soloPares]** y si nos piden obtener la sumatoria de los elementos del filtro podemos agregar la función **sum()** de la siguiente manera: **np.sum(A4x4[soloPares])**.

Ejemplo 46: Crear y mostrar la siguiente matriz de nombre **A4x4** que contenga números enteros aleatorios cuyo rango de valores oscilan entre el 1 y el 10. Luego que efectúe el promedio de los números impares y lo muestre.

```
import numpy as np
A4x4=np.random.randint(1,10, size=(4,4))
print(A4x4)
soloImpares=A4x4 % 2==1
print("El promedio de los nº impares es:",np.mean(A4x4[soloImpares]))
```

El resultado es:

```
[[3 2 9 9]
 [9 1 1 6]
 [8 7 2 9]
 [7 7 3 1]]
El promedio de los nº impares es: 5.5
```

Nota

Con la función **mean()** de NumPy y los valores impares filtrados se consigue el promedio.

Para obtener una zona específica de una matriz, como una determinada fila entera o columna, mostramos el siguiente ejemplo:

```
print(mat4x5)
print("\n")
print(mat4x5[2]) # mostrar la fila 2
print("\n")
print(mat4x5[:,3]) # mostrar la columna 3
```

La respuesta es:

```
[[ 1 10  9 21  9]
 [ 2  0 12 19 13]
 [17 29 15 18 27]
 [ 1 12  7 14 17]]

[17 29 15 18 27]

[21 19 18 14]
```

Nota
Con **mat4x5[2]** simplemente se obtiene la fila 2 y con **mat4x5[:,3]**, la columna 3. Recordar que los arreglos comienzan la fila y la columna con la numeración 0.

Desafío
Repetir el mismo ejemplo pero creando dos vectores de nombres **vecFil2** y **vecCol3**. Cada vector debe guardar los elementos de la fila 2 y los elementos de la columna 3 (respectivamente) y mostrar los dos vectores. Por último, mostrar la sumatoria de la fila 2 y el promedio de la columna 3.

5. 4. 2. Máximos y mínimos con matrices

En el caso de los valores máximos y mínimos dentro de la matriz funcionan de la misma manera que en los vectores. Vemos un ejemplo:

```
import numpy as np
mat4x3=np.round(np.random.uniform(1,10,size=(4,3)),2)
print(mat4x3)
print("El mayor valor de la matriz es",np.max(mat4x3))
print("esta ubicado en",np.argmax(mat4x3)+1)
```

El resultado es:

```
[[8.8  1.02 5.63]
 [6.72 3.71 9.34]
 [5.59 9.05 7.94]
 [8.39 9.01 7.28]]
El mayor valor de la matriz es 9.34
esta ubicado en 6
```

Nota

En este caso nos indica la ubicación solamente de un modo secuencial. En el ejemplo anterior el mayor valor se encuentra en la segunda fila y tercera columna, es decir que el lugar es el sexto comenzando el recorrido desde el primer elemento que se encuentra en (1,1).

Desafío

Hacer el mismo ejemplo pero para hallar el valor mínimo y su ubicación.

También podemos obtener los valores máximos de cada fila con la función **amax()** y su argumento **axis**.

```
import numpy as np
mat4x5=np.random.randint(0,20,size=(4,5))
print(mat4x5)
print("\n")
print("Los valores máximos de c/fila son",np.amax(mat4x5,axis=1))
```

El resultado es:

```
[[18  7  8 11  4]
 [13 12  0  8 14]
 [12  3 11  0 13]
 [ 0  5 16 13 14]]

Los valores máximos de c/fila son [18 14 13 16]
```

Nota

Si agregamos una línea más de código a este ejemplo, **print(np.argmax(mat4x5,axis=1)+1))** nos muestra además la posición de cada uno (el **+1** es para correr la ubicación por comenzar con 0 cada fila).

Desafío

Realizar el mismo ejemplo pero para hallar los valores mínimos de cada fila y sus posiciones. El argumento **axis** significa eje en inglés y para una matriz de dos dimensiones **axis=1** se refiere a las filas y **axis=0** a las columnas. En el caso de querer obtener los valores máximos de cada columna para el ejemplo anterior lo hacemos de la siguiente manera:

```
import numpy as np
mat4x5=np.random.randint(0,20,size=(4,5))
print(mat4x5)
print("\n")
print("Los valores máximos de c/columna son",np.amax(mat4x5,axis=0))
```

El resultado es:

```
[[17 16 18 17  3]
 [ 6  0  1  0  1]
 [10 15  3  3  8]
 [13  5 18 19 10]]

Los valores máximos de c/columna son [17 16 18 19 10]
```

Ejemplo 47: Obtener el promedio de cada fila de la matriz **A4x5** con números enteros aleatorios.

```
import numpy as np
A4x5=np.random.randint(1,10, size=(4,5))
print(A4x5)
print("El promedio de c/fila es:",np.mean(A4x5,axis=1))
```

El resultado es:

```
[[3 7 8 1 5]
 [9 4 8 8 3]
 [2 2 4 6 9]
 [5 6 6 4 5]]
El promedio de c/fila es: [4.8 6.4 4.6 5.2]
```

Desafío

Mostrar el promedio de cada columna del ejercicio anterior.

5. 4. 3. Operaciones aritméticas con matrices

Como con los vectores, las matrices también pueden operar aritméticamente, por ejemplo si queremos efectuar la suma de dos matrices **A+B**.

```
import numpy as np
A=np.array([[3,6,7,1],[5,1,0,9],[3,3,4,6]])
print(A)
print("\n")
B=np.array([[2,9,7,5],[3,6,1,1],[0,8,6,3]])
print(B)
print("\nLa matriz C resultado de la suma es")
C=A+B
print(C)
```

El resultado es:

```
[[3 6 7 1]
 [5 1 0 9]
 [3 3 4 6]]

[[2 9 7 5]
 [3 6 1 1]
 [0 8 6 3]]

La matriz C resultado de la suma es
[[ 5 15 14  6]
 [ 8  7  1 10]
 [ 3 11 10  9]]
```

> **Nota**
> Para la suma o la resta de matrices, ambas deben tener la misma dimensión. Probar el mismo ejemplo pero para restar **A-B**.

Para el producto de un escalar 3 con la matriz **A** resulta el siguiente ejemplo:

```
import numpy as np
A=np.array([[3,6,7,1],[5,1,0,9],[3,3,4,6]])
print(A)
print("\nLa matriz C resultado del producto de A por 3")
C=3*A
print(C)
```

El resultado es:

```
[[3 6 7 1]
 [5 1 0 9]
 [3 3 4 6]]

La matriz C resultado del producto de A por 3
[[ 9 18 21  3]
 [15  3  0 27]
 [ 9  9 12 18]]
```

Probar la siguiente combinación de operaciones con las matrices **A** y **B** utilizadas en el ejemplo anterior:

$$C = \left(\frac{A}{5} + 11 - B \right) \cdot 8$$

Para comparar dos arreglos, ya sean unidimensionales o bidimensionales que tengan la misma dimensión y el mismo tipo de elementos, se utiliza la función **all()**.

```
import numpy as np
mat1=np.random.randint(0,2,(5,5))
mat2=np.random.randint(0,2,(5,5))
if(mat1==mat2).all():
    print("Son iguales")
else:
    print("No son iguales")
```

Nota
En el ejemplo comparamos dos matrices **mat1** y **mat2** de 5x5 y de números binarios para informar si son iguales o no.

