

Python para el modelado y simulación basada en agentes en las Ciencias Sociales

ANTONIO AGUILERA ONTIVEROS
GLORIA ALFERES VARELA



Python para el modelado y simulación
basado en agentes en las ciencias sociales

COLECCIÓN INVESTIGACIONES

PYTHON PARA EL MODELADO
Y SIMULACIÓN BASADO EN
AGENTES EN LAS CIENCIAS
SOCIALES

ANTONIO AGUILERA ONTIVEROS
GLORIA ALFERES VARELA



EL COLEGIO
DE SAN LUIS

300

A283p

Aguilera Ontiveros, Antonio

Python para el modelado y simulación basado en agentes en las ciencias sociales [Libro electrónico] / Antonio Aguilera Ontiveros, Gloria Alferes Varela. -- 1ª edición. -- San Luis Potosí, San Luis Potosí : El Colegio de San Luis, A.C., 2024.

1 recurso en línea (231 páginas) : incluye gráficas -- (Colección Investigaciones)

Incluye índice y bibliografía (páginas 227-231)

ISBN de El Colegio de San Luis (978-607-2627-23-9) ebook

1. Interacción social 2. Ciencias sociales - simulación, métodos de 3. Simulación de computadora digital 4. Ciencias sociales - Modelos matemáticos I. Alferes Varela, Gloria, coaut.

Esta obra fue dictaminada por evaluadores externos a El Colegio de San Luis por el método de doble ciego.

Primera edición: 2025

Diseño de la portada: Maygualida Alba Aguilar.

© Antonio Aguilera Ontiveros y Gloria Alferes Varela

D.R. © El Colegio de San Luis

Parque de Macul 155

Fracc. Colinas del Parque

San Luis Potosí, S.L.P., 78299

ISBN 978-607-2627-23-9

Impreso y hecho en México

ÍNDICE

Introducción

Capítulo 1. Modelado basado en agentes y su uso en la investigación social

1.1. Introducción	15
1.2. Modelos basados en agentes: conceptos básicos	16
1.3. Descripción matemática de un modelo basado en agentes	20
1.4. El protocolo ODD de diseño de modelos basados en agentes	23
1.5. Construcción de sociedades artificiales	25

Capítulo 2. Introducción a Python

2.1. Introducción	35
2.2. ¿Qué es Python?	35
2.3. ¿Por qué Python?	37
2.4. Instalación de Python	38
2.5. Introducción a IDLE, entorno de desarrollo de Python	43
2.6. Recorrido básico por las herramientas esenciales de IDLE	44
2.7. Jupyter Notebook	48
2.8. Anaconda	55

Capítulo 3. Estructura y elementos del lenguaje de Python

3.1. Elementos del lenguaje	61
3.2. Entrada y salida de datos	66
3.3. Explorando las colecciones en Python: listas, tuplas, diccionarios y conjuntos	68
3.4. Estructuras de control de flujo	76
3.5. Solución de ejercicios	82

Capítulo 4. Funciones

4.1. Introducción a las funciones en Python	87
4.2. Parámetros de la función	88
4.3. Llamadas de retorno	91
4.4. ¿Cómo saber si una función existe y puede ser llamada?	93
4.5. Llamadas recursivas	94
4.6. Ejercicios del capítulo	94
4.7. Solución de ejercicios	95

Capítulo 5. Orientación a objetos

5.1. Introducción a la programación orientada a objetos	99
5.2. Clases y objetos	100
5.3. Herencia	103
5.4. Herencia múltiple	104
5.5. Polimorfismo	105
5.6. Encapsulación	106
5.7. Ejercicios del capítulo	108
5.8. Solución de los ejercicios	109

Capítulo 6. Métodos especiales

6.1. <code>__new__(cls, args)</code>	115
6.2. <code>__del__(self)</code>	116
6.3. <code>__str__(self)</code>	116
6.4. <code>__eq__()</code>	117
6.5. <code>__ne__()</code>	118
6.6. <code>__lt__()</code>	118
6.7. <code>__le__()</code>	120
6.8. <code>__gt__()</code>	121
6.9. <code>__ge__()</code>	121
6.10. <code>__len__()</code>	122

Capítulo 7. Más métodos de listas, cadenas y diccionarios

7.1. Introducción	123
7.2. Métodos de listas	123
7.3. Métodos de cadenas	126
7.4. Métodos de diccionarios	129

7.5. Ejercicios del capítulo	135
7.6. Solución de ejercicios	136
 Capítulo 8. Introducción a MESA	
8.1. Introducción a MESA	143
8.2. Aspectos básico de MESA	144
8.3. Recursos necesarios.	146
8.4. Un ejemplo del uso de MESA.	147
 Capítulo 9. Modelos socioeconómicos usando MESA	
9.1. Introducción	155
9.2. El modelo de riqueza de Boltzmann.	157
9.3. El modelo de Axelrod de difusión cultural	185
 Capítulo 10. Conclusiones	
10.1. Introducción	211
10.2. Importancia del modelado basado en agentes en las ciencias sociales .	212
10.3. Ventajas de Python y la biblioteca MESA en el MSBA.	213
10.4. Aplicaciones y casos de estudio	215
10.5. Desafíos y limitaciones del MSBA.	217
10.6. Desarrollos recientes y tendencias futuras en el MSBA	219
10.7. Implicaciones metodológicas y teóricas para las ciencias sociales	221
10.8. Guía práctica y recomendaciones para futuros investigadores	223
10.9. Reflexiones finales	226
 Referencias	

INTRODUCCIÓN

El uso de técnicas formales en general, y del análisis computacional en particular, está desempeñando un papel cada vez más importante en el desarrollo de teorías de sistemas complejos compuestos por seres humanos tales como grupos, equipos de trabajo, organizaciones y sus arquitecturas de mando y control. Una de las razones es el reconocimiento cada vez mayor de que los procesos subyacentes son complejos, dinámicos, adaptativos y no lineales, y de que el comportamiento de los grupos o equipos surge de las interacciones entre los agentes y entidades que componen la unidad (personas, subgrupos, tecnologías, etc.) y las relaciones entre estas entidades que limitan y permiten la acción a nivel individual y de unidad. Otra razón del paso a los enfoques computacionales es el reconocimiento de que las unidades compuestas por varias personas son intrínsecamente informáticas, ya que necesitan escanear y observar su entorno, almacenar hechos y programas, comunicarse entre sus miembros y con su entorno, y transformar la información mediante la toma de decisiones humanas o automatizadas (Baligh *et al.*, 1990). En general, el objetivo de la investigación computacional es construir nuevos conceptos, teorías y conocimientos sobre sistemas complejos de actividad humana. Este objetivo puede alcanzarse, y se está alcanzando, mediante el uso de una amplia gama de modelos computacionales que incluyen la simulación por computadora, la modelación numérica y los modelos de emulación que se centran en los procesos subyacentes de los sistemas (Carley, 1999).

El modelado y la simulación basados en agentes (MSBA) es un enfoque para modelar sistemas complejos desde el punto de vista *bottom up*. Se considera que el sistema complejo se compone de agentes autónomos, y para explicar la dinámica del sistema se toman en cuenta sus interacciones, así como las interacciones con su entorno. En las ciencias sociales, los sistemas basados en agentes sirven para modelar fenómenos

socioeconómicos y sociopolíticos complejos en donde las interacciones entre los agentes se fundamentan en creencias, deseos y actitudes, que tienen que ver con las creencias y actitudes políticas y las normas sociales. Las aplicaciones van desde el modelado del comportamiento de los agentes en el mercado de valores y cadenas de ofertas hasta predecir la propagación de epidemias, la amenaza de guerra biológica, modelar el crecimiento y declive de civilizaciones antiguas y la migración internacional. El MSBA pretende tener efectos de largo alcance en la forma en que los científicos sociales usan la computación para apoyarse en la generación de teorías y cómo se utilizan los lenguajes computacionales y los algoritmos de inteligencia artificial para investigar las sociedades actuales y del pasado. Algunos entusiastas del MSBA han ido tan lejos como para sostener que estos son una nueva forma de hacer ciencia social, la cual da paso al desarrollo de lo que se conoce como sociología computacional (Epstein y Axtell, 1996; Squazzoni, 2012; Aguilera y Abrica-Jacinto, 2022).

Este libro describe los fundamentos de MSBA en las ciencias sociales. Introduce al lector lego en los pormenores de la programación en el lenguaje Python. Explica cómo construir *kits* de herramientas para MSBA baados en el lenguaje de programación Python y la biblioteca MESA de Python. Instruye sobre el uso del entorno de desarrollo Jupyter y Jupyter Notebook. Describe métodos de construcción de MSBA ilustrados a través de tres ejemplos icónicos en las ciencias sociales, esto es, el modelo de difusión de opinión basado en el modelo de Ising, el modelo de riqueza de Boltzmann (Drăgulescu y Yakovenko, 2000) y el modelo de difusión cultural de Axelrod (1997)

El uso de Python para el desarrollo de los MSBA es único en un libro de estas características. Creemos que el uso del lenguaje Python aporta una ventaja al investigador interesado en aprender el MSBA mediante un lenguaje de programación sencillo y poderoso de usar. Además, el uso de un lenguaje actual diseñado para el MSBA, tal como lo es Netlogo (Wilensky, 1999) ha sido cubierto en español en otra obra (Aguilera y Posada, 2017). Años de uso de dicho lenguaje han llevado a reconocer sus carencias y limitaciones. Por ello creemos que el siguiente paso será desarrollar *paquetes* de desarrollo en Python que se puedan usar y reutilizar para el ensamblado y construcción de MSBA.

Esperamos que este libro cubra una parte faltante en la reflexión actual en español sobre el uso del modelado y simulación basados en agentes dentro de la comunidad de investigadores sociales. Es nuestra esperanza que el uso de Python cree una comunidad fuerte de usuarios del MSBA en este lenguaje de programación, el cual es rico en procesos ya previamente construidos por una comunidad de programadores en dicho lenguaje.

CAPÍTULO 1.

MODELADO BASADO EN AGENTES Y SU USO EN LA INVESTIGACIÓN SOCIAL

1.1. INTRODUCCIÓN

Los modelos basados en agentes (MBA) son una herramienta de modelado computacional de gran utilidad en las ciencias sociales,¹ la economía y las ingenierías. En términos de las ciencias sociales, se usan como el método privilegiado dentro de la sociología computacional (*cf.* Epstein y Axtell, 1996; Squazzoni, 2012; Aguilera y Abrica-Jacinto, 2022) y la sociología analítica (*cf.* Hedström y Ylikoski 2010; Manzo, 2014; Manzo y Matthews, 2014; León-Medina, 2017). Desarrollados en los años noventa del siglo XX, principalmente como una tecnología de *software* que usaba la idea de la inteligencia artificial distribuida (Aguilera y López-Paredes, 2001), los MBA se fueron posicionando como una forma sencilla, pero poderosa, de tratar con la complejidad de agregados de átomos, moléculas, animales y personas. Como metodología computacional de las ciencias sociales, se basa en modelar sistemas desde la perspectiva de sus elementos constitutivos, o agentes, y las relaciones entre ellos (Pereda y Zamarreño, 2015). Una de las principales ventajas de esta metodología de modelado es que existe una correspondencia directa entre los elementos del sistema y los elementos del modelo y las relaciones entre ellos en el sistema y las relaciones entre los elementos en el modelo (Galán *et al.*, 2009). El modelado basado en agentes, como su nombre indica, se basa en modelar un sistema usando el concepto abstracto de agente. No existe una definición generalmente aceptada de agente. Sin embargo, con base en la definición de Macal y North (2006), un agen-

¹ Para una revisión detallada de sus usos en las ciencias sociales, véase Squazzoni (2014).

te es un elemento individual e identificable, con un conjunto de características y reglas que guían su comportamiento y toma de decisiones. Este agente, además, está situado en un ambiente con el que interactúa.

Este capítulo tiene como objetivo introducir al lector en los conceptos básicos del modelado basado en agentes (en adelante, MBA) y su uso en las ciencias sociales computacionales. Se divide en tres apartados. En el primero se dan los conceptos básicos de los MBA, se describen las características genéricas de un agente, su estructura lógica y su entorno. En el segundo, se da una formulación matemática de un MBA. Dicha formulación sirve para formalizar el proceso de modelado y tener claridad en la estructura del MBA, sus relaciones e interacciones. En el tercer apartado, se dan las características generales del uso de los MBA en la generación de sociedades artificiales.

1.2. MODELOS BASADOS EN AGENTES: CONCEPTOS BÁSICOS

Antes de comenzar a hablar de los modelos basados en agentes (MBA), es necesario que establezcamos el concepto de agente en ciencias computacionales. Un agente puede entenderse como una entidad autónoma (principalmente es un *software*, pero también podría ser un robot) con una serie de comportamientos que van desde los más sencillos de tipo reactivo hasta los más sofisticados basados en la inteligencia artificial adaptativa. Algunos estudiosos de los agentes establecen condiciones más rigurosas para considerar un agente, por ejemplo, Casti (1997) argumenta que los agentes deberían tener tanto reglas de nivel básico para el comportamiento como un conjunto de reglas de nivel superior para “cambiar” las reglas de nivel básico. En términos generales, un agente tiene las siguientes características (Macal y North, 2006: 74).

- Un agente es identificable. Esto es, un agente es una entidad delimitada con un conjunto de características y reglas que rigen su comportamiento y capacidad de decisión. Los agentes son autónomos. El requisito de discreción implica que un agente tiene un límite y

uno puede determinar fácilmente si algo es parte de un agente, no es parte de un agente, o es una característica compartida entre agentes.

- Un agente se sitúa en un espacio. El agente vive en un entorno con el que interactúa junto con otros agentes. Los agentes tienen protocolos de interacción con otros agentes, tales como para la comunicación y la capacidad de responder al entorno. Los agentes tienen la capacidad de reconocer y distinguir los rasgos de otros agentes.
- Un agente puede dirigirse por objetivos. Los agentes tienen objetivos por lograr (no necesariamente objetivos para maximizar) con respecto a sus comportamientos. Esto permite a un agente comparar el resultado de su comportamiento con sus metas.
- Un agente es autónomo y autodirigido. Un agente puede funcionar con independencia en su entorno y en sus tratos con otros agentes, al menos sobre una gama limitada de situaciones que son de interés.
- Un agente es flexible, posee la capacidad de aprender y adaptar sus comportamientos con base en la experiencia, lo cual requiere alguna forma de memoria. Un agente puede tener reglas que modifican sus reglas de comportamiento.

En la figura 1.1. se muestran de forma gráfica los componentes de un agente.



FIGURA 1.1. UN AGENTE. ELABORACIÓN PROPIA.

Un modelo es una simplificación de la realidad. De esta manera, un modelo simplifica los componentes y procesos que integran el sistema que se analiza. El proceso de construcción de un modelo ayuda a identificar, seleccionar y organizar la información disponible sobre el funcionamiento del sistema en estudio. Esta actividad de modelación se puede observar en la figura 1.2.

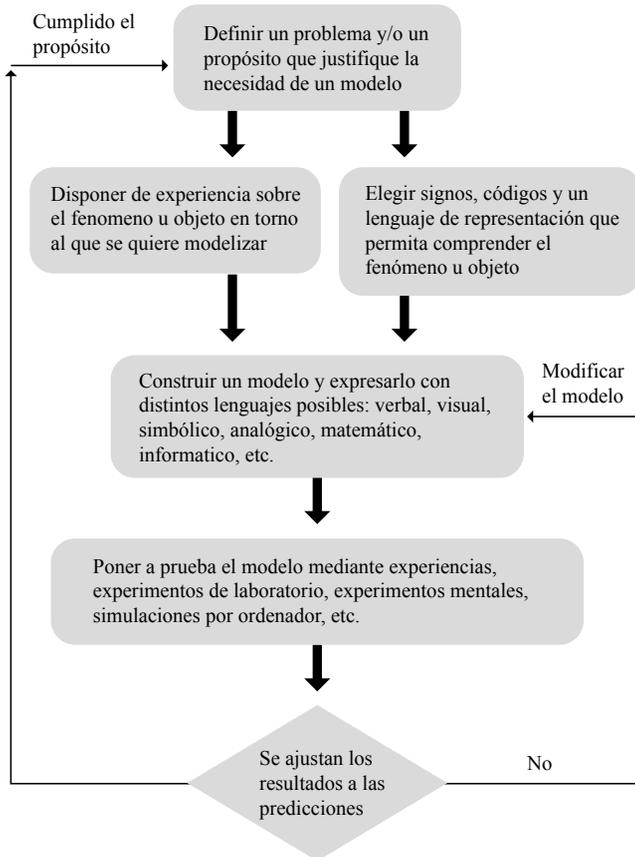


FIGURA 1.2. EL PROCESO DE MODELADO.
TOMADO DE ARAGÓN, *ET AL.* (2018: 197)

Los modelos basados en agentes son un tipo de modelado matemático-computacional en el que los elementos del sistema que se modela son

identificados con agentes y sus características. No todo sistema puede o debe ser modelado como un sistema de agentes. Por ejemplo, un gas ideal no sería un sistema apropiado para modelar con agentes, esto debido a que las partículas que lo componen se consideran homogéneas y sus interacciones no son complejas más allá de la interacción debida a la colisión entre las partículas. Dicho sistema no tiene objetivos que lograr por parte de sus componentes ni reglas de comportamiento más allá de las dadas por la física clásica. Sin embargo, un mercado sí sería un sistema para modelar con agentes, ya que los individuos que conforman el mercado son heterogéneos y sus interacciones son complejas; asimismo, los individuos tienen objetivos que lograr dentro del mercado, ya sea maximizar sus ganancias o bien llegar a acuerdos para el intercambio de productos en el mercado. Aquí lo relevante es que la interacción compleja de los agentes tanto entre ellos como con su entorno tiene repercusiones a nivel macro. Al contrario de la modelación *top-bottom*, la modelación basada en agentes es de tipo *bottom-up*.

Un MBA se compone de una colección de agentes, un ambiente a través del cual los agentes interactúan y reglas que definen las relaciones entre agentes y su ambiente y que determinan la secuencia de acciones en el modelo (Parker *et al.*, 2003). Como ya vimos, los agentes son entidades físicas o virtuales que toman decisiones de manera autónoma. Dichas decisiones afectan los estados de los agentes y los estados del sistema compuesto por los agentes y su entorno.

Los MBA tienen fuertes raíces en los campos de los sistemas multiagente (SMA) y la robótica, así como en el campo de la IA distribuida. Pero los MBA no solo se relacionan con el diseño y la comprensión de “agentes artificiales”. Sus raíces principales se encuentran en el modelado de las sociedades humanas y el comportamiento organizacional y la toma de decisiones individuales (Bonabeau 2001). Con esto, surgen áreas de investigación como son la interacción social, la colaboración, el comportamiento del grupo y el surgimiento de una estructura social de orden superior.

La dinámica macroscópica del modelo es el resultado de la dinámica de los agentes a nivel microscópico. Los agentes se representan dinámicamente a través de la evolución de sus estados, que son sus atributos dinámicos y son causados por su propia evolución o por las interacciones con

otros agentes o con el entorno. Además, un agente puede experimentar cambios de estado debido a diferentes causas, lo que permite analizar la dinámica mediante varios enfoques (Pereda y Zamarreño, 2015: 306).

1.3. DESCRIPCIÓN MATEMÁTICA DE UN MODELO BASADO EN AGENTES

Un MBA se compone en principio por una población de agentes. Los agentes pueden estar agrupados en tipos o “razas” de agentes, o bien pueden ser todos del mismo tipo. En el caso de haber varios tipos de agentes, estos los identificamos con el índice j , que toma valores desde 1 hasta el número total de tipos de agentes b (Pereda y Zamarreño, 2015: 306).

Siguiendo a Pereda y Zamarreño (2015), cada agente i_j de tipo j tiene una serie de atributos dinámicos que definen su estado en cada instante y que puede ser representado mediante un vector de estados dinámicos $\vec{x}_{i_j}(t)$ en el instante de tiempo t . Para disponer de una notación más compacta, todos los agentes del mismo tipo j se agrupan en una matriz de vectores de estado $\mathbf{X}^j(t)$. Los agentes pueden estar descritos por atributos estáticos, esto es atributos invariables en el tiempo, que favorecen la heterogeneidad del modelo y que se pueden representar con un vector $\vec{p}_{i_j}^j$. Estos parámetros pueden influir tanto en la dinámica como en el comportamiento de los agentes (Pereda y Zamarreño, 2015: 306).

En algunos modelos, el número de agentes n_j puede variar durante la simulación; esto se expresará como una dependencia temporal: $n_j(t)$. Si el número de agentes es constante en el modelo, la dependencia temporal es obviada.

En resumen, el estado de un agente en el tiempo t se presenta de la siguiente forma (Pereda y Zamarreño, 2015: 306):

- $j = 1, \dots, b$ identifica el tipo del agente.
- $i_j = 1, \dots, n_j(t)$ identifica al agente de tipo j .
- $n_j(t)$ es el número de agentes de tipo j en el tiempo t .
- $\vec{x}_{i_j}(t) \in \mathbb{R}^{m_j}$, m_j es el número de estados para el agente de tipo j .
- $\vec{p}_{i_j} \in \mathbb{R}^{q_j}$, q_j es el número de parámetros para el agente de tipo j .

Para disponer de una notación compacta podemos representar los estados del conjunto de agentes de tipo j como $X^j(t) = (\vec{x}_1^j(t) \dots \vec{x}_{i_{n_j(t)}}^j(t))$ donde $X^j(t) \in M_{m_j \times n_j(t)}(\mathbb{R})$.

De igual forma, se representan los parámetros del conjunto de agentes de tipo j como $P^j = (\vec{p}_1^j \dots \vec{p}_{i_{n_j(t)}}^j)$ donde $P^j \in M_{q_j \times n_j(t)}(\mathbb{R})$.

El entorno en el que se desenvuelven los agentes estará parametrizado de alguna forma, por ejemplo, las dimensiones del mundo en que viven los agentes. Este conjunto de parámetros que definen el entorno lo agrupamos en un vector $\vec{\alpha}$ cuyos elementos tomarán valores en un subespacio \mathbb{S} de \mathbb{R}^l siendo l el número de parámetros con el que se define el entorno. Es decir, el entorno vendrá descrito como $\vec{\alpha} \in \mathbb{S} \subset \mathbb{R}^l$ (Pereda y Zamarreño, 2015: 306).

En general, una interacción con otros agentes o con el entorno del agente i_j de tipo j se denomina 'a'. Esta interacción se expresa por medio de un método 'a' que depende de los estados y parámetros de todos los agentes (en general) y de los parámetros del entorno. Esta interacción puede originar un cambio en algunos de los estados de los demás agentes, no solamente de él mismo. Los métodos también pueden representar una consecuencia del paso del tiempo (*tick*) en vez de representar una interacción; por ejemplo, el incremento de edad de un agente o la pérdida de energía si no se ha alimentado. Expresar en formulación matemática estas interacciones no es un asunto trivial debido a que las interacciones de unos agentes pueden afectar a otros agentes (Pereda y Zamarreño, 2015: 306).

La formulación matemática de las interacciones entre agentes depende del tipo de fenómeno que se va a estudiar. Tomemos, por ejemplo, los modelos de opinión. Estos son un tipo de modelo basado en agentes que se utiliza para estudiar cómo se forman y cambian las opiniones en una sociedad. En este tipo de modelos, cada agente representa a un individuo que tiene una opinión sobre un tema determinado. Los agentes interactúan entre sí y con el entorno, y pueden cambiar su opinión en función de las opiniones de los demás agentes y de la información que reciben del entorno. La interacción entre los agentes en un modelo de opinión se puede definir mediante ecuaciones que describen cómo se propagan las opiniones a través de la red de agentes. Una de las ecuaciones más comunes es la ecuación de Deffuant-Weisbuch (Deffuant *et al.*,

2000), que se utiliza para modelar la influencia social en la formación de opiniones. Esta ecuación se define como

$$x_i(t+1) = x_i(t) \text{ si } |x_i(t) - x_j(t)| > \epsilon$$

y

$$\frac{1}{2} (x_i(t) + x_j(t)) + \mu \text{ si } |x_i(t) - x_j(t)| \leq \epsilon$$

Donde $x_i(t)$ es la opinión del agente i en el tiempo t , $x_j(t)$ es la opinión del agente j con el que interactúa el agente i , ϵ es un parámetro que define el umbral de tolerancia para la diferencia de opiniones, μ es un parámetro que define la intensidad de la influencia social, y η es un número aleatorio que representa el ruido en la opinión.

En la primera parte de la ecuación, si la diferencia entre las opiniones de los agentes i y j es mayor que el umbral de tolerancia ϵ , la opinión del agente i no cambia. En la segunda parte de la ecuación, si la diferencia entre las opiniones de los agentes i y j es menor o igual que el umbral de tolerancia ϵ , la opinión del agente i cambia hacia una opinión intermedia entre su opinión y la opinión del agente j , ponderada por la intensidad de la influencia social μ y un término de ruido η .

La simulación de un modelo basado en agentes es la forma en que se estudia el comportamiento del sistema. Según Klein *et al.* (2018), las simulaciones permiten soluciones numéricas de descripciones matemáticas de sistemas sociales que no son manejables empleando medios clásicos, lo cual es de modo particular importante para los modelos basados en agentes, donde un gran número de agentes posiblemente heterogéneos pueden tener interacciones complejas durante un periodo prolongado. Al simular las interacciones iteradas de los agentes a lo largo del tiempo, los modelos basados en agentes permiten la recreación de los procesos relevantes de emergencia paso a paso. Con mayor motivo, al variar sistemáticamente los parámetros individuales, las simulaciones basadas en agentes proporcionan una manera de obtener una comprensión detallada de cómo los patrones emergentes dependen de los tipos exactos de interacciones. En algunos casos, los fenómenos de interés pueden realizarse de múltiples maneras a nivel micro, lo que los hace robustos a nivel macro. En otros casos, la realización de los fenómenos de interés depende

de la presencia de variables bien definidas o de resultados aleatorios. Al controlar los efectos aleatorios y los parámetros de entrada, los modelos basados en agentes permiten estudiar ejecuciones individuales para el análisis de procesos no deterministas. Tal grado de precisión permite el descubrimiento y análisis de dependencias de trayectoria y puntos de inflexión: fenómenos que son omnipresentes en las redes sociales, pero son extremadamente difíciles de abordar con medios estadísticos clásicos.

1.4. EL PROTOCOLO ODD DE DISEÑO DE MODELOS BASADOS EN AGENTES

Debido a su naturaleza flexible y adaptable, el modelado basado en agentes se ha utilizado en diversos campos de la ciencia, tales como la economía, la biología, la sociología, la ciencia política, la informática, la geografía, y muchos otros más. Por ello se ha elaborado una estandarización en el desarrollo de descripciones de los MBA mediante un protocolo. Este protocolo, diseñado con el objetivo de comunicar los modelos basados en agentes, fue propuesto por Grimm y colegas (Grimm *et al.*, 2010), es llamado Protocolo ODD (*Overview, desing concepts and details*). Es una herramienta metodológica para el análisis de tales modelos, promueve la representación rigurosa de estos y facilita las revisiones y futuras comparaciones entre los diversos modelos. En términos generales, el protocolo se describe de la siguiente manera:

1. Visión general (*Overview*): describe de manera general qué es y cómo está diseñado el modelo, consta de tres partes:
 - a. Propósito: descripción general y breve del proceso que se desea captar con el modelo.
 - b. Entidades, variables de estado y escalas: expone el tipo de entidades (agentes, entorno espacial, ambiente global) que representa el modelo; las variables de estado que caracterizan y cuantifican a las entidades y sus escalas (temporales, espaciales). Es decir, en esta parte se describe el estado del modelo en cada momento.

- c. Visión general de los procesos y programación: describir los procesos que afectan a las variables de estado asociadas a las entidades del modelo. Respecto a la programación, el objetivo es dar un orden a la sucesión de los procesos que afectan al sistema.
2. Conceptos de diseño (*Design concepts*): establecer el conjunto de propiedades y conceptos básicos para la aplicación del modelo, tales como los principios básicos (teóricos), emergencia, adaptación, objetivos, aprendizaje, predicción, detección, interacción, aleatoriedad, colectivos, observación.
- I Detalles (*Details*): establecer los detalles necesarios para completar la descripción del modelo mediante:
 - a. Inicialización: condiciones iniciales establecidas en la simulación.
 - b. Entradas: detallar el inicio de las variables con el fin de conocer el estado de todas las variables y computar así las actualizaciones de tales variables.
 - c. Submodelos: descripción detallada de las ecuaciones, reglas lógicas, algoritmos y parámetros del modelo.

En la siguiente figura se muestra el protocolo ODD:

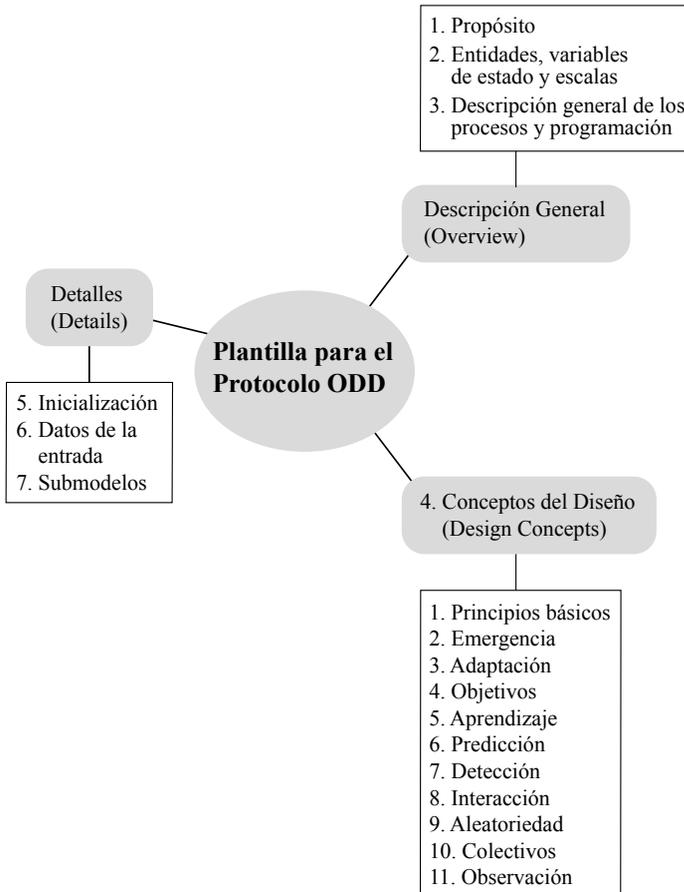


FIGURA 1.3. PROTOCOLO ODD. TOMADO DE
AGUILERA Y ABRICA-JACINTO, 2019.

1.5. CONSTRUCCIÓN DE SOCIEDADES ARTIFICIALES

La idea detrás del modelado basado en agentes en las ciencias sociales es la propuesta y construcción de sociedades artificiales con las cuales se puede indagar la relación entre seres sociales y sus efectos a nivel macro.

“Vemos las sociedades artificiales como laboratorios, donde intentamos crecer ciertas estructuras sociales en la computadora -o in silico- con el objetivo de descubrir mecanismos locales o micro-fundamentos que sean suficientes para generar las estructuras sociales macroscópicas y los comportamientos colectivos de interés” (Epstein y Axtell, 1996:4).

Las sociedades artificiales son una metodología novedosa de indagación en las ciencias sociales. Parten de un enfoque *bottom-up* en el que se considera que los individuos de una sociedad interactúan entre sí de alguna forma, la cual genera comportamientos colectivos observables.

En el estudio de las sociedades artificiales como medio de indagar la sociedad, está inmerso un conjunto de supuestos ontológicos y epistemológicos que debemos reflexionar y tener en cuenta. En primer lugar, en el contexto de la computación, una ontología es un modelo que puede representar aspectos importantes de un fenómeno o un área de interés utilizando un lenguaje formal legible por máquina basado en lógica matemática. La ontología en el estudio de las sociedades artificiales se refiere a la conceptualización y representación de las entidades que componen estas sociedades simuladas. Algunos elementos clave sobre la ontología en este campo son:

- Se define una ontología de agentes, que especifica las entidades participantes en la sociedad artificial como agentes con propiedades y comportamientos.
- Se conceptualizan relaciones entre agentes, como interacciones, vínculos sociales, jerarquías, etc. Esto define una ontología de red social.
- Emergen propiedades y estructuras colectivas a nivel de sistema, como normas, cultura, instituciones. Esto constituye una ontología sistémica.
- El entorno donde se sitúan los agentes también se conceptualiza ontológicamente, incluyendo recursos, infraestructura, etcétera.
- Se pueden definir ontologías de dominio específico según el fenómeno modelado (economía, política, epidemias, etcétera).

Los aspectos epistemológicos que se deben tener en cuenta al estudiar aspectos sociales a través de la metodología de las sociedades artificiales son los siguientes:

- El conocimiento sobre la sociedad se obtiene mediante la simulación computacional. Permite estudiar las propiedades emergentes.
- La validez del conocimiento depende de la precisión con la que se modelen los agentes y su entorno. Requiere una abstracción cuidadosa.
- El enfoque es empirista: se explora el comportamiento del sistema a través de experimentos.
- Pero también es mentalista: se modelan constructos no observables como creencias y deseos.
- Permite combinar explicaciones individualistas (nivel micro) y holistas (nivel macro).

Las sociedades artificiales plantean ontologías y formas de conocimiento mixtas, que integran elementos empiristas, racionalistas, mentalistas y de sistemas complejos. Su riqueza reside justamente en combinar múltiples enfoques.

Las sociedades artificiales tienen aplicaciones en:

- Modelado social y político: predicción de comportamientos sociales, propagación de ideas, formación de normas, etcétera.
- Economía computacional: modelado de mercados, intercambio de bienes, formación de precios.
- Coordinación multiagente: diseño de sistemas de agentes que cooperan para resolver tareas complejas.
- Entretenimiento: videojuegos, mundos virtuales habitados por agentes autónomos.

La investigación con sociedades artificiales no está libre de sesgos y aspectos éticos que pueden afectar las interpretaciones y explicaciones que emergen del uso de modelos en la investigación social. En primer lugar, el investigador que diseña la sociedad artificial puede generar sesgos y discriminación en la elaboración de los modelos, que pueden afectar el comportamiento de los agentes en el modelo. También está el problema de la transparencia en los criterios que determinan las interacciones en estas sociedades; muchas veces el investigador, si bien ve el modelo matemático computacional, no tiene manera de verificar la forma en que

están interaccionando los agentes en las simulaciones, debido muchas veces a factores estocásticos que convierten al modelo en una caja negra.

La investigación en de sociedades artificiales usando el MSBA ha desarrollado un proceso de investigación más o menos estandarizado que consiste en una secuencia de pasos. En la práctica, varios de estos pasos ocurren en paralelo, y todo el proceso a menudo se realiza de forma iterativa a medida que las ideas se refinan y desarrollan. La figura 1.4 muestra los principales pasos que los investigadores deben seguir para construir un modelo basado en agentes. La secuencia comienza con la teoría social y termina con “el objetivo” o el proceso social que el investigador está interesado en modelar.

Entre estos dos puntos, se deben lograr una serie de pasos para desarrollar un modelo sólido basado en agentes. Salgado y Nigel (2013) han identificado tres grandes etapas: 1) Especificación y formalización; 2) modelado, verificación y experimentación; y 3) calibración y validación. La primera etapa consiste en traducir la hipótesis teórica que explica el proceso social de interés, generalmente expresado en lenguajes naturales, a lenguajes formales, utilizando la lógica o las matemáticas. La segunda etapa incluye el modelado en sí, en el que el investigador construye y verifica el modelo por medios experimentales. El tercer paso incluye la calibración del modelo con datos empíricos y la consiguiente validación de este mediante pruebas estadísticas adecuadas.

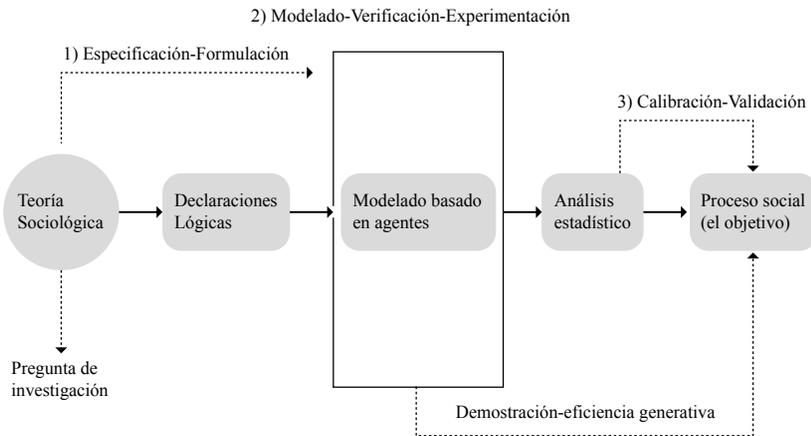


FIGURA 1.4. PRINCIPALES PASOS Y ETAPAS PARA CONSTRUIR UN MODELO BASADO EN AGENTES (ADAPTADO DE SALGADO Y GILBERT, 2013: 250).

1.5.1. Especificación y formalización

Es esencial definir con precisión la pregunta (o preguntas) de investigación que el modelo abordará en una etapa temprana. Las preguntas de investigación típicas que los investigadores intentan responder cuando utilizan el MSBA son aquellas que explican cómo las regularidades observadas en el nivel social o macro pueden surgir de las interacciones de agentes a nivel micro. Según Epstein (1999, 2007), se dan características que distinguen el enfoque tanto de la ciencia inductiva como de la deductiva. Los investigadores que utilizan MSBA intentan resolver en general la siguiente pregunta: ¿Cómo podrían las interacciones locales descentralizadas de agentes heterogéneos generar o provocar el fenómeno macro dado? Trabajar con el MSBA tiene como objetivo responder a cualquier macropropiedad que se quiera explicar.

Para definir con precisión la pregunta de investigación, también es necesario que el modelo esté integrado en las teorías sociales existentes. Es importante revisar las teorías existentes relacionadas con la pregunta de investigación del modelo para identificar los mecanismos causales que con probabilidad sean significativos en el modelo. Por tanto, es importante elegir la teoría que proporciona el mecanismo causal más plausible

y empíricamente comprobable. En este sentido, una característica importante del MSBA es que no impone ninguna restricción a priori a los mecanismos que se supone están funcionando. El MSBA no se basa en ninguna teoría de la acción o interacción (Hedström y Ylikoski, 2010). Es una metodología para derivar los resultados sociales que es probable generen grupos de actores interactuantes, cualesquiera que sean las lógicas de acción o las estructuras de interacción.

Evidentemente, un modelo es siempre una simplificación del “mundo real”. De hecho, esta es la razón por la que los científicos construyen modelos: quieren reducir la complejidad del mundo y aislar los elementos principales que provocan el fenómeno que se va a explicar. Por esta razón, los investigadores que utilizan MSBA pretenden especificar los mecanismos causales que subyacen a algún fenómeno. Según Miller y Page (2007), especificar una teoría es reducir el mundo a un conjunto fundamental de elementos (clases de equivalencia) y leyes (funciones de transición), y sobre esta base comprender mejor y quizá predecir aspectos clave del mundo. Como comentamos antes, en el MSBA los investigadores tienen que especificar los agentes que van a estar involucrados en el modelo y el entorno en el que van a participar y actuar. Para cada tipo de agente en el modelo, es necesario especificar los atributos y las reglas de comportamiento, es decir, el conjunto de reglas simples que especifican cómo se comportan y reaccionan los agentes ante su entorno local. Un atributo es una característica o rasgo del agente, y es algo que ayuda a distinguir al agente de otros en el modelo y no cambia, o algo que cambia a medida que se ejecuta la simulación.

Una vez que se ha identificado la teoría apropiada (la que proporciona explicaciones causales plausibles sobre el proceso social objetivo) y se han especificado las reglas de comportamiento, el investigador está equipado con las hipótesis teóricas que debe probar, en este caso, *in silico*, es decir, utilizando computadoras y lenguajes computacionales como laboratorios para realizar experimentos. En las ciencias sociales, las hipótesis se expresan normalmente en forma textual, usando lenguajes naturales como el inglés o el español. Sin embargo, los lenguajes naturales suelen ser ambiguos y los conceptos no siempre están definidos con rigor. Por eso, cuando tenemos una teoría de cómo se comportan los individuos en la situación que estamos analizando, es útil expresarla en for-

ma de procedimiento o lenguaje formal o artificial, utilizando la lógica o las matemáticas. Una ventaja de utilizar la lógica es que proporciona condiciones de validez. Estos resultados se mantienen cuando y si lo siguiente es cierto. Así, los investigadores pueden utilizar reglas condicionales como “si C1, C2 y C3, entonces EP”, donde C representa alguna condición y EP es la propiedad macro o emergente que quieren explicar. Este tipo de formalización facilita el objetivo final del MSBA, esto es, formalizar hipótesis teóricas en forma de un programa informático.

1.5.2 Modelado, verificación, experimentación

Una vez que la teoría ha sido especificada y formalizada en lógica o matemáticas, los modeladores pueden traducir esta formalización en programas de computadora. De este modo, el modelo formal puede programarse y ejecutarse en la computadora, y el comportamiento de la simulación puede observarse y probarse. Construir un modelo es similar a diseñar un experimento. Por tanto, dado el *explanandum* macroscópico —una regularidad que debe ser explicada—, el experimento canónico basado en agentes es el mostrado en la figura 1.5. (Salgado y Gilbert, 2013). En este sentido, el modelado computacional basado en agentes es una nueva herramienta para la investigación empírica, el cual ofrece un entorno natural para el estudio de los fenómenos conexionistas en las ciencias sociales.

Un aspecto importante para tener en cuenta es que, al escribir programas informáticos, especialmente los complicados, es muy común cometer errores. El proceso de comprobar que un programa hace lo que se planeó hacer se conoce como verificación. En el caso de los modelos basados en agentes, las dificultades de verificación se ven agravadas por el hecho de que muchas simulaciones incluyen generadores de números aleatorios, lo que significa que cada ejecución es diferente y que la teoría solo puede anticipar la distribución de los resultados. Por tanto, es esencial “depurar” la simulación con mucho cuidado, de ser posible utilizando un conjunto de casos de prueba, tal vez de situaciones extremas donde los resultados sean fácilmente predecibles y ejecutar múltiples experimentos para explorar y medir el espacio de parámetros.

Cuando el modelo basado en agentes puede generar el tipo de resultado que se va a explicar, entonces el investigador ha proporcionado

do una demostración computacional de que una microespecificación (o mecanismo) dada es de hecho suficiente para generar la macroestructura de interés. Esta demostración, denominada suficiencia generativa (Epstein, 1999), presenta características que distinguen el enfoque tanto de las ciencias inductivas y deductivas. Es lo que el mismo Epstein llama ciencia social generativa. El modelado basado en agentes puede entonces utilizar datos relevantes y estadísticas para estimar la suficiencia generativa de una microespecificación determinada probando la concordancia entre el “mundo real” y las macroestructuras generadas en la simulación por computadora. Por otro lado, cuando el modelo basado en agentes no puede generar el resultado que se debe explicar, la microespecificación no es una posible explicación del fenómeno y el investigador ha demostrado que el mecanismo hipotético es falso.

1.5.3. Calibración y validación

Una vez que el investigador ha especificado un modelo basado en agentes sustancialmente plausible, y este modelo genera el patrón macro emergente de interés, se pueden usar datos empíricos para estimar el tamaño de varios parámetros desconocidos de este modelo y la concordancia entre los datos predichos y reales. Esto se logra calibrando y validando el modelo.

Mientras que la verificación se refiere a si el programa está funcionando como espera el investigador (como se analizó en la subsección anterior), la validación se refiere a si la simulación es un buen modelo del sistema real, esto es, el “objetivo”. Un modelo en el que se puede confiar para reflejar el comportamiento del objetivo es “válido”. Una forma común de validar un modelo consiste en comparar el resultado de la simulación con datos reales recopilados sobre el objetivo. Sin embargo, hay varias advertencias que deben tenerse en cuenta al realizar esta comparación. Por ejemplo, no se debe esperar una correspondencia exacta entre los datos reales y los simulados. Entonces el investigador debe decidir si la diferencia entre los dos tipos de datos es aceptable para que el modelo se considere válido. Esto generalmente se hace mediante algún análisis estadístico para probar la importancia de la diferencia. Si bien la bondad de ajuste siempre se puede mejorar agregando más factores explicativos, existe un equilibrio entre bondad de ajuste y parsimonia.

Demasiados ajustes pueden resultar en una reducción del poder explicativo porque el modelo se vuelve difícil de interpretar. En el extremo, si un modelo se vuelve tan complicado como en el mundo real, será igual de difícil de interpretar y no ofrecerá ningún poder explicativo. Se produce aquí, por tanto, una paradoja para la que no existe una solución obvia. A pesar de su naturaleza en apariencia científica, la modelación es una cuestión de criterio.

Por último, es importante distinguir las diferentes formas en que se puede validar y calibrar un modelo basado en agentes. Según Bianchi *et al.* (2007), existen tres formas de validar un modelo basado en agentes, a saber:

- Validación descriptiva de la salida, o comparación de la salida generada computacionalmente con datos ya disponibles. Este tipo de procedimiento de validación es probablemente el más intuitivo y representa un paso fundamental hacia el desarrollo de un buen modelo.
- Validación de resultados predictivos o comparación de datos generados por computadora con datos del sistema aún por adquirir. Como es evidente, el principal problema de este procedimiento se debe esencialmente al retraso entre los resultados de la simulación y la comparación final con los datos reales.
- Validación de los *inputs*, o asegurar que las condiciones iniciales estructurales, conductuales e institucionales fundamentales incorporadas en el modelo reproduzcan los principales aspectos del sistema real.

Para saber más sobre la validación empírica en el MSBA, se recomienda revisar la siguiente literatura: Axtell *et al.* (1996); Bianchi *et al.* (2007); Fagiolo *et al.* (2007); Kleindorfer *et al.* (1998).

CAPÍTULO 2. INTRODUCCIÓN A PYTHON

2.1. INTRODUCCIÓN

Una de las dificultades que enfrenta el investigador social al tratar de aplicar el modelado y simulación basados en agentes en sus proyectos de investigación es la necesidad de contar con habilidades de programación en algún lenguaje computacional capaz de desarrollar el modelado basado en agentes. Desde hace ya muchos años, Wilensky (1999) desarrolló un lenguaje orientado al modelado basado en agentes, llamado Netlogo. Sin embargo, dicho lenguaje no es muy eficiente en computación ya que es un lenguaje basado en Java y además se necesita aprender muchos conceptos difíciles de entender para poder usar de manera experta el lenguaje Netlogo. Por otro lado, hemos visto que a los investigadores que tenemos nociones de programación como en C y C++ nos es difícil adecuarnos a la lógica de Netlogo.

Este capítulo tiene como objetivo introducir al lector no entrenado en el lenguaje de programación a Python, un lenguaje de programación contemporáneo de fácil aprendizaje.

2.2. ¿QUÉ ES PYTHON?

Python es un lenguaje de programación desarrollado por Guido van Rossum, un programador de origen holandés, a finales de los ochenta y principios de los noventa, cuando se encontraba trabajando en el sistema operativo Amoeba (González-Duque, 2011: 7). El nombre del lenguaje está inspirado en el popular grupo de cómicos ingleses Monty Python de los años sesenta y setenta. Se trata de un lenguaje de *script*, de tipado

dinámico, multiplataforma y orientado a objetos. Veamos a continuación y con profundidad qué significan estos conceptos.

El tipado dinámico es una característica de Python que permite asignar valores a variables sin necesidad de declarar explícitamente su tipo. Durante la ejecución del código, el intérprete o compilador de Python determina el tipo de cada variable según el valor que se le asigna. Esto significa que una variable puede contener un entero, un número de punto flotante, un carácter o una cadena de texto, entre otros tipos, sin requerir una especificación previa. Por otro lado, Python es un lenguaje fuertemente tipado, lo que significa que no permite tratar una variable como si fuera de un tipo distinto al que realmente tiene. Es necesario realizar conversiones explícitas si se desea trabajar con un tipo de valor diferente.

Un lenguaje interpretado o de *script* es aquel que se ejecuta utilizando un intérprete en lugar de compilarse a lenguaje máquina y ejecutarse directamente en una computadora, a diferencia de los lenguajes compilados. El intérprete de Python lee el código fuente directamente y lo ejecuta línea por línea. Si el código del programa tiene un error, la ejecución se detiene. Con frecuencia, Python es visto como un lenguaje de *script* debido a su idoneidad para la creación de *scripts* y pequeñas aplicaciones de líneas de comandos, las cuales son utilizadas para automatizar tareas repetitivas y procesar datos.

Python es un lenguaje multiplataforma, lo que significa que el intérprete está disponible para varias plataformas, como Linux, Windows y MacOS. Esto posibilita que los programas desarrollados en Python sean compatibles con diversas plataformas sin necesidad de realizar modificaciones importantes.

Otra característica importante es que Python es un lenguaje orientado a objetos. La orientación a objetos es un paradigma² de la progra-

² Un paradigma en ciencias computacionales se refiere a un enfoque o estilo fundamental de programación que proporciona la base para la construcción y estructuración de *software*. Los paradigmas guían cómo se resuelven los problemas y se escriben los programas. En la computación, los paradigmas más conocidos incluyen:

1. Imperativo:

- Concepto: Programación basada en comandos.

- Ejemplo: Lenguajes como C, donde se especifican secuencias de instrucciones que ejecutar.

2. Declarativo:

- Concepto: Programación basada en la descripción del problema.

mación, es decir, como un patrón o modelo de programación que nos guía sobre cómo podemos trabajar con base en ello. Permite que se estructure un programa en clases y objetos, lo que ayuda a reutilizar el código y la creación de estancias individuales para trabajar con datos y funciones específicas.

2.3. ¿POR QUÉ PYTHON?

¿Cuál es la razón para aprender Python entre todos los lenguajes de programación disponibles? Python tiene varias ventajas que lo hacen muy atractivo para actividades científicas y para aquellos que quieren ingresar al mundo de la programación y aprender. Python nos ofrece varias ventajas interesantes, como:

- *Lenguaje expresivo*: Con esta característica de lenguaje expresivo, nos referimos a que Python es muy compacto, suele ser más corto que su equivalente en lenguajes como C. En el siguiente ejemplo, se pueden ver las diferencias del ¡Hola mundo! en Python a la izquierda y en C a la derecha.

```
print ("¡Hola, Mundo!")           #include <stdio.h>
```

- Ejemplo: SQL, donde se declara qué datos se desean obtener, no cómo obtenerlos.

3. Orientado a objetos (OO):

- Concepto: Basado en "objetos" que son instancias de clases.

- Ejemplo: Java, C++, donde los objetos encapsulan datos y comportamientos.

4. Funcional:

- Concepto: Programación con funciones matemáticas.

- Ejemplo: Haskell, donde las funciones son ciudadanos de primera clase y no hay estado mutable.

5. Lógico:

- Concepto: Programación basada en la lógica formal.

- Ejemplo: Prolog, donde se definen hechos y reglas para la deducción lógica.

6. Procedural:

- Concepto: Subconjunto del paradigma imperativo centrado en procedimientos o rutinas.

- Ejemplo: C, donde el código se estructura en funciones o procedimientos.

Cada paradigma tiene sus propias reglas y maneras de abordar los problemas, y muchos lenguajes de programación modernos combinan aspectos de varios paradigmas para proporcionar flexibilidad a los programadores. Los paradigmas también pueden influir en cómo se piensa sobre la estructura de datos y el flujo de control en los programas.

```
int main(void) {  
    printf (“¡Hola, Mundo! \n”).  
    return 0;  
}
```

- *Python es muy legible:* La sintaxis de Python es muy legible para el usuario y le permite el desarrollo de programas cuya comprensión resulta más sencilla a comparación con otros lenguajes de programación.
- *Usos de diversos paradigmas de programación:* Python puede usarse como lenguaje imperativo, procedimental o como lenguaje orientado a objetos.

2.4. INSTALACIÓN DE PYTHON

Si ya se tienen instaladas versiones anteriores de Python en su computadora, no es necesario desinstalarlas para instalar una nueva versión. Puede tener múltiples versiones de Python instaladas en su sistema, sin problemas.

Existen varias aplicaciones de Python, siendo CPython la de referencia y la más utilizada. Sin embargo, también existen otras, como PyPy, Jython, IronPython, entre otras.

La instalación de Python es relativamente sencilla. Puede descargar la versión correspondiente a su sistema operativo desde el sitio *web* oficial de Python en el siguiente enlace: <https://www.python.org/downloads/>. En esa página encontrará los paquetes de instalación para diferentes sistemas operativos. En este caso, los pasos que se describirán a continuación son para la instalación en el sistema operativo Windows. Simplemente elija la versión adecuada y siga las instrucciones de instalación proporcionadas.



FIGURA 2.1. DESCARGA DE PYTHON.

Una vez finalizada la descarga, se ejecutan los paquetes de instalación como administrador. Haga clic en “Install now” y marque las dos casillas que se encuentran en la parte inferior de la ventana. Asegúrese de seleccionar la opción “Add Python to PATH” para poder ejecutarlo desde la línea de comandos. Luego, haga clic en “Continuar”.



FIGURA 2.2. INSTALACIÓN DE PYTHON.

Espera a que finalice la instalación, sin interrumpir el proceso.

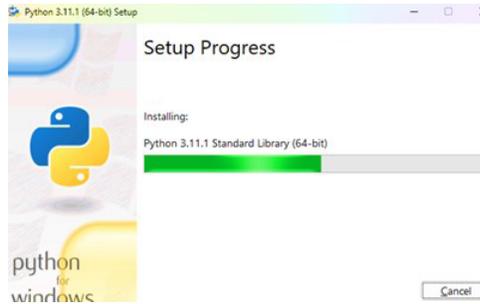


FIGURA 2.3. EN PROCESO DE INSTALACIÓN.

Después de que aparezca la ventana que confirma que el proceso ha concluido, verifique que la instalación se haya realizado correctamente. Ingrese al símbolo del sistema presionando las teclas Windows + R. Cuando aparezca el cuadro de diálogo de ejecución, escriba “cmd” y presione “Enter”.

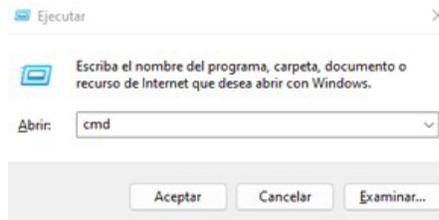


FIGURA 2.4. CUADRO DE DIÁLOGO “EJECUTAR”.

Una vez dentro del símbolo del sistema, escriba la palabra “python”. Puede observar en la figura que se muestra la versión de Python que acaba de instalar, junto con otras características. En caso de que aparezca un mensaje de error, es recomendable reiniciar el sistema para que los cambios surtan efecto.

```
C:\WINDOWS\system32\CMD.exe - python
Microsoft Windows [Versión 10.0.22000.1936]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\GLORIA.ALFERES>python
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> █
```

FIGURA 2.5. SÍMBOLO DEL SISTEMA CON COMANDO PYTHON.

Al tener finalizada la instalación de Python, puede ver algunos programas y herramientas asociados. Estos programas proporcionan funcionalidades fundamentales para el lenguaje de Python.

IDLE (Python 3.11) y (3.11 64-bit):

Este es el entorno de desarrollo predeterminado que viene con Python 3.11. Se hablará más detalladamente sobre esto en otra sección. En resumen, proporciona una interfaz gráfica y un editor de texto donde se puede escribir, ejecutar y depurar código en Python. En cuanto a IDLE de 64 bits, es una versión específica diseñada para la instalación de Python en sistemas operativos de 64 bits. Ambos son prácticamente idénticos; la única diferencia radica en la compatibilidad con la arquitectura del sistema operativo.

Python 3.11:

Es la versión principal de Python que se instala. Al abrir la línea de comandos, se inicia el intérprete donde se pueden ejecutar programas y comandos de Python directamente en esa terminal.

Python 3.11 Module Docs:

Este enlace lleva a la documentación en línea de Python 3.11. Proporciona documentación detallada y exhaustiva sobre los módulos y paquetes disponibles en esa versión de Python. Esta documentación es de gran utilidad para comprender y utilizar las diversas características de Python.

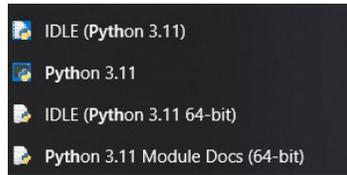


FIGURA 2.6. PROGRAMAS Y HERRAMIENTAS ASOCIADOS A PYTHON.

Tenga en cuenta que la instalación puede variar ligeramente según el sistema operativo; aunque los pasos generales de instalación son muy similares, puede haber algunas diferencias en los comandos utilizados. Los siguientes son los pasos a grandes rasgos para instalar Python en Linux y MacOS.

Linux

1. Descargar la última versión de Python desde el sitio oficial de Python (<https://www.python.org/downloads/>).
2. Una vez descargado el instalador, ejecutar los paquetes de instalación y continuar con los pasos que se indican.
3. En la ventana de componentes, marca las casillas que desee instalar. Por lo general, ya están seleccionados los componentes necesarios.
4. El proceso de instalación tardará unos minutos.
5. Cuando haya finalizado, aparecerá una ventana que informará que la instalación se ha completado.
6. Una vez instalado, abra una terminal en Linux y verifique si Python está instalado en su sistema operativo ingresando el siguiente comando:

```
python -version
```

Lo cual mostrará la versión de Python que está instalada.

MacOS

1. Descargar la última versión de Python desde el sitio oficial de Python (<https://www.python.org/downloads/>).
2. Ejecutar el instalador descargado y seguir las instrucciones que se presenten.
3. Durante el proceso de instalación, asegúrese de marcar la casilla “Agregar Python al PATH” para que Python sea reconocido desde la línea de comandos.
4. Una vez completada la instalación, puede verificar que Python se haya instalado correctamente ejecutando el siguiente comando en la terminal:

```
python3 --version
```

Esto mostrará la versión de Python que está instalada.

2.5. INTRODUCCIÓN A IDLE, ENTORNO DE DESARROLLO DE PYTHON

2.5.1. ¿Qué es un IDE?

Para introducir el tema del IDLE, que es el entorno de desarrollo que Python ofrece dentro de sus paquetes de instalación, es importante comprender qué es un IDE y qué significa. El término IDE se refiere a un “entorno de desarrollo integrado” (*integrated development environment*, en inglés), que es un sistema creado específicamente para el desarrollo y diseño de *software*. IDLE es un IDE específico para Python que viene con CPython. Un IDE combina múltiples herramientas comunes utilizadas en el proceso de desarrollo en una única interfaz gráfica de usuario, lo cual facilita y agiliza la creación de aplicaciones y programas. Por lo general, un IDE cuenta con las siguientes características:

- *Editor de código fuente*: El IDE nos proporciona un editor de código fuente que nos ayuda a escribir el código de *software*. Este editor in-

cluye funciones como autoindentación, resaltado de la sintaxis y sugerencias de autocompletado, entre otras.

- *Automatización de las compilaciones locales*: Las herramientas de automatización de compilaciones locales son utilidades que simplifican tareas básicas relacionadas con el proceso de compilación del código fuente en un entorno de desarrollo local. Aunque el IDLE de Python no ofrece una amplia gama de funcionalidades en comparación con otros IDE más completos, sí proporciona una forma sencilla de ejecutar y probar código en Python.
- *Depurador*: Un IDE nos ofrece un depurador, el cual se encarga de rastrear errores en el código y mostrar la ubicación de un error de forma gráfica en el código original.

Es importante destacar que las características básicas de un IDE pueden variar según el IDE específico y las necesidades del desarrollo de *software*. En comparación con IDE más populares y completos como Visual Studio o PyCharm, IDLE es considerado más básico. La elección del entorno de desarrollo dependerá de las necesidades y preferencias del usuario.

En el caso de fines educativos, el uso del IDLE es adecuado. Sin embargo, si el IDLE no resulta completamente satisfactorio, existe la opción de instalar otro IDE que se ajuste mejor a las necesidades individuales. Es importante tener en cuenta que cada IDE tiene sus propias características y funcionalidades, por lo que es recomendable explorar diferentes opciones y seleccionar aquella que brinde las herramientas y comodidades deseadas para el desarrollo de *software*.

2.6. RECORRIDO BÁSICO POR LAS HERRAMIENTAS ESENCIALES DE IDLE

Como se ha mencionado antes, IDLE es el entorno de desarrollo integrado de Python. IDLE es un acrónimo de “Integrated development and learning environment” (entorno integrado de desarrollo y aprendizaje). Este IDE nos ofrece varias características y herramientas útiles, como un editor de código fuente, automatización de compilaciones y un depura-

dor. A continuación, daremos un vistazo a las principales características de este entorno y cómo pueden ser utilizadas.

Al abrir el IDLE de Python con la ruta de acceso Inicio > Python 3.11 > IDLE (Python 3.11 64-bit), se muestra la ventana principal de IDLE, como se puede observar en la figura 2.7. En dicha ventana se indica la versión de Python instalada en el sistema.

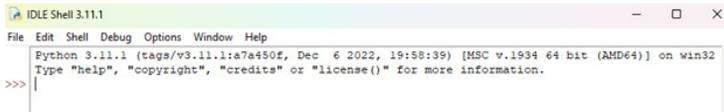


FIGURA 2.7. VENTANA PRINCIPAL DE IDLE.

La ventana principal del IDLE tiene un tamaño predeterminado, pero se puede modificar ingresando a Options > Configure IDLE > Windows. Desde allí, puede ajustar el tamaño de la ventana según sus preferencias.

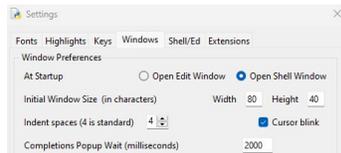


FIGURA 2.8. MODIFICAR TAMAÑO DE LA VENTANA.

El IDLE cuenta con una ventana principal que funciona como un entorno interactivo de Python. En esta ventana, puede escribir comandos de Python a la derecha del símbolo de petición “>>>” (conocido como *prompt* en inglés). Al presionar “Enter”, el IDLE ejecutará el comando de inmediato. Si el comando tiene un resultado, se mostrará en color azul y sin el símbolo de petición.

Después de ejecutar un comando en IDLE, el programa vuelve a mostrar el símbolo de petición, listo para recibir una nueva orden. Dentro del entorno interactivo de IDLE, puede recuperar comandos anteriores utilizando los siguientes atajos de teclado:

Alt+p para ver el comando anterior (puede interpretarse como “previous”, o *anterior* en inglés).

Alt+n para ver el comando siguiente (puede interpretarse como “next”, o *siguiente* en inglés).

IDLE nos brinda la posibilidad de escribir programas, guardarlos y ejecutarlos. La ventana principal del IDLE es tanto el entorno interactivo como el lugar donde se ejecutan los programas.

Para comenzar a crear un archivo de programa con IDLE, simplemente abra una nueva ventana utilizando el menú File > New File (o presione Ctrl+N como atajo de teclado).

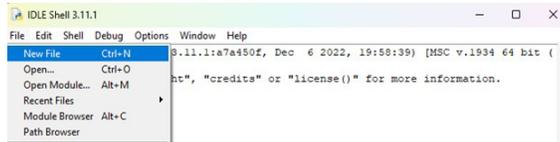


FIGURA 2.9. CREAR UN ARCHIVO DE PROGRAMA.

Al seleccionar esta opción, se abrirá una ventana adicional que se asemeja a la mostrada en la figura 2.10. En esta ventana, el símbolo de solicitud ya no está presente, ya que se trata simplemente de un editor de texto que resalta la sintaxis del código de Python. Debido a esto, los menús disponibles en esta ventana difieren de los de la ventana principal del IDLE.



FIGURA 2.10. EDITOR DE TEXTO DE PYTHON.

Después de escribir el código, si desea guardar el programa, puede seleccionar la opción correspondiente en el menú File > Save o File > Save As ... (también puede utilizar el atajo de teclado Ctrl+S). Al guardar por

primera vez un programa, se abrirá la ventana de diálogo estándar de Windows. Desde allí, puede elegir la carpeta y asignar un nombre al archivo. Es importante recordar que la extensión común para los programas de Python es .py. Si olvida escribir la extensión, el propio IDLE la agregará automáticamente.

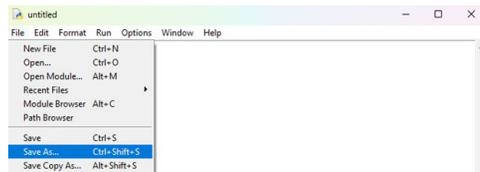


FIGURA 2.1.1. GUARDAR ARCHIVO.

Antes de ejecutar un programa editado en IDLE, es necesario guardar los cambios realizados. Una vez que el programa se haya guardado, se puede ejecutar seleccionando la opción “Run > Run module” en el menú (también se puede utilizar la tecla F5 como atajo).

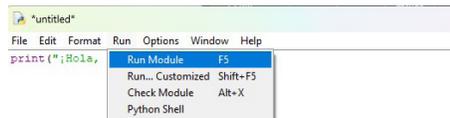


FIGURA 2.1.2. EJECUTAR CÓDIGO.

Los resultados del programa se visualizarán en la ventana principal del IDLE.

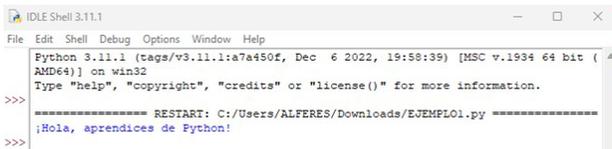


FIGURA 2.1.3. SALIDA DEL PROGRAMA.

También es posible ejecutar programas previamente creados en IDLE simplemente abriéndolos. Para hacerlo, basta con entrar al menú “File > Open” en IDLE y seleccionar el archivo que desea ejecutar.



FIGURA 2.14. ABRIR NUEVO ARCHIVO.

Después de abrir el programa en IDLE, puede ejecutarlo de dos formas: presionando la tecla F5 o seleccionando la opción “Run” en el menú y luego “Run module”.

Las indicaciones o características mencionadas antes son solo una parte de las funciones fundamentales que ofrece el IDLE de Python. Con estas herramientas, podrá ejecutar fácilmente un código sencillo y practicar los ejercicios que se presentarán más adelante, lo que le permitirá dominar los conceptos básicos del lenguaje de programación de Python.

2.7. JUPYTER NOTEBOOK

Jupyter Notebook es una aplicación *web* de código abierto que permite crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto narrativo. Es una herramienta muy útil para el análisis de datos, la visualización y la documentación de procesos y resultados. A continuación se describen sus características principales:

2.7.1. Características de Jupyter Notebook

a. Celdas de código:

- Permiten escribir y ejecutar código en lenguajes como Python, R, Julia, entre otros.

- Los resultados de la ejecución del código se muestran inmediatamente debajo de la celda, lo que facilita la comprobación y modificación del código.

b. Celdas de texto (Markdown):

- Se utilizan para escribir texto en formato Markdown,³ que puede incluir enlaces, imágenes, ecuaciones en LaTeX⁴ y formato enriquecido.
- Es ideal para documentar el código, describir métodos y resultados, y crear informes reproducibles.

c. Visualizaciones:

- Soporta bibliotecas de visualización como *Matplotlib*, *Seaborn* y *Plotly*.
- Permite crear gráficos y visualizaciones interactivas que se muestran dentro del *notebook*.

d. Integración con herramientas de análisis de datos:

- Compatible con bibliotecas como Pandas, NumPy y SciPy, lo cual facilita el análisis y manipulación de datos.

³ Markdown es un lenguaje de marcado ligero que se utiliza por lo regular para formatear texto en documentos, especialmente en entornos en línea como sitios *web*, foros, *blogs* y, en el contexto de Jupyter Notebook, en celdas de texto Markdown. Markdown fue creado por John Gruber y Aaron Swartz en 2004, y se ha vuelto ampliamente adoptado debido a su simplicidad y facilidad de uso. Markdown se destaca por su capacidad para convertir texto sin formato en contenido bien formateado mediante el uso de una sintaxis simple y legible.

⁴ LaTeX es un sistema de composición de documentos ampliamente utilizado en campos académicos y científicos para crear documentos con una calidad tipográfica excepcional, en especial cuando se trata de documentos que contienen fórmulas matemáticas, ecuaciones, referencias cruzadas y bibliografías. LaTeX se basa en el sistema TeX desarrollado por Donald Knuth en la década de 1970 y ha evolucionado para convertirse en una herramienta estándar en la producción de documentos académicos y técnicos, sobre todo en las ciencias naturales y las ingenierías que requieren el uso extendido de matemáticas.

- Posibilidad de realizar análisis exploratorio y procesamiento de datos en el mismo entorno.

e. Interactividad:

- Permite la creación de *widgets* interactivos utilizando bibliotecas como *ipywidgets*, lo que facilita la creación de aplicaciones *web* interactivas dentro del *notebook*.

2.7.2. Ejemplo de uso

A continuación se dará un pequeño ejemplo que muestra cómo se puede usar Jupyter Notebook para analizar datos. Lo primero que hay que hacer es instalar Jupyter Notebook.

Ahora es momento de abrir Jupyter Notebook. Esto se puede hacer desde el símbolo de sistema en Windows, de la siguiente manera:

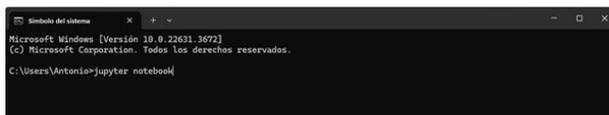


FIGURA 2.15. APERTURA DE JUPYTER NOTEBOOK DESDE EL SÍMBOLO DE SISTEMA EN WINDOWS.

Esta acción abre en su navegador predeterminado una ventana donde se desplegará Jupyter Notebook. En la figura 2.16 se muestra la ventana que se abre en el navegador Mozilla Firefox.

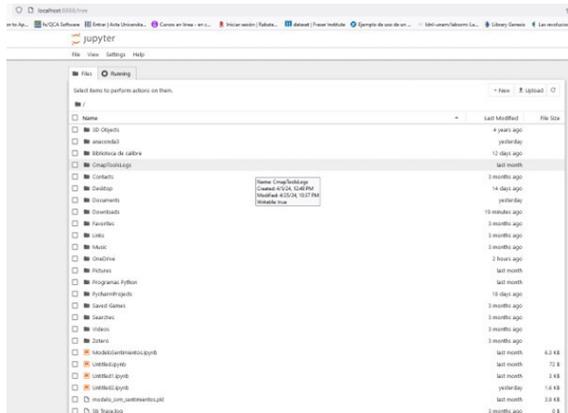


FIGURA 2.16. VENTANA DE JUPYTER NOTEBOOK EN EL NAVEGADOR FIREFOX.

Ahora debemos crear un nuevo *notebook*; para ello, nos vamos a la sección *file* y ahí le indicamos crear un nuevo *notebook*.

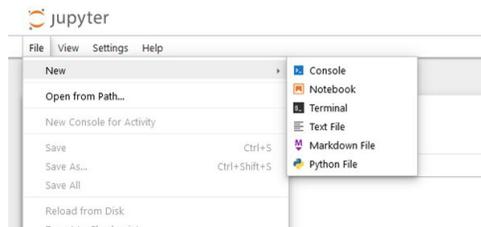


FIGURA 2.17. CREACIÓN DE UN NUEVO *NOTEBOOK*.

Al crear el *notebook* se desplegará la siguiente ventana, donde debemos especificar la versión de Python que se usará (figura 2.18). Esto se llama establecer el Kernel de Python.

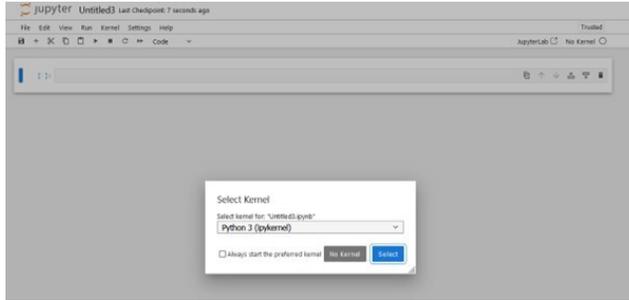


FIGURA 2.18. ESTABLECIMIENTO DEL KERNEL DE PYTHON.

Una vez establecido el Kernel de Python –en nuestro caso, Python 3–, se creará un nuevo *notebook* y se mostrará una ventana como la de la figura 2.19.

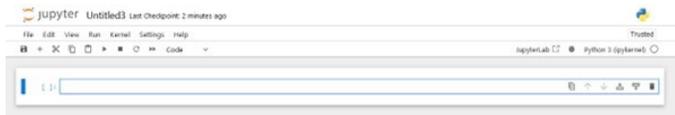


FIGURA 2.19. EL NUEVO *NOTEBOOK* CREADO.

Ahora procederemos a crear una base de datos con Python. Se muestra el código para ello en el *notebook* de Jupyter.

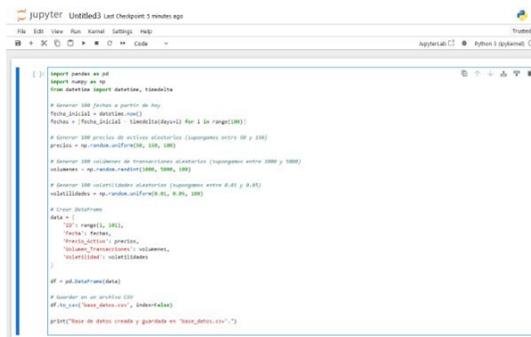
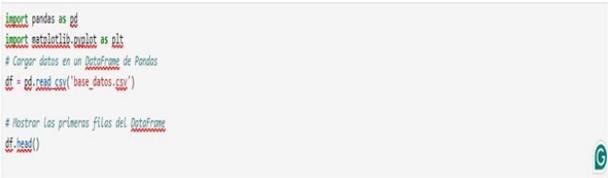


FIGURA 2.20. EL CÓDIGO PARA CREAR UNA BASE DE DATOS EN EL *NOTEBOOK* DE JUPYTER.

El código es el siguiente:

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
# Generar 100 fechas a partir de hoy
fecha_inicial = datetime.now()
fechas = [fecha_inicial - timedelta(days=i) for i in range(100)]
# Generar 100 precios de activos aleatorios (supongamos entre 50 y
150)
precios = np.random.uniform(50, 150, 100)
# Generar 100 volúmenes de transacciones aleatorios (supongamos
entre 1000 y 5000)
volumenes = np.random.randint(1000, 5000, 100)
# Generar 100 volatilidades aleatorias (supongamos entre 0.01 y 0.05)
volatilidades = np.random.uniform(0.01, 0.05, 100)
# Crear DataFrame
data = {
    'ID': range(1, 101),
    'Fecha': fechas,
    'Precio_Activo': precios,
    'Volumen_Transacciones': volumenes,
    'Volatilidad': volatilidades
}
df = pd.DataFrame(data)
# Guardar en un archivo CSV
df.to_csv('base_datos.csv', index=False)
print("Base de datos creada y guardada en 'base_datos.csv'.")
```

Nótese que si usted quiere correr este código debe tener instalado en su computadora las bibliotecas Pandas y Numpy; si no es así, el programa le marcará error. Ahora procederemos a leer en Python la base de datos con ayuda de pandas. El código para esto se puede ver en la figura 2.21.



```
import pandas as pd
import matplotlib.pyplot as plt
# Cargar datos en un DataFrame de Pandas
df = pd.read_csv('base_datos.csv')

# Mostrar las primeras filas del DataFrame
df.head()
```

FIGURA 2.21. EL CÓDIGO PARA VISUALIZAR LA BASE DE DATOS CREADA.

El código para hacer esto es el siguiente:

```
import pandas as pd
import matplotlib.pyplot as plt
# Cargar datos en un DataFrame de Pandas
df = pd.read_csv('datos.csv')
# Mostrar las primeras filas del DataFrame
df.head()
```

La salida de este código se muestra en la figura 2.22.

	ID	Fecha	Precio_Activo	Volumen_Transacciones	Volatilidad
0	1	2024-06-01 12:20:52.785612	145.012280	4107	0.047442
1	2	2024-05-31 12:20:52.785612	59.630249	1098	0.031289
2	3	2024-05-30 12:20:52.785612	100.710405	1289	0.021004
3	4	2024-05-29 12:20:52.785612	148.637303	2117	0.013875
4	5	2024-05-28 12:20:52.785612	109.089272	1818	0.017388

FIGURA 2.22. *DATAFRAME* CON LOS DATOS CREADOS EN EL CÓDIGO ANTERIOR.

2.7.3. Ventajas de usar Jupyter Notebook

- *Reproducibilidad*: Facilita la reproducción de experimentos y análisis, ya que combina código, resultados y documentación en un solo documento.
- *Colaboración*: Permite a los investigadores y analistas compartir sus trabajos fácilmente, lo que fomenta la colaboración.
- *Educación*: Es ampliamente utilizado en la enseñanza de programación y análisis de datos, pues proporciona un entorno interactivo para el aprendizaje.

2.7.4. Desventajas

- *Desempeño*: Puede no ser adecuado para trabajos que requieren un rendimiento muy alto o manejo de grandes volúmenes de datos, debido a limitaciones en la interfaz y el entorno *web*. Para un trabajo con gran volumen de datos se recomienda el uso del IDE PyCharm⁵.

⁵ PyCharm es un entorno de desarrollo integrado (IDE) para programación en Python, desarrollado por JetBrains. Es una de las herramientas más populares y potentes para desarrollar

- Gestión de dependencias: Puede ser complejo gestionar diferentes versiones de bibliotecas y dependencias en entornos compartidos.

Jupyter Notebook se ha convertido en una herramienta esencial en ciencia de datos, análisis de datos y áreas relacionadas, ya que facilita la integración de código, análisis y documentación en un solo entorno accesible y colaborativo. Su gran ventaja es que es gratuito y de código abierto.

2.8. ANACONDA

Anaconda es una distribución de código abierto para la programación en Python y R que está diseñada para facilitar la gestión de paquetes y la aplicación de entornos. Es especialmente popular en el ámbito de la ciencia de datos y el análisis de datos debido a su facilidad de uso y su capacidad para gestionar dependencias complejas. A continuación, se describen sus características y usos principales:

aplicaciones en Python, y está diseñada para mejorar la productividad del desarrollador con una amplia gama de características y funcionalidades. Tiene características como: a) Editor de código avanzado con funciones como el autocompletado de código y el análisis de código con detección automática de errores y problemas en el código en tiempo real. Asimismo, permite la refactorización, que es una herramienta para reestructurar el código sin cambiar su comportamiento externo, lo cual facilita la limpieza y optimización del código; b) Depurador integrado que permite pausar la ejecución del programa para inspeccionar variables y el flujo de ejecución, y Pruebas unitarias, que es un soporte para marcos de pruebas como pytest, unittest, y doctest, que facilitan la escritura y ejecución de pruebas; c) Integración con sistemas de control de versiones, ya que soporta para Git, SVN, Mercurial, entre otros, permitiendo la gestión de versiones y colaboraciones en proyectos de manera integrada; d) Herramientas de desarrollo *web*, ya que da soporte para *frameworks web* como Django, Flask, y Pyramid y plantillas HTML, CSS y JavaScript integradas para desarrollo *web full-stack*; Soporte para bases de datos y SQL, ya que tiene un explorador de bases de datos integrado, que permite conectarse a bases de datos, ejecutar consultas y manejar esquemas de bases de datos; e) Soporte para ciencia de datos, ya que permite la integración con Jupyter Notebooks y bibliotecas de ciencia de datos como NumPy, pandas, matplotlib, y scipy. PyCharm es una excelente elección para desarrolladores de Python que buscan un entorno robusto y repleto de funcionalidades que les ayuden a escribir, depurar y mantener código de alta calidad de manera eficiente.

2.8.1. Características de Anaconda

1. Gestión de Paquetes y Entornos:

- Anaconda incluye *conda*, una herramienta de gestión de paquetes y entornos que permite instalar, actualizar y desinstalar paquetes y gestionar entornos virtuales de manera eficiente.
- Permite crear entornos aislados para diferentes proyectos, evitando conflictos de versiones entre paquetes.

2. Paquetes preinstalados:

- Viene con más de 1 500 paquetes preinstalados, incluyendo bibliotecas populares para ciencia de datos e inteligencia artificial como NumPy, Pandas, SciPy, Matplotlib y Scikit-learn.
- Facilita la instalación de paquetes adicionales desde el repositorio de *conda* y también desde PyPI (utilizando ``pip``).

3. Interfaz gráfica (Anaconda Navigator):

Anaconda Navigator es una interfaz gráfica que permite a los usuarios gestionar paquetes y entornos, y lanzar aplicaciones como Jupyter Notebook, Spyder, y otros IDE, sin necesidad de usar la línea de comandos.

4. Distribuciones Miniconda:

Miniconda es una versión más ligera de Anaconda que solo incluye *conda* y su dependencia Python, lo cual permite a los usuarios instalar solo los paquetes que necesitan.

2.8.2. Usos de Anaconda

2.8.2.1. Instalación y configuración de entornos

Para instalar Anaconda, se descarga el instalador adecuado para el sistema operativo (Windows, MacOS, Linux) desde <https://www.anaconda.com/products/distribution>. La instalación incluye *Python* y *conda*, junto con una serie de herramientas útiles.

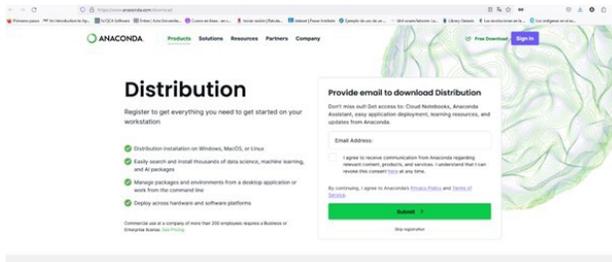


FIGURA 2.23. PÁGINA PARA DESCARGAR ANACONDA.

2.8.2.2 Creación y activación de entornos virtuales

Un entorno virtual es un entorno aislado que contiene una instalación específica de Python y paquetes. Esto es útil para mantener las dependencias separadas para diferentes proyectos. Aquí un ejemplo de cómo crear y gestionar entornos:

```

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows

PS C:\Users\Antonio> conda create --name mi_entorno python=3.8
  
```

FIGURA 2.24. CREACIÓN DE UN ENTORNO CON CONDA.

El código es el siguiente:

```
conda create --name mi_entorno python=3.8
```

Ahora activaremos el entorno.

```

Símbolo del sistema - C:\User x +
Microsoft Windows [Versión 10.0.22631.3672]
(C) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Antonio>conda activate mi_entorno

(mi_entorno) C:\Users\Antonio
  
```

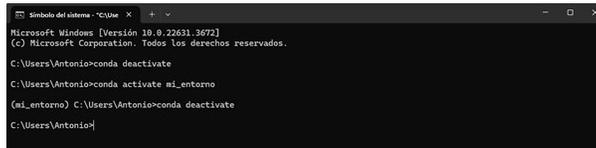
FIGURA 2.25. ACTIVACIÓN DEL ENTORNO.

Para activar el entorno, se usa el siguiente código en el símbolo de sistema.