5.4.4. Diagonales en matrices

Para obtener la diagonal principal de una matriz regular se utiliza la función **diag()**.

```
import numpy as np
A4x4=np.array([[1,3,7,9],[6,9,0,4],[0,3,8,7],[1,2,2,6]])
print(A4x4)
print("\nLa diagonal principal de la matriz es")
print(np.diag(A4x4))
```

El resultado es:

```
[[1 3 7 9]
 [6 9 0 4]
 [0 3 8 7]
 [1 2 2 6]]

La diagonal principal de la matriz es
[1 9 8 6]
```

Por el contrario, si queremos armar una matriz regular a partir de un vector hacemos lo siguiente:

```
import numpy as np
MD=np.diag([3,5,1,8])
print(MD)
```

El resultado es:

```
[[3 0 0 0]
 [0 5 0 0]
 [0 0 1 0]
 [0 0 0 8]]
```

Nota
De esta forma se completa con ceros en el resto de la matriz.

Para crear la matriz identidad (diagonal de unos y el resto de ceros) se necesita de la función **eye()**. Por ejemplo para crear la matriz identidad de 5x5:

```
import numpy as np
Miden=np.eye(5)
print(Miden)
```

El resultado es:

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

5. 4. 5. La traspuesta de una matriz

Para hallar la traspuesta de una matriz se realiza simplemente al agregar **.T** (punto T) al nombre de la matriz.

```
import numpy as np
mat=np.array([[1,2,3],[4,5,6]])
print("La matriz original es")
print(mat)
print("\nLa matriz traspuesta es")
print(mat.T)
```

La respuesta es:

```
La matriz original es
[[1 2 3]
 [4 5 6]]

La matriz traspuesta es
[[1 4]
 [2 5]
 [3 6]]
```

Nota
Las columnas pasan a ser filas y las filas pasan a ser columnas. Esto se puede aplicar para cualquier dimensión bidimensional.

5.5. Asignaciones condicionales

Supongamos que queremos crear una nueva matriz de nombre **destino** a partir de una llamada **origen** asignando datos que tengan que ver con una condición. Por ejemplo, tenemos la matriz **origen** con números enteros del 1 al 10 y le queremos asignar a la matriz **destino** la letra **A** a los que son menores que 6, y al resto la letra **B**.

```
import numpy as np
A4x5=np.random.randint(1,10, size=(4,5))
print(A4x5)
condiciones = [A4x5 < 6]
resultado = np.select(condiciones, ['A'], ['B'])
print("\n La matriz resultante es")
print(resultado)
```

El resultado es:

```
[[4 1 3 8 4]
 [7 7 9 9 6]
 [8 2 4 1 5]
 [6 3 7 9 1]]

 La matriz resultante es
[['A' 'A' 'A' 'B' 'A']
 ['B' 'B' 'B' 'B' 'B']
 ['B' 'A' 'A' 'A' 'A']
 ['B' 'A' 'B' 'B' 'A']]
```

Nota
La función **select()** permite hacer este tipo de situaciones, es decir, genera un nuevo arreglo a partir de otro usando condiciones binarias. La nueva matriz puede contener no solo letras sino otro tipo de elementos (siempre y cuando sea homogéneo).

Desafío
Generar la matriz resultante pero con ceros y unos en lugar de **A** y **B**.

5. 6. Atributos de un arreglo

Existen varios atributos y funciones que describen las características de un arreglo que pueden usarse para obtener la forma actual de una matriz o vector, pero también se pueden usar para remodelar (reconfigurar):

arreglo.ndim: devuelve el número de dimensiones del arreglo.
arreglo.shape: devuelve una tupla con las dimensiones del arreglo.
arreglo.size: devuelve el número de elementos del arreglo.
arreglo.dtype: devuelve el tipo de datos de los elementos del arreglo.

Vemos un ejemplo:

```
import numpy as np
arreglo = np.random.uniform(-3,3,size=(3,4))
print(np.round(arreglo,2))
print("Número de dimensiones",arreglo.ndim)
print("Número de tupla",arreglo.shape)
print("Número de elementos",arreglo.size)
print("El tipo de elemntos",arreglo.dtype)
```

El resultado es:

```
[[-2.9   0.84 -1.57 -0.14]
 [-1.04 -0.52  0.59  2.34]
 [-0.56  0.57 -0.89 -1.24]]
Número de dimensiones 2
Número de tupla (3, 4)
Número de elementos 12
El tipo de elemntos float64
```

5. 7. Gráficas: Matplotlib

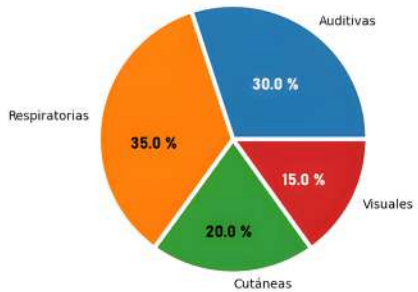
Para las gráficas tenemos que instalar la librería Matplotlib. Los pasos son los siguientes para el sistema operativo Windows:

1. Tendrás que dirigirte a **"Símbolo de sistema"**.
2. Escribir en el *prompt* lo siguiente: **pip install matplotlib**.
3. Luego de dar *enter* en el teclado esperamos la carga de la librería y listo.
4. Para probar que efectivamente funciona, probemos agregar en el Python:
   import matplotlib.pyplot as plt (no debería saltar error).

Ejemplo 48: La contaminación atmosférica genera un gran riesgo para la salud de los ecosistemas. Se estima que causa alrededor de dos millones de muertes prematuras al año en todo el mundo. Los

factores de la consecuencia de la contaminación pueden ser por distintas causas: 30 % auditivas, 35 % respiratorias, 20 % cutáneas y 15 % visuales. Representamos estos datos de modo gráfico usando el diseño pie (torta).

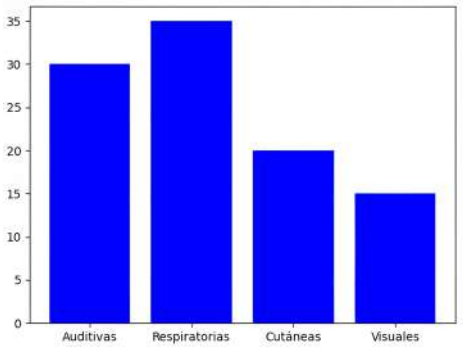
```
import numpy as np
import matplotlib.pyplot as plt
porcentajes=[30,35,20,15]
etiquetas=["Auditivas","Respiratorias","Cutáneas","Visuales"]
plt.pie(porcentajes,labels=etiquetas, autopct="%0.1f %%")
plt.show()
```



Nota
La función **pie()** de Matplotlib tiene tres parámetros: los datos numéricos, las etiquetas leyendas y las etiquetas numéricas.

Ejemplo 49: Crear un gráfico de barras vertical con los datos de la contaminación atmosférica del ejemplo anterior.

```
import matplotlib.pyplot as plt
etiquetas=["Auditivas","Respiratorias","Cutáneas","Visuales"]
porcentajes=[30,35,20,15]
plt.bar(x=etiquetas,height=porcentajes)
plt.show()
```

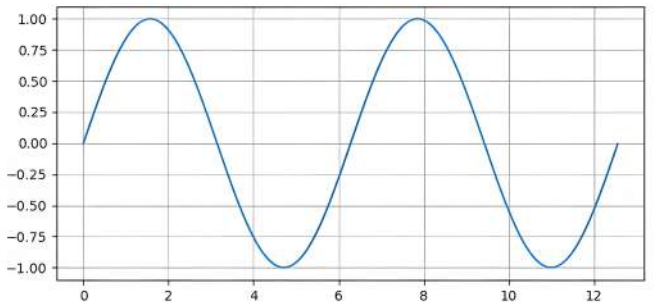


Nota

Si para este ejemplo queremos crear una gráfica de barras pero horizontal entonces debemos utilizar **plt.barh** y los parámetros son **y=etiquetas**, **width=porcentajes**. Realizar la prueba.