```
conda activate mi_entorno
```

Para desactivar el entorno se usa el siguiente código (figura 2.26):

```
conda deactivate
```



```
Microsoft Windows [Versión 10.0.22631.3672]
(c) Microsoft Corporation. Todos los derechos reservados.
C:\Users\Antonio>conda deactivate
C:\Users\Antonio>conda activate mi_entorno
(mi_entorno) C:\Users\Antonio>conda deactivate
C:\Users\Antonio>
```

FIGURA 2.26. DESACTIVACIÓN DEL ENTORNO CREADO.

2.8.2.3. Gestión de paquetes

Con *conda* se pueden instalar, actualizar y eliminar paquetes fácilmente. Algunos comandos útiles son:

- Instalar paquetes: `conda install nombre_del_paquete`
 - Instalar un paquete de una versión específica:
`conda install nombre_del_paquete=versión`
 - Instalar varios paquetes:
`conda install paquete1 paquete2 paquete3`
 - Buscar un paquete:
`conda search nombre_del_paquete`
 - Actualizar un paquete a la última versión:
`conda update nombre_del_paquete`
 - Eliminar un paquete:
`conda remove nombre_del_paquete`
 - Crear un nuevo entorno e instalar paquetes en él:
`conda create --name nombre_del_entorno nombre_del_paquete`
 - Listar todos los entornos:
`conda env list`
 - Listar todos los paquetes instalados en el entorno actual:
`conda list`

- Clonar un entorno existente:

```
conda create --name nuevo_entorno --clone nombre_del_entorno_existente
```

2.8.2.4. Uso de Anaconda Navigator

Anaconda Navigator proporciona una interfaz gráfica para las operaciones mencionadas antes, además de permitir el lanzamiento de aplicaciones como Jupyter Notebook, Spyder, VS Code, RStudio, etc., sin necesidad de usar la línea de comandos.

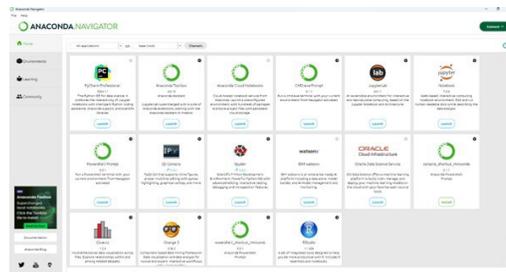


FIGURA 2.27. ANACONDA NAVIGATOR.

2.8.3. Ventajas de usar Anaconda

- **Facilidad de instalación:** Simplifica la instalación y gestión de paquetes y entornos.
- **Compatibilidad:** Reduce los problemas de compatibilidad y conflictos de versiones entre paquetes.
- **Entorno de desarrollo completo:** Proporciona un entorno de desarrollo completo y listo para usar, ideal para científicos de datos y desarrolladores.
- **Herramientas incluidas:** Incluye herramientas populares como Jupyter Notebook, Spyder y RStudio, facilitando el trabajo con datos y la programación en Python y R.

2.8.4. Desventajas

- **Tamaño de instalación:** La instalación completa de Anaconda puede ser grande, ocupando varios *gigabytes* de espacio en disco.
- **Consumo de recursos:** Algunos entornos y paquetes pueden consumir una cantidad considerable de recursos del sistema, lo que puede ser un problema en máquinas con recursos limitados.

CAPÍTULO 3.

ESTRUCTURA Y ELEMENTOS DEL LENGUAJE DE PYTHON

3.1. ELEMENTOS DEL LENGUAJE

Dentro de los lenguajes de programación informáticos, Python se clasifica como un lenguaje de alto nivel e interpretación. Al igual que la mayoría de los lenguajes de programación, Python se compone de un conjunto de elementos que conforman su estructura. A continuación, se verán algunos de estos elementos.

3.1.1. Variables

Una variable es un elemento fundamental en el desarrollo de un código. Es un espacio en la memoria principal de la computadora que se utiliza para almacenar datos. Cada variable tiene un identificador al cual se le asigna un nombre simbólico o significativo asociado a dicho espacio. En Python, la declaración de una variable se realiza utilizando la siguiente sintaxis:

```
nombre_de_la_variable = valor_de_la_variable
```

Es ampliamente recomendado utilizar nombres descriptivos y en minúsculas al declarar una variable. En el caso de nombres compuestos, se deben usar guiones bajos y debe haber un espacio antes y después del signo *igual*.

Otra forma de crear variables es mediante la asignación múltiple, lo cual es una ventaja, ya que permite declarar múltiples variables en una sola instrucción con sus respectivos valores. Por ejemplo:

```
var1, var2, var3 = 'string', 22, False
```

Existe un tipo de variable denominada *constante*, la cual se utiliza para definir valores fijos que no deben modificarse. Para las constantes, también se deben usar nombres descriptivos en mayúsculas, y en el caso de nombres compuestos, se deben separar con guiones bajos.

En ocasiones, se necesitará mostrar el contenido de una variable en pantalla para que otras personas puedan ver resultados o seguir instrucciones. Para imprimir el valor de una variable en la consola, Python utiliza la palabra reservada *print*. Por ejemplo:

```
print(var1)
```

3.1.2. Tipos de datos

Una variable o constante puede contener valores de tipos distintos; en Python, los distintos tipos básicos se dividen en:

- Números, pueden ser de tres tipos

```
Enteros: edad = 22
Flotantes o reales: número = 22.8
Complejos: número = 22 + 2j
```

- Cadenas de texto (*String*):

```
mi_cadena = "¡Hola, Mundo!"
cadena_multilinea = " " "
Esta es una cadena de
Varias líneas" " "
```

- Valores *booleanos*:

```
verdadero = True
falso = false
```

Como habrá notado, en Python, a diferencia de muchos otros lenguajes, no se declara el tipo de variable al crearla; por ejemplo, en C tendría que escribirse:

```
#include <stdio.h>
void main()
{
char mi_cadena [20] = “¡Hola, Mundo!”

}
```

3.1.3. Operadores aritméticos

Entre los operadores aritméticos que utiliza Python, se encuentran los siguientes:

<i>Símbolo</i>	<i>Significado</i>	<i>Ejemplo</i>	<i>Resultado</i>
+	Suma	d = 10 + 2	d es 12
-	Resta	d = 10 - 2	d es 10
-	Negación	d = -10	d es -10
*	Multiplicación	d = 10 * 2	d es 20
**	Exponente	d = 10 ** 2	d es 100
/	División	d = 11 /2	d es 5.5
//	División entera	d = 11//2	d es 5.0
%	Módulo	d = 11%2	d es 1

TABLA 3.1. OPERADORES ARITMÉTICOS

A continuación, puede ver un ejemplo con variables y operadores aritméticos de acuerdo con la operación:

```
# Declaración de variables
n1 = 22
n2 = 32
# SUMA/RESTA
suma= n1 + n2
print (suma)
resta = n1 - n2
print (resta)
#MULTIPLICACIÓN/EXPONENTE
multiplicación = n1 * n2
print (multiplicación)
exponente = n1 ** n2
print (exponente)
# DIVISION
division_normal = n1 / n2
print(division_normal)
```

```
division_sin_decimal = n1// n2 # Al poner dos slashes nos devuelve un
resultado entero
print(division_sin_decimal)
# MODULO residuo de la división
modulo = n1 % n2
print(modulo)
```

3.1.4. Comentarios

Los comentarios son anotaciones que los programadores incluyen en el código con el objetivo de facilitar su comprensión tanto para ellos mismos como para otros desarrolladores que puedan leerlo. Un archivo de código no solo contiene instrucciones, sino también comentarios que proporcionan explicaciones adicionales.

Existen dos tipos de comentarios: los de una sola línea y los de varias líneas.

```
# Este es el comentario de una sola línea
""" Este es un comentario
de varias líneas """
```

La diferencia entre los comentarios de una sola línea y los de varias líneas se puede distinguir fácilmente en Python. En una sola línea, utilizamos el símbolo `#` para indicar que el texto es un comentario. Por otro lado, cuando necesitamos hacer un comentario de varias líneas, utilizamos tres comillas consecutivas al principio y al final del bloque de texto.

En los comentarios, se pueden incluir palabras que nos ayuden a identificar el subtipo de comentario:

```
# TODO esto se necesita resolver
# FIXME importante corregir
```

3.1.5. Cadenas

Las cadenas son simplemente texto que se encuentra entre comillas simples o dobles. Dentro de esas comillas, podemos utilizar caracteres especiales que tienen un propósito específico. Por ejemplo, al usar `\n`, al

ejecutar el código en la consola, se mostrará un salto de línea, o `\n` para representar una tabulación. Por ejemplo:

```
print (“¡Hola, mundo! \n¿cómo están todos?”) #Código usando \n
¡Hola, mundo!          # Una vez compilado
¿Cómo están todos?
print (“¡Hola, mundo! \t¿cómo están todos?”) # Código usando \t
¡Hola, mundo!   ¿Como están todos?      # Una vez compilado
```

De igual manera, es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma, se puede escribir el texto en varias líneas. Al imprimir la cadena, se verá como si se hubiera usado un carácter especial `\n`, pero en realidad son los saltos de línea comunes que se han introducido en la cadena.

```
comillas_triples = '''Primer línea
esto se verá en otra línea
y así sucesivamente '''
```

Para concatenar cadenas, se puede utilizar el operador `+` y también `*`. Con el operador `+` se pueden unir dos cadenas, mientras que con el operador `*` se puede repetir una cadena múltiples veces según el número que se indique. Véase el siguiente ejemplo:

```
a = “uno”
b = “dos”
c = a + b # c es “unodos”
c = a * 3 # c es “unounouno”
```

3.1.6. Booleanos

Los *booleanos* solo nos pueden devolver dos valores: cierto o falso (*true* o *false*). Los diferentes tipos de operadores con los que podemos trabajar con valores *booleanos* se llaman operadores lógicos o condicionales. Más adelante, se verán en detalle las estructuras de control condicionales que permiten trabajar con este tipo de valores.

Para expresar una evaluación lógica sobre una condición, se utilizan operadores relacionales o de comparación.

<i>Símbolo</i>	<i>Significado</i>	<i>Ejemplo</i>	<i>Resultado</i>
==	Igual que	5 == 10	Falso
!=	Diferente de	5 != 10	Verdadero
<	Menor que	5 < 10	Verdadero
>	Mayor que	5 > 10	Falso
<=	Menor o igual que	5 <= 5	Verdadero
>=	Mayor o igual que	5 >= 10	Falso

TABLA 3.2. VALORES *BOOLEANOS*

Para evaluar más de una condición simultánea, se utilizan operadores lógicos (en la descripción a = True y b = False):

<i>Operador</i>	<i>Descripción</i>	<i>Ejemplo</i>
And	Se cumple a y b	c = a and b #c = False
Or	Se cumple a o b	c = a or b #c = True
Not	No se cumple a	c = Not a #c = False

TABLA 3.3. OPERADORES LÓGICOS

3.2. ENTRADA Y SALIDA DE DATOS

3.2.1. Entrada de datos

La entrada de datos en cualquier lenguaje de programación consiste en colocar en la memoria principal datos provenientes desde un dispositivo de entrada para que la computadora, de acuerdo con las instrucciones del código, realice una tarea en específico. En los ejemplos que se han visto hasta ahora siempre se asigna un valor a las variables; sin embargo, a la hora de hacer un programa, la mayoría de las veces, como programadores, no asignamos el valor, sino se lo pedimos al usuario. Pongamos un ejemplo:

```
''' Pedirle al usuario que digite su nombre, posteriormente imprimir el resultado'''
```

```
nombre = input («Digite su nombre:\t»)
print («Hola», nombre)
```

Como puede observarse en el ejemplo anterior, para poder obtener texto escrito por el teclado se usa una variable y se iguala a la función *input* (); dentro de los paréntesis se escriben las instrucciones que queremos se impriman en pantalla; al ejecutar el programa, se detiene el puntero y parpadea, de manera que está esperando a que el usuario escriba algo y una vez escrito pulse la tecla *Intro*. Posteriormente, imprime la salida. Con respecto a las salidas, más adelante veremos los ejemplos correspondientes.

Al usar la sintaxis para la entrada de texto no es necesario declarar que se trata de una cadena, ya que la función *input* guarda datos de tipo *String*. ¿Qué pasa entonces si queremos declarar un dato de tipo entero o flotante? Si usamos la sintaxis del ejemplo anterior, guardará el dato, pero será de tipo *string*; por tanto, necesitamos hacer lo siguiente:

```
número = int (input («Digite un número: \t»))
```

La sintaxis que seguimos es poner la palabra reservada *int* y encerrar el resto entre paréntesis. Lo que hace el ejemplo anterior es que el *input* guarda un valor de tipo cadena, pero con *int* se convertirá el valor a entero. Para ingresar números flotantes, es exactamente lo mismo, solo que se usa la palabra reservada *float*.

3.2.2. Salida de datos

La salida de datos consiste en mandar datos que son el resultado de algún programa que se está ejecutando desde la memoria principal hasta un dispositivo de salida. Veremos algunos ejemplos del código que podemos aplicar para la salida de datos por consola.

```
#Ejemplo No. 1
nombre = «Gloria»
edad = 22
print («Hola», nombre, «tienes», edad, «años») #la salida es: Hola
Gloria tienes 22 años
```

En el ejemplo señalado, como podrá verse, entre comillas dobles se encierran las cadenas de texto para imprimir y se separa por comas; se colocan las variables de las cuales se imprimirá el valor. En el último ejemplo mostrado suele ser un poco tedioso por la aplicación de tantas comillas, pero no hay de que preocuparse, pues veremos más opciones para la salida de datos. La siguiente forma es recurriendo a la función *format*; en ella, dentro de las comillas dobles se pone el texto que se desea imprimir, seguido de llaves, y en la posición en la que se pongan las llaves se imprimirán los valores de las variables:

```
#Ejemplo No. 2
print («Hola {} tienes {} años». Format (nombre, edad)) #la salida
es: Hola Gloria tienes 22 años
```

Hay una tercera opción para imprimir datos por consola, la cual surgió a partir de la versión 3.6 de Python, donde se introdujo el *f-Strings*:

```
#Ejemplo No. 2
print (f «Hola {nombre} tienes {edad} años») #la salida es: Hola
Gloria tienes 22 años
```

En el último ejemplo, antes de poner las comillas simples para escribir el texto, se coloca la letra *f*, para posteriormente, dentro de las comillas, entre llaves, colocar los valores de las variables que se quieren mostrar.

3.3. EXPLORANDO LAS COLECCIONES EN PYTHON: LISTAS, TUPLAS, DICCIONARIOS Y CONJUNTOS

En la sección anterior se vieron algunos elementos básicos del lenguaje, como los tipos de datos, cadenas, valores *booleanos*, etc. En esta sección se verán cuatro tipos más complejos que admiten una colección de datos. Estos tipos son:

- Listas
- Tuplas

- Diccionarios
- Conjuntos

3.3.1. Listas

Las listas son un tipo de colección ordenada, se usan para almacenar conjuntos de elementos que pueden ser de cualquier tipo, como números, cadenas o *booleanos*; también puede haber listas dentro de las listas.

Para crear una lista, solo se necesita asignarle un nombre a la lista, colocar el signo igual e indicar entre corchetes, separados por comas, los valores que se incluyen:

```
lista = ['lunes' , 'martes' , 40 , 5.28 , [1 ,2 , 3] , True]
```

Se tiene acceso a los elementos de esta lista indicando el índice del elemento entre corchetes. Algo que se debe tomar en cuenta es que el primer índice de la lista siempre será 0 y no 1:

```
variable = lista [0] #variable es igual a Lunes
```

Si buscamos acceso a elemento de una lista incluida dentro de otra lista, se tendrá que utilizar dos veces el operador anterior, es decir, primero indicar la posición de la lista y luego la del elemento dentro de ella:

```
variable = lista [4][1] #variable es igual a 2
```

Con este operador también se puede modificar un elemento de una lista; solo se necesita poner la posición en la que sustituirá a otro valor:

```
lista [2] = 'miércoles' #lista ahora es igual a ['lunes' , 'martes' , 'miercoles' , 5.28 , [1 ,2 , 3] , True]
```

En Python se tiene acceso a un elemento de la lista si utilizamos un número negativo como índice, que significa que se empieza a contar de derecha a izquierda; como ejemplo, con [-1] llegamos al último elemento, con [-2] al penúltimo y así de manera sucesiva:

```
variable = lista [-1] #variable es igual a True
```

Otro aspecto interesante de Python es el *slicing*, una operación en la que se permite seleccionar partes específicas de una lista. En lugar de un número, escribimos uno de inicio y otro de fin, separados por dos puntos. Python lo interpretará como que el primer número es la posición de inicio y el segundo número los elementos que se van a tomar.

```
lista = ['lunes', 'martes', 40 , 5.28 , [1 ,2 , 3] , True]
variable = lista[0:3] #variable es igual a ['lunes' , 'martes' , 40 ]
```

Sí se escriben tres números (inicio: fin: salto), el tercero se usa para determinar cada cuál posición añadir un elemento a la lista.

```
variable = lista [0:5:2] # variable es igual a ['lunes', 40, [1 ,2 , 3]]
```

No es necesario indicar el principio y el final del *slicing*; si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista:

```
variable = lista[:] #variable es ['lunes', 'martes' , 40 , 5.28 , [1 ,2 , 3] , True]
variable = lista[1:] #variable es ['martes', 40 , 5.28 , [1 ,2 , 3] , True]
variable = lista[:4] #variable es ['lunes', 'martes' , 40 , 5.28 ]
variable = lista[::2] #variable es ['lunes', 40, [1 ,2 , 3]]
```

En la sección de variables se pudo observar la asignación múltiple de las variables, la cual también puede darse utilizando como valores el contenido de una lista:

```
lista = ['México', 'San Luis Potosí']
país, estado = lista
print(país) #Podemos imprimir por separado las variables
print(estado)
```

3.3.2. Tuplas

Las tuplas son muy parecidas a las listas; una de las diferencias es que la forma de definir las es diferente, ya que usan paréntesis en lugar de corchetes.

```
tupla = ("cadena de texto", 11, 25.2, True)
```

Se tiene acceso también a los elementos de la tupla mediante su índice, al igual que con las listas.

```
print(tupla [0]) #La salida es "cadena de texto"
```

Al igual que con las listas, se puede recurrir a la técnica del *slicing*, indicando opcionalmente desde el índice de inicio hasta el del fin:

```
variable = tupla[:] #variable es ("cadena de texto", 11, 25.2, True)
variable = tupla[1:] #variable es (11, 25.2, True)
variable = tupla[:3] #variable es ("cadena de texto", 11, 25.2)
variable = tupla[::2] #variable es ("cadena de texto", 25.2)
```

Otra diferencia muy importante que existe con las listas es que las tuplas son inmutables; sus valores no se pueden modificar una vez creadas. Tampoco poseen un mecanismo de modificación a través de funciones útiles como los métodos de las listas.

Una ventaja sobre las listas es que las tuplas son más ligeras y ahorran espacio de memoria, ya que su uso es más básico.

3.3.3. Dicionarios

Los diccionarios, también conocidos como matrices asociativas, reciben esta denominación porque son colecciones que relacionan una clave y un valor. A continuación, se verá un ejemplo de un diccionario básico de colores del español traducidos al inglés:

```
diccionario = {"azul": "blue", "rojo": "red", "verde": "green"}
```

El primer valor corresponde a la clave y el segundo es un valor asociado a ella. Tratándose de la clave, se puede utilizar cualquiera valor inmuta-

ble: se podrían usar números, cadenas, *booleanos*, tuplas, (pero NO listas o diccionarios, dado que sí son mutables). La condición anterior sucede porque los diccionarios se aplican como tablas *hash*. Una tabla *hash* es una estructura de datos asociada a llaves o claves de valores. La función principal que soporta de manera eficiente es la búsqueda, permite el acceso a los elementos almacenados a partir de una clave generada usando el *id*. Funciona transformando la clave con una función *hash* en un *hash*, un número que la tabla utiliza para localizar el valor deseado.

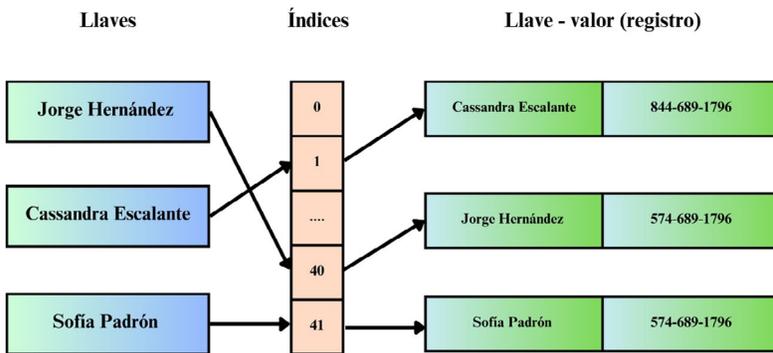


FIGURA 3.1. DIAGRAMA DE UNA FUNCIÓN *HASH*.

Mientras que a las listas y tuplas se ingresa por un número de índice, en los diccionarios no aplica ya que de hecho no tienen orden; los diccionarios permiten utilizar la clave para declarar y tomar un valor:

```
diccionario [azul] # nos devuelve "blue"
```

Al igual que en las listas, también se puede asignar valores o eliminarlos.

```
diccionario [amarillo] = "yellow" #se asigna al diccionario yellow
con la clave amarillo
del ( diccionario [azul] ) # eliminara la llave azul con su
correspondiente id
```

Sin embargo, en diccionarios no se puede hacer uso de *slicing* porque estos no son secuencias, sino mapeos o asociaciones.

3.3.4. Conjuntos

Python incluye un tipo de dato para conjuntos. Los conjuntos son grupos de elementos desordenados; su principal característica es que no puede haber duplicados. Los usos básicos de un conjunto incluyen verificación de pertenencia y eliminación de entradas duplicadas.

```
conjunto = {1, 2, 3, "Hola", True, 2.8, 1, 2, 3}
print (conjunto) #La salida es {1, 2, 3, 2.8, 'Hola', True}
```

Las llaves no pueden usarse para crear conjuntos vacíos; primero tenemos que usar la función `set ()` y después declaramos el conjunto vacío con las llaves `{}`, ya que Python lo interpreta como un diccionario vacío.

```
conjunto = set ()
conjunto = {}
```

Otra función importante de recalcar es la función `list()`, la cual nos ayuda a convertir un conjunto a una lista; por ejemplo:

```
lista = list(conjunto)
print(lista) #Salida []
```

Se puede llenar un conjunto con cualquier tipo de datos; se podrían usar números, cadenas, *booleanos*, pero no se puede hacer uso de ninguna colección⁶ para llenar un conjunto.

3.3.4.1. Unión de dos conjuntos en Python

La unión y la intersección de los conjuntos son operaciones consideradas dentro de la teoría de conjuntos. Con base en ellas se resuelven situaciones problemáticas que de no utilizarlas serían bastante complejas. La operación de la unión de dos conjuntos da como resultado un nuevo conjunto formado por todos los elementos que conforman ambos conjuntos. Se denota de la siguiente manera $A \cup B$:

⁶ Una colección permite agrupar varios objetos bajo un mismo nombre. En Python existen tres colecciones básicas: las listas, las tuplas y los diccionarios.

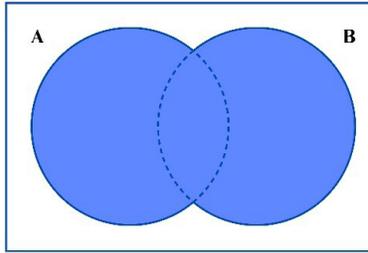


FIGURA 3.2. DIAGRAMA DE VENN, UNIÓN DE DOS CONJUNTOS

En Python se utiliza el operador `|` para hacer la unión de dos o más conjuntos.

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = a | b #c es {1, 2, 3, 4, 5, 6}
```

La operación de la intersección de dos conjuntos da como resultado un nuevo conjunto conformado por los elementos que integran ambos conjuntos. Se denota de la forma $A \cap B$:

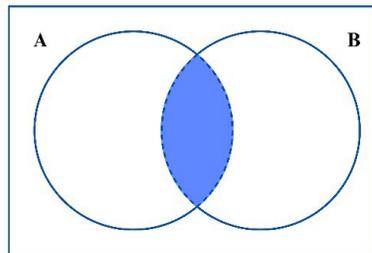


FIGURA 3.3. DIAGRAMA DE VENN, INTERSECCIÓN DE DOS CONJUNTOS.

En Python se utiliza el operador `&` para hacer la intersección de dos o más conjuntos.

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = a & b #c es {3, 4}
```

La operación de la diferencia de conjuntos da como resultado los elementos que pertenecen a al primero y no al segundo conjunto. Se denota de la forma $A - B$:

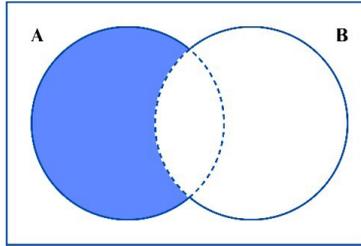


FIGURA 3.4. DIAGRAMA DE VENN, DIFERENCIA DE DOS CONJUNTOS.

En Python se utiliza el operador de $-$ para hacer la diferencia de dos conjuntos:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = a - b #c es {1, 2}
```

La operación de la diferencia simétrica de conjuntos da como resultado un nuevo conjunto que tiene por elementos la unión de las diferencias de $A - B$ y $B - A$. Es decir, $A \wedge B = (A - B) \cup (B - A)$:

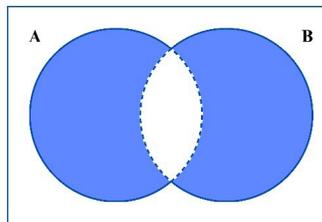


FIGURA 3.5. DIAGRAMA DE VENN, DIFERENCIA DE DOS CONJUNTOS

En Python se utiliza el operador de \wedge para obtener la diferencia simétrica de ambos conjuntos:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = a ^ b #c es {1, 2, 5, 6}
```

3.4. ESTRUCTURAS DE CONTROL DE FLUJO

Una estructura de control en Python es un conjunto de datos que se agrupa en bloques de código, mediante el indentado de las sentencias. Estas estructuras permiten que el programa tome decisiones o realice acciones en función de las condiciones que se presenten durante su ejecución. En esta sección se abordará el uso de sentencias condicionales y bucles.

3.4.1. Sentencias condicionales

Las estructuras de control o condicionales son aquellas que permiten evaluar si una o más condiciones se cumplen. Estas expresiones pueden dar solo dos valores *True* ('cierto') y *False* ('falso'). Para evaluar una condición, se usan los operadores relacionales, que se vieron en la sección de valores *booleanos*. Se pueden evaluar múltiples condiciones al mismo tiempo usando los operadores lógicos que también se vieron en la sección de valores *booleanos*.

A diario, las personas interactuamos de acuerdo con la evaluación de condiciones; un ejemplo es la manera en la que nos transportamos. Si se hizo tarde, se puede tomar un taxi. Si hay suficiente tiempo, podría usar el transporte público; si no, también podría usar la bicicleta. Este tipo de decisiones son las que se trabajan con las sentencias condicionales en Python.

Las estructuras condicionales están definidas mediante las palabras reservadas *if*, *elif*, *else*.

3.4.1.1. If

El condicional *if* ayuda a evaluar si una condición se cumple, con el fin de ejecutar una sentencia de código que conlleva a una acción. La forma

simple de usarlo es emplear la palabra reservada *if* seguida de la condición que se evaluará y dos puntos. En las siguientes líneas hay que hacer uso de la *indentación*, que es el uso de la sangría después de enunciar una sentencia, usando el tabulador.

Es importante tomar en cuenta que en cualquier sentencia de estructura de control en Python se debe hacer uso de la indentación, ya que lo toma como el código que se va a incluir en la condición; de hacer caso omiso, nos marcará error.

Tomando el ejemplo anterior del transporte:

```
if transporte == tarde:
    print ("Tomar un taxi")
```

Que en lenguaje coloquial se leería: si transporte es igual a tarde, toma un taxi.

3.4.1.2. If... else

Para ejecutar ciertas órdenes en el caso de que una condición no se cumpla, se pueden tomar varias alternativas para cumplir el mismo objetivo. Una forma consiste en usar un *if* anidado:

```
if transporte == tarde:
    print ("Tomar un taxi")
    if transporte == tiempo_suficiente:
        print ("Toma el transporte público")
```

Pero la ruta más corta es usando el *if... else*, que traducido al español se puede leer como 'si... no'.

```
if transporte == tarde:
    print ("Tomar un taxi")
else:
    print ("Toma el transporte público")
```

3.4.1.3. Elif

El condicional *elif* se usa para ejecutar una acción en caso de que más de una condición no se cumpla; es una contracción de lo que en otros lenguajes de programación sería el *else if*.

```
if transporte == tarde:
    print ("Tomar un taxi")
elif transporte == tiempo_suficiente:
    print ("Toma el transporte público")
else:
    print ("Ve en bicicleta")
```

Primero se va a evaluar la condición del *if*. Si no se cumple, pasa a evaluar la condición del *elif*; por último, si ninguna de las dos condiciones se cumple, pasa a evaluar a la sentencia del *else* y ejecutar el código.

3.4.2. Bucles

La diferencia de los bucles con las condicionales es que los primeros permiten ejecutar una sentencia de código de manera repetitiva mientras se cumpla la condición.

3.4.2.1. Bucle while

Un bucle *while*, cuya traducción del inglés al español es ‘mientras que’, ejecuta una condición de manera iterativa mientras sea verdadera. Quien haya tenido experiencia con otros lenguajes de programación sabrá que el bucle *while* tiene dos variantes: *while* y *do – while*, pero en Python solo se admite la primera.

En el siguiente ejemplo se puede ver que el bucle evalúa la condición de que la variable iteradora *i* sea menor de 10. Mientras eso pase, se imprimirá el mensaje.

```
while i < 10:
    print ("Hola Mundo")
    i = i + 1
```

Como se puede observar, con cada ciclo que transcurre se incrementa el valor de la variable *i*, que es la que condiciona el bucle *while*. Si no se

pusiera en la sintaxis que conforme a cada ciclo vaya sumando un número más, siempre imprimiría el “hola mundo” sin fin, hasta que dejáramos de ejecutar el programa, ya que *i* siempre sería menor de 10.

Hay una forma más simple para hacer la declaración de los operadores de asignación; se debe tener cuidado ya que no debe haber espacios entre el símbolo del operador y el signo de igualación. Los siguientes son algunos ejemplos de las declaraciones de los operadores de asignación.

```
m += 5      # suma en asignación m = m + 5
m -= 5      # resta en asignación m = m - 5
m *= 5      # Multiplicación en asignación
m /= 5      #División en asignación
m **= 5     #Potencia en asignación
m %= 2      #modulo en asignación
```

Dependiendo de las situaciones no siempre se puede hacer uso de la condicional numérica y el ciclo infinito nos resulta realmente útil. Veamos un ejemplo:

```
while True:
    numero = input ("Inserte su número telefónico (solo 10 dígitos)
\t")
    print(len(numero))
    if (len(numero) != 10):
        print ("El número no cumple con los 10 dígitos, Intente
nuevamente")
    else:
        print ("Número correcto")
        break
```

En el último ejemplo, se le pide al usuario que ingrese su número de teléfono, pero el programa solo acepta diez caracteres, de lo contrario, el programa pedirá al usuario que ingresen de nuevo y de manera correcta los diez dígitos. Se podría decir que es un bucle anidado; *if* verifica si la variable cumple con la condición, y de lo contrario pasa al *else*, donde el *break* terminará el ciclo.

3.4.2.2.-Bucle for

En Python, la sintaxis del ciclo *for* es un poco diferente; es una forma más genérica de hacer una iteración con base en una secuencia, pero con el fin de facilitar su uso. Un bucle *for* permite hacer iteraciones sobre variables de tipo lista, diccionario o tupla; repite el código del bucle para cada valor de la variable. Por ejemplo, por cada valor en lista, imprime el número:

```
lista = [1, 2, 3, 4, 5]
for i in lista:
    print(f"Elemento {i} ")
```

Por cada valor en el diccionario, imprime el id:

```
diccionario = {"Gloria":22,"Maria":23,"Alex":28}
for i in diccionario:
    print(f"{diccionario[i]}")
```

Por cada carácter en el *string*, imprime la cadena:

```
cadena = "Python"
for i in cadena:
    print(f"{i}", end=" ") #end = " " nos sirve para imprimir sin salto
de línea
```

Por cada clave y valor en el diccionario, imprime los elementos del diccionario:

```
diccionario = {"Gloria":22,"Maria":23,"Alex":28}
for clave, valor in diccionario.items():
    print(f"{clave} -> {valor}")
```

Hasta ahora se ha visto cómo iterar una variable compleja del tipo lista o tupla. Pero ¿qué pasa si quisiéramos aplicar el ciclo *for* para imprimir variables de tipo *int*? No será necesario crear una lista y añadir los números porque Python tiene una función llamada *range*.

La función *range* proporciona números enteros delimitados por argumentos de la función. Dentro del paréntesis, se hace uso del *slising*, en donde se indica desde dónde se quiere que comience la iteración y hasta qué número termina.

```
for i in range(0,10):
    print(i, end=" ")          #imprime: 0 1 2 3 4 5 6 7 8 9
```

También se puede indicar cuántos saltos y cuántas veces se quiere que se cumpla el ciclo.

```
for i in range(0,20,2):
    print(i, end=" ")          #imprime: 0 2 4 6 8 10 12 14 16 18
```

Como otro escenario, suponga que desea omitir la iteración si se cumple una condición, pero que siga ejecutando el ciclo. Para ello, se puede hacer uso de la palabra reservada *continue*:

```
for i in range(0,10):
    if (i == 6):
        continue
    print(i, end=" ")          #imprime: 0 1 2 3 4 5 7 8 9
```

Por último, se puede usar un bucle *for* que se encuentre dentro de un bloque de código de otro bucle *for*. A esta estructura de código se le conoce como bucles anidados. Como podrá ver en el siguiente ejemplo, se tienen dos bucles anidados que van desde el 0 hasta el 5. Lo que busca el ejercicio es imprimir una figura cuadrada con el carácter "*". El primer ciclo controla las filas y el segundo las columnas. Hacemos uso de un *print()* vacío para que exista un salto de línea al término de una fila.

```
for i in range(0,5):
    for j in range(0,5):
        print(" * ", end= '')
    print( )
#la salida es:
* * * *
* * * *
* * * *
* * *
```

Es fundamental poner en práctica lo aprendido en programación para el desarrollo de cualquier programador. Practicar y aprender de los errores son aspectos esenciales para comprender y dominar la lógica de la programación. En esta sección, se le presentarán algunos ejercicios relacionados con estructuras de control con el objetivo de entrenar las habilidades adquiridas hasta el momento. Lo ideal sería que se intente resolverlos por uno mismo y luego se revise la solución para comparar resultados.

1. Hacer un programa que pida tres números y determine cuál es el mayor de los tres, además que indique si los tres son iguales.
2. Hacer un programa que imprima una pirámide de nueve pisos con números del uno al nueve; es decir, el primer piso imprimirá el número uno, el segundo el uno y dos y así sucesivamente.
3. Construir un programa que simule el funcionamiento de una calculadora que puede realizar las cuatro operaciones aritméticas básicas (suma, resta, multiplicación y división). Debe existir un menú en donde el usuario especifique la operación que se realizará con el primer carácter del nombre de la operación, además de la opción de salir.
4. Búsqueda de elementos: escriba un programa que tome una lista y un elemento como entrada, y verifique si el elemento está presente en la lista. Puede utilizarse el operador *in* para realizar la verificación.

3.5. SOLUCIÓN DE EJERCICIOS

- 1 Para resolver este ejercicio, haremos uso de las condicionales.
 - 1.1 Solicitar al usuario que ingrese tres números, almacenándolos en las variables correspondientes.
 - 1.2 Comparar el primer número con los otros dos números.
 - 1.3 Verificar si el segundo número es menor o igual que el primer número y si el tercer número es menor o igual que el primer número.
 - 1.4 Si la condición del paso cuatro es verdadera, imprimir en pantalla que el mayor de los tres números es el primer número.

- 1.5 Realizar la misma lógica de comparación para los otros dos números.
- 1.6 Terminar la ejecución del programa o continuar según sea necesario.

```
num1 = int(input("Digite un número: "))
num2 = int(input("Digite otro número: "))
num3 = int(input("Digite uno más: "))

if num2 <= num1 and num3 <= num1:
    print("El mayor es {}".format(num1))
elif num1 <= num2 and num3 <= num2:
    print("El mayor es {}".format(num2))
elif num1 <= num3 and num2 <= num3:
    print("El mayor es {}".format(num3))
else:
    print("Todos los números son iguales")
```

- 2 Para resolver este ejercicio, haremos uso de un ciclo *for* anidado.
 - 2.1 Iniciar el primer ciclo *for* para controlar las filas del patrón.
 - 2.2 Incluir un segundo ciclo *for* anidado dentro del primero, para controlar las columnas del patrón.
 - 2.3 En el primer ciclo *for*, establecer el rango de filas de 1 a 10 para determinar cuántas filas se generarán.
 - 2.4 Explicar la razón por la que el rango del primer ciclo es hasta 10 a pesar de que solo se necesitan números hasta 9, debido a que el ciclo se detiene un número antes del límite.
 - 2.5 En el segundo ciclo *for*, establecer que el número de columnas sea igual al número de la fila actual del primer ciclo.
 - 2.6 Explicar que en el primer ciclo, cuando hay solo una fila, se imprimirá un número en esa fila.
 - 2.7 Mencionar el incremento de +1 en la impresión de las columnas para que los números no comiencen desde 0, sino desde 1.
 - 2.8 Continuar con la ejecución del programa o finalizar según sea necesario.

```
for fila in range(1, 10):
    for columnas in range(fila):
        print(columnas + 1, end=' ')
```

```
print()
```

- 3 Para construir el siguiente programa, hacemos uso de un bucle *while*.
 - 3.1 Utilizar un bucle *while* para establecer un ciclo infinito en el programa hasta que el usuario decida salir.
 - 3.2 Solicitar al usuario que ingrese un carácter correspondiente a la operación que desea realizar, y almacenar este carácter en la variable “var”.
 - 3.3 Utilizar la función *upper* para convertir el carácter ingresado en mayúsculas, asegurando así que no haya problemas con las comparaciones.
 - 3.4 Verificar si el carácter ingresado es uno de los dos tipos de caracteres esperados para las operaciones. Si es uno de ellos, se continúa con la lógica del programa.
 - 3.5 Dentro del bloque *if*, realizar la operación correspondiente según la opción ingresada por el usuario.
 - 3.6 Si el carácter ingresado no es compatible con ninguna de las opciones esperadas, ejecutar el bloque *else*.
 - 3.7 En el bloque *else*, mostrar el mensaje: “El valor que seleccionaste no pertenece al menú”, informando al usuario que su elección no es válida.
 - 3.8 Volver al ciclo principal, permitiendo al usuario seleccionar otra opción válida desde el menú.
 - 3.9 Continuar con el ciclo infinito hasta que el usuario decida salir explícitamente del programa.

```
while True:
    print("\n---CALCULADORA BÁSICA---")
    print("Inserte el carácter de acuerdo a la operación:\n\tS -
    suma\n\tR - resta\n\tM - multiplicación\n\tD - división\n\tE -
    salir")
    var = input("\n\t--> ").upper()

    if var == 'E':
        print("Adiós")
        break

    print("Ingrese los números para las operaciones:")
    num1 = float(input("N1: "))
```

```

        num2 = float(input("N2: "))
if var == 'S':
    suma = num1 + num2
    print(f"La suma es: {suma}")
elif var == 'R':
    esta = num1 - num2
    print(f"El valor de la resta: {resta}")
elif var == 'M':
    multi = num1 * num2
    print(f"El valor de la multiplicación: {multi}")
elif var == 'D':
    if num2 != 0:
        div = num1 / num2
        print(f"El valor de la división: {div}")
    else:
        print("Error: No se puede dividir entre cero.")
else:
    print("El valor que seleccionaste no pertenece al menú.")

```

- 4 En este código, se definirá una lista con cuatro números y se solicitará al usuario que ingrese un elemento.
- 4.1 Utilizar una estructura condicional “*if*” para verificar si el elemento ingresado está en la lista.
 - 4.2 Utilizar el operador “*in*” para realizar la comprobación de pertenencia del elemento a la lista.
 - 4.3 Si el elemento está en la lista (la condición es verdadera), imprimir un mensaje indicando que el elemento está en la lista.
 - 4.4 Si el elemento no está en la lista (la condición es falsa), imprimir un mensaje indicando que el elemento no está en la lista.
 - 4.5 Finalizar la ejecución del programa o continuar según sea necesario.

```

lista = [1, 2, 3, 4]
elemento = int(input("Ingrese un elemento: "))

if elemento in lista:
    print("El elemento está en la lista.")
else:
    print("El elemento no está en la lista.")

```

CAPÍTULO 4. FUNCIONES

4.1. INTRODUCCIÓN A LAS FUNCIONES EN PYTHON

En programación, una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor (González-Duque, 2011: 36). Cuando el programador no especifica un valor de retorno, la función devuelve automáticamente el valor *none*, que es equivalente al concepto de *null* en Java. Las funciones son fundamentales en la programación ya que permiten organizar y reutilizar el código de manera eficiente. En Python, como en muchos otros lenguajes, las funciones tienen una estructura definida que consta de:

1. Definición de función: Se define una función utilizando la palabra clave *def*, seguida del nombre de la función y paréntesis que pueden contener parámetros de entrada. Es importante poner al final de la definición de la función el símbolo de dos puntos (:).
2. Cuerpo de la función: El cuerpo de la función contiene las instrucciones que se ejecutarán cuando la función sea llamada. Estas instrucciones están indentadas para indicar que son parte de la función.
3. Retorno de valores: De manera opcional, una función puede devolver un valor utilizando la palabra clave *return*. Esto permite que la función proporcione un resultado que puede ser utilizado en otras partes del programa.

A continuación, se muestra un esquema básico de una función en Python:

```
def nombre_de_la_funcion(parametro1, parametro2):  
    # Cuerpo de la función  
    resultado = parametro1 + parametro2  
    return resultado
```

Cuando se llama a una función, se proporcionan argumentos que se asignan a los parámetros definidos. La función ejecuta su lógica interna y, si es necesario, devuelve un valor. Las funciones son esenciales para *modularizar* el código y facilitar su mantenimiento.

Veamos ahora un ejemplo de una función muy simple que ejecuta la acción de imprimir en pantalla el mensaje “Hola mundo”.

```
def funcion ():  
    print (“Hola Mundo”)
```

La función no se ejecutará si no se hace un llamado de esta. Para invocar la función, simplemente seguimos la sintaxis:

```
funcion()
```

Cuando la función retorna un valor de cualquier tipo, este puede ser asignado a una variable:

```
def funcion():  
    return “Hola Mundo”  
cadena = funcion ()  
print (cadena)
```

4.2. PARÁMETROS DE LA FUNCIÓN

Los parámetros son valores que una función puede recibir cuando es invocada, con los cuales se ejecutará dicha función. Una función puede o no tener parámetros. Los parámetros irán separados por comas.

```
def funcion (carrera, materia):  
    #Cuerpo de la función
```

Los parámetros que la función espera serán usados de manera local, es decir, son variables locales a las cuales solo se puede llegar dentro de la función.

```
def funcion (carrera, materia):
    asignatura_carrera = carrera, materia
    print(asignatura_carrera)
```

Al llamar una función, y en caso de que esta cuente con argumentos, siempre se le deben pasar en el mismo orden que los espera. Para evitarlo, se puede hacer uso del paso de argumentos como *keywords*, es decir, como clave = valor:

```
def alumno(nombre, matricula):
    print (matricula,nombre)
alumno(matricula="180486", nombre="Daniela Varela" )
```

Es posible hacer uso de los parámetros por omisión, en donde asignamos valores por defecto a los parámetros. Entonces puede ser llamada con menos argumentos de los que se espera.

```
def alumno(matricula="180486", nombre):
    print (matricula,nombre)
alumno("Daniela Varela")
```

Es recomendable que después de haber declarado una función se dejen dos espacios en blanco. Al asignar los parámetros por omisión, no debe haber espacios en blanco ni antes ni después del signo =.

Como en otros lenguajes de programación de alto nivel, es posible que la función reciba un número desconocido de argumentos, que llegan a la función en forma de tupla. Para definir que los argumentos son arbitrarios, se antecede el parámetro con un asterisco *.

```
def parámetros_arbitrarios (parámetro_fijo, *valores_arbitrarios):
    for i in valores_arbitrarios:
        print(i)
parámetros_arbitrarios('valor fijo', 'arbitrario 1', 'arbitrario
2','arbitrario 3')
```

Si la función espera recibir un parámetro fijo junto con arbitrarios, los fijos siempre deben ir antes que los arbitrarios. También los parámetros

arbitrarios pueden ser parámetros clave. En estos casos, al declarar el parámetro, se le debe anteceder con dos asteriscos **:

```
def parámetros_arbitrarios (parámetro_fijo, *valores_arbitrarios,
**clave_valor):

    for i in valores_arbitrarios:
        print(i)
    for i in clave_valor:
        print ("El valor de", i, "es", clave_valor[i])

parámetros_arbitrarios ('valor fijo', "arbitrario 1", "arbitrario 2",
"arbitrario 3", clave1="valor uno", clave2="valor dos")
```

Al contrario del ejemplo anterior, suele suceder que la función esté esperando una lista fija de parámetros, pero que estos no se encuentren de manera separada sino contenidos en una lista o tupla. Por tanto, el signo * debe estar antes del nombre de la lista o tupla al momento que se pasa como un parámetro durante la llamada de la función:

```
def iva_producto(precio, iva ):

    return ((precio * iva)/100)

datos = [179,16]
print(iva_producto(*datos))
```

Lo mismo pasa cuando los valores están contenidos en un diccionario y se quieren pasar como parámetros a la función, deben ser procedidos de dos asteriscos (**):

```
def iva_producto(precio, iva):
    return ((precio * iva)/100)

datos = {"precio": 179, "iva": 16}
print(iva_producto(**datos))
```

4.3. LLAMADAS DE RETORNO

Muchas veces es necesario que una función retorne o devuelva un valor fuera de ella, para trabajar con ellos en el exterior de la función. Para lograrlo, se hace uso de la palabra reservada *return*. Esta función se utiliza solo dentro de las funciones, pues no puede existir un *return* en otro lugar que no sea dentro de una función, y se utiliza para establecer un valor de retorno de una dicha función. Como ejemplo:

```
def funcion():
    return "Hola mundo"
funcion()
```

Pero en el ejemplo anterior, al hacer la llamada a la función, no se está recibiendo el valor que contiene; necesitamos declarar una variable que sea de tipo contenedor para que se almacene el valor de retorno de la función.

```
def funcion():
    return "Hola mundo"
variable = funcion()
print(variable)
```

En Python, podemos llamar a una función dentro de otra, de forma fija como si la llamáramos fuera de la función:

```
def funcion():
    return "Hola mundo"

def saludo(nombre, mensaje= "Hello word"):
    print(mensaje,nombre)
    print(funcion())
```

Sí tuvo la oportunidad de ejecutar el ejemplo anterior se pudo dar cuenta que no logra imprimir ningún valor de la función. Una forma de lograr hacerlo es hacer una llamada de retorno, es decir, de manera dinámica, sin que se conozca el nombre de la función a la que se quiere llamar.

Para comprender mejor la idea de cómo podemos llamar una función dentro de otra tenemos que entender lo que son las variables loca-

les y globales. En Python, una variable local es la que se define dentro de una función y a la que solo podemos llegar dentro de ella. Las globales se definen al principio del programa fuera de cualquier función, permitiendo entrar a ellas en cualquier parte del código.

Continuando con el ejemplo de la llamada de retorno, las funciones nativas, como lo son *locals()* y *globals()*, nos ayudan a llamar a una función de manera dinámica. Ninguna lleva argumento alguno dentro de los paréntesis. Si imprimimos en pantalla alguna de las dos funciones, obtendremos una lista de un objeto diccionario, las cuales tienen la sintaxis `'_nombremetodo_'`, que se verá más a fondo en otros capítulos. Entonces, ambas retornan un diccionario y sus valores serán los que hayamos asignado, cuyo valor será su ubicación en memoria. En el caso de *locals()*, este diccionario se compone de todos los elementos de ámbito local, mientras que el de *globals()*, en el nivel global.

```
def funcion():
    return 'Hola Mundo'

def llamada_retorno(fun=""):
    #llamada de retorno global
    return globals()[fun]()

print(llamada_retorno("funcion"))

#retorno de la función a nivel local
retorno_local = "funcion"
print(locals()[retorno_local]())
```

Pasar argumentos en una llamada de retorno, se puede hacer sin ningún problema.

```
def funcion(nombre):
    return 'Hola Mundo' + nombre

def llamada_retorno(fun=""):
    #llamada de retorno global
    return globals()[fun]("Sebastian")

print(llamada_retorno("funcion"))
```

```
#retorno de la función a nivel local
retorno_local = "funcion"
print(locals()[retorno_local]("Sofia"))
```

4.4. ¿CÓMO SABER SI UNA FUNCIÓN EXISTE Y PUEDE SER LLAMADA?

Cuando se hace una llamada a una función, el nombre de la función puede que no sea el correcto; por ende, siempre que se hace una llamada de retorno es necesario comprobar que la función exista y puede ser llamada.

El operador *in* ayuda a saber si un elemento pertenece a una colección de datos, mientras que la función *callable()* permite saber si esa función puede ser llamada.

```
def funcion(nombre):
    return "Hola " + nombre

def llamada_de_retorno(fun= ""):
    if fun in globals():
        if callable(globals()[fun]):
            return globals()[fun]("Laura")
    else:
        return "Función no encontrada"

print(llamada_de_retorno("funcion"))

nombre_de_la_funcion = "funcion"

if nombre_de_la_funcion in locals():
    if callable(locals()[nombre_de_la_funcion]):
        print (locals()[nombre_de_la_funcion]("Emma"))
else:
    print("funcion no encontrada")
```

4.5. LLAMADAS RECURSIVAS

Se le llama recursividad o llamada recursiva a la función que en su cuerpo o en el código que contiene hace referencia a sí misma. Python permite las llamadas recursivas, pues una función puede llamarse a sí misma como cuando se llama a otra función.

La recursividad es muy útil en casos específicos, pero puede generar iteraciones infinitas, por lo cual solo debe usarse cuando sea estrictamente necesario y no exista una forma alternativa que pueda ayudar y la recursividad sea la única opción.

```
def jugar(intento=1):  
  
    #con lower convertimos todos los caracteres a minúsculas  
    respuesta = input("¿De qué color es una naranja? ").lower()  
    if respuesta != "naranja":  
        if intento < 3:  
            print ("\nFallaste! Inténtalo de nuevo")  
            intento += 1  
            jugar(intento) # Llamada recursiva  
        else:  
            print ("\nPerdiste!")  
        else:  
            print ("\nGanaste!")  
    jugar()
```

Una función puede contener cualquier tipo de algoritmo y usar cualquiera de los temas vistos hasta ahora. No obstante, una buena práctica indica que la finalidad de las funciones es que deben realizar una única acción, reutilizable y, por tanto, tan genérica como sea posible.

4.6. EJERCICIOS DEL CAPÍTULO

Las funciones desempeñan un papel fundamental en la programación en general. Un aspecto muy importante es la reutilización de código, ya que aquellas permiten reutilizar bloques de código, lo que vuelve a nuestro código más legible. A continuación, se presentan tres ejercicios en los que podrá ponerse en práctica lo que aprendido durante el capí-

tulo. Intente resolverlos, y una vez que lo haya logrado, podrá ver una de las posibles formas de resolverlos.

1. Escriba una función que calcule el área de un círculo. La función debe tomar el radio como argumento.
2. Escriba una función recursiva que calcule el enésimo número de la sucesión de Fibonacci.
3. Escriba una función recursiva que calcule el factorial de un número entero positivo.

4.7. SOLUCIÓN DE EJERCICIOS

- 1 El código toma un valor de radio ingresado por el usuario, calcule el área de un círculo utilizando una función definida y muestre el resultado redondeado en pantalla.
 - 1.1 La primera línea define `area_circulo` que toma un parámetro *rad* (que se asume es el radio del círculo). Esta función calcula el área del círculo utilizando la fórmula $\pi * \text{radio}^2$ y luego redondea el resultado a dos decimales.
 - 1.2 Se solicita al usuario que ingrese el valor del radio de la circunferencia. El valor ingresado se convierte en un número de punto flotante (decimal) utilizando la función *float*. En este ejemplo, se asume que el usuario ingresa el valor 20.
 - 1.3 Se llama a la función `area_circulo` pasando el valor del radio ingresado por el usuario como argumento. Esto calcula el área del círculo con el radio proporcionado.
 - 1.4 Finalmente, se imprime un mensaje que muestra el resultado del cálculo del área del círculo. El formato *f-string* se utiliza para insertar el valor de resultado en el mensaje. En este caso, la salida será “El área del círculo es: 1256.64”, basada en la entrada de radio 20 y el resultado calculado.

```
def area_circulo(rad):  
    return round(3.1416 * (rad**2), 2)
```

```
radio = float(input("Ingrese el radio de la circunferencia\t"))
#Ingresamos 20
resultado = area_circulo(radio)
print(f"El área del círculo es:\t{resultado}") #Salida 1256.64
```

- 2 El código usa una función recursiva para calcular el término de la sucesión de Fibonacci según el valor ingresado por el usuario, y luego muestra el resultado.

2.1 Se define una función que calcula el término *num* de la sucesión de Fibonacci. Si *num* es 0 o 1, retorna 1 (casos base).

2.2 Si *num* no es 0 ni 1, la función se llama a sí misma dos veces con valores decrementados (*num* - 1 y *num* - 2) y suma sus resultados.

2.3 Se pide al usuario el término de la sucesión que desea calcular.

2.4 Se calcula el término de la sucesión de Fibonacci correspondiente al valor ingresado por el usuario.

2.5 Se muestra el valor calculado en pantalla.

```
def fibonacci (num):
    if num == 0 or num == 1:
        return 1
    else:
        return fibonacci(num - 1) + fibonacci(num - 2)

num = int(input("Ingrese el valor que quiere conocer de la
sucesión: "))
#Se ingresa el número 8
fib = fibonacci(num)
print(fib) #salida 34
```

- 3 En el código, calcule el factorial de un número utilizando la técnica recursiva descrita, donde el factorial de *n* se calcula en términos del factorial de *n* - 1.

3.1 En la primera línea, se define una función, llamada factorial, que toma un parámetro *num*. El factorial de 0 y 1 es 1, por lo que si *num* es 0 o 1, la función devuelve 1.

3.2 Si *num* no es 0 ni 1, entonces se ejecuta la parte del *else*. En este punto, la función factorial se llama a sí misma con un

argumento decrementado ($\text{num} - 1$) y luego multiplica ese resultado por el valor actual de num . Esto se debe a que el factorial de un número n es n multiplicado por el factorial de $n - 1$.

- 3.3 Se solicita al usuario que ingrese un número para calcular su factorial. En este ejemplo, el usuario ingresa el número 4.
- 3.4 Se llama a la función factorial pasando el valor ingresado por el usuario como argumento. Esto calcula el factorial del número ingresado.
- 3.5 Finalmente, se imprime el valor del factorial calculado utilizando el valor ingresado por el usuario. En este caso, la salida será "24", basada en la entrada "4" y el cálculo del factorial.

```
def factorial (num):
    if num == 1 or num == 0:
        return 1
    else:
        return num * factorial(num - 1)

num = int(input("Ingrese el número: "))
#Se ingresa el número 4
fac = factorial(num)
print(fac) #Salida 24
```

CAPÍTULO 5.

ORIENTACIÓN A OBJETOS

5.1. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (POO) es un paradigma de programación que se basa en la idea de organizar y estructurar el código de manera que los datos y las operaciones relacionadas se agrupen en objetos. A continuación, se da una explicación esquemática de los conceptos clave de la programación orientada a objetos:

1. **Objeto:** Un objeto es una instancia concreta de una clase. Representa una entidad del mundo real y contiene datos (atributos) y comportamiento (métodos) relacionados.
2. **Clase:** Una clase es un plano o plantilla para crear objetos. Define la estructura y el comportamiento común que tendrán los objetos de esa clase.
3. **Atributos:** Los atributos son variables que almacenan datos relacionados con el objeto. Representan las características de un objeto.
4. **Métodos:** Los métodos son funciones que definen el comportamiento de un objeto. Representan las acciones que un objeto puede realizar.
5. **Encapsulación:** La encapsulación es un principio que consiste en ocultar los detalles internos de un objeto y proporcionar una interfaz clara para interactuar con él. Se utiliza para proteger los datos y garantizar que solo se puedan ingresar y modificar de manera controlada.
6. **Herencia:** La herencia permite crear nuevas clases basadas en clases existentes. Una clase derivada hereda atributos y métodos de una clase base, lo que fomenta la reutilización de código y la creación de jerarquías de clases.

7. Polimorfismo: El polimorfismo permite que objetos de diferentes clases respondan de manera similar a un conjunto común de operaciones. Esto facilita la flexibilidad y la extensibilidad del código.

Un ejemplo en Python sería el siguiente:

```
# Definición de una clase 'Persona'
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")

# Creación de objetos 'Persona'
persona1 = Persona("Juan", 30)
persona2 = Persona("María", 25)

# Llamada a métodos de objetos
persona1.saludar()
persona2.saludar()
```

En este ejemplo, la clase 'Persona' define atributos (nombre y edad) y un método (saludar). Se crean dos objetos (persona1 y persona2) a partir de esta clase y se llaman a los métodos en cada objeto.

La POO es ampliamente utilizada en la programación debido a su capacidad para modelar conceptos del mundo real de manera eficiente y estructurada.

5.2. CLASES Y OBJETOS

Para entender el tema de la orientación a objetos, primero hay que distinguir entre una clase y un objeto. Un objeto es un ente que contiene atributos, así como funciones relacionadas, a las que se les conoce como métodos del objeto.

Una clase es la organización de datos en los que se describe y codifica el comportamiento de un objeto. El ejemplo podría ser que se tenga una clase llamada teléfono celular, sus propiedades pueden ser la marca,

la cámara, los botones, la pantalla, mientras que algunas funciones que lo definen pueden ser realizar llamadas, tomar fotografías, encender o apagar. A nuestro alrededor existe un conjunto de objetos a los que podemos llamar teléfonos celulares, los cuales contienen atributos en común con otros, que es a lo que precisamente llamamos clases. A pesar de ello todos los objetos son distintos por muy parecidos que parezcan, aunque pertenecen a la misma clasificación o bien clase de objetos.

En Python, las clases se definen mediante la palabra clave “class”, y en seguida el nombre de la clase, para finalizar con dos puntos (:) y después, con la indentación correspondiente, el cuerpo de la clase.

```
class Celular:

    def __init__(self, bateria):
        self.bateria = bateria
        print (f"Tenemos {bateria}% de bateria")
        if self.bateria < 15:
            print("Nivel bajo de bateria")
        def encender(self):
            if self.bateria > 0:
                print ("Enciende")
            else:
                print ("No enciende")
        def usar(self):
            if self.bateria > 0:
                self.bateria -= 1
                print (f"Queda {self.bateria}% de bateria")
                if self.bateria < 15:
                    print("Nivel bajo de bateria")
            else:
                print("No enciende")
```

Algo que muy probablemente despierte interés es la forma del método `__init__`. El método anterior, que está compuesto por una doble barra baja al principio y final del nombre, se ejecuta después de haber creado el objeto a partir de la clase. A dicho proceso se le conoce como *instanciación*.

El primer parámetro de la clase siempre es “self”, el cual sirve para referirse al objeto actual. Este mecanismo es necesario para ingresar a los atributos y métodos del objeto diferenciando una variable local de un atributo del objeto “self”.

Al regresar al método `_init_` de la clase celular, se puede ver cómo se utiliza `self` para asignar al atributo `carga` el valor que el programador asignó para el parámetro `carga`. El parámetro de la carga se elimina al final de la función, mientras que el atributo `carga` se conserva, al cual se puede ingresar mientras exista el objeto.

Para crear un objeto, se escribiría el nombre de la clase, seguido de cualquier parámetro entre paréntesis. Estos parámetros son los que se pasarán al método `_init_`, que, como decíamos, es el método que se llama al *instanciar* la clase.

```
mi_celular = Celular (50)
```

¿Cómo es posible que al crear nuestro primer objeto pasemos solo un parámetro a `__init__` el número 50, al cual corresponde el porcentaje de batería, cuando claramente, al definir la función, existen dos parámetros, que son (`self` y `batería`). Esto sucede porque Python pasa el primer argumento automáticamente.

Una vez creado el objeto, podemos ingresar a sus atributos mediante la sintaxis `objeto.atributo` o `objeto.metodo()`:

```
print(mi_celular.bateria)           # salida 50
print(mi_celular.encender())        #Enciende
print(mi_celular.usar())            #Queda 49% de bateria
print(mi_celular.usar())            #Queda 48% de batería
```

Como último recordatorio, en Python todo es un objeto. Las cadenas, como ejemplo, tienen métodos como `upper()`, que devuelve texto en mayúsculas, o `lower()`, que devuelve el texto en minúsculas. Se ha estado usando todo el tiempo objetos sin darnos cuenta, coinciden con cualquier tipo de dato en Python, como los enteros, flotantes, listas, tuplas, diccionarios. Siempre recuerde que un objeto es una colección de datos y comportamientos.

5.3. HERENCIA

Cuando se habla de las clases y objetos, se hace referencia a que hay objetos que comparten propiedades y métodos con otros. En los lenguajes de programación orientados a objetos, cuando se hace que una superclase herede a una clase subclase, se está haciendo que la última tenga los atributos y métodos de la superclase.

Suponga que se quieren modelar los utensilios de cocina más básicos; se tendría una clase que se llame *sartén*, *cuchillo*, *espátula*, *tabla*, entre otros. Cada una de estas clases tendrá sus propios atributos, pero, por pertenecer a una clasificación, compartirá atributos y métodos. Un ejemplo sería el método `cocinar()`.

Para evitar redundancias, es más fácil crear un tipo de objeto con los atributos y métodos que comparten en común, e indicar al programa que las clases que se heredarán forman parte de la clasificación y, retomando el ejemplo anterior, heredan los atributos y métodos de “utensilios”.

Para indicar que existe una herencia, se coloca el nombre de la clase de la que se hereda, y después el nombre de la clase:

```
class Utensilios:
    def __init__(self, precio):
        self.precio = precio
    def cocinar(self):
        print("Estamos cocinando")
    def romper(self):
        print("Necesitamos comprar otro")
        print(f"son{self.precio}")
    class sarten(Utensilios):
        pass
    class espatula(Utensilios):
        pass
```

Como “sartén” y “espátula” heredan de “utensilios”, ambos tienen los mismos métodos que “utensilios” y se inician pasando el parámetro del precio. Ahora bien, ¿qué ocurre si necesitamos especificar un nuevo parámetro a la hora de crear un nuevo objeto, como “cuchillo”? Sería suficiente escribir un nuevo método para la clase “cuchillo”, que se ejecutaría en el lugar del `__init__` de “utensilios”. A la acción que se describe anteriormente se le conoce como sobrescribir métodos.

En algunos casos surgirá la necesidad de sobrescribir un método de la clase padre porque se quiere hacer uso de la clase padre pero se necesitan agregar instrucciones. En el caso correspondiente, se usaría la sintaxis `Superclase.metodo(self, argumentos)` para llamar al método de igual nombre de la clase padre. Por ejemplo, `Utensilios.__init__(self, precio)`, aquí podemos ver que llamamos al método `__init__` de `utensilios` desde una nueva clase, que sería “cuchillo”.

5.4. HERENCIA MÚLTIPLE

La herencia múltiple es una de las características que solo ciertos lenguajes de programación poseen. A diferencia de otros lenguajes, como Java o C#, en Python se permite la herencia múltiple. La herencia múltiple permite la reutilización de código, ya que una clase puede adquirir las propiedades de varias clases a la vez. Como ejemplo, tenemos la clase “coche”, que hereda a la clase “transporte”, con atributos como la marca. Lo único que hay que hacer es colocar las clases de las que se hereda, separadas por comas:

```
class Transporte (Carro,Bicicleta):
    def main(self):
        print(" Esta es la clase principal")
```

En el caso de una clase padre que tuviera métodos con el mismo nombre y parámetros que otra clase padre, estas clases sobrescribirían la aplicación de los métodos de las clases que se encuentren más a la derecha de la definición.

En el ejemplo que se muestra a continuación, como la clase “carro” está más hacia la izquierda con respecto a la llamada del método, será la definición de “carro” la que prevalecerá sobre “autobús”; por tanto, si llamamos al método `B.mercedesbenz()` del objeto de tipo “transporte”, lo que se imprimirá será “Este es un carro Mercedes-Benz”.

```
class Carro():
    def mercedesbenz(self):
        print(" Este es un carro Mercedes-Benz")
```

```

class Bicicleta():
    def ventum(self):
        print(" Esta es una bicicleta Ventum ")
class Autobus():
    def mercedesbenz(self):
        print(" Este es un autobús Mercedes-Benz ")
class Avion():
    def Indigo(self):
        print(" Este es un avion Indigo")
class Transporte(Carro,Bicicleta,Autobus,Avion):
    def main(self):
        print(" Esta es la clase principal")
B = Transporte()
B.mercedesbenz()
B.ventum()
B.Indigo()
B.main()

```

5.5. POLIMORFISMO

El origen de la palabra *polimorfismo* es del griego πολύμορφος, de las raíces πολύ ('muchos') y μορφος ('formas'). Y, como la propia definición lo dice, los objetos pueden tomar diferentes formas. Esto en POO ¿qué significa? Pues bien, quiere decir que los objetos de distintas clases pueden responder a un mismo mensaje o ser conseguidos utilizando la misma interfaz, pero con un comportamiento diferente y, valga la redundancia, adquiriendo distintas formas.

Para comprender un poco mejor el tema de polimorfismo es importante entender el *duck typing*, que traducido al español sería 'tipado de pato', y se basa en una frase según la cual si camina como pato y suena como pato, tiene que ser un pato.

Esta analogía hace referencia a que los patos son objetos, y hablar/caminar, métodos. Entonces, lo único que nos interesa de un objeto es que tenga los métodos que necesitamos, y su tipo no nos importa. En pocas palabras, en Python no importan los tipos de objetos, lo único que nos interesa son los métodos.

Como ya hemos mencionado, Python es un lenguaje de tipado dinámico; entonces, no es necesario que los objetos compartan una estructura del código, sino basta con que tengan métodos que se quieren

llamar. Supongamos que tenemos una clase llamada “aves” con el método de “trinar”.

```
class Ave:
    def trinar(self):
        pass

class Pato(Ave):
    def trinar(self):
        print("¡Cuac, Cuac!")

class Pollo(Ave):
    def trinar(self):
        print("¡Pio, pio!")
```

A continuación, creamos un objeto de cada clase y llamamos al método trinar(). Podemos observar la diferencia de cómo se comporta cada ave. La variable “ave” ha tomado el polimorfismo, ya que toma la forma de pato y pollo.

```
for ave in Pato(), Pollo():
    ave.trinar()
```

En muchas ocasiones se relaciona el polimorfismo con la sobrecarga de métodos. “La sobrecarga de métodos, término que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa” (González-Duque, 2011: 48). Pero algo importante es que en Python la sobrecarga de métodos es inexistente, ya que solo se sobrescribe el método anterior.

5.6. ENCAPSULACIÓN

Con encapsulación nos referimos al proceso de restringir el acceso a métodos y atributos seleccionados, para establecer los elementos a los cuales sí podemos ingresar fuera de la clase.

Seguramente, si ha trabajado en Java lo relacione con las palabras reservadas (public) o (private), en donde se utilizan modificadores de acce-

so y se establece si cualquiera puede entrar a esa función o variable, o si está restringido el acceso. En Python es muy diferente, pues no existen los modificadores de acceso, sino más bien se determina por su nombre. Con la afirmación anterior se hace referencia a que la forma de declarar los atributos y métodos es la que define si son públicos o privados:

```
class Clase:
    def __init__(self):
        self._a_privado = "Atributo Privado"

    def metodo_publico(self):
        print("Método publico")

    def _metodo_privado(self):
        print("Método privado")

ob = Clase()
print(ob._a_privado)
ob.metodo_publico()
ob._Clase__metodo_privado()
```

En el ejemplo anterior, vimos que la declaración para los métodos privados se hace comenzando con dobles guiones bajos (técnica a la que se le conoce como *name mangling*). Los elementos `_a_privado` y `_metodo_privado` son internos de la clase, por lo que no es conveniente ingresar a ellos fuera de esta. Pero no quiere decir que no sea posible hacerlo; como pudo observar, se ingresó al método privado haciendo uso de la siguiente declaración:

```
ob._Clase__metodo_privado()
```

Por el contrario, `metodo_publico` puede ser utilizado por otros objetos y fuera de la clase. Suele suceder que se necesite permitir el acceso a algún atributo y que el acceso sea de manera controlada. Para ello, se pueden crear métodos cuyo objetivo sea ese. Normalmente esos métodos tienen la estructura de un `getNombre` y `setNombre`; de donde surge la clasificación de *getters* y *setters*.

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre
```

```

def get_nombre(self):
    return self._nombre

def set_nombre(self, nuevo_nombre):
    self._nombre = nuevo_nombre

nombre = property(get_nombre, set_nombre)

persona = Persona(«Antonio»)
print(persona.nombre)

persona.nombre = «Pedro»
print(persona.nombre)

```

En el ejemplo que se acaba de ver, se definen los métodos de `get_nombre` y `set_nombre` para ingresar a y modificar el atributo `_nombre`. Posteriormente, se hizo uso de la propiedad “nombre” para enlazar los métodos *setter* y *getter*. La propiedad de nombre se define usando los métodos `get_nombre` y `set_nombre` como argumentos de la función `property()`.

La función `property()` está integrada a Python y se usa para crear propiedades de clase y objetos. Esta función toma uno o hasta tres argumentos. Si se proporciona un solo argumento, debe ser un argumento que se comporte como *getter*. Si se proporcionan dos argumentos, el primero debe ser un *getter* y el segundo un *setter*.

En el ejemplo anterior, la función `property()` se usa para crear la propiedad “nombre”. Se le pasan los métodos `get_nombre()` y `set_nombre()` como argumentos. El método `get_nombre()` devuelve el valor del atributo `_nombre`, y el método `set_nombre()` asigna el valor del parámetro `nuevo_nombre` al atributo `_nombre`.

5.7. EJERCICIOS DEL CAPÍTULO

La programación orientada a objetos permite representar objetos del mundo real. Pueden crearse clases que encapsulen datos y comportamientos, lo cual facilita la comprensión y el diseño del programa. En esta sección, al igual que en secciones de capítulos anteriores, se verán algunos ejercicios de la POO. Intente resolverlos y luego vea la solución de dichos ejercicios.

1. Agregue un método a la clase “persona” llamado “saludar”, que imprima en pantalla un saludo con el nombre de la persona.
2. Cree una clase llamada “CuentaBancaria” con los atributos “titular” y “saldo”. Agregue métodos para depositar y retirar dinero de la cuenta, así como un método para mostrar el saldo actual.
3. Cree una clase llamada “libro” con los atributos “título”, “autor” y “año de publicación”. Agregue métodos para mostrar la información del libro y calcular cuántos años han pasado desde su publicación.
4. Cree una clase llamada “calculadora” con métodos estáticos para realizar operaciones matemáticas básicas, como sumar, restar, multiplicar y dividir.

5.8. SOLUCIÓN DE LOS EJERCICIOS

- 1 El código define una clase “persona” con métodos para asignar nombres y mostrar saludos personalizados. Luego se crea una instancia, se llama al método para imprimir un saludo usando el nombre de la instancia.
 - 1.1 Se crea la clase “persona” con un método constructor `__init__` para establecer el atributo “nombre”. También se define el método “saludo” para imprimir un saludo personalizado usando el atributo “nombre”.
 - 1.2 Se crea una instancia de la clase “persona” llamada “Sofía” con el nombre “Sofía” como parámetro.
 - 1.3 Se llama al método “saludo” en la instancia “Sofía” para imprimir un saludo personalizado. El resultado en la consola será “Hola Sofía, ¿cómo estás?”.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def saludo(self):
        print(f"Hola {self.nombre}, ¿cómo estás?")

persona = Persona("Sofía")
persona.saludo()
```

- 2 El código se basa en la definición de una clase “CuentaBancaria” con métodos para realizar operaciones en la cuenta; solicite al usuario operaciones específicas (depósito o retiro), ejecute las operaciones y muestre el saldo resultante.
 - 2.1 Se crea una clase con constructor para “titular” y “saldo”, y tres métodos para depositar, retirar y mostrar saldo.
 - 2.2 Se pide el nombre del titular y se establece un saldo inicial de 1 000.
 - 2.3 El programa le pide al usuario que elija una opción de operación basada en el menú proporcionado. Las opciones típicas son depósito o retiro.
 - 2.4 Basado en la elección del usuario, se ejecutan operaciones en la cuenta usando los métodos de la clase.
 - 2.5 Las operaciones de depósito o retiro modifican el saldo actual de la cuenta bancaria y reflejan los cambios realizados.
 - 2.6 Después de realizar la operación deseada, el programa muestra el saldo actualizado en pantalla.

```

class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, dinero):
        self.saldo += dinero
        print(f"Depósito de {dinero} exitoso.")

    def retirar(self, dinero):
        if dinero <= self.saldo:
            self.saldo -= dinero
            print(f"Retiro de {dinero} exitoso.")
        else:
            print("Fondos insuficientes.")

    def mostrar_saldo(self):
        print(f"Saldo actual: {self.saldo}.")

titular = input("Ingrese el nombre del titular: ")
saldo_inicial = float(input("Ingrese el saldo inicial: "))
cuenta = CuentaBancaria(titular, saldo_inicial)

opcion = int(input("Seleccione la opción de acuerdo a la
operación:\n1 = Depositar\n2 = Retirar\n"))

```

```

if opcion == 1:
    cantidad = float(input("Ingrese la cantidad a depositar: "))
    cuenta.depositar(cantidad)
    cuenta.mostrar_saldo()
else:
    cantidad = float(input("Ingrese la cantidad
a retirar: "))
    cuenta.retirar(cantidad)
    cuenta.mostrar_saldo()

```

3 El código define una clase “libro” con métodos para mostrar detalles y calcular antigüedad. Luego se recopila información del usuario, se crea una instancia y se emplean los métodos para mostrar detalles y antigüedad.

3.1 Como primer paso, se crea la clase “libro” con un constructor para título, autor y año de publicación. Además, se definen métodos “informacion()” y “antigüedad(anio_actual)” para mostrar detalles y calcular la antigüedad en años.

3.2 Se crea una instancia de “libro” con los datos ingresados.

3.3 Al final, se llaman los métodos “informacion()” y “antigüedad(anio_actual)” para mostrar detalles y calcular la antigüedad.

```

class Libro:
    def __init__(self, titulo, autor, year):
        self.titulo = titulo
        self.autor = autor
        self.year = year

    def informacion(self):
        print("INFORMACIÓN DEL LIBRO")
        print(f"Título: {self.titulo}")
        print(f"Autor: {self.autor}")
        print(f"Año de publicación: {self.year}")

    def antigüedad(self, anio_actual):
        antigüedad = anio_actual - self.year
        print(f"Antigüedad de la publicación:
{antigüedad} años")

print("Ingrese la siguiente información:")
titulo = input("Título del libro:\t")
autor = input("Nombre del autor:\t")
year = int(input("Año de publicación:\t"))
anio_actual = int(input("Año actual:\t"))

```

```
libro = Libro(titulo, autor, year)
libro.informacion()
libro.antiguedad(año_actual)
```

- 4 El código define una clase “calculadora” con métodos estáticos para operaciones matemáticas básicas. Luego, el usuario ingresa dos números y se utilizan los métodos para realizar y mostrar los resultados de las operaciones.
 - 4.1 Se define la clase “calculadora”, que contiene cuatro métodos estáticos: suma, resta, multiplicar y dividir.
 - 4.2 El método estático “suma(numero1, numero2)” recibe dos números como parámetros, calcula la suma y la imprime en la consola.
 - 4.3 El método estático “resta(numero1, numero2)” es similar al método anterior; este método calcula la resta de los dos números y la imprime.
 - 4.4 El método estático “multiplicar(numero1, numero2)” calcula el producto de los números ingresados y lo imprime.
 - 4.5 El método estático “dividir(numero1, numero2)” calcula la división de los números ingresados, pero antes verifica si el divisor no es cero para evitar una división por cero. Si el divisor es cero, muestra un mensaje de error.
 - 4.6 El usuario ingresa dos números enteros, num1 y num2, y se hace la lectura de los números.Al final se llama a los métodos estáticos de la clase “calculadora” para realizar las operaciones de suma, resta, división y multiplicación con los números ingresados.

```
class Calculadora:
    @staticmethod
    def suma(numero1, numero2):
        suma = numero1 + numero2
        print(f"SUMA:\t{suma}")

    @staticmethod
    def resta(numero1, numero2):
        resta = numero1 - numero2
        print(f"RESTA:\t{resta}")

    @staticmethod
```

```
def multiplicar(numero1, numero2):
    multiplicacion = numero1 * numero2
    print(f"MULTIPLICACIÓN:\t{multiplicacion}")

    @staticmethod
    def dividir(numero1, numero2):
        if numero2 != 0:
            division = numero1 / numero2
            print(f"DIVISIÓN:\t{division}")
        else:
            print("ERROR: No se puede dividir entre cero.")

num1 = int(input("Ingrese el primer número:\t"))
num2 = int(input("Ingrese el segundo número:\t"))

Calculadora.suma(num1, num2)
Calculadora.resta(num1, num2)
Calculadora.dividir(num1, num2)
Calculadora.multiplicar(num1, num2)
```

CAPÍTULO 6. MÉTODOS ESPECIALES

Se ha estado trabajando con el método “init”, el cual se utiliza para iniciar un objeto cuando se crea una nueva instancia de alguna clase. Existen otros métodos con significados distintos, cuyos nombres se caracterizan por empezar y terminar con dos guiones bajos. Estos métodos especiales se utilizan para instalar comportamientos específicos en las clases de Python y se invocan por ciertas funciones de Python. A continuación, se verán los métodos más comunes.

6.1. `__NEW__(CLS, ARGS)`

Se utiliza para crear e iniciar una instancia de alguna clase. Es muy útil para personalizar la creación de instancias y controlar el número máximo de instancias que se pueden crear. Es un método estático que recibe como primer parámetro la clase y después los argumentos necesarios para crear la instancia.

```
class MiClase:
    def __new__(cls, *ar1, **ar2):
        print("Creando una nueva instancia de MiClase")
        instancia = super().__new__(cls)
        return instancia

    def __init__(self, argumento1, argumento2):
        print("Inicializando la instancia de MiClase")
        self.argumento1 = argumento1
        self.argumento2 = argumento2

# Creamos una instancia de la clase MiClase
mi_instancia = MiClase("valor1", "valor2")
```

En el ejemplo anterior, la definición del método imprime un mensaje que indica que se está creando una instancia de la clase. Luego, se usa el método para crear y devolver la instancia de la clase. La definición del método imprime el mensaje de que la instancia de la clase se está iniciando.

Como paréntesis, se puede observar que se hizo uso de la función “super()”. Esta función, incorporada en Python, se usa para ingresar a los métodos definidos en la superclase de una clase. Se usa de manera común en la definición de métodos de clases derivadas, para que puedan ingresar y llamar a los métodos de la superclase y extender su funcionalidad.

6.2. `__DEL__`(SELF)

Este método se utiliza para realizar la limpieza final y eliminar el objeto deseado. También se encarga de realizar tareas de limpieza adicionales, como cerrar archivos, liberar recursos de memoria, cerrar conexiones de red, entre otras. La sintaxis básica es la siguiente:

```
class MiClase:
    def __del__(self):
        # Código de limpieza final
```

6.3. `__STR__`(SELF)

El método “_str_” se usa para definir la creación de una cadena de texto que va a representar al objeto. La sintaxis básica es la siguiente:

```
class MiClase:
    def __str__(self):
        # Crear la representación en forma de cadena
        return "Cadena que representa al objeto"
```

A continuación, se verá un ejemplo del uso del método “str”:

```
class MiClase:
    def __init__(self, nombre, edad):
        self.nombre = nombre
```

```

        self.edad = edad

    def __str__(self):
        return f"{self.nombre} ({self.edad} años)"

# Creacion del objeto para MiClase
persona = MiClase("Gloria",22)
print(persona) # Salida Gloria (22 años)

```

En el ejemplo, se define la clase “MiClase” con los atributos de nombre y edad. La función “__str__()” se utiliza para definir la representación de un objeto de “MiClase” como una cadena de caracteres. En este caso, el método se convierte en una cadena que contiene el nombre y la edad del objeto persona. Posteriormente, se crea un objeto de “MiClase” con el nombre de “persona”, y se le asignan los parámetros de nombre y edad. Finalmente, se imprime el contenido del método “__str__”.

6.4. __EQ__()

Método que se utiliza para comparar dos objetos de la misma clase y determinar si son iguales; asimismo, se emplea en conjunto con el operador de igualdad. El método de la función devuelve un valor “true” o “false” dependiendo de si son o no iguales. A continuación, se muestra un ejemplo de aplicación de dicho método:

```

class Persona:
    def __init__(self, nombre,edad):
        self.nombre = nombre
        self.edad = edad

    def __eq__(self, other):
        return self.edad == other.edad

persona1 = Persona("Santiago", 25)
persona2 = Persona("Sofía", 23)

if persona1 == persona2:
    print("Santiago y Sofia tienen la misma edad")
else:
    print("Santiago y Sofia no tienen la misma edad")

```

En el ejemplo anterior, cuando hacemos la comparación utilizando el operador “==” entre los objetos “persona1” y “persona2”, inmediatamente se llama al método “__eq__()”. Debido a que las edades son diferentes, al entrar en la condición se imprimirá el mensaje negativo que indica que las edades no coinciden.

6.5. __NE__()

Método especial que se utiliza para comparar si dos métodos son diferentes. Se utiliza en conjunto con el “operador !=”. En caso de que ambos sean diferentes, el método devolverá “true”; de lo contrario, devolverá “false”. El ejemplo siguiente muestra la forma de funcionar del método.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __ne__(self, other):
        return self.edad != other.edad

persona1 = Persona("Santiago", 25)
persona2 = Persona("Sofía", 23)

if persona1 != persona2:
    print("Santiago y Sofia no tienen la misma edad")
else:
    print("Santiago y Sofia tienen la misma edad")
```

6.6. _LT_()

Este método especial se utiliza para hacer uso del operador de comparación *menor que* (<). La estructura de codificación es la siguiente:

```
def __lt__(self, other):
    #Código para comparar el objeto actual con otro
    # Debe devolver True o False
```

“Self” hace referencia al objeto actual, y “other” al objeto con el que se desea comparar. La definición de menor que es subjetiva y depende de la lógica de la clase. Pondremos un ejemplo: si tenemos una clase de números, el método “lt” podría comparar únicamente los valores numéricos de los objetos, aunque también se pueden realizar comparaciones de propiedades o atributos.

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __lt__(self, other):
        if self.x < other.x:
            return True
        elif self.x == other.x and self.y < other.y:
            return True

        else:
            return False

coordenada1 = Punto(9,5)
coordenada2 = Punto(5,6)
coordenada3 = Punto(1,4)
coordenada4 = Punto(5,8)

coor = [coordenada1, coordenada2, coordenada3, coordenada4]
ordenar = sorted(coor)

for i in coor:
    print(f"{i.x},{i.y}")
''' Resultado de la ejecución del código
(1,4)
(5,6)
(5,8)
(9,5) '''
```

En este ejemplo, la clase “Punto” representa una coordenada en un plano cartesiano. La función integrada “__lt__()” permite comparar dos objetos, que representan las coordenadas x e y . Si se va a la sexta línea de código, se puede ver que se realiza una comparación. Si la coordenada x del objeto actual (“self”) es menor que la coordenada x del objeto anterior (“other”), se devuelve “true”. En la siguiente comparación, se establece que si la coordenada x del objeto actual (“self”) es igual a la

coordenada x del objeto (“other”), se compara la coordenada, y para verificar si la coordenada y del objeto actual es menor que la del objeto anterior (“other”). Si ambas condiciones se cumplen, se devuelve “true”; de lo contrario, si no se cumple la condición, se devuelve “false”.

Al final, fuera de la clase, se crean cuatro objetos de coordenadas con diferentes valores de x e y . Estos objetos se guardan en la lista “Coor” para luego ordenar las coordenadas en orden creciente utilizando la función “sorted()”, primero por la coordenada x y luego por la coordenada y .

6.7. `__LE__()`

En Python, esta función reservada se aplica para hacer uso de *menor o igual que* (\leq). Su uso es común cuando se necesita utilizar el operador “menor o igual que” con un objeto de una clase determinada. Este método toma dos argumentos: “self” y “other”. La sintaxis, al igual que en otras funciones, es la siguiente: “self” es el objeto actual y “other” es el objeto con el cual se realiza la comparación. En el siguiente ejemplo, podemos ver cómo utilizar el método:

```
class Fraccion:

    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador

    def __le__(self, other):
        return self.numerador / self.denominador
           <= other.numerador / other.denominador

frac1 = Fraccion(1, 2)
frac2 = Fraccion(3, 4)

print(frac1 <= frac2) #True
```

Podemos observar que en la clase “Fracción”, dentro del método “le”, se realiza una comparación entre dos fracciones: “frac1” ($1/2$) y la fracción “frac2” ($3/4$), utilizando el operador “ \leq ”. El resultado de esta comparación es un valor *booleano*.

6.8. `__GT__()`

Esta función se llama cuando se compara un objeto con otro de la misma clase utilizando el operador de comparación *mayor que* (`>`). A continuación, se muestra un ejemplo de cómo se puede aplicar el método especial:

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def __gt__(self, other):
        return self.precio > other.precio

p1 = Producto("shampoo", 50 )
p2 = Producto("desodorante", 60)

print(p1>p2) #Salida: Falso
print(p2>p1) #Salida: True
```

En este ejemplo, la clase "Producto" contiene un método "gt" en el cual se comparan los precios de dos productos, retornando un valor *booleano* ("true" o "false") dependiendo de si un producto es más caro que otro.

6.9. `__GE__()`

Esta función reservada se llama cuando se compara si un objeto de la clase es mayor o igual que otro objeto de la misma u otra clase. A continuación, podemos ver un ejemplo del método "ge":

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def __ge__(self, other):
        return self.precio >= other.precio

p1 = Producto("shampoo", 50 )
```

```
p2 = Producto("desodorante", 60)

print(p1>=p2) #Salida: Falso
print(p2>=p1) #Salida: True
```

6.10. `__LEN__()`

Este es un método que se utiliza para conocer la longitud de una cadena de texto que proviene de un objeto. El valor que devuelve es de tipo entero. Se utiliza comúnmente con objetos que contienen listas, tuplas o diccionarios. Un ejemplo de su uso es cuando se llama a la función "`len(obj)`" en nuestro objeto.