Ejemplo 50: Representar gráficamente la función trigonométrica seno. Los valores de X tendrán el siguiente dominio [0 a 4π] y con un salto de 0.01.

```
import numpy as np
import math
import matplotlib.pyplot as plt
X=np.arange(0,4*math.pi,0.01)
Y=np.sin(X)
plt.grid(axis='both')
plt.plot(X,Y)
plt.show()
```



Nota

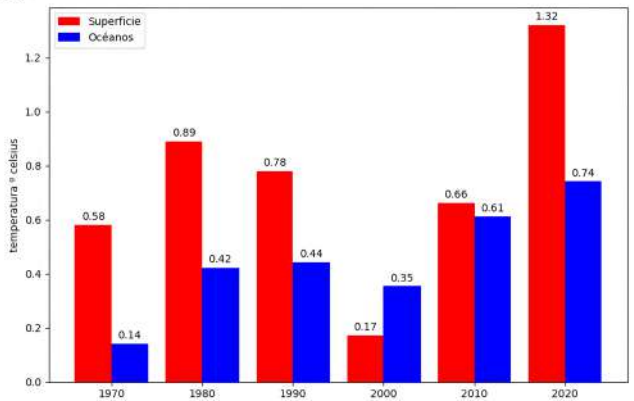
La función **plot()** sirve para hacer este tipo de gráficas (curvas). Además se tuvo que importar la librería Math que permite usar PI para calcular el dominio hasta 2 vueltas. La función **grid()** permite en este caso dibujar la grilla.

Ejemplo 51: Gran parte del calentamiento global es producido por las emisiones de gases de efecto invernadero que no solo afecta a la superficie terrestre, sino también a los océanos. Los siguientes datos obtenidos del organismo internacional National Center for Enviromental Information representan los cambios de temperaturas en grados Celsius registrados en el hemisferio sur en las distintas décadas desde 1970 al 2020.

Año	Superficie	Océanos
1970	0,58	0,14
1980	0,89	0,42
1990	0,78	0,44
2000	0,17	0,35
2010	0,66	0,61
2020	1,32	0,74

Por lo tanto, se debe crear un gráfico de barras agrupando la información de la variación de temperatura (superficie y océanos) en cada decenio.

```
import matplotlib.pyplot as plt
import numpy as np
plt.tittle=('Aumentos de temperatura en la superficie y océanos')
labels = ['1970', '1980', '1990', '2000', '2010','2020']
superficie = [0.58, 0.89, 0.78, 0.17, 0.66, 1.32]
oceanos = [0.14, 0.42, 0.44, 0.35, 0.61,0.74]
x=np.arange(len(labels)) # define en x las etiquetas de los años
width=0.40 # ancho de barrras
fig, ax = plt.subplots()
# Definimos las barras por ubicación, datos, tamaño, etiqueta y color
bar1 = ax.bar(x-width/2,superficie,width,label='Superficie',color="tab:red")
bar2 = ax.bar(x+width/2,oceanos,width,label='Océanos',color="tab:blue")
ax.set_ylabel('temperatura º celsius') # etiqueta para el eje y
ax.legend() # leyenda
ax.set_xticks(x, labels) # agregar los años en el eje x
ax.bar_label(bar1, padding=3) # agregar los valores sobre las barras
ax.bar_label(bar2, padding=3)
plt.show()
```



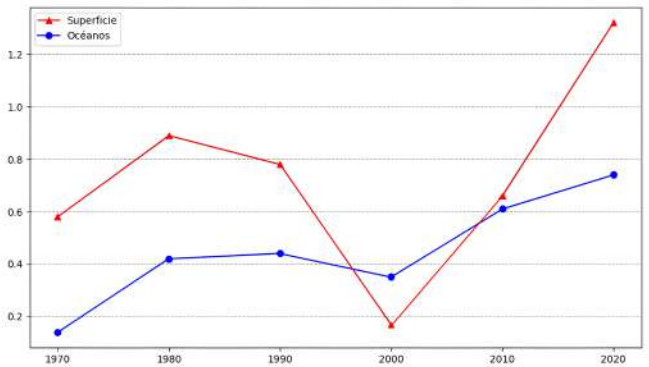
Nota

Recordar que los nombres de las variables no llevan tildes, como en el caso de **"oceanos"**, pero sí en las etiquetas (leyenda). Los colores de las barras vienen por defecto, pero en este caso decidimos modificar dentro de la función **ax.bar** (en el quinto parámetro).

La librería Matplotlib tiene definido una serie de colores, pero si no lo especificamos son establecidos por defecto. Los colores básicos se pueden mencionan con una letra (**b** azul, **g** verde, **r** rojo, **c** cian, **m** magenta, **y** amarillo, **k** negro, **w** blanco). También tiene una paleta básica (que es la que usamos en el ejemplo anterior) que contiene la palabra **tab:** seguido de la palabra completa **tab:blue, tab:orange, tab:green, tab:red, tab:purple, tab:brown, tab:pink, tab:gray, tab:olive, tab:cyan**. A través de la lista CSS podemos obtener 148 colores o por medio del modelo aditivo RGB (Red, Green, Blue) que determina 948 colores.

Ejemplo 52: Efectuar la gráfica de líneas (comparativas) utilizando los datos del cambio climático del ejercicio anterior.

```
import matplotlib.pyplot as plt
import numpy as np
plt.tittle=('Aumentos de temperatura en la superficie y océanos')
fig, ax = plt.subplots()
labels = ['1970','1980','1990','2000','2010','2020']
superficie = [0.58, 0.89, 0.78, 0.17, 0.66, 1.32]
oceanos = [0.14, 0.42, 0.44, 0.35, 0.61, 0.74]
line1=ax.plot(labels,superficie,color="tab:red",label='Superficie',marker='^')
line2=ax.plot(labels,oceanos,color="tab:blue",label='Océanos',marker='o')
x=np.arange(len(labels)) # define en x las etiquetas de los años
ax.legend() # leyenda
ax.grid(axis = 'y', color = 'gray', linestyle = '--') # líneas punteadas
plt.show()
```



Nota
Con la función **grid** (grilla) se puede agregar las líneas, en este caso punteadas **'--'**. Y dentro de la función **plot** se agregaron las marcas (triángulos para la curva de superficie y círculo para los oceanos) con el parámetro **marker**.

Desafío
Realizar intentos en el que se efectúen cambios en los colores, en el tipo de líneas y grilla en estos cinco ejemplos gráficos.

5. 8. Problemas: arreglos y matrices

68. Crea una matriz de 3x3 con valores enteros, generados en forma aleatoria en el rango de 1 a 10.
    a. Mostrar la matriz.
    b. Ingresar un valor entero entre **x** para determinar cuántas veces aparece en la matriz. ¿Cuál es la estructura de control adecuada para realizar el recorrido?
    c. Mostrar la matriz traspuesta.
69. Crear cuatro vectores con 10 números flotantes aleatorios (de 2 decimales) y convertir los arreglos en una matriz de nombre **Mat**.
70. Crear las dos matrices **w1** y **w2** conteniendo: [1,4,5,2],[3,5,3,1],[7,4,2,0] y [6,3,5,1],[9,4,3,7],[1,1,0,5], mostrarlas. Luego se pide hallar y mostrar:
    La operación :
    $$\frac{6w1 - w2}{5}$$
    La sumatoria de las dos diagonales **w1** y **w2**.
    Cantidad de números múltiplos de 3 en **w2**.
71. Crear una matriz de 3x3 para un tatetí. Por teclado debe ingresar **(x**,**o**) para cada posición, mostrar el tablero y luego:
    a. Detectar e informar si la **x** completó el tatetí en el tablero.
    b. Detectar e informar si la **0** completó el tatetí en el tablero.
72. Crear una matriz regular de NxN con los números enteros aleatorios (0, 1, 2) y mostrarlo. La dimensión **N** se ingresa por teclado. Luego hallar:
    a. Porcentaje de ceros.
    b. Porcentaje de unos.
    c. Porcentaje de dos.
73. Crear otra matriz con la misma dimensión. Los números se representan por lenguaje coloquial [0 (cero), 1 (uno) y 2 (dos)].