```
class Mylista:

    def __init__(self, lista1):
        self.lista1 = lista1
    def __len__(self):
        return len(self.lista1)

lista1 = Mylista([5,4,7,8,9,6,1,0])
lista2 = Mylista("Hola")

print(f"la longitud de la lista 1 es de {len(lista1)}")      #salida:
8
print(f"la longitud de la lista 2 es de {len(lista2)}")      #salida:
4
```

Hay un sinfín de métodos especiales en Python que son importantes en la programación orientada a objetos, ya que permiten que las clases se comporten de manera más consistente en la sintaxis de Python. Sin embargo, solo mencionamos algunos de los más destacados y utilizados, ya que hablar de cada uno de ellos se sale del objetivo del capítulo.

CAPÍTULO 7.

MÁS MÉTODOS DE LISTAS, CADENAS Y DICCIONARIOS

7.1. INTRODUCCIÓN

En los capítulos anteriores nos enfocamos en ver los objetos tal como son: números *booleanos*, cadenas de texto, diccionarios, conjuntos, listas y tuplas. Ahora que se tienen más claras las características de cada uno de estos objetos, nos enfocaremos en este capítulo en mostrar algunos de los métodos más útiles de estos objetos en Python.

Un método puede considerarse como una función que pertenece a un objeto. Son palabras reservadas que se pueden utilizar para facilitar la vida a la hora de trabajar con colecciones. A continuación, se muestra un listado de los métodos más comunes y utilizados:

7.2. MÉTODOS DE LISTAS

len(objeto)

Consultar el tamaño de una lista devuelve un valor entero. Este método no es exclusivo de las listas, pues también se puede usar con cadenas, tuplas o diccionarios, entre otros.

```
lista = [1, 2, 3, 4, 5, 7]
len (lista) #nos devuelve un 6.
```

Lista.append(objeto)

Agrega un elemento al final de la lista.

```
lista.append(6) # lista ahora es [1, 2, 3, 4, 5, 7, 6]
```

Lista.insert(indice, objeto)

La función “insert” agregará un elemento en la posición dada. El primer argumento es el índice del elemento delante del cual se va a insertar, y el segundo argumento es el elemento que se insertará.

```
lista.insert(5, 6) # lista ahora es [1, 2, 3, 4, 5, 6, 7, 6]
```

Lista.extend(objeto)

Para agregar varios elementos al final de una lista, se concatenan los elementos que se insertarán con la lista existente.

```
Lista2 = [5,4,3]  
lista.extend(lista2) #lista ahora es [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3]
```

Lista.index(valor[, inicio[,fn]])

Se puede conocer en qué índice se encuentra dicho elemento, pero será el elemento que aparecerá primero en la lista.

```
lista = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3]  
lista.index(4) # El elemento 4 se encuentra en el índice 3
```

En el ejemplo pasado solo buscamos un valor específico en la lista; en el método, es opcional insertar un índice de inicio y un índice final.

```
indice = lista.index(5,0,6)  
print(indice) # El elemento 5 se encuentra en el índice 4
```

Lo que hace esta sentencia es buscar el primer índice donde aparece el valor 5 en la sublista que va desde el índice 0 hasta el índice 5 (ya que el índice 6 es exclusivo). Si encuentra el valor 5 en ese rango, devuelve el índice de la primera aparición de 5. Si no lo encuentra, lanza una excepción “ValueError”.

Lista.reverse()

Se invierten los elementos de una lista; trabaja sobre la misma lista y no sobre una copia.

```
lista = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3]
lista.reverse() # lista ahora es [3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1]
```

Lista.sort()

Ordena los elementos de una lista de manera ascendente, ya sean números o caracteres.

```
lista = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3]
lista.sort() # lista ahora es [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7]
```

Listas.count(valor)

Da como resultado el número de veces que cierto valor aparece en la lista.

```
lista = [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7]
lista.count(6) # El número 6 aparece 2 veces
```

Nueva_lista = lista.copy()

Retornar una copia de la lista; se puede usar una variable auxiliar para almacenar la copia.

```
lista1 = lista.copy() # lista1 es [1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7]
```

lista.remove(valor)

Quita el primer elemento de la lista cuyo valor sea x . Sí el elemento no se encuentra, muestra un mensaje de error.

```
lista.remove(4) #lista ahora es [1, 2, 3, 3, 4, 5, 5, 6, 6, 7]
```

lista.pop(indice)

Quita el elemento en la posición dada de la lista. Si no existe un índice específico, “`lista.pop()`”, por ende, remueve el último elemento de la lista.

```
lista.pop() #lista ahora es [1, 2, 3, 3, 4, 5, 5, 6, 6]
lista.pop(3) # lista ahora es [1, 2, 3, 4, 5, 5, 6, 6]
```

lista.clear()

Elimina todos los elementos de la lista.

```
lista.clear() #equivale a del lista[:] lista ahora es []
```

7.3. MÉTODOS DE CADENAS

c.upper()

Este método especial convierte todos los caracteres de la cadena en mayúsculas.

```
cadena = "Hola"
print(cadena.upper()) #Salida: HOLA
```

c.lower()

Convierte todos los caracteres de una cadena en minúsculas.

```
cadena = "Hola"
print(cadena.lower()) #Salida: hola
```

c.capitalize()

Convierte la primera letra de la cadena a mayúscula.

```
cadena = "hola mundo"
print(cadena.capitalize()) #Salida: Hola mundo
```

c.title()

Convierte cada inicial de una palabra de la cadena en mayúsculas.

```
cadena = "hola mundo"
print(cadena.title()) #Salida: Hola Mundo
```

c.split()

Divide la cadena en una lista con subcadenas; los espacios en blanco son, por defecto, los separadores.

```
cadena = "Hola mundo"
print(cadena.split()) #Salida: ['Hola', 'mundo']
```

c.replace(concurrencia, remplazo)

Remplaza todas las coincidencias que existen entre la cadena principal y la subcadena por otra.

```
cadena = "Hola Mundo"
print(cadena.replace("Hola", ¿Como les va?")) #Salida: ¿Como les va?
Mundo
```

c.find(cadena)

Devuelve la posición de la primera coincidencia que encuentre en la cadena principal con respecto a la subcadena. Si no se encuentra coincidencia alguna, nos devuelve un 0 o -1.

```
cadena = "Hola mundo"
print(cadena.find("a")) #Salida: 3
```

c.count(cadena)

Nos devuelve el número de veces que una subcadena aparece en la cadena con la que se hace la comparación.

```
cadena = "Hola mundo!"
print(cadena.count("o")) #Salida: 2
```

join(c)

Se utiliza para concatenar todos los elementos que contiene una cadena en una sola cadena y los separa con un carácter que funge como separador.

```
cadena = "Hola"
print("*".join(cadena)) #Salida: H*o*l*a
```

c.partition(separador)

Esta función especial se usa para separar una cadena en dos partes; busca el separador, que es en sí el primer espacio entre dos palabras, y devuelve una tupla con la primera parte de la cadena, el separador y la parte después de la cadena hasta el final. Si no encuentra separador alguno, la tupla contiene la cadena original, además de dos cadenas vacías. Veamos dos ejemplos.

```
cadena = "¡Hola mundo!"
print(cadena.partition(" ")) #Salida: ('¡Hola', ' ', 'mundo!')
```

```
cad= 'No_hay_separador'
```

```
partes = cad.partition(" ")
print(partes) #Salida: ('No_hay_separador', ' ', '')
```

7.4. MÉTODOS DE DICCIONARIOS

d.clear()

Este método no es exclusivo de las listas ya que también se puede usar en diccionarios para eliminar todos los elementos que contiene.

```
diccionario = {"azul": "blue", "rojo": "red", "verde": "green"}
diccionario.clear() #Salida: None
```

d.copy()

El método “copy()” se usa para copiar un diccionario, por ende, crea un nuevo diccionario con las mismas claves y valores. Hay que tener en cuenta que si se modifica un elemento en el diccionario original, se verán afectados ambos diccionarios. En el caso de la copia, si se modifica un valor, afectará al original solo si se modifican valores mutables dentro del diccionario, como lo son listas o diccionarios. Con los valores inmutables, el diccionario original no se ve afectado. A continuación se muestran algunos ejemplos del método “copy()”:

```
#Se crea el diccionario
diccionario = {'nombre': 'Daniela', 'edad': 24, 'ciudad': 'San Luis Potosí'}

#copiamos el diccionario
copia = diccionario.copy()

#Imprimimos copia y original
print(f"Original\t{diccionario}")

#Salida: Original {'nombre': 'Daniela', 'edad': 24, 'ciudad': 'San Luis Potosí'}
print(f"Copia\t{copia}")
#Salida: Copia {'nombre': 'Daniela', 'edad': 24, 'ciudad': 'San Luis Potosí'}
```

En el siguiente código se plasma el ejemplo de cómo al modificar un elemento mutable en la copia de un diccionario se afecta al original:

```
#Se crea el diccionario con el valor mutable
diccionario = {'nombres': ['Daniela', 'Abigail'], 'edad': 24, 'ciudad':
'San Luis Potosí'}

#copiamos el diccionario
copia = diccionario.copy()

#Modificamos el valor en la copia
copia['nombres'].remove("Abigail")

#Imprimimos copia y original
print(f"Original\t{diccionario}")
#Salida: Original {'nombres': ['Daniela'], 'edad': 24, 'ciudad': 'San
Luis Potosí'}
print(f"Copia\t{copia}")
#Original: Copia {'nombres': ['Daniela'], 'edad': 24, 'ciudad': 'San
Luis Potosí'}
```

fromkeys(seq[, v])

Este método sirve para crear diccionarios con un conjunto de claves definidas y predeterminadas. La palabra *seq* está en la posición en donde se permite especificar valores de diccionarios, mientras que en la posición del argumento de la letra *v* se establece el valor predeterminado para las claves del diccionario. Una forma de usar este método es primero estableciendo una lista, tupla o conjunto con las claves y después hacer uso de la palabra reservada *dict*, que sirve para manipular los diccionarios:

```
#Creamos la lista con las claves
claves = ['nombre', 'edad', 'dirección']

#Creamos diccionario con los valores predeterminados
persona = dict.fromkeys(claves, 'anónimo')

#Imprimimos el diccionario resultante
print(persona) #Salida: {'nombre': 'anónimo', 'edad': 'anónimo',
'dirección': 'anónimo'}
```

d.get()

Este método devuelve la clave del valor asociado a ella. Si la clave existe, devuelve el valor predeterminado; en caso de que no sea, así devolverá el valor de “none”. En el siguiente ejemplo podemos ver cómo usar el método “get”:

```
#creación del diccionario
datos_personales = {'nombre': 'Daniela', 'edad': 24, 'ciudad': 'San
Luis Potosí'}

#Obtenemos la clave asociada
clave = datos_personales.get('ciudad')

#Imprimimos el valor para ver si existe
print(clave) #Salida: San Luis Potosí
```

d.items()

Este método es utilizado para tener una vista de los elementos pares de un diccionario de la forma (clave, valor). El siguiente ejemplo es un diccionario que contiene la lista de tres alumnos en conjunto con sus matrículas, con ayuda del método “items” se puede ver la lista completa del diccionario dentro de la estructura “dict_items()”:

```
#Se crea el diccionario
alumnos = {'Daniela': '180485', 'Fátima': '150180', 'Alberto':
'171214'}

#Mostramos los Items
elementos = alumnos.items()

#Imprimimos la lista
print(elementos)
#Salida: dict_items([('Daniela', '180485'), ('Fátima', '150180'),
('Alberto', '171214')])
```

d.keys()

Este método devuelve un objeto “dict_keys”, el cual es una vista de todas las claves del diccionario.

```

#Se crea el diccionario
estudiante = {"nombre": 'daniela', "matricula": '180485', "edad": 19,
"carreira": 'Licenciatura'}

#Mostramos las claves
ver_claves = estudiante.keys()

#imprimimos las claves
print(ver_claves)
#Salida: dict_keys(['nombre', 'matricula', 'edad', 'carreira'])

```

El objeto que se devuelve podemos convertirlo a una lista usando la función "list":

```

#Se crea el diccionario
estudiante = {"nombre": 'daniela', "matricula": '180485', "edad": 19,
"carreira": 'Licenciatura'}

#Mostramos las claves
lista_claves = list(estudiante.keys())

#imprimimos las claves
print(lista_claves)
#Salida: ['nombre', 'matricula', 'edad', 'carreira']

```

diccionario.pop(key[,d])

Al igual que en listas, se puede usar este método especial en diccionarios, precisamente para eliminar un elemento del diccionario según la clave especificada. En caso de que la clave no exista, se puede especificar un valor predeterminado que se devolverá en su lugar.

```

#Se crea el diccionario
estudiante = {"nombre": 'Daniela', "matricula": '180485', "edad": 19,
"carreira": 'Licenciatura'}

#eliminamos una clave
clave = estudiante.pop("carreira")

#imprimimos la lista sin la clave eliminada
print(estudiante)
#Salida: {'nombre': 'Daniela', 'matricula': '180485', 'edad': 19}

```

En la siguiente parte del código, lo que se hace es intentar eliminar un elemento que no existe en el diccionario; especificamos el valor de “no existe”, que se almacena en la variable `carrera`, al final se imprime el valor en consola.

```
#especificamos un valor predeterminado
Buscar_valor = "No existe"

carrera = estudiante.pop("carrera", buscar_valor)

print(carrera) #Salida: No existe
```

diccionario.popitem()

En este método se elimina el último elemento del diccionario y devuelve entre paréntesis el par de (clave, valor). Si el diccionario está vacío, marcará un error de tipo “`KeyError`”.

```
#Se crea el diccionario
estudiante = {"nombre": 'Daniela', "matricula": '180485', "edad": 19,
"carreira": 'Licenciatura'}

#Variable que almacena el par eliminado
clave = estudiante.popitem()

#Imprimimos el valor eliminado
print(clave) #Salida: ('carrera', 'Licenciatura')
```

diccionario.setdefault(key[,d])

Dicho método ayuda a buscar algún valor asociado a alguna clave. Si la clave no existe, el método puede agregar una clave con un valor especificado, como ejemplo:

```
#Se crea el diccionario
estudiante = {"nombre": 'Daniela', "matricula": '180485', "edad": 19,
"carreira": 'Licenciatura'}

#Obtenemos un valor asociado a la clave nombre
nombre = estudiante.setdefault("nombre", "desconocido")
print(nombre) #Salida: Daniela
```

```

#Obtenemos el valor asociado a la clave "semestre" y se agrega al
diccionario
semestre = estudiante.setdefault("semestre", "3")
print(semestre) #Salida: 3
#Imprimimos el diccionario
Print(estudiante)
#Salida: {'nombre': 'Daniela', 'matricula': '180485', 'edad': 19,
'carrera': 'Licenciatura', 'semestre': '3'}

```

diccionario.update()

El método "update" se usa para actualizar un diccionario con las claves y valores de otro diccionario. En el ejemplo siguiente, a la variable "diccionario" es a la que se le van agregar los pares de clave-valor.

```

#Creamos el 1er diccionario
diccionario = {'a':1, 'b':2, 'c':3}

#Creación del 2do diccionario
dic2 = {'d':4, 'e':5}

#Agregamos los valores del dic2 al diccionario
diccionario.update(dic2)
print(diccionario)
#Salida: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

```

Algo que se debe tomar en cuenta es que si la clave ya existe en el diccionario original, el valor se va a actualizar con el valor que ya esté en el segundo diccionario.

diccionario.values()

Esta función devuelve una lista de los valores de un diccionario. Al tratarse de un objeto iterable, se puede ingresar a los valores almacenados en el diccionario.

```

#Creamos el 1er diccionario
diccionario = {'a':1, 'b':2, 'c':3}

#En la variable valores se agregan los valores del diccionario
valores = diccionario.values()

#Imprimir los valores

```

```
print(valores) #Salida: dict_values([1, 2, 3])

#Iteramos tupla e imprimimos sus valores
for valor in valores:
    print(valor) #Salida: 1 2 3
```

En Python existen otros métodos especiales que se distinguen por su formato de dos guiones bajos al principio y al final del nombre. Estos métodos únicos se emplean para aplicar comportamientos específicos dentro de las clases y son invocados por funciones particulares de Python. La utilización de métodos especiales en Python nos brinda la capacidad de dotar nuestras clases con comportamientos específicos, lo que enriquece nuestras capacidades de programación al permitir una personalización detallada y una interacción más intuitiva con nuestros objetos.

7.5. EJERCICIOS DEL CAPÍTULO

Como es ampliamente conocido, los métodos especiales en Python son funciones que pertenecen a los objetos y brindan la capacidad de personalizar su comportamiento en diferentes contextos. En esta sección nos sumergiremos en una serie de ejercicios relacionados con métodos especiales. Como se ha mencionado en capítulos anteriores, se recomienda intentar resolver los ejercicios de manera independiente antes de revisar las soluciones proporcionadas. De esta manera, podrá fortalecer sus habilidades y comprensión de los métodos especiales en Python.

1. Crear una clase “Punto” que represente un punto en un plano cartesiano. Aplique los métodos especiales “init”, “str”, “eq” y “add” para que pueda iniciar un punto, mostrarlo como una cadena de texto, comparar si dos puntos son iguales y sumar dos puntos, respectivamente.
2. Defina una clase “Caja” que represente una caja rectangular. Aplique los métodos especiales “init”, “str”, “eq”, “lt” y “gt” para que pueda iniciar una caja, mostrarla como una cadena de texto, comparar si dos cajas son iguales y determinar si una caja es menor o mayor que otra en función de su volumen.

3. Cree una clase “ListaOrdenada” que represente una lista que mantenga sus elementos ordenados de menor a mayor. Aplique los métodos especiales “init”, “str”, “contains”, “len”, “getitem”, “setitem”, “delitem” e “iter” para que pueda iniciar la lista ordenada, mostrarla como una cadena de texto, verificar si un elemento está presente en la lista, obtener la longitud de la lista, ingresar a los elementos mediante índices, eliminar elementos por índice e iterar sobre la lista.

7.6. SOLUCIÓN DE EJERCICIOS

- 1 El planteamiento del problema sugiere qué métodos utilizar para dar solución al presente. A continuación, se enumeran los pasos con la explicación del código:
 - 1.1 Se define la clase “Punto” para representar un punto en un plano cartesiano. El método “init” se encarga de iniciar las coordenadas x e y del punto.
 - 1.2 El método “str” devuelve una cadena de texto que representa las coordenadas del punto.
 - 1.3 El método “eq” compara si dos puntos son iguales, verifica si las coordenadas x e y de ambos puntos son iguales.
 - 1.4 El método “add” sobrecarga el operador de suma (+) para permitir sumar dos puntos. Realiza la suma de las coordenadas x e y de los puntos y devuelve la suma resultante.
 - 1.5 Se crean dos instancias de la clase “Punto” llamadas “coor1” y “coor2”, con diferentes coordenadas como parámetros.
 - 1.6 Se compara si los puntos “coor1” y “coor2” son iguales utilizando el operador de igualdad (==). Se imprime un mensaje indicando si los puntos son iguales o diferentes.
 - 1.7 Se realiza la suma de los puntos “coor1” y “coor2” utilizando el operador de suma (+). El resultado se almacena en la variable “suma_puntos”.
 - 1.8 Finalmente, se imprime la suma de las coordenadas utilizando el valor de “suma_puntos”.

```
class Punto:
```

```

def __init__(self, x, y):
    self.x = x
    self.y = y

def __str__(self):
    return f"Coordenada X -->
{self.x}\nCoordenada Y --> {self.y}"

def __eq__(self, other):
    return self.x == other.x and self.y == other.y

def __add__(self, other):
    sum_x = self.x + other.x
    sum_y = self.y + other.y
    return sum_y + sum_x

coor1 = Punto(5,9)
coor2 = Punto(9,6)

print(coor1)
print(coor2)

if coor1 == coor2:
    print("Los puntos son iguales")
else:
    print("Los puntos son diferentes")

suma_puntos = coor1 + coor2
print(f"Suma de coordenadas: {suma_puntos}")

```

- 2 Este ejercicio también plantea los métodos especiales para dar solución al problema. A continuación, se enumeran los pasos con la explicación del código:
 - 2.1 Se utiliza el método especial “init” para iniciar los atributos de la caja, que son largo, ancho y alto.
 - 2.2 El método “str” devuelve una cadena de texto que representa las características de la caja, mostrando sus dimensiones.
 - 2.3 El método “eq” compara si el volumen de ambas cajas es igual, utilizando el método “volumen()” en ambas instancias de la clase.
 - 2.4 El método “lt” compara si el volumen de una caja es menor al de la otra, utilizando el método “volumen()” en ambas instancias de la clase.

- 2.5 Por el contrario, al método anterior ahora se busca comparar cuál volumen de la caja es mayor. Se aplica el método “gt” usando el método “volumen()”.
- 2.6 El método “volumen()” calcula y devuelve el volumen de la caja, multiplicando sus dimensiones.
- 2.7 Se crean dos instancias de la clase “Caja” mediante la declaración de variables y asignando las dimensiones correspondientes.
- 2.8 Se imprime la representación en texto de cada caja, mostrando sus dimensiones.
- 2.9 Se utiliza una estructura condicional “if” para comparar si ambas cajas son iguales. Si son iguales, se imprime un mensaje indicando que las cajas son iguales. De lo contrario, se imprime un mensaje indicando qué caja es mayor en tamaño.

```

class Caja:
    def __init__(self, largo, ancho, alto):
        self.largo = largo
        self.ancho = ancho
        self.alto = alto

    def __str__(self):
        return f"Dimensiones:\nLargo: {self.largo}\nAncho: {self.
ancho}\nAlto: {self.alto}"

    def __eq__(self, other):
        return self.volumen() == other.volumen()

    def __lt__(self, other):
        return self.volumen() < other.volumen()

    def __gt__(self, other):
        return self.volumen() > other.volumen()

    def volumen(self):
        return self.largo * self.ancho * self.alto

caja1 = Caja(5, 7, 8)
caja2 = Caja(8, 9, 1)

print("--> Caja No.1 <--")
print(caja1)
print("--> Caja No.2 <--")
print(caja2)

```

```
if caja1 == caja2:
    print("Ambas cajas son iguales")
elif caja1 < caja2:
    print("La caja 1 es menor que la caja 2")
else:
    print("La caja 1 es mayor que la caja 2")
```

- 3 Este ejercicio también plantea los métodos que se pueden utilizar para resolver el problema, solo es cuestión de saber cómo aplicarlos. A continuación, se enumeran los pasos seguidos para dar solución a dicho problema.
 - 3.1 Se define la clase “ListaOrdenada” con un método constructor que recibe una lista opcional como parámetro. Si no se proporciona una lista, se crea una lista vacía; de lo contrario, se ordena la lista proporcionada.
 - 3.2 El método “str” devuelve la representación de la lista ordenada como una cadena de texto.
 - 3.3 El método “contains” verifica si un valor dado está presente en la lista ordenada.
 - 3.4 El método “len” devuelve la longitud de la lista ordenada.
 - 3.5 El método “getitem” permite acceder a los elementos de la lista ordenada mediante el operador de indexación.
 - 3.6 El método “setitem” permite asignar un valor a un índice específico de la lista ordenada. Luego, se vuelve a ordenar la lista.
 - 3.7 El método “delitem” permite eliminar un elemento de la lista ordenada mediante el operador “del”.
 - 3.8 El método “iter” devuelve un iterador de la lista ordenada.
 - 3.9 Se crea una instancia de “ListaOrdenada” llamada “mi lista” con una lista dada.
 - 3.10 Se imprime “mi lista” utilizando el método “str”.
 - 3.11 Se verifica si un número dado está presente en “mi lista” utilizando el operador “in”. Se imprime un mensaje para indicar si el número está o no en la lista.
 - 3.12 Se obtiene la longitud de “mi lista” utilizando el método “len”.
 - 3.13 Se imprime la longitud de la lista.

- 3.14 Se elimina un elemento de “mi lista” mediante el operador “del”.
- 3.15 Se imprime la lista actualizada después de eliminar el elemento.
- 3.16 Se itera sobre “mi lista” e imprime cada elemento en una línea separada.

```
class ListaOrdenada:
    def __init__(self, lista=None):
        if lista is None:
            self.lista = []
        else:
            self.lista = sorted(lista)

    def __str__(self):
        return f"La lista ordenada es: {self.lista}"

    def __contains__(self, valor):
        return valor in self.lista

    def __len__(self):
        return len(self.lista)

    def __getitem__(self, indice):
        return self.lista[indice]

    def __setitem__(self, indice, valor):
        self.lista[indice] = valor
        self.lista.sort()

    def __delitem__(self, indice):
        del self.lista[indice]

    def __iter__(self):
        return iter(self.lista)

lista = [4, 8, 9, 1, 45, 6, 7, 8, 9, 60]
mi_lista = ListaOrdenada(lista)
print(mi_lista) #Mostrar como cadena de texto

#Verificar si un elemento está en la lista
numero = 4
estado = numero in mi_lista

if estado:
    print(f"El número {numero} está en la lista")
else:
    print(f"El número {numero} NO está en la lista")
```

```
#Obtener longitud de la lista
longitud = len(mi_lista)
print(f"La longitud de la lista: {longitud}")

#Acceder a los elementos mediante indices
indice = 1
del mi_lista[indice]
print(f"La lista sin el elemento del índice {indice}:")
print(mi_lista)

#iteracion de la lista
print("Iteración sobre la lista:")
for elemento in mi_lista:
    print(elemento, end=" ")
```

CAPÍTULO 8. INTRODUCCIÓN A MESA

8.1. INTRODUCCIÓN A MESA

MESA es una plataforma de código abierto que facilita la creación de modelos basados en agentes, lo que significa que se puede simular sistemas complejos donde los agentes individuales interactúan entre sí y con su entorno. Su objetivo es ser la contraparte basada en Python 3 de NetLogo, Repast y MASON.

MESA brinda a los usuarios la capacidad de generar de manera ágil modelos que se fundamentan en agentes, ya sea mediante el uso de elementos centrales preexistentes, como rejillas espaciales y programadores de agentes, o mediante personalizaciones específicas. Además, ofrece la posibilidad de observar estos modelos a través de una interfaz *web* y llevar a cabo análisis de sus resultados utilizando las herramientas de análisis de datos disponibles en Python.

Los siguientes puntos abordan información adicional sobre la interfaz de MESA:

- *Componentes incorporados y personalización:* MESA ofrece componentes centrales predefinidos, como rejillas espaciales y planificadores de agentes, que permiten a los usuarios crear modelos rápido. Sin embargo, también es altamente personalizable, lo que significa que se pueden adaptar y extender estos componentes según sus necesidades específicas.
- *Interfaz basada en navegador:* proporciona una interfaz basada en navegador que facilita la visualización de sus modelos y su ejecución. Esto hace que sea fácil observar cómo se desarrollan sus simulaciones y cómo interactúan los agentes en tiempo real.

- *Análisis de datos con Python:* MESA se integra a la perfección con las herramientas de análisis de datos de Python, lo cual permite utilizar bibliotecas como Pandas, Matplotlib y NumPy para analizar y visualizar los resultados de sus simulaciones, de manera eficiente.
- *Alternativa a otras plataformas:* MESA se presenta como una alternativa en Python 3 para otras herramientas de modelado basado en agentes populares, como NetLogo y MASON, los cuales se basan en Java y Repast basado en C++. Ofrece a los usuarios de Python la capacidad de crear modelos de agentes sofisticados y realizar análisis detallados sin tener que aprender un nuevo lenguaje de programación o entorno de modelado.

MESA es una herramienta poderosa para crear, visualizar y analizar modelos basados en agentes en el entorno de programación de Python. Su flexibilidad y capacidad de personalización lo hacen adecuada tanto para principiantes como para usuarios avanzados que deseen abordar problemas de simulación y modelado en ciencias sociales, economía o biología, entre otros campos.

8.2. ASPECTOS BÁSICO DE MESA

Como se ha visto, la plataforma de desarrollo de modelos basada en agentes MESA está diseñada para la creación, análisis y visualización de modelos de agentes en Python. Aquí está cómo funciona MESA, estructurado de forma esquemática:

1. Agentes:
 - Definición de clases: Los agentes se definen como clases de Python con sus propios atributos y métodos.
 - Comportamiento: Cada agente tiene un método *step* que encierra su comportamiento, ejecutado en cada paso de la simulación.
2. Ambiente/espacio:
 - Tipos de espacios: MESA permite diferentes tipos de entornos, como espacios de cuadrícula o continuos.

- Interacción: Los agentes pueden moverse o interactuar con otros agentes o con el espacio.
3. Modelo:
 - Contenedor principal: El modelo engloba agentes y el espacio, controlando la creación, interacción y eliminación de agentes.
 - Método *step*: El modelo tiene un método *step* que avanza la simulación, generalmente llamando al método *step* de cada agente.
 4. Tiempo:
 - Tiempo discreto: MESA trabaja con tiempo discreto, avanzando en pasos (*ticks*).
 - Programación de eventos: La plataforma puede gestionar eventos basados en tiempo a través de la planificación de agentes o acciones.
 5. Recopilación de datos:
 - Recolección: MESA tiene componentes para recopilar datos de los agentes y del modelo en cada paso.
 - Análisis: Los datos pueden ser analizados o visualizados durante o después de la simulación.
 6. Visualización:
 - Servidor: MESA proporciona un servidor para visualizar modelos en un navegador *web*.
 - Módulos: Usa módulos para representar visualmente los agentes y el espacio de forma interactiva.
 7. Interacción y experimentación:
 - Parámetros: Permite la modificación de parámetros para experimentar con diferentes configuraciones del modelo.
 - Análisis de sensibilidad: Facilita la realización de análisis de sensibilidad y experimentos de Monte Carlo.

8.3. RECURSOS NECESARIOS

Para utilizar MESA, los recursos necesarios se pueden dividir en varias categorías:

8.3.1. *Software*

- Python: MESA es una biblioteca de Python, por lo que es imprescindible tener Python instalado. Por lo general, se requiere una versión reciente de Python.
- Biblioteca MESA: Se debe instalar MESA, lo cual se hace típicamente a través de pip, el gestor de paquetes de Python (“pip install mesa”).

8.3.2. *Hardware*

- Computadora: Cualquier computadora capaz de ejecutar Python y sus bibliotecas asociadas.
- Memoria y procesamiento: La capacidad necesaria dependerá del tamaño y complejidad de los modelos. Los modelos más grandes y complejos con miles de agentes pueden requerir más recursos de memoria y una mayor capacidad de procesamiento. Por lo regular, una computadora con un procesador de 6 núcleos y 32 GB de RAM es suficiente para modelos grandes de miles de agentes.

8.3.3. *Conocimientos*

- Programación en Python: Es esencial poseer conocimiento de programación en Python, ya que MESA se utiliza a través de este lenguaje.
- Teoría de sistemas complejos y modelado basado en agentes: Entender los principios básicos detrás del modelado basado en agentes y cómo los sistemas complejos pueden ser modelados de esta manera.

8.3.4. *Dependencias de Python*

- Bibliotecas auxiliares: MESA tiene varias dependencias que se deben instalar de forma manual con *pip*, como NumPy para el manejo de *arrays* y operaciones matemáticas, y Pandas para la manipulación y análisis de datos y archivos de Excel y *cvs*.
- Visualización: Para la visualización de modelos, es posible que se necesiten bibliotecas adicionales de Python como Matplotlib o in-

cluso herramientas de JavaScript si se desea una visualización más interactiva en la red.

8.3.5. *Desarrollo y pruebas*

- Entorno de desarrollo: Un entorno de desarrollo integrado (IDE) para escribir y probar el código, como IDLE, PyCharm, Visual Studio Code, o Jupyter Notebooks.
- Control de versiones: Herramientas de control de versiones como Git pueden ser útiles para gestionar los cambios en el código del modelo a medida que se desarrolla y prueba.

8.3.6. *Recursos de aprendizaje y documentación*

- Documentación de MESA: Ingresar a la documentación oficial de MESA para entender cómo utilizar la biblioteca y sus diferentes módulos. Los recursos de documentación para MESA se encuentran en la plataforma ReadTheDocs. Puede entrar a la documentación completa de MESA para el modelado basado en agentes en Python a través de la siguiente URL: <https://mesa.readthedocs.io>. Aquí encontrará información sobre la instalación de la biblioteca, ejemplos de cómo construir modelos, guías para visualizar y analizar los resultados de la simulación y referencias para el uso de los componentes centrales de MESA.
- Tutoriales y ejemplos: Estudiar ejemplos de modelos y tutoriales para aprender las mejores prácticas en el modelado basado en agentes con MESA.

Estos recursos proporcionan la base para comenzar a trabajar con MESA en el desarrollo de modelos basados en agentes.

8.4. UN EJEMPLO DEL USO DE MESA

Un ejemplo del uso de MESA sería modelar la dinámica de opinión en una sociedad artificial. Un modelo de dinámica de opinión basado en agentes es un tipo de modelo computacional utilizado en las ciencias sociales computacionales para simular y analizar cómo las opiniones

de individuos o agentes cambian y se distribuyen a lo largo del tiempo dentro de una población o sistema. Estos modelos son particularmente útiles para estudiar fenómenos sociales complejos, como la formación de consenso, la polarización de opiniones o la difusión de información y creencias. El modelo que se presenta a continuación simulará la difusión de opiniones entre agentes en una cuadrícula de dos dimensiones utilizando el modelo de Ising como base. Explicaremos cómo se calcula la interacción entre agentes y el cambio de opinión en términos conceptuales y con ecuaciones.

Conceptualmente, el modelo de Ising se utiliza para describir fenómenos en sistemas físicos, como la magnetización de materiales ferromagnéticos. En este contexto, se ha adaptado para representar la difusión de opiniones en una cuadrícula de agentes. La idea principal es que cada agente tiene una opinión, que puede ser a o b , y puede cambiar su opinión a lo largo del tiempo en función de la interacción con sus vecinos.

Los elementos clave en este modelo son las siguientes:

1. Vecinos: Se determina la vecindad de Von Neumann de un agente. La vecindad de Von Neumann se define como el conjunto de celdas que son adyacentes a una celda dada en las direcciones norte, sur, este y oeste. Es importante destacar que no incluye las celdas diagonales.
2. Energía: Se calcula la energía de interacción entre el agente y sus vecinos. En el modelo de Ising, la energía depende de la diferencia de opiniones entre el agente y sus vecinos. Si un vecino tiene la misma opinión que el agente, la energía es baja (favorable), y si tienen opiniones diferentes, la energía es alta (desfavorable).
3. Parámetro beta (β): Este parámetro controla la influencia de los vecinos en la decisión de cambiar de opinión. Un valor más alto de β significa que los agentes son más influenciados por sus vecinos.
4. Probabilidad de cambio de opinión: Finalmente, se calcula la probabilidad de que el agente cambie de opinión. Esto se hace mediante la siguiente fórmula:

$$P(\text{cambio de opinión}) = \frac{1}{1 + e^{-2\beta\Delta E}}$$

Donde:

- $(P(\text{cambio de opinión}))(\text{cambio de opinión})$ es la probabilidad de que el agente cambie de opinión.
- (β) es el parámetro que controla la influencia de los vecinos.
- (ΔE) es la diferencia de energía entre la situación actual del agente y una en la que cambia de opinión.

Si la probabilidad de cambio de opinión es mayor que un número aleatorio entre 0 y 1, el agente cambia su opinión. Si no, conserva su opinión actual.

El código que hemos usado se describe a continuación. En primer lugar, se deben especificar las bibliotecas y los parámetros de MESA que se emplearán.

```
import random
import socket
import math
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
```

Nótese que se está importando las bibliotecas “Random”, “Socket” y “Math” para el correcto uso de los métodos de aleatorización, el manejo de la interfase gráfica que se desplegará usando el *browser* que tenga usted como predeterminado y el uso de funciones matemáticas como el número e .

Después, es necesario comenzar a definir las clases que componen el modelo. En primer lugar, se define la clase modelo con el siguiente código:

```
# Definición de la clase del modelo
class DifusionOpinionesModel(Model):
    def __init__(self, width, height, num_agents):
        self.num_agents = num_agents
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)

    # Crear una lista de todas las coordenadas posibles
    all_coords = [(x, y) for x in range(width)
                  for y in range(height)]
```

```

# Mezclar la lista de coordenadas para
un muestreo sin reemplazo
random.shuffle(all_coords)

# Crear agentes y asignarles opiniones aleatorias
for i in range(self.num_agents):
    if len(all_coords) > 0:
        x, y = all_coords.pop()
        opinion = random.choice(['A', 'B'])
        # Dos opiniones posibles
        agent = DifusionOpinionesAgent((x, y), self, opinion)
        self.grid.place_agent(agent, (x, y))
        self.schedule.add(agent)

def step(self):
    self.schedule.step()

```

Este procedimiento crea la clase “DifusionOpinionesModel()”. En dicha clase, se crean los agentes, se colocan cada uno sobre un enrejado bidimensional de forma que no haya traslapes y se les asignan opiniones aleatorias que pueden ser *a* o *b*.

El siguiente código sirve para la definición de la clase agentes. En dicha clase se definen las reglas de interacción y la probabilidad de cambio de opinión:

```

# Definición de la clase de agentes
class DifusionOpinionesAgent(Agent):
    def __init__(self, pos, model, opinion):
        super().__init__(pos, model)
        self.opinion = opinion

def step(self):
    # Implementa las reglas de interacción entre
    agentes basadas en el modelo de Ising aquí
    neighbors = self.model.grid.get_neighbors
    (self.pos, moore=False, include_center=False)
    neighbor_opinions = [neighbor.opinion for neighbor
    in neighbors]
    my_opinion = self.opinion

    # Calcula la probabilidad de cambiar de opinión
    según el modelo de Ising
    beta = 0.5 # Parámetro que controla la influencia
    de los vecinos

```

```

energy = 2.0 * sum([1 if neighbor_opinion !=
my_opinion else -1 for neighbor_opinion in
neighbor_opinions])
prob_change = 1.0 / (1.0 + math.exp((-2.0 * beta * energy)))

# Cambia la opinión con base en la probabilidad
if random.random() < prob_change:
    self.opinion = 'A' if my_opinion == 'B' else 'B'

```

Nótese que aquí se definen el parámetro β y la ΔE . La vecindad que se está usando es la de Von Neumann.

El siguiente código sirve para la visualización en el navegador (*browser*) que tenga usted predeterminado para navegar por internet:

```

# Definición de la representación visual de los agentes
def agent_portrayal(agent):
    # Define cómo se mostrarán los agentes en la interfaz gráfica
    portrayal = {"Shape": "circle",
                "Color": "red" if agent.opinion == 'A' else 'blue',
                "Filled": "true",
                "Layer": 0,
                "r": 0.5}
    return portrayal

# Configuración de la visualización
width = 10
height = 10
grid = CanvasGrid(agent_portrayal, width, height, 500, 500)

# Configuración del servidor de simulación
server = ModularServer(
    DifusionOpinionesModel,
    [grid],
    "Difusion de Opiniones",
    {"width": width, "height": height, "num_agents": 50}
)

# Establecer el puerto para la interfaz web y lanzar la simulación
server.port = 8521
server.launch()

```

Nótese que el tamaño del enrejado es de 10×10 y se crean solo cincuenta agentes. La interfaz que se crea se muestra a continuación:

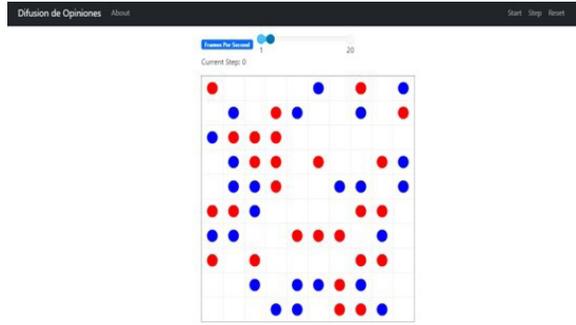


FIGURA 8.1. INTERFAZ GRÁFICA DE MESA.

A continuación, se mostrarán algunas instantáneas de la evolución del sistema.

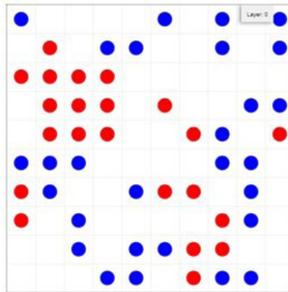


FIGURA 8.2. PRIMERA ITERACIÓN DEL MODELO DE OPINIÓN BASADO EN LA DINÁMICA DE ISING.

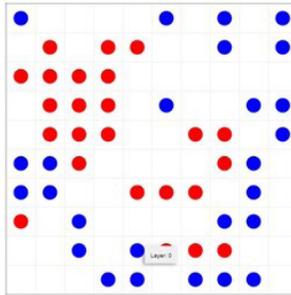


FIGURA 8.3. SEGUNDA ITERACIÓN DEL MODELO DE OPINIÓN BASADO EN LA DINÁMICA DE ÍSING.

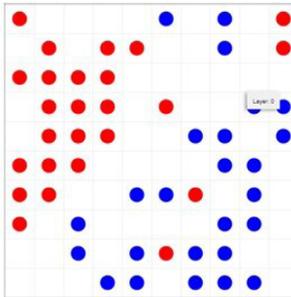


FIGURA 8.4. SEXAGÉSIMA ITERACIÓN DEL MODELO DE OPINIÓN BASADO EN LA DINÁMICA DE ÍSING.

Nótese cómo el sistema evoluciona hacia un estado que muestra dos regiones claramente establecidas de opinión. Una de ellas se ubica hacia la izquierda del enrejado y la otra hacia la derecha. Existen islas de agentes con opiniones distintas. Obviamente, el sistema es sensible al estado inicial. Si comenzamos con un número similar de agentes ubicados en distintas coordenadas, la evolución del sistema diferirá del anterior.

Para ejemplificar lo anterior, veamos a continuación el siguiente estado inicial (figura 8.5.).

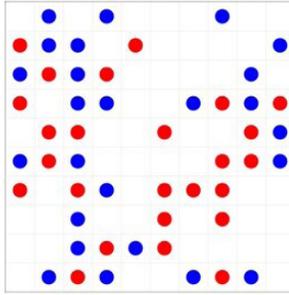


FIGURA 8.5. NUEVO ESTADO INICIAL PARA EL MODELO DE OPINIÓN BASADO EN LA DINÁMICA DE ISING.

Si hacemos evolucionar el sistema sesenta veces para compararlo con el sistema anterior, veremos que se presentan tres regiones de opinión claramente establecidas (figura 8.6.).

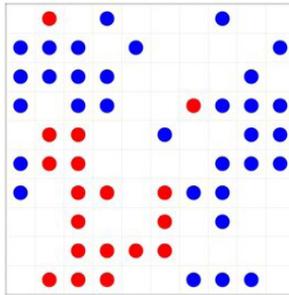


FIGURA 8.6. ESTADO DESPUÉS DE SESENTA ITERACIONES.

Por último, queremos aclarar que para la elaboración del modelo se usó el IDLE que viene con Python y el *browser* fue Edge. El lector es libre de usar el IDE y navegador que más le acomode.

CAPÍTULO 9. MODELOS SOCIOECONÓMICOS USANDO MESA

9.1. INTRODUCCIÓN

Como se ha visto en el capítulo anterior, MESA es una poderosa biblioteca de Python para desarrollar procesos basados en la arquitectura de *software* de programación orientada a objetos (OOB, por sus siglas en inglés). Dichos objetos permiten trabajar con los conceptos de agentes, entornos, modelos y simulaciones de modelos de agentes. Hasta ahora se ha visto cómo instalar la biblioteca de MESA en la computadora y usar las distintas bibliotecas de Python para interactuar con MESA. Asimismo, se mostró un ejemplo del uso de MESA para simular la dinámica de la opinión en una sociedad artificial.

En este capítulo, se desarrollará, primero, un modelo de intercambio económico para mostrar el uso de MESA en la econofísica⁷. Dicho modelo servirá para ejemplificar, paso a paso, el desarrollo de un modelo basado en agentes desde una perspectiva *bottom-up*. Además, el modelo servirá de ejemplo del uso avanzado de Jupyter Notebook.

En el capítulo 2 se vio que Jupyter es un entorno de desarrollo interactivo ampliamente utilizado en ciencias de la computación y ciencias de datos. Se diseñó para facilitar la creación y compartición de documentos que contienen código, texto, visualizaciones y ecuaciones matemáticas. Jupyter se deriva de los proyectos IPython (Interactive Python) y se ha expandido para admitir varios lenguajes de programación y no solo Python.

⁷ La econofísica es un campo interdisciplinario que aplica teorías y métodos desarrollados originalmente en la física para resolver problemas en economía. Este campo emergente comenzó a ganar reconocimiento en la década de 1990, cuando los físicos empezaron a usar sus técnicas y modelos con el fin de estudiar fenómenos económicos. Para saber más del tema, se recomienda el libro de Mantegna y Stanley(1999).

Jupyter Notebook, por otro lado, es una de las aplicaciones más populares dentro del ecosistema de Jupyter. Es una interfaz *web* que permite crear y ejecutar documentos interactivos llamados “notebooks”, los cuales pueden contener tanto código ejecutable como elementos de texto enriquecido, imágenes, ecuaciones en LaTeX y visualizaciones. Esto lo hace ideal para tareas como análisis de datos, modelado, simulaciones y presentaciones.

Jupyter y Jupyter Notebook son herramientas esenciales para la computación interactiva y la ciencia de datos. Permiten a los científicos, ingenieros y analistas trabajar de manera efectiva al combinar código, texto y visualizaciones en un solo entorno colaborativo y flexible. Para quien está incursionando en el uso de Python con el fin de modelar sistemas sociales, Jupyter Notebook será de gran utilidad.

El segundo modelo que se explorará es el de Axelrod de difusión cultural (Axelrod, 1997). Este ejemplo, tiene como objetivo ilustrar cómo usar MESA para el modelado de problemas clásicos dentro de las ciencias sociales computacionales. El modelo de Axelrod se ha utilizado para estudiar una gran variedad de fenómenos culturales y sociales, incluidos conflictos étnicos, polarización política y segregación residencial. Ofrece una herramienta valiosa para analizar cómo las dinámicas culturales pueden dar forma a la política y la estructura social. Además, es un modelo que no ha perdido vigencia, es más, en nuestra era de las redes sociales y la comunicación global, comprender cómo se difunden las ideas y cómo evoluciona la diversidad cultural es más importante que nunca. El modelo de Axelrod proporciona ideas y herramientas para abordar estas cuestiones en un entorno contemporáneo. Puede ser extendido de su forma original, que es una cuadrícula y un conjunto de autómatas celulares a una sociedad artificial compuesta por agentes heterogéneos. Además, su “mundo”, que en origen es una cuadrícula, puede extenderse a una red compleja, lo cual permite una exploración de los fenómenos más realista.

9.2. EL MODELO DE RIQUEZA DE BOLTZMANN

El modelo de riqueza de Boltzmann es una idea teórica que se inspira en la mecánica estadística de Ludwig Boltzmann, un físico austriaco del siglo XIX conocido por sus contribuciones al entendimiento de la termodinámica y la física estadística. El modelo de riqueza de Boltzmann (Drăgulescu y Yakovenko, 2002) se utiliza en el campo de las ciencias sociales computacionales y la economía para analizar la distribución de la riqueza en una sociedad o sistema económico. Este modelo se basa en la premisa de que la distribución de la riqueza puede entenderse en términos de probabilidades y fluctuaciones aleatorias, de manera similar a como se entiende la distribución de las partículas en un gas en la mecánica estadística.

Los conceptos clave del modelo de riqueza de Boltzmann son los siguientes:

1. Partículas (agentes): En este contexto, las “partículas” representan a los individuos o agentes económicos en una sociedad. Cada agente tiene una cierta cantidad de riqueza, que puede considerarse como una propiedad de la partícula en términos de mecánica estadística.
2. Estados de energía (niveles de riqueza): Al igual que en la mecánica estadística, los agentes (partículas) pueden ocupar diferentes “estados de energía”, que en este caso corresponden a diferentes niveles de riqueza. Por ejemplo, un estado de energía más alto podría representar a individuos más ricos, mientras que un estado de energía más bajo representaría a individuos menos ricos.
3. Distribución de probabilidad: En el modelo de Boltzmann, se asume que la probabilidad de que un agente tenga una cierta cantidad de riqueza sigue una distribución de probabilidad específica. Esta distribución puede ser una distribución de probabilidad exponencial o alguna otra, dependiendo de las suposiciones del modelo.
4. Interacciones y transferencia de riqueza: Los agentes interactúan entre sí de alguna manera que permite la transferencia de riqueza. Por ejemplo, podrían participar en transacciones económicas donde algunos ganan dinero mientras que otros lo pierden. Estas interacciones

- y transferencias de riqueza son fundamentales para modelar cómo evoluciona la distribución de la riqueza con el tiempo.
5. Entropía y equilibrio: El modelo de Boltzmann también utiliza conceptos de entropía y equilibrio para describir cómo la distribución de la riqueza evoluciona hacia un estado estable o de equilibrio. La entropía representa la medida de la dispersión o desorden de la distribución de la riqueza.
 6. Ley de Zipf: A menudo, el modelo de riqueza de Boltzmann se relaciona con la ley de Zipf, que describe una distribución de riqueza en la que unas pocas personas o agentes poseen la mayoría de la riqueza, mientras que la mayoría tiene cantidades mucho menores. Esta ley se ha observado en muchas sociedades y sistemas económicos.

9.2.1 Aspectos matemáticos del modelo de riqueza de Boltzmann

Las ecuaciones que ayudan a definir los aspectos matemáticos del modelo de riqueza de Boltzmann permiten definir los conceptos clave de este modelo. Se utilizan distribuciones de probabilidad para describir cómo se distribuye la riqueza entre los agentes económicos. Usaremos como ejemplo la distribución de Pareto, la cual es comúnmente asociada con la distribución de riqueza en la realidad. La función de densidad de probabilidad (FDP) de la distribución de Pareto se expresa de la siguiente manera:

$$f(x; \alpha, x_{\min}) = \frac{\alpha x_{\min}^{\alpha}}{x^{\alpha+1}}$$

Donde:

- (x) es la riqueza de un agente.
- (α) es el parámetro de forma que determina la cola larga de la distribución.
- (x_{\min}) es la riqueza mínima posible.

Esta FDP describe cómo se distribuye la riqueza entre los agentes en términos de probabilidades. Cabe aclarar que la “cola larga” en la distribución de Pareto se refiere a la característica distintiva de esta distribución estadística en la que un pequeño número de eventos o valores extrema-

damente grandes son responsables de la mayor parte de los resultados o valores observados. En otras palabras, la cola larga indica que la mayoría de los eventos o valores se concentran en una región relativamente pequeña, mientras que un número significativamente menor de eventos o valores se extienden hacia el extremo derecho de la distribución, que representan valores muy altos o raros.

La distribución de Pareto se utiliza por lo regular para describir fenómenos en los que la desigualdad es pronunciada, como la distribución de la riqueza, la distribución de ingresos, la popularidad de sitios *web*, la frecuencia de palabras en textos y muchos otros casos. En estos contextos, la “cola larga” significa que un pequeño porcentaje de la población o los elementos (por ejemplo, personas, productos, palabras) poseen una gran parte de la riqueza, ingresos, popularidad o frecuencia, mientras que la mayoría de la población o elementos contribuyen con una fracción relativamente pequeña de estos recursos o resultados.

La “cola larga” se refleja en el hecho de que esta función de densidad de probabilidad decae de manera muy lenta en el extremo derecho de la distribución, lo que significa que es posible encontrar valores extremadamente altos con una baja probabilidad, pero que aún son parte de la distribución.

Para modelar la dinámica temporal de la distribución de riqueza, se pueden utilizar ecuaciones diferenciales estocásticas que describen cómo las transacciones económicas afectan a los agentes. Una simple ecuación de dinámica de riqueza podría ser:

$$\frac{dx}{dt} = \lambda \cdot x$$

Donde:

- $(\frac{dx}{dt})$ representa la tasa de cambio de la riqueza de un agente en función del tiempo (t).
- (λ) es una tasa que representa la tasa de crecimiento de la riqueza.

Esta ecuación modela el crecimiento exponencial de la riqueza de un agente con el tiempo.

Para calcular la entropía de la distribución de riqueza, podemos utilizar la fórmula de entropía de Shannon, que mide la dispersión o desorden en la distribución de probabilidad:

$$H(X) = - \sum_i p(x_i) \cdot \log(p(x_i))$$

Donde:

- $H(X)$ es la entropía de la distribución de riqueza.
- $p(x_i)$ es la probabilidad de que un agente tenga una riqueza (x_i).

Esta fórmula calcula la entropía de la distribución de riqueza, que cuantifica cuán desigual es la distribución.

El modelo de riqueza de Boltzmann utiliza ecuaciones de distribución de probabilidad, dinámicas de riqueza y cálculos de entropía para analizar cómo se distribuye y evoluciona la riqueza en una sociedad o sistema económico. Estas ecuaciones matemáticas proporcionan un marco para comprender la dinámica de la desigualdad de riqueza y su relación con la mecánica estadística.

9.2.2. Modelado computacional del modelo de riqueza de Boltzmann

En el ejemplo que trataremos en esta sección, nos inspiramos en el trabajo de Drăgulescu y Yakovenko (2002) para obtener información adicional sobre los supuestos utilizados en dicho modelo. Los supuestos que rigen este modelo son:

1. Hay una serie de agentes.
2. Todos los agentes comienzan con una unidad de dinero.
3. En cada paso del modelo, un agente da una unidad de dinero (si es que lo tiene) a algún otro agente.

Incluso como modelo de nivel inicial, los resultados arrojados son interesantes e inesperados para quienes no están familiarizados con el tema específico. Como tal, este modelo es un buen punto de partida para examinar las características principales de MESA.

Se recomienda iniciar con un cuaderno de Jupyter Notebook para este ejemplo, ya que esto permite examinar pequeños segmentos de códigos de uno en uno. Como opción, esto se puede crear utilizando archivos de script de Python usando el IDE de su preferencia.

En primer lugar, hay que instalar Jupyter y Jupyter Notebook en la computadora. Si se está usando Windows, esto se hace desde la consola de símbolo de sistema. Para instalar Jupyter, solo es necesario usar el comando “pip install jupyter”⁸. Una vez instalado Jupyter, se puede comenzar de inmediato a usar Jupyter Notebook. Para iniciar a usar Jupyter Notebook es necesario usar el comando “jupyter notebook” en la consola de símbolo de sistema.

Para las visualizaciones del modelo, será necesario instalar la biblioteca de Python Matplotlib. Esto se hace desde la consola de símbolo de sistema de la siguiente manera: “pip install matplotlib”.

Una buena práctica es colocar su modelo en la propia carpeta/directorio. Esto no es específicamente necesario para el modelo que usaremos, pero a medida que otros modelos se vuelven más complicados y se expanden, se pueden agregar múltiples *scripts* de Python, documentación, discusiones y cuadernos.

Usando los comandos del sistema operativo, cree a continuación una nueva carpeta/directorio llamada “starter_model”. Cambie a la nueva carpeta/directorio.

Abra Jupyter Notebook y cree el modelo de forma interactiva. Se verá algo similar a la figura 9.1. Después, es necesario que comience importando las dependencias que se usarán en el modelo. A continuación, se muestra el código:

```
# biblioteca especializada en agentes
import mesa

# Herramientas de visualización de datos.
import seaborn as sns

# Tiene arrays y matrices multi-dimensionales.
# Tiene una gran colección de funciones matemáticas para operar
estos arreglos.
import numpy as np
```

⁸ Recuerde tener instalado Python, ya que *pip* es una función de Python.

```
# manipulación y análisis de datos
import pandas as pd
```

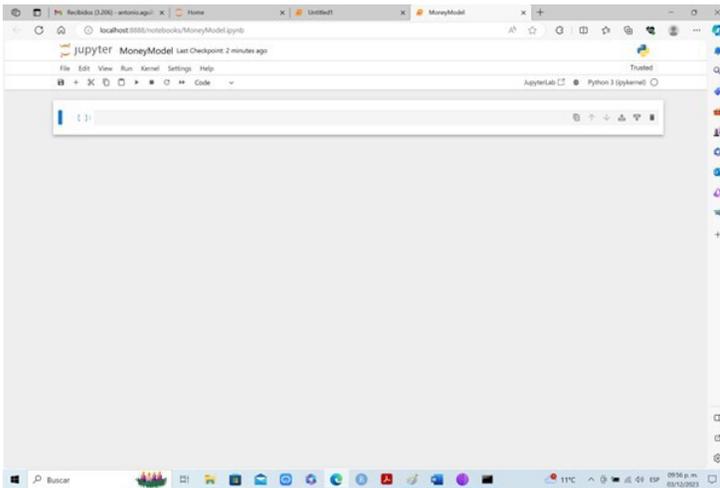


FIGURA 9.1. VENTANA DE JUPYTER NOTEBOOK.

En primer lugar, cree el agente. A medida que avance el capítulo, se agregarán más funciones al agente. Los agentes son las entidades individuales que actúan en el modelo. Es una buena práctica de modelado asegurarse de que cada agente pueda ser identificado de forma única.

En el modelo, los agentes son los individuos que intercambian dinero; en este caso, la cantidad de dinero que tiene un agente individual se representa como riqueza. Además, cada agente tiene un identificador único.

Para la aplicación del código, se hace creando una nueva clase (u objeto) que se extiende creando una subclase de la clase de mesa. La nueva clase se denomina “class MoneyAgent(mesa.Agent)”. El código se muestra a continuación.

```
class MoneyAgent(mesa.Agent):
    """Un agente con riqueza inicial fija."""

    def __init__(self, unique_id, model):
        # Pasa los parámetros a la clase padre
        super().__init__(unique_id, model)
```

```
# Crea las variables de los agentes
y establece los valores iniciales.
self.wealth = 1
```

Una vez creada la clase agente, es necesario crear la clase modelo. En nuestro caso, el modelo se puede visualizar como una cuadrícula que contiene todos los agentes. El modelo crea, mantiene y gestiona todos los agentes de la red. El modelo evoluciona en pasos de tiempo discretos.

Cuando se crea un modelo, se especifica el número de agentes dentro de él. Luego, el modelo crea los agentes y los coloca en la cuadrícula. El modelo contiene asimismo un programador que controla el orden en que se activan los agentes. El planificador también es responsable de hacer avanzar el modelo un paso. El modelo contiene además un recolector de datos que recopila datos del modelo. Estos temas se cubrirán con más detalle más adelante en el capítulo.

Para crear el modelo, es necesario crear una clase de MESA. Esto se hace con el siguiente código: “class MoneyModel(mesa.Model)”. El código para la creación de la clase y la creación e inicio de los agentes se da a continuación:

```
class MoneyModel(mesa.Model):
    """UN modelo con un número dado de agentes."""

    def __init__(self, N):
        self.num_agents = N
        # Creación de los agentes
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
```

Ahora, es necesario incorporar a nuestro programa un planificador (*scheduler*). El planificador controla el orden en el que se activan los agentes, lo que hace que el agente realice la acción definida. El planificador también es responsable de hacer avanzar el modelo un paso. Un paso es la unidad de tiempo más pequeña del modelo, y a menudo se lo denomina *tic*. El planificador se puede configurar para activar agentes en diferentes órdenes. Esto es importante ya que el orden en el que se activan los agentes puede afectar los resultados del modelo. En cada paso del mode-

lo, uno o más agentes —normalmente todos— se activan y dan su propio paso, cambiando internamente o interactuando entre sí o con el entorno.

Para crear el planificador, se crea una nueva clase llamada “RandomActivationByAgent”, para lo que se usa la extensión “mesa.time.RandomActivation” y se crea una subclase de la clase “RandomActivation” de MESA. Esta clase activa a todos los agentes una vez por paso, en orden aleatorio. Se espera que cada agente tenga un método *step*. El método *step* es la acción que realiza el agente cuando es activado por el planificador del modelo. Agregamos un agente al planificador usando el método *add*; cuando llamamos al método *step* del planificador, el modelo baraja el orden de los agentes, luego activa y ejecuta el método *step* de cada agente. Luego, el planificador se agrega al modelo.

Una vez que hemos comprendido lo que vamos a hacer, procederemos a modificar las clases “MoneyAgent” y “MoneyModel”. El código es el siguiente:

```
class MoneyAgent(mesa.Agent):
    """Un agente es creado con un nivel de riqueza inicial fijo"""

    def __init__(self, unique_id, model):
        # Pasa los parámetros a la clase padre.
        super().__init__(unique_id, model)

        # Crea los atributos de los agentes y establece
        # los valores iniciales.
        self.wealth = 1

    def step(self):
        # El proceso step va aquí
        # Para propósitos de demostración imprimiremos
        # el id único del agente
        print(f"Hola, Soy un agente, me puedes llamar
        {str(self.unique_id)}.")

class MoneyModel(mesa.Model):
    """Un modelo con algún número de agentes."""

    def __init__(self, N):
        self.num_agents = N
        # Crea el planificador y lo asigna al modelo
        self.schedule = mesa.time.RandomActivation(self)

        # Crea agentes
        for i in range(self.num_agents):
```

```

    a = MoneyAgent(i, self)
    # Añade el agente al planificador
    self.schedule.add(a)

def step(self):
    """Avanza el modelo un paso."""

    '''El step del modelo iría aquí, por ahora solo llamará
    al método de cada agente e imprimirá su Id único'''
    self.schedule.step()

```

Ahora se ha creado un modelo básico. El modelo se puede ejecutar creando un objeto de modelo y llamando al método de paso. El modelo se ejecutará durante un paso e imprimirá el ID único de cada agente. Puede ejecutar el modelo para varios pasos llamando al método de paso varias veces.

El siguiente código, muestra cómo se ejecutaría el modelo:

```

starter_model = MoneyModel(10)
starter_model.step()

```

Note que si usted está no está usando Jupyter Notebook sino otro IDE, su extensión será “.py” en vez de “.ipynb”. Para correr su programa, es necesario que cree un archivo de Python con extensión .py, que se puede llamar “run.py” y que debe contener el siguiente código:

```

from money_model import MoneyModel

starter_model = MoneyModel(10)
starter_model.step()

```

Recuerde guardar su programa principal como el *script* “money_model.py”, ya que de ahí se importará la clase “MoneyModel”.

El resultado de la simulación es el siguiente:

```

Hola, Soy un agente, me puedes llamar 9.
Hola, Soy un agente, me puedes llamar 3.
Hola, Soy un agente, me puedes llamar 6.
Hola, Soy un agente, me puedes llamar 1.
Hola, Soy un agente, me puedes llamar 5.
Hola, Soy un agente, me puedes llamar 8.
Hola, Soy un agente, me puedes llamar 7.

```

```
Hola, Soy un agente, me puedes llamar 0.  
Hola, Soy un agente, me puedes llamar 4.  
Hola, Soy un agente, me puedes llamar 2.
```

Si volvemos a correr el modelo, gracias a la aleatoriedad de la asignación de *steps*, tendremos una lista diferente. Véase la siguiente salida:

```
Hola, Soy un agente, me puedes llamar 8.  
Hola, Soy un agente, me puedes llamar 2.  
Hola, Soy un agente, me puedes llamar 0.  
Hola, Soy un agente, me puedes llamar 9.  
Hola, Soy un agente, me puedes llamar 4.  
Hola, Soy un agente, me puedes llamar 6.  
Hola, Soy un agente, me puedes llamar 5.  
Hola, Soy un agente, me puedes llamar 1.  
Hola, Soy un agente, me puedes llamar 7.  
Hola, Soy un agente, me puedes llamar 3.
```

Vamos ahora a modificar la clase “MoneyAgent” para que nos imprima, además del Id único, la riqueza de cada agente. El código es:

```
class MoneyAgent(mesa.Agent):  
    """Un agente con una riqueza fija."""  
  
    def __init__(self, unique_id, model):  
        # Pasa los parámetros a la clase padre.  
        super().__init__(unique_id, model)  
  
        # Crea los atributos de los agentes y establece  
        los valores iniciales.  
        self.wealth = 1  
  
    def step(self):  
        # El step del agente iría aquí  
        # Corregido para imprimir el Id y la riqueza  
        print(f"Hola, Soy el agente {str(self.unique_id)} y tengo",  
              self.wealth)
```

La salida de este nuevo programa es la siguiente:

```
Hola, Soy el agente 8 y tengo 1  
Hola, Soy el agente 7 y tengo 1  
Hola, Soy el agente 3 y tengo 1  
Hola, Soy el agente 6 y tengo 1
```

```
Hola, Soy el agente 4 y tengo 1
Hola, Soy el agente 1 y tengo 1
Hola, Soy el agente 5 y tengo 1
Hola, Soy el agente 2 y tengo 1
Hola, Soy el agente 0 y tengo 1
Hola, Soy el agente 9 y tengo 1
```

Volvamos ahora el concepto del *step* del agente. Como se pudo observar, el *step* es donde se define el comportamiento del agente en relación con cada paso o *tick* del modelo. Para nuestro modelo, el agente comprobará su riqueza, y si tiene dinero, regalará una unidad a otro agente aleatorio.

El planificador llama al método después de cada paso del agente durante cada paso del modelo. Para permitir que el agente elija otro agente al azar, se usará el método “`model.random`”, que es un generador de números aleatorios. Esto funciona igual que el módulo `random`⁹ de Python, pero con un conjunto de semillas fijo cuando se crea una instancia del modelo, que se puede usar para replicar un modelo específico ejecutado más adelante.

Para elegir un agente al azar, necesitamos una lista de todos los agentes. Tome en cuenta que no existe de forma explícita dicha lista en el modelo. El planificador, sin embargo, posee una lista interna de todos los agentes que está programado para activar.

El código quedaría de la siguiente forma:

```
import copy

class MoneyAgent(mesa.Agent):
    """Un agente con un nivel de riqueza fijo."""

    def __init__(self, unique_id, model):
        # Pasa los parámetros a la clase padre.
        super().__init__(unique_id, model)

        # Crea los atributos de los agentes y establece
        los valores iniciales.
        self.wealth = 1

    def step(self):
        # Verifica si el agente tiene alguna riqueza
```

⁹ Recuérdese que los lenguajes de computadora no generan números realmente aleatorios, sino pseudoaleatorios, es decir, usan una semilla y un algoritmo para generar los números; si se usa dicha semilla, se pueden replicar los mismos números aleatorios muchas veces.

```

if self.wealth > 0:
    other_agent = self.random.choice
    (self.model.schedule.agents)
    if other_agent is not None:
        other_agent.wealth += 1
        self.wealth -= 1

```

Ahora estamos ya ante nuestro primer modelo de intercambio de riqueza. Para correrlo, usaremos el siguiente código:

```

model = MoneyModel(10)
for i in range(10):
    model.step()

```

Nótese que pareciera que el programa no hizo nada; lo que realmente ha pasado es que no tenemos todavía forma de capturar la información que surge del modelo. Para ello, es necesario que programemos nuevos procesos que nos permitan visualizar los resultados de la simulación.

Para mostrar gráficamente el comportamiento de la distribución de la riqueza en el conjunto de los diez agentes que generamos, recurriremos al siguiente código:

```

# Para un Notebook de jupyter añada la siguiente línea:
%matplotlib inline

# La línea de abajo es necesaria tanto para scripts como para
Notebooks
import matplotlib.pyplot as plt

agent_wealth = [a.wealth for a in model.schedule.agents]
# Crea un histograma con seaborn
g = sns.histplot(agent_wealth, discrete=True)
g.set(
    title="Distribución de la riqueza", xlabel="Riqueza",
    ylabel="Número de agentes"
);
# El punto y coma es usado para evitar que se imprima la
representación del objeto

```

Corriendo el código anterior en nuestro Notebook, obtenemos la gráfica que aparece en la figura 9.2.

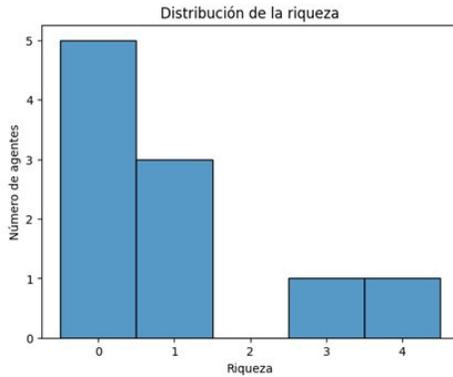


FIGURA 9.2. HISTOGRAMA DONDE SE MUESTRA LA DISTRIBUCIÓN DE LA RIQUEZA PARA DIEZ AGENTES Y DIEZ ITERACIONES.

Con este simple modelo, vamos a explorar un poco el comportamiento de este, ampliemos a cien agentes y hagamos mil iteraciones del modelo. Esto se hace con el siguiente código:

```

model = MoneyModel(100)
for i in range(1000):
    model.step()

```

Luego corra de nuevo el código para construir el histograma y vea lo que obtiene. En nuestro caso hemos obtenido la gráfica que muestra la figura 9.3. Nótese que el comportamiento corresponde a una distribución de Pareto, es decir, existen muchos agentes con nada de riqueza (“wealth” = 0) y solo unos cuantos que tienen el máximo de la riqueza. Es sorprendente que sin usar una distribución de probabilidad, como marca el modelo básico de Boltzmann, se haya logrado solo con una simple regla replicar los resultados de una distribución tipo Pareto.

Con esto podemos ver cómo en nuestra economía artificial la mayoría de los agentes quedan en situación de pobreza extrema y solo unos cuantos de ellos detentan la riqueza de la sociedad. Una situación nada alejada de lo que se vive en las economías del mundo real.

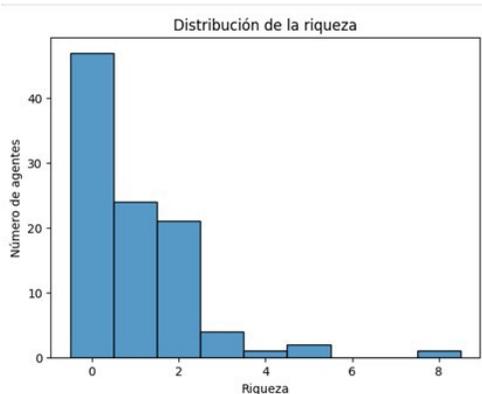


FIGURA 9.3. DISTRIBUCIÓN DE LA RIQUEZA PARA EL CASO DE CIENTO AGENTES Y MIL ITERACIONES.

Muchos MBA tienen un elemento espacial en el que los agentes se mueven e interactúan con los vecinos cercanos. MESA admite dos tipos generales de espacios: cuadrícula y continuo. Las cuadrículas se dividen en celdas y los agentes solo pueden ubicarse en una celda en particular, como las piezas de un tablero de ajedrez. El espacio continuo, por el contrario, permite a los agentes ocupar cualquier posición arbitraria. Tanto las cuadrículas como los espacios continuos son frecuentemente toroidales, lo que significa que los bordes se envuelven, con las celdas del borde derecho conectadas a las del borde izquierdo y de arriba abajo. Esto evita que algunas celdas tengan menos vecinos que otras o que los agentes puedan salirse del borde del entorno.

Agregaremos un elemento espacial simple a nuestro modelo económico colocando a nuestros agentes en una cuadrícula y haciéndolos caminar al azar. En lugar de darle su unidad de dinero a cualquier agente aleatorio, se la darán a un agente en la misma celda.

MESA tiene dos tipos de cuadrículas: SingleGrid y MultiGrid. SingleGrid permite colocar solo un agente por celda, mientras que MultiGrid permite que varios agentes ocupen la misma celda. Dado que necesitamos que varios agentes ocupen la misma celda, usaremos el comando “MultiGrid”.

Creamos una instancia de cuadrícula con parámetros de ancho y alto, y un valor *booleano* para determinar si la cuadrícula es toroidal. Hagamos parámetros del modelo de ancho y alto, además del número de agentes, y hagamos que la cuadrícula sea siempre toroidal. Podemos colocar agentes en una cuadrícula con el método “place_agent” de la cuadrícula, que toma un agente y una tupla (x, y) de coordenadas para colocar el agente. Estos cambios se hacen en la clase “Modelo”. Veamos el código:

```
class MoneyModel(mesa.Model):
    """Un modelo con algún número de agentes."""

    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)

        # Crea agentes
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

            # Añade los agentes a una celda
            # aleatoria de una cuadrícula
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))
```

La posición de cada agente se almacena de dos maneras: el agente está contenido en la cuadrícula en la celda en la que se encuentra actualmente y el agente tiene una variable *pos* con una tupla de coordenadas (x, y) . El método “place_agent” agrega la coordenada al agente de forma automática.

Ahora necesitamos aumentar el comportamiento de los agentes, permitiéndoles moverse y solo dar dinero a otros agentes en la misma celda. Para ello, primero manejemos el movimiento y hagamos que los agentes se trasladen a una celda vecina. El objeto “grid” proporciona un método “move_agent” que mueve un agente a una celda determinada. Eso todavía nos deja encontrar las posibles celdas vecinas a las que trasladarnos. Para ello, utilizaremos el método “get_neighborhood” integrado de la cuadrícula, que devuelve todos los vecinos de una celda determinada. Este método puede obtener dos tipos de vecindades de celdas: Moore

(incluye los ocho cuadrados circundantes) y Von Neumann (solo arriba/abajo/izquierda/derecha). También necesita un argumento sobre si se debe incluir la celda central como una de las vecinas.

El código, entonces, quedaría como sigue:

```
import copy
class MoneyAgent(mesa.Agent):
    """Un agente con un nivel de riqueza fijo."""

    def __init__(self, unique_id, model):
        # Pasa los parámetros a la clase padre.
        super().__init__(unique_id, model)

        # Crea los atributos de los agentes y establece
        los valores iniciales.
        self.wealth = 1

#Mueve los agentes en la cuadrícula
def move(self):
    possible_steps = self.model.grid.get_neighborhood(
        self.pos,
        moore=True,
        include_center=False)
    new_position = self.random.choice(possible_steps)
    self.model.grid.move_agent(self, new_position)
```

A continuación, debemos reunir a todos los demás agentes presentes en una celda y darle algo de dinero a uno de ellos. Podemos obtener el contenido de una o más celdas usando el método “get_cell_list_contents” de la cuadrícula, o entrando de forma directa a una celda. El método acepta una lista de tuplas de coordenadas de celda, o una sola tupla si solo nos importa una celda. El código será el siguiente:

```
class MoneyAgent(mesa.Agent):
    """Un agente con un nivel de riqueza fijo."""

    def __init__(self, unique_id, model):
        # Pasa los parámetros a la clase padre.
        super().__init__(unique_id, model)

        # Crea los atributos de los agentes y establece
        los valores iniciales.
        self.wealth = 1
```

```

#Mueve los agentes en la cuadrícula
def move(self):
    possible_steps = self.model.grid.get_neighborhood(
        self.pos,
        moore=True,
        include_center=False)
    new_position = self.random.choice(possible_steps)
    self.model.grid.move_agent(self, new_position)

# Da dinero a un agente al azar que esté ocupando
la misma celda
def give_money(self):
    cellmates =
self.model.grid.get_cell_list_contents([self.pos])
    if len(cellmates) > 1:
        other = self.random.choice(cellmates)
        other.wealth += 1
        self.wealth -= 1

```

Ahora solo nos falta hacer que el agente avance con “step”. Esto quedaría de la siguiente forma:

```

def step(self):
    self.move()
    if self.wealth > 0:
        self.give_money()

```

El código completo se muestra a continuación:

```

class MoneyAgent(mesa.Agent):
    """Un agente con una riqueza fija."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
        cellmates =
self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:

```

```

        other_agent = self.random.choice(cellmates)
        other_agent.wealth += 1
        self.wealth -= 1

def step(self):
    self.move()
    if self.wealth > 0:
        self.give_money()

class MoneyModel(mesa.Model):
    """Un modelo con algún número de agentes"""

    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)

        # Crea agentes
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

        # Añade el agente a una celda aleatoria de la cuadrícula
        x = self.random.randrange(self.grid.width)
        y = self.random.randrange(self.grid.height)
        self.grid.place_agent(a, (x, y))

    def step(self):
        self.schedule.step()

```

Para correr el modelo, es necesario que cambiemos los parámetros de nuestro código que hemos estado usando para correr las simulaciones. El código nuevo será:

```

model = MoneyModel(100, 10, 10)
for i in range(20):
    model.step()

```

Nótese que MoneyModel ahora tiene tres parámetros de entrada, el primero corresponde al número de agentes que se crearán y los dos siguientes son el ancho y alto de la cuadrícula. Si corremos este código, no veremos todavía nada; es necesario introducir un gráfico especial para visualizar lo que está ocurriendo, y para ello necesitamos usar “matplotlib” y “numpy” para visualizar la cantidad de agentes que residen en cada

celda. Para eso, creamos una matriz “numpy” del mismo tamaño que la cuadrícula, llena de ceros. Luego usamos la función “coord_iter()” del objeto de cuadrícula, que nos permite recorrer cada celda de la cuadrícula, proporcionándonos las posiciones y el contenido de cada celda por turno. El código sería el siguiente:

```
agent_counts = np.zeros((model.grid.width, model.grid.height))
for cell_content, (x, y) in model.grid.coord_iter():
    agent_count = len(cell_content)
    agent_counts[x][y] = agent_count
# Graficación usando seaborn, con un tamaño de 5x5
g = sns.heatmap(agent_counts, cmap="viridis", annot=True, cbar=False,
square=True)
g.figure.set_size_inches(4, 4)
g.set(title="Número de agentes sobre cada celda de la cuadrícula");
```

La salida gráfica que obtenemos con este código se muestra en la figura 9.4.



FIGURA 9.4. CUADRÍCULA MOSTRANDO EL NÚMERO DE AGENTES EN CADA CELDA.

Hasta ahora, al final de cada ejecución del modelo tenemos que escribir nuestro propio código para obtener los datos del modelo. Esto tiene dos problemas: no es muy eficiente y solo nos da resultados finales.

Si quisiéramos conocer la riqueza de cada agente en cada paso, tendríamos que agregarla al ciclo de ejecución de pasos y encontrar alguna manera de almacenar los datos.

Dado que uno de los principales objetivos del modelado basado en agentes es generar datos para el análisis, MESA proporciona una clase que puede manejar la recopilación y el almacenamiento de datos y facilitar su análisis.

El recopilador de datos almacena tres categorías de datos: variables a nivel de modelo, variables a nivel de agente y tablas (que son un comodín para todo lo demás). Las variables a nivel de modelo y agente se agregan al recopilador de datos junto con una función para recopilarlas. Las funciones de recopilación en el nivel de modelo toman un objeto de modelo como entrada, mientras que las funciones de recopilación en el nivel de agente toman un objeto de agente como entrada. Luego, ambos devuelven un valor calculado a partir del modelo o de cada agente en su estado actual. Cuando se llama al método de recopilación del recopilador de datos, con un objeto de modelo como argumento, aplica cada función de recopilación en el nivel de modelo al modelo y almacena los resultados en un diccionario, asociando el valor actual con el paso actual del modelo. De manera similar, el método aplica cada función de recopilación en el nivel de agente a cada agente actualmente en el planificador, asociando el valor resultante con el paso del modelo y el ID único del agente.

Agreguemos un “DataCollector” al modelo con “`mesa.DataCollector`” y recopilaremos dos variables. En el de agente, queremos recolectar la riqueza de cada agente en cada paso. En cuanto al modelo, calcularemos el coeficiente de Gini¹⁰ del modelo, un indicador muy usado en economía para medir de la desigualdad de riqueza. El código es el siguiente:

```
def compute_gini(model):
    agent_wealths =
    [agent.wealth for agent in model.schedule.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
```

¹⁰ El coeficiente de Gini es un número entre 0 y 1, donde 0 corresponde a la perfecta igualdad (todos tienen los mismos ingresos) y 1 corresponde a la perfecta desigualdad (una persona tiene todos los ingresos y los demás ninguno).

```

B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
return 1 + (1 / N) - 2 * B

class MoneyAgent(mesa.Agent):
    """Un agente con una riqueza fija."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos, moore=True, include_center=False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(
            self, new_position)

    def give_money(self):
        cellmates = self.model.grid.get_cell_list_contents(
            [self.pos])
        cellmates.pop(
            cellmates.index(self)
        ) # Ensure agent is not giving money to itself
        if len(cellmates) > 1:
            other = self.random.choice(cellmates)
            other.wealth += 1
            self.wealth -= 1
            if other == self:
                print("Me acabo de dar dinero a mi mismo, jejeje")

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()

class MoneyModel(mesa.Model):
    """Un modelo con algún número de agentes."""

    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = mesa.space.MultiGrid(width, height, True)
        self.schedule = mesa.time.RandomActivation(self)

        # Crea agentes
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)
            # Add the agent to a random grid cell
            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)

```

```

self.grid.place_agent(a, (x, y))

self.datacollector = mesa.DataCollector(
    model_reporters={"Gini": compute_gini},
    agent_reporters={"Wealth": "wealth"}
)

def step(self):
    self.datacollector.collect(self)
    self.schedule.step()

```

En cada paso del modelo, el recolector de datos recopilará y almacenará el coeficiente de Gini actual en el nivel del modelo, así como la riqueza de cada agente, asociando cada uno con el paso actual. Ejecutamos el modelo tal como lo hicimos antes. Ahora es cuando una sesión interactiva, en especial a través de un “Notebook”, resulta útil: el “DataCollector” puede exportar los datos recopilados como un “DataFrame” de pandas¹¹, para facilitar el análisis interactivo.

Volvamos ahora a correr nuestro programa modificado para calcular el coeficiente de Gini y veamos el código que nos permite visualizarlo.