74. Crear una matriz de NxM (dimensiones ingresadas por teclado) para cargar números flotantes aleatorios en el rango -20 a 20 con dos decimales y mostrarla. Se pide además:
    a. Informar cuántos números negativos existen en la matriz.
    b. Mostrar el valor máximo y mínimo.
    c. Detectar el primer valor 0 (si existe) informando la ubicación (fila, columna).
    d. Informar el promedio (solo de los números positivos).
75. Generar los números aleatorios de la ruleta a la matriz **Rulet** de 10x10, recordando que oscilan en el 0 y el 36. Mostrar la matriz y efectuar lo siguiente:
    a. Hallar los porcentajes de la 1º, 2º y 3º docena por separado.
    b. Se ingresa un número por teclado para saber cuántas veces apareció.
    c. Cuántos fueron pares y cuantos impares (el número 0 no cuenta).
    d. Se ingresa la coordenada por teclado para informar cuál fue el número que salió.
76. Crear una matriz de 4x6, primero con ceros y luego completar con los números al azar con las caras de un dado (1...6). Mostrar la matriz y luego:
    a. Mostrar los bordes de la matriz (primera y última fila y primera y última columna). Usar funciones de NumPy.
    b. Ingresando por teclado un número x para que el programa informe el porcentaje que obtuvo.
77. Representar la función $f(x) = \frac{1-x^2}{e^x}$, el dominio debe ir desde -2 a 7.
78. Representar la función $f(x) = x^3 - \frac{2x-1}{x^2}$, el dominio debe ir desde -6 a 8.
79. Crear la gráfica de tipo pie (torta) con los datos del último censo, para representar los porcentajes de población de las cinco provincias patagónicas.

Capítulo 6

Modularización: funciones y procedimientos

> Por norma, los sistemas *software* no funcionan bien hasta que han sido utilizados y han fallado repetidamente en entornos reales.
>
> Dave Parnas*

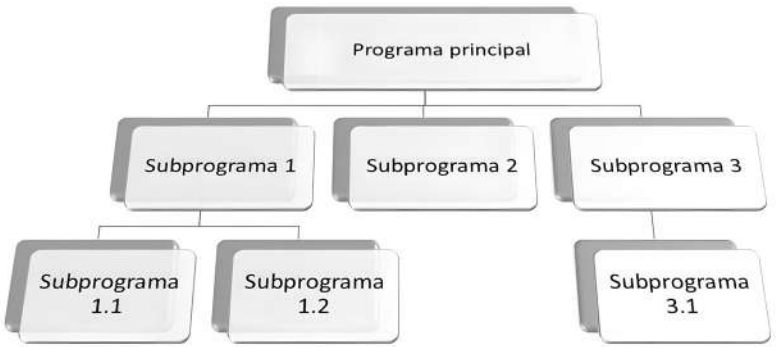
Temas:

Concepto de modularización. Parámetros. Funciones. Procedimientos. Combinación de funciones y procedimientos. Variables locales y globales. Palabras reservadas. Problemas.

6.1. Modularización

Hasta el momento hemos visto en Python un lenguaje que permite solamente trabajar con un paradigma, la programación estructurada (secuencia, selección e iteración). Pero existen otros paradigmas como el orientado a objetos (utilizando clases POO) y la modularización. Este último es el que vamos a estudiar en este capítulo porque nos brinda beneficios muy importantes.

La modularidad o modularización es una propiedad que existe prácticamente en todos los lenguajes de programación. Este método nos permite subdividir nuestro programa en subprogramas que habitualmente llamamos *módulos* los cuales deben ser independientes.



* Dave Lorge Parnas es un ingeniero en software canadiense; transformó el concepto del desarrollo de software convirtiendo códigos caóticos en sistemas elegantes y manejables.

Esta metodología también se la conoce como *top-down*. Cuando hablamos de independencia nos referimos a que pueden ser reutilizados por otras aplicaciones y, además, que tienen una baja interacción con el resto de los módulos.

Python cuenta con diversos módulos que podemos traer a nuestro código y utilizarlos para facilitar y ahorrar la programación de código entre otros beneficios, por ejemplo, el módulo *math* que incluye varias operaciones matemáticas que pueden ser llamadas como una función luego de importar el módulo.

Además podemos crear nuestros propios módulos para utilizar en nuestros proyectos cuando sea necesario. Por ejemplo, si queremos crear un módulo que muestre el resultado de una función logarítmica o el punto de vértice de una cuadrática, simplemente, se importa y se utiliza sin necesidad de crear nuevamente todo el código del algoritmo. Con este método se obtienen ventajas significativas al dividir el problema en varios subproblemas, siendo más fácil de entender ya que solo se tiene que concentrar en un pequeño problema a la vez. Estos módulos o subprogramas pueden ser funciones o procedimientos. Ambos tienen mucha similitud aunque poseen algunas diferencias que se detallan a continuación.

6. 2. Funciones

Las *funciones* son subprogramas que devuelven un valor de cualquier tipo (entero, flotante, carácter, booleano, etcétera). Si bien Python ya proporciona funciones integradas como el **print()**, **round()**, **concatenate()** y otras, también se pueden definir sus propias funciones para ser usadas en los algoritmos.

Para definir funciones en Python debemos utilizar la palabra reservada **def** y la siguiente sintaxis:

```
def nombre de la función (parámetros):
    Bloque de operaciones
    return valor de salida
```

La función siempre culmina con la palabra clave **return** y su valor de salida. Esta última línea, al igual que el bloque de operaciones, debe estar indentada.

A continuación, vemos un ejemplo sencillo.

Ejemplo 53: Hallar el eje de simetría a partir de los coeficientes **a** y **b** de la función cuadrática: $f(x) = ax^2 \pm bx \pm c$

```
def simetria(a,b):
    X=-b/2*a
    return X

# Programa principal
print("El eje de simetría es",simetria(3,5))
```

El resultado es:

```
El eje de simetría es -7.5
```

Nota
Primero se define la función y luego en el programa principal (cuerpo del programa) se efectúa la llamada a la función. En este caso, su nombre es **simetría** (con sus dos parámetros **a** y **b**) y se lo invoca dentro del **print()**.

En el ejemplo, los datos ya están establecidos como 3 y 5, pero qué sucede si quisiéramos que el programa pida los coeficientes por teclado.

```
def simetria(a,b):
    X=-b/2*a
    return X

# Programa principal
ca=float(input("Ingrese el coeficiente cuadrático "))
cb=float(input("Ingrese el coeficiente lineal "))
print("El eje de simetría es",simetria(ca,cb))
```

El resultado es:

```
Ingrese el coeficiente cuadrático 4.7
Ingrese el coeficiente lineal -2.3
El eje de simetría es 5.404999999999999
```

La función no cambió, pero sí el programa principal.