```

gini = model.datacollector.get_model_vars_dataframe()
# Grafica el indice de Gini sobre el tiempo
g = sns.lineplot(data=gini)
g.set(title="Coeficiente de Gini sobre el tiempo", ylabel="Coeficiente
de Gini");

```

El resultado de este código es la figura 9.5.

¹¹ Si es nuevo en Python, tenga en cuenta que pandas ya está instalado como una dependencia de MESA y que pandas es una “herramienta de manipulación y análisis de datos de código abierto, rápida, potente, flexible y fácil de usar”. Pandas es un gran recurso para ayudar a analizar los datos recopilados en sus modelos.



FIGURA 9.5. ÍNDICE DE GINI PARA EL MODELO.

Podemos ver que en el tiempo cero el coeficiente de Gini es cero, lo que corresponde a las condiciones iniciales que se establecieron al momento de crear los agentes, esto es, cada agente tiene una unidad de riqueza al tiempo cero y por tanto la riqueza ha sido distribuida de manera uniforme. Según avanza el tiempo, el índice de Gini tiende al alza, lo que indica condiciones de desigualdad en nuestra economía artificial.

Verá que el índice del “DataFrame” son pares de paso del modelo e ID del agente. Esto se debe a que el recopilador de datos almacena los datos en un diccionario, con el número de paso como clave y un diccionario de pares de ID de agente y valores de variable como valor. Luego, el recopilador de datos convierte este diccionario en un “DataFrame”, razón por la cual el índice es un par de (paso del modelo, ID del agente). Puede analizarlo como lo haría con cualquier otro “DataFrame”. Por ejemplo, para obtener un histograma de la riqueza de los agentes al final del modelo, podemos usar el siguiente código:

```
last_step = agent_wealth.index.get_level_values("Step").max()
end_wealth = agent_wealth.xs(last_step, level="Step")["Wealth"]
# Crea un histograma de la riqueza para el último step.
g = sns.histplot(end_wealth, discrete=True)
g.set(
    title="Distribución de la riqueza al final de la simulación",
    xlabel="Riqueza",
    ylabel="Número de agentes",
);
```

El histograma se puede ver en la figura 9.6.

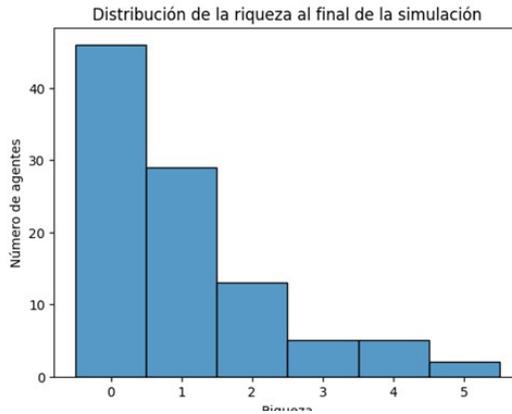


FIGURA 9.6. HISTOGRAMA DE LA DISTRIBUCIÓN DE LA RIQUEZA AL FINAL DE LA SIMULACIÓN.

Ahora vamos a modificar el modelo para que utilice una función de probabilidad como establece las condiciones del modelo de Boltzmann. Para ello, vamos a recurrir a la función de probabilidad de Pareto mencionada en los aspectos matemáticos del modelo de Boltzmann.

Para modificar el programa y usar una función de probabilidad de Pareto para calcular el intercambio de riqueza entre los agentes, primero debemos comprender cómo la distribución de Pareto puede aplicarse en este contexto. La distribución de Pareto se emplea comúnmente para describir distribuciones desiguales, como la distribución de la riqueza en una sociedad.

En este caso, podríamos utilizar la distribución de Pareto para determinar la cantidad de riqueza que un agente transfiere a otro. La idea sería que los agentes más ricos tienen una mayor probabilidad de perder más riqueza en un intercambio, mientras que los más pobres tienen menos probabilidades de perder mucho. Esta distribución imita algunos aspectos de los sistemas económicos reales donde los ricos tienden a una mayor variabilidad en sus cambios de riqueza.

A continuación, se presenta el código modificado con comentarios para explicar cada paso:

```

import mesa
import numpy as np

class MoneyAgent(mesa.Agent):
    """Un agente con un nivel de riqueza fijo."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def step(self):
        if self.wealth > 0:
            other_agent = self.random.choice
            (self.model.schedule.agents)
            if other_agent is not None:
                # Determinar la cantidad de riqueza
                a transferir usando la distribución de Pareto
                amount = self.pareto_wealth_transfer()
                # Asegurar que la cantidad transferida
                no sea mayor que la riqueza actual
                amount = min(amount, self.wealth)

                other_agent.wealth += amount
                self.wealth -= amount

    def pareto_wealth_transfer(self):
        """Determina la cantidad de riqueza a transferir
        basándose en una distribución de Pareto."""
        # Definir los parámetros de la distribución de Pareto
        a = 3.0 # Parámetro de forma; puede ajustarse
        para cambiar la "desigualdad"
        m = 1.0 # Escala mínima

        # Generar un valor de la distribución de Pareto
        return (np.random.pareto(a) + 1) * m

```

En este código hemos añadido el método “pareto_wealth_transfer”, que utiliza la función “np.random.pareto” de NumPy para generar un valor de la distribución de Pareto, que representa la cantidad de riqueza que se transfiere en cada interacción. Los parámetros a y m de la distribución de Pareto pueden ajustarse para modificar la forma de la distribución, y por tanto, la “desigualdad” en la transferencia de riqueza.

Este modelo ofrece un enfoque más realista del intercambio de riqueza al permitir que las interacciones entre agentes reflejen de mejor manera las dinámicas complejas y desiguales observadas en sistemas económicos reales.

Para agregar un proceso que permita graficar la distribución de la riqueza entre los agentes después de terminada la simulación, necesitaremos realizar algunos pasos adicionales. Primero, vamos a recolectar los datos de la riqueza de cada agente al final de la simulación. Luego, utilizaremos la biblioteca de gráficos “matplotlib” para crear un gráfico que muestre esta información.

A continuación, se muestra el código modificado con los cambios necesarios, incluyendo comentarios para explicar cada paso:

```
import mesa
import numpy as np
import matplotlib.pyplot as plt

class MoneyAgent(mesa.Agent):
    """Un agente con un nivel de riqueza fijo."""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def step(self):
        if self.wealth > 0:
            other_agent = self.random.choice
            (self.model.schedule.agents)
            if other_agent is not None:
                # Determinar la cantidad de riqueza
                a transferir usando la distribución de Pareto
                amount = self.pareto_wealth_transfer()
                # Asegurar que la cantidad transferida
                no sea mayor que la riqueza actual
                amount = min(amount, self.wealth)

                other_agent.wealth += amount
                self.wealth -= amount

    def pareto_wealth_transfer(self):
        """Determina la cantidad de riqueza a transferir
        basándose en una distribución de Pareto."""
        # Definir los parámetros de la distribución de Pareto
        a = 3.0 # Parámetro de forma; puede ajustarse para
        cambiar la “desigualdad”
        m = 1.0 # Escala mínima

        # Generar un valor de la distribución de Pareto
        return (np.random.pareto(a) + 1) * m

class MoneyModel(mesa.Model):
```

```

"""Modelo que contiene varios agentes MoneyAgent."""

def __init__(self, N):
    self.num_agents = N
    self.schedule = mesa.time.RandomActivation(self)
    self.all_wealth = [] # Lista para almacenar la
    riqueza total al final

# Crear agentes
for i in range(self.num_agents):
    agent = MoneyAgent(i, self)
    self.schedule.add(agent)

def step(self):
    """Ejecutar un paso del modelo."""
    self.schedule.step()

    # Al final de cada paso, agregar la riqueza
    total a la lista
    total_wealth = sum([agent.wealth for agent in
    self.schedule.agents])
    self.all_wealth.append(total_wealth)

def run_model(steps):
    """Función para ejecutar el modelo y mostrar
    el gráfico al final."""
    model = MoneyModel(100)
    for i in range(steps):
        model.step()

# Recolectar la riqueza de cada agente al final
de la simulación
agent_wealth = [agent.wealth for agent
in model.schedule.agents]

# Asegurar que el valor máximo para los bins sea un entero
max_wealth = int(max(agent_wealth))

# Graficar la distribución de la riqueza por agente
plt.hist(agent_wealth, bins=range(max_wealth + 1))
plt.title("Distribución de la Riqueza por Agente al Final
de la Simulación")
plt.xlabel("Riqueza")
plt.ylabel("Número de Agentes")
plt.show()

# Ejecutar la simulación
run_model(100)

```

En este código, hemos añadido

```
agent_wealth=[agent.wealth for agent in model.schedule.agents]
```

para recolectar la riqueza de cada agente. Además, para asegurarnos de que el valor máximo de los bins es un entero, se usa el código:

```
max_wealth = int(max(agent_wealth))
```

El gráfico que se obtiene aparece en la figura 9.7. Nótese que el resultado es consistente con el modelo simple, en donde la dinámica era solo intercambiar riqueza entre los agentes si es que ellos contaban por lo menos con una unidad de riqueza. Los dos modelos se basan en dinámicas diferentes, pero dan resultados similares. Si estuviéramos pensando cuál modelo refleja mejor la realidad, ¿con cuál se quedaría usted?

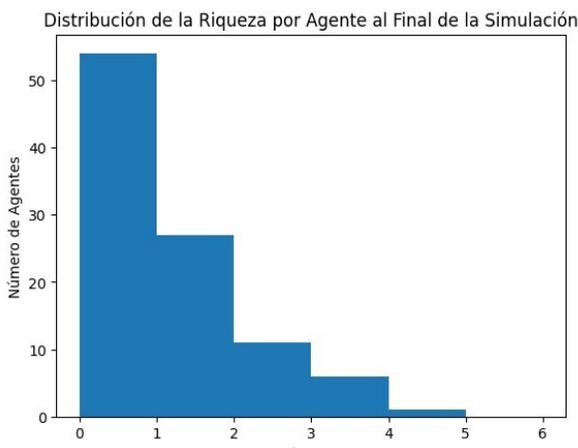


FIGURA 9.7. DISTRIBUCIÓN DE LA RIQUEZA AL FINAL DE LA SIMULACIÓN PARA EL MODELO DE BOLTZMANN CON DINÁMICA DE INTERCAMBIO DADA POR LA FUSIÓN DE DISTRIBUCIÓN DE PARETO.

9.3. EL MODELO DE AXELROD DE DIFUSIÓN CULTURAL

El modelo de difusión cultural de Robert Axelrod es un marco teórico fundamental en el campo de las ciencias sociales computacionales, especialmente en el estudio de la dinámica cultural¹². Este modelo se basa en el concepto de modelado basado en agentes y fue propuesto por Axelrod en 1997 para investigar cómo se difunde la cultura en una sociedad artificial y cómo se forman y mantienen las fronteras culturales. A continuación, se desglosan los aspectos clave de este modelo.

9.3.1. *Conceptos fundamentales*

- a. Agentes y cultura: En el modelo de Axelrod, los agentes representan individuos o actores sociales. Cada agente tiene un conjunto de atributos o rasgos culturales, que pueden incluir idioma, vestimenta, costumbres, valores, etc. Estos atributos están representados por variables que pueden adoptar una gama de estados.
- b. Interacción local: Los agentes interactúan con sus vecinos inmediatos. La probabilidad de que dos agentes interactúen depende de su similitud cultural: cuanto más similares son, más probable es que interactúen.
- c. Influencia y homogeneización: Cuando dos agentes interactúan, hay una posibilidad de que uno influya en el otro, lo que lleva a una mayor similitud cultural entre ellos. Esto puede llevar a una homogeneización cultural en el ámbito local.

9.3.2. *Dinámica del modelo*

- a. Configuración inicial: Los agentes se sitúan en una red o estructura espacial, comúnmente una cuadrícula. En un inicio, sus rasgos culturales se asignan al azar.
- b. Interacción y adaptación: En cada paso del modelo, un agente se selecciona al azar junto con uno de sus vecinos. Si comparten algunos, pero no todos los rasgos culturales, entonces hay una

¹² Para una discusión detallada de los fenómenos de la aculturización y la transculturización, véase el interesante y clásico trabajo de Foladori (1971).

probabilidad de que uno de los rasgos discordantes del vecino se adapte para coincidir con el agente seleccionado.

- c. Proceso iterativo: Este proceso se repite muchas veces, lo que permite observar cómo evolucionan las pautas culturales a lo largo del tiempo.

9.3.3. Implicaciones y observaciones

- a. Emergencia de regiones culturales: Una de las observaciones clave del modelo es la formación de regiones culturales homogéneas, separadas por fronteras donde los rasgos culturales cambian bruscamente.
- b. Importancia de la interacción local: El modelo muestra que la interacción local puede llevar a la diversidad cultural a gran escala, a pesar de la tendencia hacia la homogeneización a pequeña escala.
- c. Sensibilidad a las condiciones iniciales: El resultado final del modelo es altamente sensible a las condiciones iniciales, destacando la naturaleza compleja y no lineal de la dinámica cultural.

9.3.4. Relevancia en las ciencias sociales

El modelo de Axelrod es significativo porque ofrece una forma de entender cómo las microinteracciones entre individuos pueden llevar a patrones culturales macroscópicos. También proporciona un marco para explorar cómo las pequeñas diferencias iniciales en las culturas pueden persistir e incluso amplificarse con el tiempo, un concepto relevante en la discusión sobre multiculturalismo, integración social y dinámicas de grupo. En resumen, el modelo de difusión cultural de Axelrod es una herramienta poderosa para analizar la complejidad de la interacción social y la formación de patrones culturales, utilizando un enfoque basado en la simulación de agentes que ha sido básica en el desarrollo de las ciencias sociales computacionales.

9.3.5. Aspectos matemáticos del modelo de Axelrod

Como se ha visto el modelo de difusión cultural de Axelrod se centra en la interacción y difusión de rasgos culturales entre agentes en una po-

blación. Se basa en la idea de que la similitud promueve la interacción, y se puede describir mediante las siguientes nociones:

1. Agentes: La población se compone de N agentes, donde cada agente i tiene un vector de rasgos culturales representado como (\vec{C}_i) .
2. Similitud cultural: La similitud entre dos agentes i y j se mide mediante la distancia de Hamming¹³, que cuenta cuántos elementos de sus vectores de rasgos culturales son diferentes:

$$d_{ij} = \sum_{k=1}^F |C_{ik} - C_{jk}|$$

Donde F es la cantidad de rasgos culturales.

3. Interacción: Los agentes interactúan con otros que son culturalmente similares. La probabilidad de que un agente i interactúe con un agente j depende de su similitud cultural:

$$P_{ij} = \frac{1 - d_{ij}/F}{1 + s}$$

Donde s es un parámetro de sensibilidad que controla la influencia de la similitud.

4. Difusión cultural: Durante la interacción, los agentes pueden adoptar rasgos culturales del otro. Si i interactúa con j , el agente i puede adoptar uno de los rasgos culturales de j con probabilidad proporcional a la similitud:

$$C_{ik} \rightarrow C_{jk} \text{ con probabilidad } \frac{d_{ij}}{F}$$

¹³ La distancia de Hamming, nombrada así por el científico Richard Hamming, es una medida utilizada en diversas disciplinas como la teoría de la información, la criptografía y las ciencias computacionales, para cuantificar el grado de diferencia entre dos secuencias de igual longitud. En un contexto más general, puede aplicarse a cualquier par de secuencias (o conjuntos de símbolos) de la misma longitud para determinar cuán diferentes son entre sí. La distancia de Hamming entre dos cadenas de la misma longitud se calcula contando el número de posiciones en las que los símbolos correspondientes son diferentes. Por ejemplo, en el caso de cadenas binarias, se cuenta cuántos *bits* son diferentes entre dos cadenas. Para ilustrar, si tenemos dos cadenas binarias A = 1101101 y B = 1001111, la distancia de Hamming sería 2, ya que difieren en dos posiciones (tercera y cuarta posición).

5. Evolución temporal: Repetir las interacciones y la difusión cultural a lo largo del tiempo para ver cómo evolucionan los rasgos culturales de la población.

Este modelo de difusión cultural de Axelrod se basa en la idea de que la similitud cultural y la interacción promueven la homogeneización de los rasgos culturales en una población.

Para proporcionar un ejemplo numérico del modelo de difusión cultural de Axelrod, consideremos una población de cinco agentes ($N = 5$) y tres rasgos culturales ($F = 3$). Supongamos que inicialmente, los agentes tienen los siguientes vectores de rasgos culturales:

Agente 1: [1, 0, 1]
Agente 2: [0, 1, 0]
Agente 3: [1, 1, 0]
Agente 4: [0, 0, 1]
Agente 5: [0, 1, 1]

También, establezcamos un parámetro de sensibilidad $s = 0.5$ para controlar la influencia de la similitud cultural en la probabilidad de interacción.

Ahora realizaremos iteraciones de interacción y difusión cultural en el modelo. Supongamos que, en cada iteración, los agentes interactúan con otros agentes según la probabilidad de interacción basada en la similitud cultural, y pueden adoptar rasgos culturales de los demás.

Después de algunas iteraciones, los vectores de rasgos culturales podrían cambiar de la siguiente manera:

Iteración 1:
Agente 1 interactúa con agente 3 y adopta el rasgo 0 de agente 3.
Iteración 2:
Agente 2 interactúa con agente 5 y adopta el rasgo 1 de agente 5.
Iteración 3:
Agente 4 interactúa con agente 5 y adopta el rasgo 0 de agente 5.
Después de estas iteraciones, los vectores de rasgos culturales podrían verse así:
Agente 1: [1, 0, 0] (adoptó el rasgo 0 de agente 3)
Agente 2: [0, 1, 1] (adoptó el rasgo 1 de agente 5)
Agente 3: [1, 1, 0] (sin cambios)
Agente 4: [0, 1, 1] (adoptó el rasgo 1 de agente 5)
Agente 5: [0, 1, 1] (sin cambios)

Este proceso de interacción y difusión cultural continúa, y con el tiempo, los rasgos culturales tienden a homogeneizarse en la población debido a la influencia de la similitud cultural y la adopción de rasgos de agentes similares.

La distancia de Hamming entre dos agentes, como el agente 1 y el agente 3 en el ejemplo anterior, se calcula contando la cantidad de elementos en sus vectores de rasgos culturales que son diferentes.

Para el Agente 1: [1, 0, 0]

Para el Agente 3: [1, 1, 0]

La distancia de Hamming se calcula como:

$$d_{13} = \sum_{k=1}^3 |1 - 1| + |0 - 1| + |0 - 0| = 1$$

Por tanto, la distancia de Hamming entre el agente 1 y el agente 3 es igual a 1, lo que significa que tienen un rasgo cultural diferente en uno de los tres rasgos en sus vectores.

La probabilidad de interacción entre el agente 1 y el agente 3 según el modelo de Axelrod se calcula utilizando la siguiente fórmula:

$$P_{13} = \frac{1 - d_{13}/F}{1 + s}$$

Donde: (d_{13}) es la distancia de Hamming entre el agente 1 y el agente 3, que calculamos previamente como ($d_{13} = 1$). (F) es la cantidad de rasgos culturales, en este caso, ($F = 3$). (s) es el parámetro de sensibilidad, que establecimos como ($s = 0.5$) en el ejemplo anterior.

Sustituyendo los valores:

$$P_{13} = \frac{1 - 1/3}{1 + 0.5} = \frac{2/3}{1.5} = \frac{2}{3} \cdot \frac{2}{3} = \frac{4}{9}$$

Por tanto, la probabilidad de interacción entre el agente 1 y el agente 3, según el modelo de Axelrod, es ($\frac{4}{9}$)=0.444. Esta probabilidad determina la posibilidad de que interactúen y, en consecuencia, la posibilidad de que el agente 1 adopte rasgos culturales del agente 3 durante la interacción.

Para calcular la interacción entre el agente 1 y el agente 3 utilizando la probabilidad calculada ($P_{13} = \frac{4}{9}$), podemos utilizar un enfoque pro-

abilístico. En este contexto, la interacción se modela como un evento estocástico.

Aquí hay un procedimiento paso a paso:

1. Se genera un número aleatorio entre 0 y 1, por ejemplo, (r).
 2. Se compara con la probabilidad calculada (P_{13}).
- Si ($r \leq P_{13}$), entonces la interacción ocurre entre el agente 1 y el agente 3.
 - Si ($r > P_{13}$), entonces la interacción no ocurre.
3. Si la interacción ocurre (es decir, ($r \leq P_{13}$)), el siguiente paso es decidir qué rasgo cultural del agente 3 adoptará el agente 1. Esto también se hace mediante un enfoque probabilístico.
 4. Para determinar qué rasgo se adoptará, se puede generar otro número aleatorio entre 0 y 1, por ejemplo, (r').
 5. Divida el intervalo $[0, 1]$ en partes iguales según la cantidad de rasgos culturales. En este caso, como hay tres rasgos culturales, se divide el intervalo en tres partes iguales, ya que, según la fórmula de Axelrod, se tiene que el cambio de un aspecto cultural es:

Recordemos que $F = 3$ y $d_{ij} = 1$

Si ($0 \leq r' < 1/3$), el agente 1 adopta el primer rasgo del agente 3.

Si ($1/3 \leq r' < 2/3$), el agente 1 adopta el segundo rasgo del agente 3.

Si ($2/3 \leq r' < 1$), el agente 1 adopta el tercer rasgo del agente 3.

Este proceso simula la interacción entre el agente 1 y el agente 3 según el modelo de Axelrod, donde la probabilidad de interacción (P_{13}) determina la frecuencia esperada de interacciones exitosas, y la selección de rasgos culturales se realiza de manera probabilística.

9.3.6. Modelo computacional del modelo de Axelrod

Vamos a construir un programa en Python que simule el modelo de difusión cultural de Axelrod, siguiendo las siguientes especificaciones:

- Número de agentes (N): 50.
- Número de rasgos culturales (F): 3.

- Dimensiones de la cuadrícula: 10×10.
- Vecindad: Vecindad de von Neumann o Moore.

La cuadrícula será gestionada por SingleGrid de MESA, y la vecindad de von Neumann se usará para determinar con qué otros agentes interactúa cada agente. Cada agente tendrá un conjunto de rasgos culturales (en este caso, 3), y la probabilidad de interacción entre dos agentes será proporcional a la cantidad de rasgos que comparten.

El esqueleto básico del código es:

1. Definir la clase del agente: Cada agente tendrá un conjunto de rasgos culturales.
2. Definir el modelo: Inicia la cuadrícula y los agentes.
3. Paso del modelo: Define cómo los agentes interactúan en cada paso del tiempo.

Vamos a construir este código paso a paso.

1. Definir la clase del agente

Primero, definimos la clase “CulturalAgent”, que hereda de la clase “Agent” de MESA. Cada agente tendrá un conjunto de rasgos culturales, que serán representados por números enteros.

2. Definir el modelo

Luego, creamos la clase “CulturalDiffusionModel”, que hereda de la clase “Model” de MESA. Aquí iniciamos la cuadrícula y colocamos a los agentes en ella.

3. Paso del modelo

Finalmente, apliquemos la lógica de la interacción entre agentes en el método “step” del modelo.

Vamos a escribir el código para estas partes. Como vimos, lo primero es importar las clases que se van a utilizar de MESA. Estas clases son: “Agent”, “Model”, “SingleGrid” y “RandomActivation”. De Python se importa la clase “Random”. El código es el siguiente:

```
from mesa import Agent, Model
from mesa.space import SingleGrid
from mesa.time import RandomActivation
```

```
import random
```

Como vimos anteriormente, el código consta de dos partes principales: la definición de la clase “CulturalAgent” y la clase “CulturalDiffusionModel”. El código para la clase “CulturalAgent” es el siguiente:

```
class CulturalAgent(Agent):
    """ Un agente con rasgos culturales. """

    def __init__(self, unique_id, model, num_features):
        super().__init__(unique_id, model)
        # Inicializa los rasgos culturales del agente
        self.features = [random.randint(0, 1)
                        for _ in range(num_features)]

    def step(self):
        # Elegir un vecino al azar
        neighbors = self.model.grid.get_neighbors
        (self.pos, moore=False)
        if neighbors:
            other = random.choice(neighbors)
            # Interactuar con el vecino si comparten
            al menos un rasgo
            if any(self.features[i] == other.features[i]
                  for i in range(len(self.features))):
                # Seleccionar un rasgo que sea diferente y copiarlo
                different_features = [i for i in
                                     range(len(self.features)) if self.features[i] !=
                                     other.features[i]]
                if different_features:
                    feature_to_copy = random.choice(different_features)
                    self.features[feature_to_copy] =
                    other.features[feature_to_copy]
```

Cada agente tiene una lista de rasgos culturales (“features”), iniciados aleatoriamente. Dichos rasgos son tomados del conjunto {0,1}. Ya que son solo tres rasgos se tienen vectores de longitud 3. En el método “step”, el agente interactúa con un vecino elegido al azar. La interacción ocurre si comparten al menos un rasgo. En tal caso, el agente copia un rasgo diferente del vecino.

A continuación, definimos el código para la clase “CulturalDiffusionModel”.

```

class CulturalDiffusionModel(Model):
    """ Modelo de difusión cultural. """

    def __init__(self, N, width, height, num_features):
        self.num_agents = N
        self.grid = SingleGrid(width, height, torus=True)
        self.schedule = RandomActivation(self)
        self.num_features = num_features

        # Crear una lista de todas las posiciones
        # posibles y mezclarla
        all_positions = [(x, y) for x in range(width)
                          for y in range(height)]
        random.shuffle(all_positions)

        # Crear agentes y asignarles una posición única
        for i in range(self.num_agents):
            agent = CulturalAgent(i, self, self.num_features)
            # Asegúrate de que no haya más agentes que
            # posiciones disponibles
            if i < len(all_positions):
                pos = all_positions.pop()
                self.grid.place_agent(agent, pos)
                self.schedule.add(agent)
            else:
                break # Salir si no hay más posiciones disponibles

    def step(self):
        # Avanzar un paso en el modelo
        self.schedule.step()

```

El modelo crea una cuadrícula de 10×10 usando “SingleGrid”. Se generan cincuenta agentes “CulturalAgent” con tres rasgos culturales cada uno. Para asegurarnos de que cada agente esté en una única coordenada sin traslapes en la cuadrícula, necesitamos asignar posiciones de manera que cada agente ocupe un espacio único. Una forma de hacerlo es utilizar un enfoque de distribución aleatoria sin reemplazo. Esto significa que seleccionaremos posiciones al azar para cada agente, pero una vez que una posición ha sido asignada a un agente, no estará disponible para otros. Para ello, podemos generar una lista de todas las posibles coordenadas de la cuadrícula y luego seleccionar posiciones de esta lista de forma aleatoria, eliminando cada posición seleccionada de la lista para evitar traslapes y añadidos a un planificador “RandomActivation”, que determina el orden de sus acciones. En este código se crea una lis-

ta de todas las posiciones posibles en la cuadrícula “all_positions” y se baraja. Luego, para cada agente, se saca una posición de esta lista y se coloca al agente en esa posición. Esto garantiza que no haya traslapes, ya que cada posición se usa una sola vez. El método “step” del modelo avanza un paso en el tiempo, permitiendo que cada agente ejecute su método “step”.

Una vez que tenemos el código listo, este se puede ejecutar por un número determinado de pasos, por ejemplo cien pasos. El código para esto es:

```
# Crear y ejecutar el modelo
model = CulturalDiffusionModel(N, width, height, num_features)
for i in range(100): # Simular 100 pasos
    model.step()
```

Para visualizar los resultados, podríamos instalar una visualización basada en Matplotlib o utilizar el servidor de visualización de MESA, esto último requiere una configuración adicional. En primer lugar, vamos a importar las clases “ModularServer”, “CanvasGrid” y “UserSettableParameter”. Estas clases están en `mesa.visualization.ModularVisualization`, `mesa.visualization.modules` y `mesa.visualization.UserParam`, respectivamente. Entonces, ahora modificamos el programa importando todas las clases necesarias. El código es el siguiente:

```
from mesa import Agent, Model
from mesa.space import SingleGrid
from mesa.time import RandomActivation
from mesa.visualization.ModularVisualization import ModularServer
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.UserParam import UserSettableParameter
import random
```

Luego, después de definir nuestras clases “CulturalAgent” y “CulturalDiffusionModel” se inserta el siguiente código:

```
def agent_portrayal(agent):
    """ Representa un agente con un color basado en
    la suma de sus rasgos culturales. """
    portrayal = {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
    "Layer": 0}
```

```

# Mapear la suma de rasgos culturales a un color
color_map = {0: "red", 1: "blue", 2: "green", 3: "yellow"}
sum_features = sum(agent.features)
portrayal["Color"] = color_map[sum_features]

return portrayal

# Configurar la visualización
grid = CanvasGrid(agent_portrayal, width, height, 500, 500)
server = ModularServer(CulturalDiffusionModel,
                       [grid],
                       "Modelo de Difusión Cultural",
                       {"N": 50, "width": 10, "height": 10, "num_features": 3})

server.port = 8521 # El puerto predeterminado
server.launch()

```

Para la correcta visualización de los rasgos culturales de cada agente hemos usado la suma de dichos rasgos. Considerando que, por ejemplo, el rasgo cultural $[0, 1, 1]$ es equivalente al rasgo cultural $[1, 0, 1]$ y $[1, 1, 0]$, ya que ambos tienen la misma distancia de Hamming, que es 2. Esta equivalencia es importante en el modelo de difusión cultural, ya que indica un grado similar de similitud o compatibilidad cultural entre agentes con esos conjuntos de rasgos.

Si corre usted el código tal como se muestra, aparecerá una ventana en su *browser* predestinado, en nuestro caso, Microsoft Edge. La ventana con el modelo aparece en la figura 9.8.



FIGURA 9.8. VENTANA DE MESA CREADA PARA VISUALIZAR EL MODELO DE AXELROD.

Nótese que tenemos cuatro culturas, las de suma cero, las de suma 1, las de suma 2 y las de suma 3. La cultura de suma cero se representa por el color rojo, la cultura de suma uno se representa por el azul, la cultura de suma dos por el verde y la cultura de suma tres por el amarillo.

Nótese que el visualizador tiene al lado derecho tres botones (figura 9.9). Estos botones sirven para iniciar la simulación (botón “Start”), para la simulación (botón “Step”) y reiniciar a la configuración inicial (botón “Reset”).

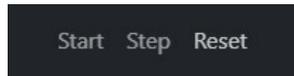


FIGURA 9.9. BOTONES DE INICIO, PARO Y REINICIO DE LA SIMULACIÓN.

Ahora que ya hemos creado una visualización del modelo, contamos con un laboratorio que nos permite observar cómo evolucionan las culturas en el tiempo en esta sociedad artificial de cincuenta agentes con cuatro culturas diferentes. Corramos el modelo para ver qué resultados nos arroja.

Partiendo de la configuración inicial mostrada en la figura 9.8, observamos que el modelo converge a la siguiente distribución espacial que se muestra en la figura 9.10.

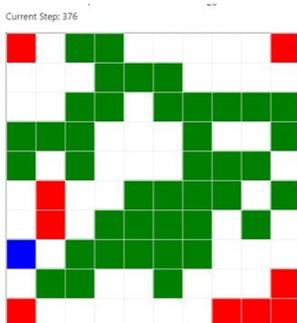


FIGURA 9.10. CONFIGURACIÓN DE LAS CULTURAS DESPUÉS DE 376 PASOS.

Como puede observarse, la cultura dominante después de 376 pasos es la de suma 2. Existen “islas” culturales debido a las condiciones topológicas de la distribución inicial y a que la dinámica se especificó usando una vecindad de von Neumann. Estas “islas” se pueden observar en las esquinas de la cuadrícula donde vemos que predomina la cultura de suma cero. La “isla” de cultura de suma cero que se observa en el lado izquierdo se debe a las condiciones de usar una vecindad de von Neumann para establecer los vecinos con los que se tiene interacción. Lo mismo pasa con la “isla” de cultura de suma uno de la izquierda.

Para experimentar con el modelo, necesitamos replicar las condiciones iniciales tanto para los rasgos culturales como para la posición de los agentes en la cuadrícula. Esto lo haremos usando el concepto de *semilla* (“seed”) en la generación de números pseudoaleatorios. El código lo cambiaremos de la siguiente forma:

```
import random

# ... (otras importaciones y definiciones de clases)

class CulturalDiffusionModel(Model):
    """ Modelo de difusión cultural. """

    def __init__(self, N, width, height, num_features, seed=None):
        self.num_agents = N
        self.grid = SingleGrid(width, height, torus=True)
        self.schedule = RandomActivation(self)
        self.num_features = num_features
```

```

# Establecer la semilla para la replicabilidad
if seed is not None:
    random.seed(seed)

# Crear una lista de todas las posiciones
posibles y mezclarla
all_positions = [(x, y) for x in range(width)
for y in range(height)]
random.shuffle(all_positions)

# Crear agentes y asignarles una posición única
for i in range(self.num_agents):
    agent = CulturalAgent(i, self, self.num_features)
    # Asegúrate de que no haya más agentes que
    posiciones disponibles
    if i < len(all_positions):
        pos = all_positions.pop()
        self.grid.place_agent(agent, pos)
        self.schedule.add(agent)
    else:
        break # Salir si no hay más posiciones disponibles

def step(self):
    # Avanzar un paso en el modelo
    self.schedule.step()

# ... (código restante para agent_portrayal y configuración del
servidor)

# Ejemplo de cómo lanzar el servidor con una semilla específica
if __name__ == "__main__":
    server = ModularServer(CulturalDiffusionModel,
        [grid],
        "Modelo de Difusión Cultural",
        {"N": 50, "width": 10, "height": 10, "num_features":
3, "seed": 12345})

    server.port = 8522 # El puerto predeterminado
    server.launch()

```

Veamos ahora como se comporta el modelo utilizando la *semilla* (“seed”) 12345. En la figura 9.11, se muestra la configuración inicial corriendo el programa con la *semilla*.

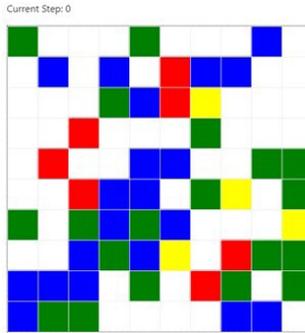


FIGURA 9.11. CONFIGURACIÓN INICIAL CON LA SEMILLA 12345.

En la figura 9.12 se muestra la convergencia del modelo a una configuración estable.

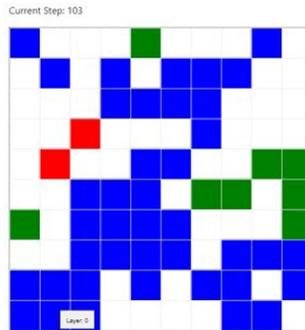


FIGURA 9.12. CONFIGURACIÓN ESTABLE DESPUÉS DE 103 ITERACIONES.

Ahora, lo que nos interesa es volver a correr el modelo con las mismas condiciones iniciales. Para ello, solo necesitamos usar el botón “Reset” y veremos que nos devuelve la misma configuración inicial mostrada en la figura 9.11. Para fines de ilustración, colocaremos ambas condiciones iniciales en una sola figura.

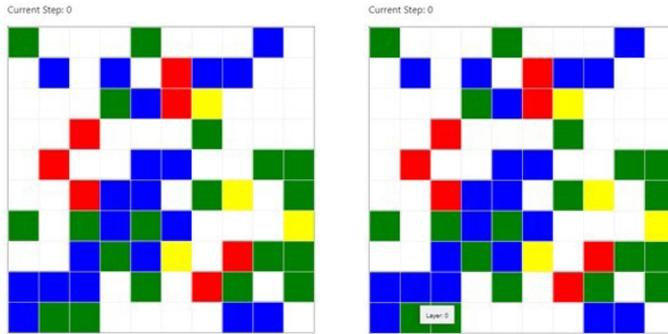


FIGURA 9.13. CONFIGURACIONES INICIALES, LA FIGURA A) CORRESPONDE A LA PRIMERA SIMULACIÓN Y LA B) A LA SEGUNDA SIMULACIÓN DESPUÉS DE PULSAR “RESET”.

Veamos ahora cómo evoluciona el modelo, la evolución después de 117 iteraciones se muestra en la figura 9.14.

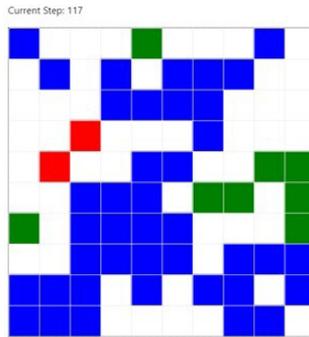


FIGURA 9.14. EVOLUCIÓN DEL NUEVO MODELO CON LAS MISMAS CONDICIONES INICIALES.

Nótese que si no se varía la *semilla* (“seed”) de los números aleatorios en el modelo de difusión cultural de Axelrod, y todos los demás parámetros del modelo permanecen constantes, se espera que el modelo produzca los mismos resultados en cada ejecución. Esto se debe a que la semilla fija del generador de números aleatorios garantiza la replicabilidad de la se-

cuencia de números aleatorios generados, lo que a su vez determina tanto la posición inicial de los agentes como sus rasgos culturales iniciales.

Sin embargo, la utilidad de las condiciones iniciales no es solo poder ver que el modelo converge siempre a la misma distribución espacial; veamos ahora cómo se comporta el modelo si hacemos una modificación en nuestro código para que la vecindad no sea la de von Neumann, sino la de Moore. Para ello vamos a nuestro código en la clase “CulturalAgent” y en la parte del código siguiente:

```
neighbors = self.model.grid.get_neighbors(self.pos, moore=False)
```

Sustituyamos el parámetro “moore = False”, por “moore = True”. Ahora, la dinámica del modelo ha cambiado, pero no las condiciones iniciales. Volvamos ahora a correr nuestro programa y veamos cómo evoluciona. La configuración inicial y la configuración de convergencia se muestra en la figura 9.15.

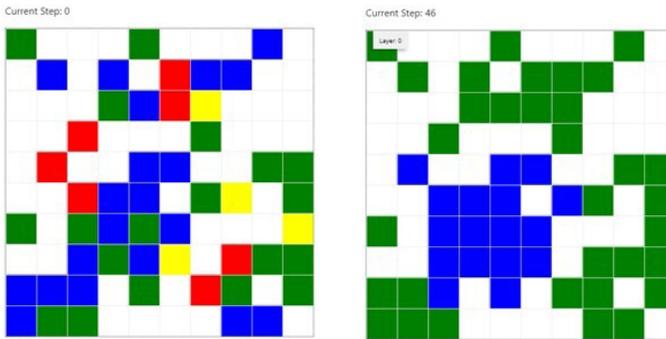


FIGURA 9.15. INICIO Y FINAL DE LA SIMULACIÓN USANDO UNA VECINDAD DE MOORE.

Nótese que en el caso de la vecindad de von Neumann tenemos un claro predominio de la cultura de suma uno, mientras que en la vecindad de Moore el predominio es de la cultura de suma dos. Además, podemos observar que culturas minoritarias persisten aisladas en la vecindad de von Neumann, mientras que en el caso de la vecindad de Moore solo hay dos culturas.

Ahora veremos cómo explorar el modelo ante pequeños cambios en sus condiciones iniciales. Para cambiar la posición y los rasgos culturales de un subconjunto específico de agentes (digamos, cinco agentes) en el que podemos modificar el proceso de inicio de los agentes en la clase “CulturalDiffusionModel”. El proceso sería el siguiente:

1. Elegir los agentes para modificar: Primero, decidimos qué agentes específicos queremos modificar. Podríamos elegirlos al azar o seleccionar los primeros cinco, por ejemplo.
2. Modificar la posición inicial: Podemos asignar nuevas posiciones a estos agentes seleccionados, asegurándonos de que las nuevas posiciones no estén ocupadas por otros agentes.
3. Modificar los rasgos culturales: Cambiar los rasgos culturales de estos agentes seleccionados asignando nuevos valores a sus características.

Vamos a modificar la clase “CulturalDiffusionModel” para ilustrar cómo hacer esto:

```
class CulturalDiffusionModel(Model):
    """ Modelo de difusión cultural. """

    def __init__(self, N, width, height, num_features, seed=None):
        self.num_agents = N
        self.grid = SingleGrid(width, height, torus=True)
        self.schedule = RandomActivation(self)
        self.num_features = num_features

        # Establecer la semilla para la replicabilidad
        if seed is not None:
            random.seed(seed)

        # Crear una lista de todas las posiciones
        # posibles y mezclarla
        all_positions = [(x, y) for x in range(width)
                        for y in range(height)]
        random.shuffle(all_positions)

        # Crear agentes y asignarles una posición única
        for i in range(self.num_agents):
            agent = CulturalAgent(i, self, self.num_features)
            if i < len(all_positions):
                pos = all_positions.pop()
                self.grid.place_agent(agent, pos)
```

```

        self.schedule.add(agent)

# Modificar la posición y rasgos culturales
de 5 agentes específicos
for i in range(5): # Elegir los primeros 5 agentes
como ejemplo
    agent = self.schedule.agents[i]
    # Asignar una nueva posición (asegurarse de que
sea una posición no ocupada)
    new_pos = all_positions.pop()
    self.grid.move_agent(agent, new_pos)
    # Cambiar rasgos culturales
    agent.features = [random.randint(0, 1)
for _ in range(num_features)]

# ... (resto del código)

# ... (resto del código para configuración del servidor y lanzamiento)
'''

```

En este código, después de colocar a todos los agentes en posiciones únicas, seleccionamos los primeros cinco agentes y modificamos tanto su posición como sus rasgos culturales. Las nuevas posiciones se eligen de las posiciones restantes para asegurar que no haya traslapes, y los rasgos culturales se generan de nuevo de manera aleatoria.

Este enfoque permite explorar cómo pequeñas variaciones en la posición y rasgos culturales de un subconjunto de agentes pueden afectar la dinámica global del modelo.

Veamos en la figura 9.16 la nueva configuración inicial; recuerde-se que seguimos usando la *semilla* (“seed”) 12345 para la generación de los números pseudoaleatorios.

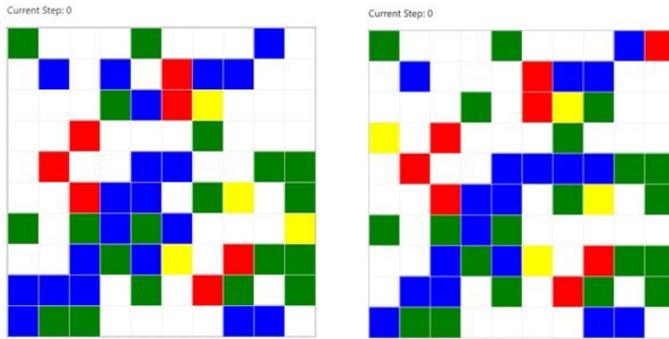


FIGURA 9.16. A) CONFIGURACIÓN INICIAL ORIGINAL QUE SE HA ESTADO USANDO Y B) CONFIGURACIÓN INICIAL CON SOLO CINCO AGENTES CAMBIADOS.

Exploremos ahora el comportamiento de esta nueva configuración inicial. El resultado de la simulación se muestra en la figura 9.18. Nótese que, de manera sorprendente, el comportamiento del modelo ha convergido a una sociedad artificial monocultural. Una monocultura se caracteriza por la ausencia de otros rasgos culturales en la sociedad, lo cual significa que todos los agentes comparten los mismos rasgos culturales. Lo notable es que solo se modificaron cinco de los agentes con respecto a su posición y rasgos culturales. Este pequeño cambio fue tan significativo que llevó al modelo converger rápidamente (en sesenta iteraciones) a un estado estable, esto es, un estado donde ya no puede haber cambios y además presentando la supremacía de una sola cultura, en este caso la cultura de suma tres. Algo interesante, ya que dicha cultura desaparecía con las condiciones iniciales que se usaron en un principio con la *semilla* (“seed”) 12345. Debemos hacer notar que se usó para esta simulación una vecindad de Moore.

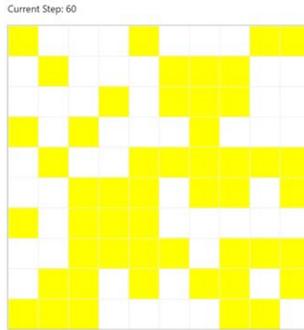


FIGURA 9.17. RESULTADO DE LA SIMULACIÓN PARA LA CONFIGURACIÓN INICIAL CON SOLO CINCO AGENTES CAMBIADOS.

Ahora veamos cómo analizar el modelo utilizando algunas de las características de la biblioteca Matplotlib y la función “DataCollector” de MESA. Lo primero que se necesita hacer es recolectar datos sobre los rasgos culturales de los agentes a lo largo del tiempo y luego graficar estos datos. Una forma efectiva de hacerlo es utilizando la función “DataCollector” en MESA. El “DataCollector” permite recopilar datos específicos en cada paso del modelo. En nuestro caso, queremos recoger información sobre la cantidad de agentes con cada conjunto de rasgos culturales en cada paso. Luego, podemos usar estos datos para generar un gráfico de líneas que muestre la evolución de las culturas a lo largo del tiempo.

Para visualizar los datos recolectados al final de la simulación con un gráfico de líneas en la consola, necesitamos ejecutar una simulación del modelo fuera del servidor de visualización y luego graficar los resultados. La ejecución del servidor (“server.launch()”) y la generación del gráfico (“data.plot()”) son procesos separados y deben manejarse de manera independiente. Aquí se muestra cómo hacerlo:

- Ejecutar el modelo sin el servidor: Se crea y ejecuta una instancia del modelo por separado de la visualización del servidor para recoger los datos.
- Graficar los resultados: Después de ejecutar el modelo, se utilizan los datos recolectados para generar el gráfico.

El código modificado es el siguiente:

```
from mesa import Agent, Model
from mesa.space import SingleGrid
from mesa.time import RandomActivation
from mesa.datacollection import DataCollector
from mesa.visualization.ModularVisualization import ModularServer
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.UserParam import Slider
import matplotlib.pyplot as plt
import random

width, height = 10, 10

class CulturalAgent(Agent):
    """ Un agente con rasgos culturales. """

    def __init__(self, unique_id, model, num_features):
        super().__init__(unique_id, model)
        # Inicializa los rasgos culturales del agente
        self.features = [random.randint(0, 1) for _ in range
            (num_features)]
    def step(self):
        # Elegir un vecino al azar
        neighbors = self.model.grid.get_neighbors
            (self.pos, moore=True)
        if neighbors:
            other = random.choice(neighbors)
            # Interactuar con el vecino si comparten
            al menos un rasgo
            if any(self.features[i] == other.features[i]
                for i in range(len(self.features))):
                # Seleccionar un rasgo que sea diferente y copiarlo
                different_features = [i for i in
                    range(len(self.features)) if self.features[i]
                    != other.features[i]]
                if different_features:
                    feature_to_copy = random.choice
                        (different_features)
                    self.features[feature_to_copy] =
                        other.features[feature_to_copy]

    def hamming_distance(features1, features2):
        """Calcula la distancia de Hamming entre dos conjuntos
            de rasgos."""
        return sum(f1 != f2 for f1, f2 in zip(features1, features2))

    def count_hamming_distance(model, distance):
        """Cuenta agentes con una distancia de Hamming específica."""
```