Nota
Es muy importante tener en cuenta los parámetros de la función. En este caso, se definieron dos (**a** y **b**), por lo tanto, se debe proporcionar el mismo número de parámetros, el tipo y el orden cuando se llame a la función. La variable **ca** se relaciona directamente con el parámetro **a** y la variable **cb** con el parámetro **b**, ambas de tipo flotante. ¿Qué tendríamos que hacer para que el resultado tenga solo dos decimales?

Ejemplo 54: En capítulos anteriores hemos visto cómo se obtiene el factorial de un número natural, por ejemplo: 6!=6x5x4x3x2x1=720.

Si queremos crear una función que resuelva el factorial de cualquier número, entonces:

```
def factorial(numero):
    f=1
    for i in range(1,numero+1):
        f=f*i
    return f

# Programa principal
num=int(input("Ingrese un número natural "))
print("El factorial es",factorial(num))
```

Nota
Observar cómo el parámetro **numero** *machea* con el dato **num**. La acción machear proviene del inglés *match*, que en términos técnicos significa emparejar.

Luego, agregamos en las funciones arreglos como parámetros.

Ejemplo 55: Queremos saber qué cantidad de números pares se encuentra en un vector de 20 números naturales generados al azar.

```
import numpy as np
def cant_pares(vector):
    cant=0
    for i in range(0,len(vector),1):
        if(vector[i]%2==0):
            cant=cant+1
    return cant

# Programa principal
ve=np.random.randint(1,50,size=20)
print(ve)
print("La cantidad de pares es",cant_pares(ve))
```

Nota
Recordar que para los arreglos es necesario importar el NumPy. En este caso, el vector **ve** creado en el programa principal con números al azar del 1 al 49 pasa a ser **vector** en la función **cant_pares** y la salida será **cant** donde se contabiliza la cantidad de números pares.

Si en el ejemplo 55 pedimos que además se muestre la cantidad de números impares que hay dentro del vector, podemos agregar otra función llamada **cant_impares()** o bien una única función para obtener ambos resultados.

```
import numpy as np
def cantParImpar(vec):
    cp=0;ci=0
    for i in range(20):
        if(vec[i]%2==0):
            cp=cp+1
        else:
            ci=ci+1
    return cp,ci

#Programa Principal
ve=np.random.randint(1,50,20)
print(ve)
c_par,c_impar=cantParImpar(ve)
print("La cantidad de pares es",c_par)
print("La cantidad de pares es",c_impar)
```

Nota
Las funciones permiten devolver más de un resultado **return cp,ci**, mientras respetemos el orden al invocar la función **c_par,c_impar=cantParImpar(ve)**.

Ejemplo 56: Crear dos arreglos con números binarios al azar de 15 elementos cada uno y luego indicar cuál de los dos tiene mayor cantidad de 1. Considerar que pueden tener la misma cantidad, en ese caso mostrar el mensaje **"tienen lo mismo"**.

```
import numpy as np
def cant_unos(vector):
    cant=0
    for i in range(0,len(vector),1):
        if(vector[i]==1):
            cant=cant+1
    return cant

# Programa principal
vec1=np.random.randint(0,2,size=15)
vec2=np.random.randint(0,2,size=15)
print("Vector 1")
print(vec1)
print("Vector 2")
print(vec2)
if cant_unos(vec1)>cant_unos(vec2):
    print("El vector 1 tiene mas unos")
elif cant_unos(vec1)<cant_unos(vec2):
    print("El vector 2 tiene mas unos")
else:
    print("Tienen la misma cantidad")
```

Resultado

```
Vector 1
[1 1 1 1 0 1 1 0 1 1 1 1 1 0 0]
Vector 2
[0 1 1 1 1 1 1 0 1 0 1 1 1 1 1]
El vector 2 tiene mas unos
```

Nota

En este ejemplo se presenta otra de las ventajas de la modularización: permite utilizar una misma función para el mismo propósito (obtener la cantidad de unos). Por otra parte, las llamadas a las funciones pueden ser parte de la estructura de decisión **if**.

Ejemplo 57: Crear un algoritmo que genere matrices de 4x4 con números aleatorios binarios hasta encontrar una que tenga menos de 6 ceros. En este caso termina el programa mostrando esa última matriz.

```
import numpy as np
def cant_ceros(matriz):
    cant=0
    for i in range(4):
        for j in range(4):
            if(matriz[i][j]==0):
                cant=cant+1
    return cant

# Programa principal
mat=np.random.randint(0,2,size=(4,4))
while cant_ceros(mat)>=6:
    mat=np.random.randint(0,2,size=(4,4))
print("La matriz final es")
print(mat)
```

Resultado:

```
La matriz final es
[[1 0 1 0]
 [1 1 1 1]
 [1 0 0 1]
 [1 0 1 1]]
```

Nota

El **while** va generando matrices de 4x4 hasta encontrar una que cumpla con la condición contraria. La función **cant_ceros** obtiene simplemente la cantidad de ceros.

Desafío

En el ejemplo anterior no sabemos cuántas veces se han generado matrices hasta llegar a tener una con menos de 6 ceros. El objetivo del ejercicio es doble: mostrar la última matriz y la cantidad que se generaron. Se debe tener presente que previo al comienzo del **while** ya tenemos una matriz creada.

Ejemplo 58: Crear vectores que contengan 4 números flotantes (del -10 al 10) aleatorios, hasta llegar a obtener un vector cuyo promedio esté cercano al 0, entre -0,2 y 0,2 (tolerancia 0,2). Mientras tanto se pide al algoritmo mostrar el vector o los vectores y contabilizar la cantidad generada.

```
import numpy as np
def cercano(vector):
    if (np.mean(vector)<=0.2 and np.mean(vector)>=-0.2):
        return "SI"
    else:
        return "NO"

# Programa principal
cant_V=1
vec4=np.random.uniform(-10,10,size=4)

while(cercano(vec4)=='NO'):
    print("Vector Nº ",cant_V)
    print(vec4)
    cant_V=cant_V + 1
    vec4=np.random.uniform(-10,10,size=4)

print("El vector Nº",cant_V,"final")
print(vec4)
```

Nota

Tenemos la función **cercano** que sirve para detectar si el promedio de los 4 elementos del vector se encuentra entre el rango (-0,2 y 0,2), devolviendo los caracteres **"SI"** o **"NO"** según corresponda. En este ejemplo no solo la función devuelve caracteres, también permite poner más de un **return**. En el **while** del programa principal se va mostrando el vector y el número de vector a medida que itera, hasta culminar con el resultante **"final"** culminando el algoritmo.

Observación

Cuando lo ejecutamos obtenemos una lista larga de vectores hasta llegar a cumplir con el objetivo. Si queremos ver uno por uno, detenidamente cada vector y su número de orden, podemos agregar un **input** para cortar la ejecución paso a paso. Por ejemplo: **input("Presione una tecla para continuar...")**. ¿Dónde se podría insertar esa línea de comando?

Ejemplo 59: En el cuerpo del programa crear dos vectores **vec1** y **vec2** de dimensión 10 con los números de los dados al azar, luego crear una función que devuelva otro vector de la misma dimensión y con los siguientes caracteres (**A**, **B**, **i**).

El criterio es el siguiente: si el valor del **vec1** es superior al **vec2,** entonces le corresponde una **"A"**, de lo contrario una **"B"** y si son iguales una **"i"**.

```
import numpy as np
def balance(v1,v2):
    v3=np.array(['','','','','','','','','',''])
    for i in range(10):
        if (v1[i]==v2[i]):
            v3[i]='i'
        elif (v1[i]>v2[i]):
            v3[i]='A'
        else:
            v3[i]='B'
    return v3

# Programa principal
vec1=np.random.randint(1,6,size=10)
vec2=np.random.randint(1,6,size=10)
print("El vector 1")
print(vec1)
print("El vector 2")
print(vec2)
vec3=balance(vec1,vec2)
print("El vector 3")
print(vec3)
```

Respuesta:

```
El vector 1
[4 2 5 3 3 3 4 3 1 4]
El vector 2
[5 3 1 4 3 4 4 4 3 3]
El vector 3
['B' 'B' 'A' 'B' 'i' 'B' 'i' 'B' 'B' 'A']
```

Nota

Dentro de la función **balance** creamos el vector **v3** con 10 elementos vacíos que son llenados con las letras según la condición establecida en el enunciado. Las funciones también pueden devolver arreglos de tipo carácter como sucede en este caso con **return v3**.

También podemos generar arreglos vacíos en el cuerpo del programa para pasarlo como parámetro de una función. A continuación, vemos un ejemplo al respecto.

Ejemplo 60: Cargar por teclado un vector con 5 números enteros y mostrarlo.

```
import numpy as np
def ingresarM(vec):
    for i in range(5):
        print("Ingrese el valor",i,end=" ")
        vec[i]=int(input())
    return vec

#Programa Principal
vecA=np.array(['','','','',''])
print(ingresarM(vecA))
```

Resultado:

```
Ingrese el valor 0 4
Ingrese el valor 1 6
Ingrese el valor 2 7
Ingrese el valor 3 2
Ingrese el valor 4 9
['4' '6' '7' '2' '9']
```

En este caso, la función **ingresar** recibe como parámetro un vector vacío que luego es cargado por teclado y retornado.

Nota
El **end=" "** del **print** permite dejar el cursor a la derecha del cartel **"Ingrese el valor X"**.

---

6. 3. Procedimientos

Los *procedimientos* en Python no son muy diferentes a las funciones, solo que no devuelven ningún valor. La forma de definir procedimientos es igual al que se utiliza con las funciones.

Para definir procedimientos en Python también debemos utilizar la palabra reservada **def** con la siguiente sintaxis:

```
def nombre del procedimiento (parámetros):
        Bloque de operaciones
```

Primero vemos un ejemplo muy sencillo.

Ejemplo 61: El usuario debe cargar el tamaño en centímetros (cm) de los tres lados de un triángulo para que el programa evalúe y muestre el tipo de triángulo (escaleno, equilátero o isósceles).

```
def tipo_triangulo(a,b,c):
    if(a==b and b==c):
        print("Triángulo equilátero")
    else:
        if(a!=b and a!=c and b!=c):
            print("Triángulo escaleno")
        else:
            print("Triángulo isósceles")

# Programa principal
ladoA=int(input("Ingrese el lado A en cm "))
ladoB=int(input("Ingrese el lado B en cm "))
ladoC=int(input("Ingrese el lado C en cm "))
tipo_triangulo(ladoA,ladoB,ladoC)
```

Nota

Al no devolver nada, el procedimiento es llamado simplemente en una línea de comando. ¿Cambiaría en algo si en este ejercicio invertimos el orden de los parámetros?

Ejemplo 62: Ingresar los coeficientes **a**, **b** y **c** de una función cuadrática.

$$f(x) = ax^2 \pm bx \pm c$$

Para que devuelva las raíces o ceros (si existen), los coeficientes deben ser de tipo flotante.

```
import math as m
def raices(aa,bb,cc): #Procedimiento 1
    dis=discriminante(aa,bb,cc)
    if (dis>0):
        obtener_raices(dis,aa,bb)
    elif (dis==0):
        obtener_raices(0,aa,bb)
    else:
        print("No tiene raices reales")

def obtener_raices(dis,aa,bb): #Procedimiento 2
    if(dis==0):
        print("La raíz doble es",round(-bb/(2*a),2))
    else:
        raiz1=round((-bb+m.sqrt(dis))/(2*a),2)
        raiz2=round((-bb-m.sqrt(dis))/(2*a),2)
        print("Las dos raíces son",raiz1,"y",raiz2)

def discriminante(aa,bb,cc): #Función
    di=pow(bb,2)-4*aa*cc
    return di

#Programa principal
a=float(input("ingrese el coeficiente cuadrático a "))
b=float(input("ingrese el coeficiente lineal b "))
c=float(input("ingrese el coeficiente independiente c "))
raices(a,b,c)
```

En este algoritmo encontramos dos procedimientos y una función. Una vez ingresados los datos, lo primero que hace el programa es llamar al procedimiento **raices**, dentro de esta se llama a la función **discriminante** que devuelve lo que va dentro de la raíz cuadrada de la ecuación de segundo grado también llamada Bhaskara. Según el resultado se invoca al otro procedimiento **obtener_raices**, donde finalmente se obtiene e imprime la raíz o las raíces. En el caso que el discriminante sea negativo, la respuesta directamente se da dentro del procedimiento **raices** sin pasar por **obtener_raices**.

Nota
No es importante el orden de los módulos, mientras las librerías se mantengan en el encabezado y el programa principal o cuerpo del programa se mantenga debajo de los módulos.

Vemos un ejemplo que combina los procedimientos con las gráficas.

Ejemplo 63: Prácticamente todos los países del mundo que tienen como límite el mar generan desechos plásticos. El siguiente

algoritmo sirve para mostrar las toneladas métricas anuales de los seis países más contaminantes, teniendo la posibilidad de optar por mostrar la gráfica de barra vertical o de barra horizontal.

```
import matplotlib.pyplot as plt
def grafica_vertical(pa,des):
    plt.bar(x=pa,height=des)
    plt.show()

def grafica_horizontal(pa,des):
    plt.barh(y=pa,width=des)
    plt.show()

#Programa principal
paises=["China","Indonesia","Filipinas","Vietnam","Siri Lanka","Egipto"]
desechos=[8.8,3.2,1.9,1.8,1.6,1]
tipograf=input("Elija una gráfica V o H: ")
if (tipograf.upper()=='H'):
    grafica_horizontal(paises,desechos)
elif (tipograf.upper()=='V'):
    grafica_vertical(paises,desechos)
else:
    print("Error en el ingreso")
```

Nota

Tenemos dos procedimientos, cada uno genera una gráfica. Ambos tienen como parámetros los países y los datos numéricos de los desechos en toneladas anuales. En el programa principal se permite optar por **'V'** o **'H'** (pudiendo ingresar la letra en minúscula o mayúscula). Si el ingreso no es correcto el programa avisa del error cometido.

En resumen, los procedimientos y las funciones se parecen. La implementación es la misma usando la palabra de inicio **def**, solo cambia la salida con **return** en las funciones**.**

6. 4. Variables locales y globales

A partir del conocimiento de la modularización surge una diferencia en el uso de las variables locales y globales. Estos dos conceptos pueden ser confusos al principio, pero en realidad son bastante simples.

Las *variables locales* son las que se definen en una función o procedimiento y que solo permiten su acceso desde estas, mientras que las *variables globales* se definen en el cuerpo del programa principal y permiten su acceso desde cualquier parte del algoritmo, incluso dentro de las funciones y procedimientos.

En Python, a diferencia de otros lenguajes, no hace falta declarar las variables pues directamente toma sentido cuando se les da un valor por primera vez. Hasta el momento las variables de todos los ejemplos del libro no las hemos declarado, porque se asume un valor al crearlas (carácter, número flotante, número entero, arreglo, etcétera).