```

base_culture = [0] * model.num_features # Puedes cambiar
esto según lo que consideres como base
count = sum(hamming_distance(agent.features, base_culture)
== distance for agent in model.schedule.agents)
return count

class CulturalDiffusionModel(Model):
    def __init__(self, N, width, height, num_features, seed=None):
        self.num_agents = N
        self.grid = SingleGrid(width, height, torus=True)
        self.schedule = RandomActivation(self)
        self.num_features = num_features

    # Establecer la semilla para la replicabilidad
    if seed is not None:
        random.seed(seed)

    # Crear una lista de todas las posiciones posibles y mezclarla
    all_positions = [(x, y) for x in range(width) for
y in range(height)]
    random.shuffle(all_positions)

    # Crear agentes y asignarles una posición única
    for i in range(self.num_agents):
        agent = CulturalAgent(i, self, self.num_features)
        if i < len(all_positions):
            pos = all_positions.pop()
            self.grid.place_agent(agent, pos)
            self.schedule.add(agent)
        else:
            break # Salir si no hay más posiciones disponibles

    # Inicializar DataCollector
    self.datacollector = DataCollector(
        {f"Hamming_{d}": lambda m, d=d: count_hamming_distance
(m, d) for d in range(num_features + 1)}
    )

    def step(self):
        # Recoger datos y luego avanzar el modelo
        self.datacollector.collect(self)
        self.schedule.step()

def agent_portrayal(agent):
    """ Representa un agente con un color basado en la suma
de sus rasgos culturales. """
    portrayal = {"Shape": "rect", "w": 1, "h": 1, "Filled":
"true", "Layer": 0}

    # Mapear la suma de rasgos culturales a un color
    color_map = {0: "red", 1: "blue", 2: "green", 3: "yellow"}

```

```

    sum_features = sum(agent.features)
    portrayal["Color"] = color_map[sum_features]

    return portrayal

# Ejecutar el modelo y recolectar datos
def run_model():
    model = CulturalDiffusionModel(50, 10, 10, 3, seed=12345)
    for i in range(100): # Número de pasos que quieres simular
        model.step()

    # Obtener y devolver los datos recolectados
    return model.datacollector.get_model_vars_dataframe()

# Ejemplo de cómo lanzar el servidor con una semilla específica
if __name__ == "__main__":
    # Ejecutar el modelo y graficar los resultados
    data = run_model()
    data.plot()
    plt.xlabel('Paso')
    plt.ylabel('Número de Agentes')
    plt.title('Evolución de las Culturas en el Modelo de Axelrod')
    plt.show()

    # Opcional: También puedes lanzar el servidor de
    # visualización si lo deseas
    # grid = CanvasGrid(agent_portrayal, width, height, 500, 500)
    # server = ModularServer(CulturalDiffusionModel,
    #                         [grid],
    #                         "Modelo de Difusión Cultural",
    #                         {"N": 50, "width": 10, "height": 10,
    #                          "num_features": 3, "seed": 12345})
    # server.port = 8533 # El puerto predeterminado
    # server.launch()

```

En este código, se ha separado la ejecución del modelo y la recolección de datos en una función “run_model()”. El código es el siguiente:

```

# Ejecutar el modelo y recolectar datos
def run_model():
    model = CulturalDiffusionModel(50, 10, 10, 3, seed=12345)
    for i in range(100): # Número de pasos que quieres simular
        model.step()
    # Obtener y devolver los datos recolectados
    return model.datacollector.get_model_vars_dataframe()

```

Después de ejecutar el modelo, se recopilan los datos y se devuelven para ser graficados. El código para hacer esto es el siguiente:

```

# Ejemplo de cómo lanzar el servidor con una semilla específica
if __name__ == "__main__":
    # Ejecutar el modelo y graficar los resultados
    data = run_model()
    data.plot()
    plt.xlabel('Paso')
    plt.ylabel('Número de Agentes')
    plt.title('Evolución de las Culturas en el Modelo de Axelrod')
    plt.show()

```

La visualización del gráfico ocurre fuera del contexto del servidor de visualización (ModularServer), lo que permite ver los resultados en la consola después de que la simulación ha finalizado. El gráfico que se obtiene aparece en la figura 9.18.

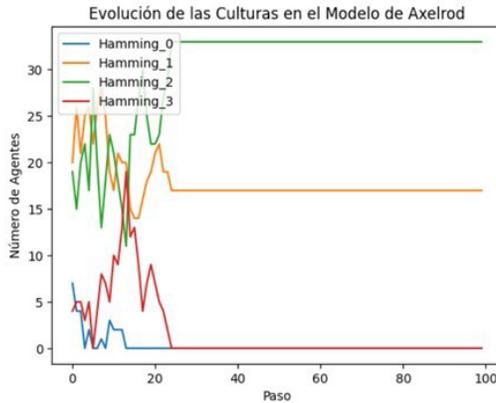


FIGURA 9.18. EVOLUCIÓN DE LAS CULTURAS PARA LA SEMILLA 12345.

CAPÍTULO 10. CONCLUSIONES

10.1. INTRODUCCIÓN

Desde el advenimiento de las computadoras y los métodos computacionales, las ciencias naturales y las ingenierías han progresado enormemente, adueñándose de estas tecnologías tanto para la investigación como para la formación de futuros científicos. Por su lado, las ciencias sociales han mostrado una aproximación que podríamos calificar de temerosa y tibia hacia el uso de las tecnologías computacionales para la indagación científica de los fenómenos sociales. La creciente complejidad de las interacciones humanas debidas a la globalización y la creciente digitalización de las sociedades hacen que las estructuras organizacionales tradicionales se vean rebasadas y presenten desafíos nuevos al entendimiento de los procesos de socialización de los individuos. El usar métodos computacionales y laboratorios de cómputo para realizar experimentos *in silico* que generen datos sobre fenómenos sociales difíciles de recabar *in situ*, y desarrollar programas educativos que usen la simulación para el entrenamiento de científicos sociales, es todavía un sueño en ciernes.

Este libro parte de una visión proactiva hacia los métodos computacionales y la simulación de los fenómenos sociales. Sus conclusiones, que se presentan a continuación, reflejan un resumen de los temas que se han tratado a lo largo del texto y que consideramos clave para la reflexión final de este trabajo académico.

10.2. IMPORTANCIA DEL MODELADO BASADO EN AGENTES EN LAS CIENCIAS SOCIALES

Como se ha visto a lo largo del libro, específicamente en la introducción y el primer capítulo, el modelado y simulación basada en agentes (MSBA) ha emergido como una herramienta crucial para comprender los sistemas sociales complejos. La complejidad de los sistemas sociales contemporáneos y sus interacciones con otros sistemas como los ecológicos hacen que el MSBA sea un enfoque computacional adecuado para tratar con dicha complejidad, que estriba en el número de agentes en juego, la no linealidad de sus interacciones y la estructura evolutiva de la toma de decisiones por parte de los seres humanos. El MSBA permite simular e indagar sobre la estructura de las interacciones de agentes autónomos entre sí y con su entorno, sea este social o físico (Helbing y Balietti, 2012).

Algunas de las principales ventajas del uso del MSBA son:

1. Permite simular y analizar la interacción de individuos (agentes) y su impacto colectivo en los sistemas sociales (Helbing y Balietti, 2012). Puede capturar emergencia, donde patrones macroscópicos surgen de las interacciones microscópicas.
2. Captura la heterogeneidad de los agentes, cada uno con comportamientos, atributos y relaciones sociales únicos (Dawid *et al.*, 2019).
3. Incorpora redes sociales y comunicación boca a boca para modelar cómo las decisiones individuales son influidas por interacciones sociales (Zadbood y Hoffenson, 2022) .
4. Permite análisis *what-if*⁴ para explorar escenarios contrafácticos y probar políticas antes de aplicarlas.

Dados estos beneficios, el MSBA se ha aplicado ampliamente en la literatura científica, en especial en la angloparlante, para estudiar fenómenos sociales como la difusión de innovaciones, la formación de opiniones,

¹⁴ Un análisis *what-if* (análisis de escenarios hipotéticos) es una técnica utilizada para evaluar los efectos de un cambio en las variables de entrada de un modelo o sistema. Consiste en alterar manualmente el valor de una o más variables independientes para observar el impacto de ese cambio en una o más variables dependientes de interés.

ña la segregación residencial, evacuación de emergencias, delincuencia y más (Helbing y Balmelli, 2012). En lo personal, hemos usado los modelos basados en agentes y la simulación de dichos modelos para el estudio de fenómenos psicocognitivos en toma de decisiones en organizaciones y como modelos de enseñanza de fenómenos políticos.

En resumen, el MSBA es una metodología prometedora en las ciencias sociales computacionales que permite nuevas perspectivas sobre la complejidad de los sistemas sociales modernos. Su creciente uso por académicos y formuladores de políticas públicas (Weisburd *et al.*, 2022) resalta su valor para comprender y, potencialmente, mejorar los resultados sociales.

10.3. VENTAJAS DE PYTHON Y LA BIBLIOTECA MESA EN EL MSBA

Python y la biblioteca MESA emergen como recursos poderosos para la simulación de fenómenos sociales complejos, ofreciendo una combinación de accesibilidad, flexibilidad y eficiencia. Python es un lenguaje de programación de alto nivel que destaca por su legibilidad y simplicidad. Su sintaxis clara y estructurada promueve un enfoque de programación más intuitivo, lo cual es esencial para científicos sociales que quizá no tengan una formación extensa en programación. Además, Python es un lenguaje interpretado, lo que facilita la iteración rápida de modelos y la experimentación.

Otra ventaja significativa de Python es su extensa comunidad de usuarios y la disponibilidad de una amplia gama de bibliotecas. Esto permite a los investigadores ingresar a herramientas avanzadas para el análisis de datos, visualización y modelado matemático, lo cual es crucial para el análisis de resultados en modelado basado en agentes.

MESA es una biblioteca de Python específicamente diseñada para el modelado basado en agentes. Ofrece una plataforma robusta y flexible para la construcción de modelos complejos. Sus principales ventajas incluyen:

1. Modelado intuitivo: MESA permite definir agentes con comportamientos individuales y reglas de interacción, lo que facilita la representación de sistemas sociales complejos.
2. Escalabilidad: Con MESA, los modelos pueden ser tan simples o complejos como se requiera. La biblioteca soporta desde modelos pequeños y conceptuales hasta simulaciones a gran escala con miles de agentes.
3. Componentes reutilizables: MESA proporciona componentes modulares que pueden ser reutilizados en diferentes modelos, lo que mejora la eficiencia y reduce el tiempo de desarrollo.
4. Visualización y análisis: La biblioteca incluye herramientas para la visualización de datos y el análisis de resultados, lo cual es crucial para la interpretación y presentación de los hallazgos de la simulación.

En las ciencias sociales, el modelado basado en agentes se utiliza para estudiar fenómenos como la difusión de información, las dinámicas de opinión, los patrones de movilidad humana y más. Python y MESA permiten a los investigadores modelar estos sistemas con un alto grado de detalle que refleje la complejidad inherente de las interacciones sociales. Además, la capacidad de incorporar fácilmente datos empíricos hace que Python y MESA sean particularmente adecuados para modelos que requieren una base empírica sólida.

La combinación de Python y la biblioteca MESA representa una poderosa herramienta para el modelado y simulación basados en agentes en las ciencias sociales. Su accesibilidad, junto con su capacidad para manejar complejidades y proporcionar análisis detallados, los convierte en una elección excelente para investigadores que buscan explorar la dinámica de los sistemas sociales complejos. A medida que estas herramientas continúan evolucionando, es probable que su impacto en las ciencias sociales se amplíe aún más.

10.4. APLICACIONES Y CASOS DE ESTUDIO

Este apartado explora diversas aplicaciones prácticas del MSBA en las ciencias sociales que se mostraron en el libro, exponiendo su versatilidad y profundidad en distintos contextos.

10.4.1. Dinámicas de opinión

En el estudio de las dinámicas políticas y de opinión, los modelos basados en agentes permiten simular cómo las opiniones individuales pueden influir en la formación de consensos o la polarización. En el apartado de introducción a MESA, se ejemplificó la construcción de un modelo de dinámica de opinión usando la dinámica de Ising. Este es un modelo clásico de la sociofísica¹⁵ que puede servir de inspiración a los lectores para hacer las modificaciones necesarias al programa para que los agentes obedezcan otro tipo de dinámicas, por ejemplo, las dinámicas sugeridas por Deffuant-Weisbuch.

10.4.2. Economía y mercados

En economía, el MSBA se utiliza para modelar mercados y sistemas económicos. Un caso de estudio relevante es el análisis de la mecánica estadística del dinero, el bienestar y el ingreso (Dragulescu y Yakovenko, 2002), donde se ofrece una visión más granular que los modelos económicos tradicionales. En el ejemplo que se presentó, se analizó un modelo de econofísica¹⁶ en donde los supuestos que rigen este modelo son:

¹⁵ La sociofísica es un campo interdisciplinario que aplica teorías y métodos de la física, especialmente de la física estadística, para modelar y entender fenómenos sociales complejos. Esta disciplina surge de la idea de que muchos sistemas sociales, al igual que los sistemas físicos, pueden entenderse mediante leyes y patrones cuantitativos. La sociofísica se enfoca en la identificación de dichos patrones y en la formulación de modelos que puedan predecir el comportamiento colectivo de los sistemas sociales. La sociofísica es una herramienta poderosa para entender y predecir el comportamiento colectivo en sistemas sociales, y su relevancia se ha incrementado en la era de los grandes datos, donde la cantidad de información disponible sobre sistemas sociales es enorme y altamente detallada.

¹⁶ La econofísica es una disciplina interdisciplinaria que se encuentra en la intersección de la economía y la física. Este campo aplica conceptos, métodos y herramientas de la física, en particular de la física estadística y teórica, para resolver problemas y analizar fenómenos dentro de la economía. La premisa fundamental de la econofísica es que muchos de los sistemas y procesos económicos, al igual que los sistemas físicos, pueden ser modelados y comprendidos mediante

1. Hay una cierta cantidad de agentes.
2. Todos los agentes comienzan con una unidad de dinero.
3. En cada paso del modelo, un agente entrega una unidad de dinero (si la tiene) a otro agente.

La premisa de este modelo fue representar a los agentes que intercambian dinero. Este intercambio de dinero afecta la riqueza. A continuación, se agregó espacio para permitir que los agentes se muevan en función del cambio en la riqueza a medida que avanza el tiempo. Este simple modelo exhibe comportamientos muy interesantes e inesperados. Dicho modelo se revisó en la parte dedicada a los modelos socioeconómicos. Se muestra cómo el modelo se rige bajo los supuestos físicos de la termodinámica de Boltzmann.

10.4.3. Difusión de la cultura

El modelo de Axelrod de difusión de la cultura, que se presentó en el capítulo 9, es un marco teórico fundamental en la sociofísica y en el estudio de los sistemas sociales complejos. Este modelo se utiliza para entender cómo la interacción entre individuos puede llevar a la emergencia de culturas distintas y homogéneas en diferentes regiones, así como para estudiar los fenómenos de diversidad cultural y su mantenimiento en las sociedades. La relevancia del modelo de Axelrod radica en varios aspectos:

- Estudio de la dinámica cultural: Proporciona un marco para entender cómo las culturas evolucionan y se difunden a través de la interacción social. Permite examinar cómo la conformidad y la diversidad cultural pueden coexistir.
- Aplicabilidad en diversos campos: Aunque se originó en la sociofísica, el modelo ha encontrado aplicaciones en la sociología, la antro-

patrones y leyes cuantitativas. La econofísica, a menudo, adopta un enfoque más empírico y menos teórico en comparación con la economía tradicional, enfocándose más en el análisis de datos y en la construcción de modelos que se ajusten a los patrones observados. Este enfoque ha llevado a algunas críticas por parte de economistas tradicionales, quienes a veces cuestionan la aplicabilidad y la precisión de los métodos físicos en contextos económicos. Sin embargo, la econofísica ha proporcionado valiosas herramientas y perspectivas para entender la complejidad de los sistemas económicos.

pología, la ciencia política y otros campos que estudian la dinámica social y cultural.

- Simulaciones basadas en agentes: El modelo es un ejemplo prominente de cómo las simulaciones basadas en agentes pueden ser utilizadas para estudiar sistemas complejos. Permite a los investigadores explorar el impacto de diferentes parámetros y condiciones iniciales en la dinámica cultural.
- Interdisciplinariedad: El modelo de Axelrod es un ejemplo de cómo los conceptos de la física y la computación pueden aplicarse para resolver problemas en las ciencias sociales, fomentando al mismo tiempo un enfoque interdisciplinario.
- Implicaciones en políticas públicas: Ayuda a entender los desafíos asociados con la integración cultural y la pluralidad, lo cual es relevante para el diseño de políticas públicas en áreas como la educación, la inmigración y la integración social.

10.5 DESAFÍOS Y LIMITACIONES DEL MSBA

El modelado y simulación basada en agentes ofrece un marco excepcionalmente flexible y potente para el análisis de sistemas sociales complejos. Su capacidad para modelar interacciones individuales y observar los fenómenos emergentes resultantes es particularmente valiosa en las ciencias sociales. Desde la política hasta la planificación urbana, el MSBA permite a los investigadores y responsables de políticas experimentar con escenarios hipotéticos, entender mejor las dinámicas sociales y tomar decisiones informadas basadas en simulaciones detalladas y realistas. A medida que la tecnología y la capacidad de cómputo siguen avanzando, es probable que el impacto y la aplicación del MSBA en las ciencias sociales continúen expandiéndose, ofreciendo perspectivas aún más profundas en los complejos tejidos de nuestras sociedades.

No obstante lo anterior, el MSBA presenta desafíos y limitaciones que deben tenerse en cuenta al momento de elegir esta metodología de investigación para el acercamiento a algún fenómenos social, político o

económico. Los principales desafíos y limitaciones que se presentan son los siguientes:

1. Complejidad computacional: Los MSBA pueden ser extremadamente complejos y requerir una gran cantidad de recursos computacionales, en especial cuando se modelan grandes poblaciones de agentes o se integran múltiples capas de interacción y decisión. Esta complejidad puede limitar la escalabilidad de los modelos y la capacidad para realizar simulaciones extensas.
2. Validación del modelo: La validación de los MSBA es un desafío crucial. Debe demostrarse que el modelo es una representación adecuada de la realidad. Esto puede ser difícil, sobre todo cuando se modelan sistemas sociales o económicos, donde la disponibilidad de datos empíricos para la validación puede ser limitada.
3. Sensibilidad a las condiciones iniciales y parámetros: Los MSBA pueden ser muy sensibles a sus condiciones iniciales y a los parámetros del modelo. Pequeñas variaciones en estos pueden conducir a resultados significativamente diferentes, lo que plantea preguntas sobre la robustez y la generalización de las conclusiones derivadas de un único conjunto de simulaciones.
4. Simplificación y abstracción: Los agentes en estos modelos a menudo se basan en reglas de decisión simplificadas, lo que puede no capturar por completo la complejidad de las decisiones y comportamientos humanos en el mundo real. Esta simplificación es necesaria para volver manejable el modelo, pero también puede limitar su aplicabilidad.
5. Interpretación y extrapolación de resultados: Existe el riesgo de sobreinterpretar o extrapolar incorrectamente los resultados de los MSBA. Los hallazgos de las simulaciones deben ser interpretados con precaución, teniendo en cuenta las limitaciones y suposiciones del modelo.
6. Diversidad y heterogeneidad de agentes: La representación adecuada de la diversidad y heterogeneidad en las poblaciones de agentes es un desafío. Es crucial asegurarse de que los modelos capturen de forma adecuada la variedad de comportamientos y características de los agentes en el sistema real.

7. Integración de datos empíricos: La integración efectiva de datos empíricos en los MSBA para hacerlos más realistas es un desafío, lo cual incluye la incorporación de datos históricos, tendencias actuales y proyecciones futuras.
8. Limitaciones en la teoría y metodología: Aunque los MSBA son una herramienta poderosa, todavía están en desarrollo y existen limitaciones en la teoría y las metodologías subyacentes. Esto incluye la necesidad de mejores métodos para la calibración de modelos, la estimación de parámetros y el análisis de sensibilidad.

En conclusión, mientras que los MSBA ofrecen una poderosa herramienta para la modelación y el análisis de sistemas complejos, su uso requiere un cuidadoso diseño, ejecución y análisis para asegurar que sus resultados sean válidos, confiables y relevantes para las preguntas de investigación planteadas.

10.6 DESARROLLOS RECIENTES Y TENDENCIAS FUTURAS EN EL MSBA

Recientemente, la integración de técnicas avanzadas como el aprendizaje automático y el análisis de *big data* han impulsado aún más las capacidades de los MSBA, y abren nuevas posibilidades y desafíos. Los desarrollos recientes son los siguientes:

1. Integración con aprendizaje automático (*machine learning*, ML): Una de las innovaciones más significativas en MSBA es la integración con técnicas de ML. Los modelos de agentes pueden ser mejorados con algoritmos de ML para simular decisiones más realistas y adaptativas de los agentes. Por ejemplo, los agentes pueden usar aprendizaje reforzado para optimizar sus estrategias basándose en experiencias pasadas.
2. Uso de *big data*: La disponibilidad de grandes conjuntos de datos ha permitido a los investigadores calibrar y validar modelos de agentes con una precisión sin precedentes. Los datos de redes sociales, por ejemplo, se utilizan para modelar la difusión de información y

opiniones, lo cual permite una simulación más precisa de las dinámicas sociales.

3. Modelos híbridos: Se están desarrollando modelos híbridos que combinan MSBA con otras técnicas de modelado, como los modelos de ecuaciones diferenciales o sistemas dinámicos. Esto permite abordar la complejidad de los sistemas desde múltiples perspectivas, aprovechando las fortalezas de cada enfoque.

Las tendencias futuras y el potencial impacto de estos desarrollos de IA y el MSBA se explican a continuación:

1. Modelos más realistas y personalizados: La integración continua de *big data* y ML en MSBA conducirá a la creación de modelos más realistas que reflejen con mayor precisión la diversidad y complejidad del comportamiento humano. Esto podría mejorar significativamente la precisión de las predicciones en campos como la economía, la sociología y la salud pública.
2. Automatización en la calibración de modelos: El futuro de los MSBA podría ver una mayor automatización en la calibración de modelos, donde los algoritmos de ML ajustan de manera autónoma los parámetros del modelo para optimizar su ajuste con los datos reales.
3. Interdisciplinariedad y colaboración: Dada la naturaleza interdisciplinaria de los MSBA, es probable que veamos un aumento en la colaboración entre científicos de datos, sociólogos, economistas y otros especialistas. Esto fomentará una comprensión más holística de los sistemas sociales.
4. Impacto en políticas públicas: Los MSBA podrían jugar un papel crucial en la formulación de políticas públicas, proporcionando herramientas para simular y predecir los efectos de diferentes estrategias de intervención en temas como el cambio climático, la salud pública y la planificación urbana.
5. Desafíos éticos y de privacidad: La utilización de *big data* en MSBA plantea importantes cuestiones éticas y de privacidad. Será crucial desarrollar marcos normativos y éticos para guiar la recopilación y uso de datos personales y asegurar que los modelos no perpetúen sesgos existentes.

En conclusión, los desarrollos recientes en MSBA, especialmente la integración de ML y *big data*, están abriendo nuevas fronteras en el estudio de sistemas complejos. Estas innovaciones prometen modelos más precisos y útiles, pero también traen consigo desafíos significativos. En el futuro, se espera que los MSBA continúen evolucionando, ampliando su impacto en la ciencia, la industria y la formulación de políticas, a la vez que enfrentan cuestiones éticas y técnicas emergentes. La clave será equilibrar la innovación tecnológica con la responsabilidad social y ética, asegurando que los beneficios de estos modelos sean accesibles y beneficiosos para la sociedad en su conjunto.

10.7. IMPLICACIONES METODOLÓGICAS Y TEÓRICAS PARA LAS CIENCIAS SOCIALES

Como hemos visto, el modelado y simulación basada en agentes (MSBA) es una poderosa herramienta en las ciencias sociales computacionales que ha desafiado y complementado los enfoques tradicionales en esta disciplina. El MSBA se ha convertido en un enfoque fundamental para comprender la complejidad social y la emergencia de patrones a partir de interacciones simples. En este apartado, exploraremos las implicaciones metodológicas y teóricas del MSBA en las ciencias sociales, destacando cómo este enfoque ha ampliado nuestro entendimiento de los fenómenos sociales.

En primer lugar, es esencial comprender el MSBA como un enfoque metodológico que se basa en la simulación computacional de agentes individuales y sus interacciones. Cada agente es modelado con atributos, comportamientos y reglas específicas que rigen su comportamiento. Estos agentes interactúan entre sí y con su entorno, y generan resultados macroscópicos a partir de microinteracciones. Esta metodología difiere significativamente de los enfoques cuantitativos tradicionales en las ciencias sociales, que a menudo se basan en métodos estadísticos y análisis de datos agregados.

Una de las principales implicaciones metodológicas del MSBA es su capacidad para modelar la heterogeneidad y la variabilidad en las poblaciones. En los enfoques tradicionales, es común asumir homogeneidad

en las poblaciones, lo que puede llevar a simplificaciones excesivas. En cambio, el MSBA permite modelar agentes con características y comportamientos diversos, lo que refleja de manera más precisa la realidad social. Esto es especialmente relevante en el estudio de fenómenos sociales como la difusión de información e infecciones, donde las diferencias individuales pueden desempeñar un papel crucial en la propagación.

Desde una perspectiva teórica, el MSBA también ha desafiado las teorías tradicionales al permitir la exploración de la emergencia de patrones y propiedades no lineales en sistemas sociales. En lugar de comenzar con suposiciones teóricas rígidas, el MSBA permite que los patrones emerjan de las interacciones de agentes individuales. Esto ha llevado al desarrollo de teorías basadas en agentes que pueden explicar fenómenos complejos y dinámicos, como la polarización de opiniones o la propagación de enfermedades.

Para ilustrar las implicaciones teóricas del MSBA, podemos considerar el modelo de difusión de información en las redes sociales. Tradicionalmente, se podría usar un enfoque de difusión de información basado en modelos epidemiológicos, como el modelo SIR (susceptible-infectado-recuperado), que asume la homogeneidad en la población. Sin embargo, en el MSBA, se pueden modelar agentes con diferentes características, como la probabilidad de adoptar una nueva información o la influencia de sus conexiones en línea. Esto permite explorar cómo los patrones de difusión emergen de las interacciones entre usuarios individuales, lo que es más realista y enriquecedor desde una perspectiva teórica.

En resumen, el análisis basado en agentes ha tenido un impacto significativo en las ciencias sociales al desafiar y complementar los enfoques tradicionales. Desde una perspectiva metodológica, ha permitido modelar la heterogeneidad y la variabilidad en las poblaciones, lo que es fundamental para comprender fenómenos sociales complejos. Desde una perspectiva teórica, ha promovido la exploración de la emergencia de patrones a partir de interacciones simples, lo cual enriquece nuestras teorías sobre la dinámica social. En última instancia, el MSBA ofrece nuevas vías para entender la complejidad social y es una herramienta invaluable en la caja de herramientas de las ciencias sociales computacionales.

10.8. GUÍA PRÁCTICA Y RECOMENDACIONES PARA FUTUROS INVESTIGADORES

El modelado basado en agentes (MSBA) ha demostrado ser una herramienta valiosa en las ciencias sociales, pues permite a los investigadores abordar problemas complejos y comprender fenómenos sociales de manera más realista. Sin embargo, ingresar al mundo del MSBA puede ser desafiante. En este apartado, ofreceremos una guía práctica y recomendaciones para futuros investigadores interesados en explorar o profundizar en este enfoque.

10.8.1. Formación y fundamentos del MSBA

Antes de aventurarse en el MSBA, es esencial adquirir una base sólida en sus fundamentos. Recomendamos comenzar con la comprensión de conceptos clave, como *agentes*, *entornos*, *reglas* y *emergencia*. Libros como *Agent based modelling*, de Nigel Gilbert (2007), *Agent-based and individual-based modeling: A practical Introduction*, de Steven F. Railsback y Volker Grimm (2019), *Introduction to agent-based modeling*, de Uri Wilensky y William Rand (2015), e *Introducción al modelado basado en agentes: una aproximación desde Netlogo*, de Antonio Aguilera y Marta Posada (2017) son recursos valiosos para adquirir esta base de conocimientos.

Además, es crucial dominar las herramientas de desarrollo de modelos, como alguno de los lenguajes especializados en modelado de agentes, como Netlogo, Swarm, RePast y MESON. Una alternativa es la que se presenta en este libro, donde se muestra cómo usar el entorno de programación Python y la biblioteca orientada a agentes MESA. La familiaridad con estas herramientas es esencial para construir y ejecutar modelos efectivos.

10.8.2. Selección del problema de investigación

Elegir el problema de investigación adecuado es un paso crucial para su investigación. Recomendamos seleccionar un problema que realmente le interese y que tenga relevancia en su campo. El MSBA es versátil y se aplica en diversas áreas, desde la economía hasta la epidemiología. Identifique una pregunta de investigación clara y específica que pueda abordar mediante el modelado basado en agentes.

No todos los problemas sociales pueden modelarse usando la metodología del MSBA. Hay que ser muy precisos en que la visión debe ser *bottom-up* y tener claras las reglas de interacción entre agentes. Es recomendable revisar la literatura referente a los modelos de cognición desarrollados en la psicología social y a los modelos matemáticos desarrollados en la sociología matemática para inspirar los modelos basados en agentes.

10.8.3. Colaboración interdisciplinaria

El MSBA a menudo se beneficia de la colaboración interdisciplinaria. Trabajar con expertos en campos relacionados, como la sociología, la psicología, la epidemiología o la economía puede enriquecer su investigación. Las perspectivas diversas pueden ayudar a formular modelos más realistas y a interpretar los resultados de manera más profunda.

10.8.4. Diseño del modelo

La construcción de un modelo efectivo requiere atención al detalle. Defina claramente los agentes, sus atributos y comportamientos, así como las reglas que gobiernan sus interacciones. Utilice la literatura existente como guía para informar su diseño. Además, asegúrese de que el modelo sea lo más simple posible para responder a su pregunta de investigación, pero lo suficientemente complejo como para capturar la esencia del fenómeno.

10.8.5. Validación y sensibilidad

La validación y la sensibilidad son componentes críticos del proceso de modelado. Comparar los resultados de su modelo con datos empíricos siempre que sea posible es una buena práctica y necesidad para validar el modelo. Realice análisis de sensibilidad para evaluar cómo los cambios en los parámetros afectan los resultados. Esto ayudará a verificar la robustez y la confiabilidad de su modelo.

10.8.6. Documentación y comunicación

Mantenga una documentación detallada de su modelo, incluyendo las reglas, los parámetros y las fuentes de datos. Es recomendable usar el protocolo ODD. Esto facilitará la reproducción de su investigación y permitirá a otros investigadores comprender y construir sobre su traba-

jo. Comunique sus resultados de manera clara y efectiva, utilizando visualizaciones y gráficos para ilustrar sus hallazgos.

Es importante ser muy cuidadoso en la comunicación matemática de sus reglas de interacción. Muchos investigadores sociales se pueden ver imposibilitados de comprender las matemáticas si estas son muy complejas. Piense en la comunidad científica a la que quiere comunicar su investigación. Esto le permitirá escoger, además, el tipo de revista científica a la cual mandar su trabajo.

10.8.7. Ética en el MSBA

Considere las implicaciones éticas de su investigación. El MSBA puede involucrar la simulación de comportamientos humanos y sociales, lo que plantea cuestiones de privacidad y consentimiento si usted está usando datos recopilados de fuentes como internet o entrevistas para calibrar su modelo. Asegúrese de seguir las pautas éticas y obtener la aprobación necesaria cuando corresponda.

10.8.8. Aprendizaje continuo

El MSBA es un campo en constante evolución. Manténgase al tanto de los avances y las mejores prácticas en el modelado basado en agentes. Participe en conferencias, talleres y comunidades en línea para aprender y colaborar con otros investigadores. Le sugerimos que se una a la Red Iberoamericana de Ciencias Sociales Computacionales (<https://redicisco.org/>) en donde se dan cursos gratuitos sobre MSBA, Netlogo, Python orientado al MSBA y se organizan anualmente simposios donde puede comunicar sus investigaciones. Esta red cuenta con un acuerdo con la Facultad de Economía de la Universidad Autónoma de San Luis Potosí y el Programa de Estudios Políticos e Internacionales de El Colegio de San Luis, A.C., para organizar el seminario de Ciencias Sociales Computacionales y Complejidad que cada mes presenta a un investigador que hace contribuciones a las ciencias sociales computacionales.

10.9. REFLEXIONES FINALES

En este libro hemos proporcionado los elementos básicos para que el lector conozca el MSBA, su uso en la elaboración de modelos de índole socioeconómica y un recorrido profundo para aprender Python y la biblioteca MESA especializada para el desarrollo de MSBA. Se han recomendado un conjunto de pasos estándar que se utilizarán cuando se está realizando la construcción de modelos basados en agentes para la investigación en ciencias sociales. El primero y el más importante en el proceso de modelado es identificar el propósito del modelo y las preguntas que abordar. La importancia de utilizar las teorías existentes para justificar dicho modelo y las reglas de interacción entre agentes es también tratada. Además, se ha hecho hincapié en los supuestos de un modelo y en validar sus resultados. Para explicar estos conceptos, hemos ejemplificado la descripción de tres modelos basado en agentes que aborda la dinámica de opinión en una sociedad artificial, la difusión de la cultura en una sociedad artificial y la dinámica del intercambio de dinero en una economía artificial.

Esperamos que este libro haya sido de interés y utilidad para los lectores y una motivación para explorar las oportunidades que brindan las ciencias sociales computacionales a la investigación y educación en ciencias sociales.

¡Les deseamos a todos mucho éxito en sus futuras investigaciones y proyectos!

Con gratitud y entusiasmo, los autores.

REFERENCIAS

- AGUILERA, A. y Abrica-Jacinto, N. L. (2022). Sociología computacional: conceptos, métodos y retos. *Revista Internacional de Ciencias Sociales Interdisciplinarias*, 10(1), 159-170. <https://doi:10.18848/2474-6029/CGP/v10i01/159-170>
- AGUILERA, A. y Abrica-Jacinto, N. L. (2019). *Modelos de simulación basados en agentes aplicados a las ciencias políticas*. El Colegio de San Luis.
- AGUILERA, A. y López-Paredes, A. (2001). *Modelado multiagente de sistemas socioeconómicos: una introducción al uso de la inteligencia artificial en la investigación social*. El Colegio de San Luis.
- AGUILERA, A. y Posada, M. (2017). *Introducción al modelado basado en agentes: una aproximación desde Netlogo*. El Colegio de San Luis.
- ARAGÓN, L., Jiménez-Tenorio, N., Oliva-Martínez, J. M. y Aragón-Méndez, M. M. (2018). La modelización en la enseñanza de las ciencias: criterios de demarcación y estudio de caso. *Revista Científica*, 32(2), 193-206. DOI: <https://doi.org/10.14483/23448350.12972>
- AXELROD, R. (1997) The dissemination of culture - A model with local convergence and global polarization. *Journal of Conflict Resolution* 41(2), pp. 203-226.
- AXTELL, R., Axelrod, R., Epstein, J. M. y Cohen, M. D. (1996). Aligning simulation models: a case study and results. *Computational & Mathematical Organization Theory*, 1(2), 123—141.
- BALIGH, H. H., Burton, R. M. y Obel, B. (1999). Devising expert systems in organization theory: The organizational consultant. En M. Masuch (ed.), *Organization, management, and expert systems* (pp. 35-57). Walter De Gruyter.
- BIANCHI, C., Cirillo, P., Gallegati, M. y Vagliasindi, P. (2007). Validating and calibrating agent-based models: a case study. *Computational Economics*, 30(3), 245-264.

- BONABEAU, E. (2001). Agent-based modeling: methods and techniques for simulating human systems. *In Proc. National Academy of Sciences*, 99(3), 7280-7287.
- CARLEY, K. M. (1999). On generating hypotheses using computer simulations. *Syst. Engin.*, 2, 69-77. [https://doi.org/10.1002/\(SICI\)1520-6858\(1999\)2:2<69::AID-SYS3>3.0.CO;2-0](https://doi.org/10.1002/(SICI)1520-6858(1999)2:2<69::AID-SYS3>3.0.CO;2-0)
- CASTI, J. (1997). *Would-be worlds: how simulation is changing the world of science*. Wiley
- DAWID, H., Harting, P., van der Hoog, S. y M. Neugart, M. (2019). Macroeconomics with heterogeneous agent models: fostering transparency, reproducibility and replication. *Journal of Evolutionary Economics*, 29, 467-538.
- DRĂGULESCU, A., Yakovenko, V.M. (2000) Statistical mechanics of money. <https://arxiv.org/abs/cond-mat/0001432>
- DEFFUANT, G., Neau, D., Amblard, F. y Weisbuch, G. (2000). Mixing beliefs among interacting agents. *Adv. Complex Sys.*, 03(01n04), 87-98.
- EPSTEIN, J. M. y Axtell, R. (1996). *Growing artificial societies: social science from the bottom up*. The MIT Press.
- EPSTEIN, J. M. (2007). *Generative social science: studies in agent-based computational modeling*. Princeton University Press.
- EPSTEIN, J. M. (1999). Agent-based computational models and generative social science. *Complexity*, 4(5), 41-60.
- FAGIOLO, G., Moneta, A. y Windrum, P. (2007). A critical guide to empirical validation of agent-based models in economics: methodologies, procedures, and open problems. *Computational Economics*, 30(3), 195-226.
- FOLADORI, G. (1971). El contacto cultural. *Revista Mexicana de Sociología*, 33(3) (julio-septiembre), 581-592.
- GALÁN, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes, A. y Edmonds, B. (2009). Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1), 1. URL: <http://jasss.soc.surrey.ac.uk/12/1/1.html>
- GILBERT, N. (2007). *Agent-based models*. Sage Publications Ltd.

- GONZÁLEZ-DUQUE, R. (2011). Python para todos. Licencia Creative Commons Reconocimiento 2.5. España. <http://mundogeek.net/tutorial-python/>
- GRIMM, V., Berger, U., De Angelis, D. L., Polhill, J. G., Giske, J. y Railsback, S. F. (2010). The ODD protocol: a review and first update. *Ecological Modelling*, 221(23), 2760-2768.
- HEDSTRÖM, P. y Ylikoski, P. (2010). Causal mechanisms in the social sciences. *Annual Review of Sociology*, 36, 49-67.
- HELBING, D. y Baliotti, S. (2012). How to do agent-based simulations in the future: from modeling social mechanisms to emergent phenomena and interactive systems design, en Dirk Helbing, *Social self-organization*. Springer, pp. 25-70. Disponible en SSRN: <https://ssrn.com/abstract=2339770>
- KLEIN, D., Marx, J. y Fischbach, K. (2018). Agent-based modeling in social science, history, and philosophy. An introduction. *Historical Social Research / Historische Sozialforschung*, 43(1)(163), número especial: Agent-based modeling in social science, history, and philosophy, 7-27.
- KLEINDORFER, G. B., O'Neill, L. y Ganeshan, R. (1998). Validation in simulation: various positions in the philosophy of science. *Management Science*, 44(8), 1087-1099.
- LEÓN-MEDINA, F. J. (2017). Analytical sociology and agent-based modeling: is generative sufficiency sufficient?, *Sociological Theory*, 35(3), 157-178. DOI: 10.1177/0735275117725766.
- MACAL, C. M. y North. M. J. (2006). Tutorial on agent-based modeling and simulation part 2: how to model with agents. *Proceedings of the 2006 Winter Simulation Conference*, 73-83. DOI: <http://dx.doi.org/10.1109/wsc.2006.323040>
- MANTEGNA, R. N. y Stanley, H. E. (1999). *Introduction to econophysics: correlations and complexity in finance*. Cambridge University Press.
- MANZO, G. (2014). *Analytical sociology: actions and networks*. John Wiley & Sons.
- MANZO, G. y Matthews, T. (2014). The potential and limitations of agent-based simulation: an introduction. *Revue française de sociologie* (English edition), 55(4), 433-462.

- MILLER, J. H., y Page, S. E. (2007). *Complex adaptive systems: An introduction to computational models of social life*. Princeton University Press.
- PARKER, D., Manson, S. M., Janssen, M. A., Hoffmann, M. J. y Deadman P. (2003). Multi-agent systems for the simulation of land-use and land-cover change. *Annals of the Association of American Geographers*, 94, 314-337.
- PEREDA, M. y Zamarreño, J. M. (2015). Modelado basado en agentes: un enfoque desde la ingeniería de sistemas. *Revista Iberoamericana de Automática e Informática industrial*, 12, 304-312.
- RAILSBACK, S. F. y Grimm, V. (2019). *Agent-based and individual-based modeling. A practical introduction*. Princeton University Press.
- RESNICK, M. (1997). *Turtles, termites and traffic jams: explorations in massively parallel microworlds*. MIT Press.
- SALGADO, M. y Gilbert, N. (2013). Agent based modelling. En T. Teo (ed.), *Handbook of quantitative methods for educational research* (pp. 247-265). Sense Publishers.
- SQUAZZONI, F. (2014). The agent-based modeling approach through some foundational monographs. *Revue française de sociologie* (English edition), 55(4), 827-840.
- SQUAZZONI, F. (2012). *Agent-based computational sociology*. Wiley.
- WEISBURD, D., Wolfowicz, M., Hasisi, B., Paolucci, M. y Andrighetto, G. (2022). What is the best approach for preventing recruitment to terrorism? Findings from ABM experiments in social and situational prevention. *Criminology & Public Policy*, 21(2), 461-485.
- WILENSKY, U. (1999). Netlogo. <http://ccl.northwestern.edu/netlogo>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, Il.
- WILENSKY, U. y Rand, W. (2015) *An introduction to agent based modeling: modeling natural, social, and engineered complex systems with NetLogo*. MIT Press.
- ZADBOOD, A., y Hoffenson, S. (2022). Social network word-of-mouth integrated into agent-based design for market systems modeling. *ASME. J. Mech. Des.*, 144(7), 071701. <https://doi.org/10.1115/1.4053684>

Python para el modelado y simulación basado en agentes en las ciencias sociales, de Antonio Aguilera Ontiveros y Gloria Alferes Varela, se terminó de editar el 13 de febrero de 2025. La composición tipográfica se realizó en Editorial Página Seis, S.A. de C.V., Lorenzo Barcelata 5105, col. Paraíso Los Pinos, C.P. 45239, Zapopan, Jalisco, tel. 33 3657 3786. La edición estuvo al cuidado de la Unidad de Publicaciones de El Colegio de San Luis. El tiraje fue de 1 ejemplar digital.