Las variables existen en un contexto que llamamos *ámbito*, es decir que una variable existe en dicho ámbito a partir del momento en que se crea y deja de existir cuando desaparece su ámbito. El ámbito local corresponde con el de una función/procedimiento, que existe desde que se hasta que termina su ejecución. No se puede acceder a las variables de una función/procedimiento desde fuera de dichos módulos. Los parámetros de las funciones/procedimientos son considerados también locales. Por ejemplo: **def raices(aa,bb,cc):** El ámbito global corresponde con el que existe desde el comienzo de la ejecución de un programa. Todas las variables definidas fuera de cualquier función/procedimiento corresponden al ámbito global, que es accesible desde cualquier punto del programa, incluidas las funciones/procedimientos. Desde el cuerpo principal del programa no se puede acceder a las variables locales de ninguna función/procedimiento.

Vemos un ejemplo de un pequeño algoritmo que simplemente desde el programa principal llama a la función **operacion** para que efectúe y devuelva una suma.

```
def operacion(b): # b es variable local
    c=10        # c es variable local
    print(a) ⇦ imprime 20
    return b+c

#Programa principal
a=20    # a es variable global
m=3     # m es variable global
print(operacion(m)) ⇦ imprime 13
print(b) ⇦
print(c) ⇦ Da error
```

Nota

Las variables locales **b** (parámetro de la función **operacion**) y **c** no alcanzan el ámbito global, por lo tanto, no se las reconoce en el cuerpo principal. Al contrario, la variable global **a** puede ser reconocida dentro de la función imprimiendo el valor 20 que fue asignado en el programa principal.

Vemos otro ejemplo con arreglos.

Ejemplo 64: Se crean dos vectores **vec1** y **vec2** en el programa principal y luego llama a la función **consecutivo** para devolver el vector pero sumando 1 a cada elemento. Luego llama a **precedente** para devolver el vector pero restando 1 a cada elemento.

```
def consecutivo(vec):
    for i in range(6):
        vec[i]=vec[i]+1
    return vec

def precedente(vec):
    for i in range(6):
        vec[i]=vec[i]-1
    return vec

#Programa principal
vec1=[2,4,3,6,1,2]
vec2=[3,6,1,5,1,4]
print("Vector 1",vec1)
print("-->",consecutivo(vec1))
print("Vector 2",vec2)
print("-->",precedente(vec2))
```

Resultado:

```
Vector 1 [2, 4, 3, 6, 1, 2]
--> [3, 5, 4, 7, 2, 3]
Vector 2 [3, 6, 1, 5, 1, 4]
--> [2, 5, 0, 4, 0, 3]
```

Nota

Cada función tiene sus variables **i** y **vec** que al ser locales no producen cortocircuito entre las funciones **consecutivo** y **precedente** por ser declaradas en sus ámbitos locales, es decir que pueden tranquilamente repetirse entre funciones/procedimientos. El problema sería que las variables se utilicen en el cuerpo principal del programa.

Ejemplo 65: Vemos qué sucede si en el ejemplo anterior utilizamos **vec1** y **vec2** globales en cada función, sin pasar como parámetro.

```
def consecutivo():
    for i in range(6):
        vec1[i]=vec1[i]+1
    return vec1

def precedente():
    for i in range(6):
        vec2[i]=vec2[i]-1
    return vec2

#Programa principal
vec1=[2,4,3,6,1,2]
vec2=[3,6,1,5,1,4]
print("Vector 1",vec1)
print("-->",consecutivo())
print("Vector 2",vec2)
print("-->",precedente())
```

El resultado es el mismo ya que lo que está declarado globalmente (**vec1** y **vec2**) puede ser usado dentro de cada módulo (función/procedimiento).

Nota
Cuando los módulos (función/procedimiento) no tienen parámetros simplemente se le agregan los paréntesis, **consecutivo()** y **precedente()**.

---

6. 5. Palabras reservadas

Como sabemos hay palabras que están reservadas en Python para definir reglas y estructuras, por lo que no se pueden utilizar como variables. En total son 36 palabras reservadas. A continuación, mostramos algunas (las que están en negrita son las más conocidas).

and	continue	**else**	**for**	**import**	**not**	raise
assert	**def**	except	from	**in**	**or**	**return**
break	del	exec	**global**	**is**	pass	try
class	**elif**	finally	**if**	lambda	**print**	**while**
False	**True**	**with**	await	async	**as**	None

Nota
Puede que difiera (muy poco) la lista de palabras reservadas según la versión que usamos de Python. Para exponer la lista escribimos en el *prompt* >>>**help()** y luego **keywords**.

6. 6. Problemas

Para la resolución de los problemas es necesario contener un programa principal para invocar a los módulos (funciones/procedimientos).

80. Crear un programa que pida el tamaño de un arreglo (vector) y el rango de números enteros (a, b), luego que lo complete aleatoriamente al arreglo y lo muestre. Por último que obtenga el promedio. Usar dos procedimientos y una función.
81. Crear un programa que cargue de modo aleatorio 10 números enteros (entre el -10 y el 10) en un arreglo, luego otra función que reciba el arreglo y devuelva otro arreglo con sus cuadrados. Debe mostrar ambos arreglos.
82. Crear un algoritmo que permita cargar por teclado 4 vectores de 4 elementos flotantes cada uno y convierta esos vectores en una matriz de 4x4 y lo muestre. Usar una función para cargar cada vector y un procedimiento para armar la matriz y mostrarla.
83. Crear un programa que cargue de modo aleatorio números enteros (entre el 1 y el 6) en dos vectores (cada uno con 10 elementos) y luego implementar los siguientes módulos: **fusionar(vector1,vector2)** que devuelva un arreglo con los elementos de cada vector intercalados y **ordenar(vector_intercalado)** que muestre el vector final ordenado. Finalmente, el vector intercalado tendrá 20 elementos.
84. Crear un programa que genere y muestre matrices de 3x3 con números binarios aleatorios hasta que aparezca una matriz identidad, es decir, solo unos en la diagonal principal y el resto ceros. Al finalizar el programa debe informar cuantas matrices se generaron. El análisis lo debe hacer una función que devuelva **"SI"** o **"NO".**
85. Crear un algoritmo que genere al azar matrices de 4x4 con los números del dado. Mientras tanto debe compararlo con la siguiente matriz de nombre **origen**:

1	3	1	5
6	2	2	4
1	5	3	3
4	5	1	6

El programa se termina cuando la diferencia entre la matriz **origen** y la nueva matriz no pase el 50 %, para eso debe comparar elemento por elemento.
Usar una función que devuelva el porcentaje de similitud.

Nota
La matriz **origen** se debe ingresar por teclado dentro de una función y la salida es la matriz completa.

---

86. Crear un algoritmo que siga los siguientes pasos:
    a. Crear un vector **v_origen** de 10 números enteros positivos generados al azar cuyo rango de valores se encuentre entre el 1 y el 5.
    b. Obtener otro vector paralelo que contenga los factoriales **v_factorial** de cada elemento del vector del paso anterior.
    c. Obtener el promedio de este último.

    Estos tres pasos lo debe hacer el algoritmo hasta llegar a obtener un promedio del vector factorial **v_factorial** mayor a 63, mientras tanto se debe imprimir **v_origen**, **v_factorial** y el **promedio.<**.
    El punto **b** y **c** deben ser funciones que devuelvan un vector y un flotante, respectivamente. Luego en el programa principal se los imprime.
    Se recomienda usar una pausa para ver detenidamente los 2 vectores y su promedio **input("Presione una tecla para continuar...")**.
87. Crear un algoritmo para calcular la cantidad de habitantes por km cuadrado de las cinco provincias patagónicas. Para eso se necesitan tres vectores: 1) los nombres de las provincias, 2) la cantidad de habitantes según el último censo, y 3) la cantidad de km cuadrados de cada provincia. El cuarto vector lo debe generar y devolver una función del programa.
    Luego mostrar las provincias y la población de modo ordenado de mayor a menor según la población. Para esto último, usar un procedimiento.
88. Crear el juego similar al tatetí donde el programa genere una matriz de ceros y unos aleatoriamente. Luego detecte si se produce la secuencia del tatetí. Para eso debe crear tres funciones que devuelvan **"SI"** o **"NO",** una para detectar verticalmente, otra horizontalmente y, por último, diagonalmente. Lo debe hacer tanto para los ceros como para los unos. Puede pasar que se genere más de un tatetí. Los posibles resultados serán: ganador el 1 con **x** tatetí, ganador el número 0 con **x** tatetí, o bien, un empate. La **x** es la cantidad de tatetís logrados.

89. Crear un algoritmo que genere al azar una matriz de 3x10 con los números del dado y lo muestre. Luego que informe cuál es la cara más repetida y cuántas veces aparece. Considerar que puede aparecer más de una cara con mayor repetición.
90. Crear un algoritmo que genere al azar una matriz de 7x7 con los números del 1 al 9 y lo muestre, luego que genere y muestre una nueva matriz pero invertida. Es decir, el primer valor pasa a ser el último; el segundo, el penúltimo y así sucesivamente.
    Por último que genere y muestre la traspuesta de la primera matriz y la inversa de la segunda. En total se obtendrán 4 matrices. Para obtener la inversa de una matriz es necesario averiguar cómo se hace (se debe usar la librería NumPy).
91. Crear un algoritmo para generar un vector de 30 letras al azar entre el **A**, **B** y **C**, luego el usuario a través de un pequeño menú debe elegir entre las siguientes opciones:
    a. Mostrar el vector.
    b. Mostrar gráfica pie.
    c. Mostrar gráfica de barras.
    d. Salir del programa.

    Es decir, mientras el usuario no ingresa la opción 4 debe permanecer el programa activo. La creación del vector se hace por única vez dentro del programa principal y antes del inicio del menú. Las gráficas representan las cantidades de las letras **A**, **B** y **C**.

Autorías y colaboraciones

Martín Mariano Julio Goin

Universidad Nacional de Río Negro, Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad Argentina. Río Negro, Argentina.

Es magister en Educación en Entornos Virtuales (Universidad Nacional de la Patagonia Austral), licenciado en Ciencias de la Computación (Universidad de Buenos Aires). Además de haber trabajado en el nivel primario, secundario y terciario en distintas instituciones de Buenos Aires y San Carlos de Bariloche, ejerció como docente en las universidades: UBA, UTN, UNComa y FASTA. Desde el 2009 en la UNRN (Sede Andina) trabaja como docente regular con dedicación completa en el área de informática y matemática, además, participa en proyectos de investigación y extensión. Desde 2016 lleva adelante actividades de investigación en el Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad (CIEDIS). En el marco de actividades de educación continua, dicta cursos y talleres vinculados a la enseñanza de la programación y a las herramientas vinculadas con la enseñanza mediada por las TIC. Ha realizado publicaciones en diversas revistas de tecnología y educación, así como presentaciones en congresos y jornadas. Cuenta con la publicación de dos libros de cátedra en la Editorial UNRN *Caminando junto al lenguaje c* y *Problemas y algoritmos. Un enfoque práctico*. Dirige desde el 2020 el proyecto de extensión y de trabajo social denominado «JAM, Juego Algorítmico de Mesa» que fue registrado en la Dirección Nacional de Derecho de Autor (DNDA) del Ministerio de Justicia y Derechos Humanos de la Nación Argentina.

Edith Noemí Lovos

Universidad Nacional de Río Negro, Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad Argentina. Río Negro, Argentina.

Es magister en Tecnología Informática Aplicada en Educación (UNLP, Facultad de Informática), especialista en Docencia Universitaria (UNRN), ingeniera en Sistemas de Información y Analista Programador Universitario (UNICEN, Facultad de Ciencias Exactas). Desde el año 1998 participa como docente en cursos universitarios vinculados a la resolución de problemas, algoritmos y programación en diferentes instituciones universitarias (PEUZO-UNS, CURZA-UNComa, UNRN). Desde 2014 realiza actividades de

investigación en el Centro Interdisciplinario de Estudios sobre Derechos, Inclusión y Sociedad (CIEDIS), Sede Atlántica de la Universidad Nacional de Río Negro (UNRN). Se especializa en tecnologías disruptivas y procesos de inclusión de las TIC en las prácticas docentes de nivel medio y superior. Dirige un proyecto de investigación sobre procesos de apropiación de tecnologías digitales por parte de los estudiantes de los diferentes niveles educativos y modalidades que tienen lugar en la provincia de Río Negro. Desde el año 2009 se desempeña como docente en el área de lenguajes y algoritmos de la licenciada en Sistemas (UNRN, Sede Atlántica) y desde 2020 en el área de tecnología educativa para los ciclos de complementación de la licenciada en Educación Primaria/Inicial (UNRN). Asimismo, desde 2010 realiza actividades de extensión universitaria vinculadas al desarrollo del pensamiento computacional en el contexto de la UNRN. Ha realizado publicaciones en diversas revistas de tecnología y educación, así como presentaciones en congresos y jornadas vinculas a la temática. Cuenta con la publicación de un libro de cátedra en la Editorial UNRN titulado *Problemas y algoritmos. Un enfoque práctico*.

Martín Goin ; Edith Lovos.
Primera edición - Viedma : Universidad Nacional de Río Negro, 2025.
Libro digital, PDF - (Lecturas de cátedra)

Archivo Digital: descarga y online

ISBN 978-987-8258-98-0

1. Computación. 2. Lenguaje de Programación. I. Lovos, Edith II. Título
CDD 005.1

© Universidad Nacional de Río Negro, 2025.
editorial.unrn.edu.ar
Belgrano 526, Viedma, Río Negro, Argentina.

© Goin, Martín, 2025.
© Lovos, Edith, 2025.
Queda hecho el depósito que dispone la Ley 11.723.

Dirección editorial: Ignacio Artola
Coordinación editorial: Diego Martín Salinas
Edición de textos: Verónica García Bianchi
Corrección de pruebas: Diego Martín Salinas
Diagramación y diseño: Sergio Campozano
Imagen de tapa: Editorial UNRN, 2025.

Esta obra tuvo el apoyo de la Secretaría de Investigación, Creación Artística, Desarrollo y Transferencia de Tecnología y la Secretaría de Docencia y Vida Estudiantil de la Universidad Nacional de Río Negro.

Licencia Creative Commons. BY-NC-ND
Usted es libre de compartir, copiar, distribuir, ejecutar
y comunicar públicamente esta obra bajo las condiciones de:
Atribución - No-comercial - Sin obra derivada

Pensamiento lógico y programación en Python
fue compuesto con la familia tipográfica Alegreya y Liberation
en sus diferentes variables.

Se editó en noviembre de 2025 en la Dirección de Publicaciones-Editorial de la UNRN.

Pensamiento lógico y programación en Python

En un mundo cada vez más digital, saber programar es una habilidad fundamental. Este libro es un punto de partida hacia el lenguaje de programación más relevante de la actualidad: Python.

El libro es una guía integral para personas que se inician en el mundo de la programación. Aborda desde los conceptos más fundamentales hasta la aplicación práctica. Cada capítulo invita a poner manos a la obra con ejercicios y ejemplos pensados para facilitar el aprendizaje progresivo. Sin dudas, el aprendizaje de Python es una oportunidad para desarrollar el pensamiento lógico y la creatividad.

L U A

EDITORIAL UNRN

Universidad Nacional de Río Negro

9 789878 258980