The Code Behind the Story

Data Skills with Python for Journalists

Your Toolkit to Analyze and Visualize Data Like a Pro

Amr Eleraqi

Data Skills with Python for Journalists

Your Toolkit to Analyze and Visualize Data Like a Pro

By: Amr Eleraqi

➔ This book is part of the **Data + Stories** series, dedicated to helping professionals transform data into compelling narratives.

Introduction

Welcome to a New Era of Journalism

In an age where information flows faster than ever, journalism is undergoing a profound transformation. The stories we tell are no longer just about who, what, when, and where—they're about understanding the why and the how behind the headlines. And increasingly, that understanding comes from data.

Data is everywhere: in government reports, corporate filings, social media feeds, environmental databases, and more. It's not just for tech experts or data scientists—it's for all of us. Whether you're covering politics, health, education, or culture, data can deepen your reporting, uncover hidden truths, and bring clarity to complex issues.

But here's the challenge: data isn't always easy to work with. Spreadsheets can only take you so far, and manually sifting through thousands of records is time-consuming and often impractical. That's where coding comes in.

This book is your guide to embracing this new era of journalism. It's about equipping you with the skills to harness the power of data and coding, not as a replacement for traditional reporting, but as a way to enhance it. Together, we'll explore how Python—a versatile and beginner-friendly programming language—can become your ally in uncovering stories that matter.

Learning to Code: A New Superpower for Journalists

Learning to code is like acquiring a superpower. When I wrote my first line of code—print("Hello, World!")—it wasn't just a technical exercise; it was the beginning of a journey that would redefine my approach to journalism. Coding isn't just a tool; it's a language that unlocks new ways to communicate, investigate, and tell stories.

While traditional tools like spreadsheets have their limits, Python empowers you to tackle massive datasets and ask questions that were once impossible. Imagine trying to analyze a million records in Excel—it would buckle under the weight. Python, however, thrives on such challenges. It's not just about efficiency; it's about possibility. Coding removes the constraints of traditional tools and opens up a world of opportunities for storytelling and investigation.

In a profession built on curiosity and storytelling, coding has become my gateway to uncovering stories hidden within rows and columns of data. Just as we interview human sources to uncover insights, coding allows us to "interview" data—exploring its patterns, anomalies, and contradictions as if we were having a conversation with it.

Coding isn't just for "data journalists." It's for any journalist who wants to work smarter, dig deeper, and tell more compelling stories. It's about using technology to enhance your reporting, not replace it. With coding, you can:

- ➔ Automate repetitive tasks, like collecting or cleaning data, so you can focus on the story.
- ➔ Analyze large datasets to uncover trends, patterns, and anomalies that might otherwise go unnoticed.
- ➔ Visualize data in ways that make complex information accessible and engaging for your audience.

Think of coding as a new tool in your toolkit—one that complements your existing skills and expands what's possible.

Big Data and Journalism

Data is as integral to our world as language itself. As society has become increasingly data-driven, journalists have had to adapt, integrating data into their reporting in the same way they would a trusted source or a key piece of evidence. This shift is essential: in a world where everything from public policies to corporate behaviors is quantified and recorded, the ability to understand, analyze, and interpret data has become a fundamental skill for journalists.

Big Data refers to the enormous volumes of structured and unstructured information that are constantly generated from sources like social media, IoT devices, transaction logs, and digital documents. For journalists, Big Data means access to a wealth of information that can provide insights into topics ranging from electoral patterns to environmental crises, public health trends, and consumer behaviors.

But while data has become more available, the challenge lies in making sense of it. This is where coding and computer-assisted reporting (CAR) come into play: they enable us to parse through mountains of information and extract meaningful insights, even as the volume of data grows exponentially.

Take, for instance, an investigation into campaign finance records for political candidates. With a dataset spanning thousands of contributions from individuals and organizations, a journalist could spend days combing through records manually. But with a few lines of Python code, this tedious process can be streamlined: filtering data by donor type, contribution size, or geographic region, allowing us to pinpoint trends and patterns that might have otherwise gone unnoticed. The ability to wrangle Big Data with code doesn’t just save time—it enables us to ask questions and uncover stories that are only accessible through data.

Interviewing Data

In many ways, working with data mirrors the process of conducting an interview. Just as we listen to human sources, follow up on leads, and probe deeper to gain clarity, interviewing data requires us to approach it with curiosity, skepticism, and rigor. Each dataset holds a story, but it doesn’t reveal itself without a bit of coaxing. We need to ask the right questions, filter through the noise, and identify what is relevant to our narrative.

Coding allows us to “speak” to data in a structured way, asking it specific questions, identifying trends, and even cross-referencing it with other datasets for context. For example, let’s imagine a journalist investigating the allocation of public funds in education. By interviewing the data—querying different subsets, running analyses, and visualizing relationships between variables—patterns emerge, such as disparities in funding distribution or correlations with student outcomes. These insights are invaluable; they inform the story and give it a backbone rooted in data.

Working with data is also like sitting down with a spreadsheet over a cup of coffee, just as you would a human source. You comb through the rows and columns, making notes, highlighting anomalies, and, just as in a human conversation, listening carefully for what stands out. This process is journalism in its digital form: reporting from the desk, equipped with a computer as a primary tool, to uncover stories hidden within data.

Journo-Coder

Computer-assisted reporting (CAR) is not a new concept, but it has become more vital than ever in the digital age. Early pioneers of CAR used rudimentary databases and spreadsheets to analyze numbers and detect anomalies, transforming raw data into investigative leads. Today, coding has brought CAR to a new level of sophistication. Python and other programming languages enable journalists to automate tasks, scrape data from websites, analyze patterns in vast datasets, and even build predictive models that can forecast trends.

Python is central to modern CAR because of its versatility and accessibility. With its extensive libraries—such as Pandas for data manipulation, BeautifulSoup for web scraping, and Matplotlib for visualization—Python enables journalists to handle virtually any type of data they might encounter in an investigation. This adaptability is crucial, especially when working on stories that demand quick turnarounds, such as tracking government spending, analyzing social media sentiment, or examining public health data.

For example, let’s say we’re tracking the spread of misinformation on social media. With Python, we can create a script that pulls data from social media platforms, analyzes trends in language or hashtags, and identifies clusters of activity. This process, which would take weeks manually, can be completed in hours or even minutes with code. By automating repetitive tasks and enhancing our ability to parse large datasets, CAR allows us to spend more time on analysis and storytelling rather than manual data entry.

A Practical Guide: Real-World Examples and Case Studies

This book is a practical guide for journalists who want to harness the power of coding to uncover stories buried in data. It’s about learning to "interview" data much like you would a human source—exploring its nuances, patterns, and sometimes even contradictions. Python, with its simplicity and extensive ecosystem of libraries, has become central to this practice. It allows journalists to automate tedious tasks, sort through massive datasets, and generate visuals that reveal insights at a glance.

Throughout the chapters, you will find numerous real-world examples and case studies that illustrate how Python can be applied to journalism. We’ll look at scenarios like investigating campaign finance, tracking public spending, analyzing social media trends, and uncovering hidden connections in large datasets. Each chapter provides hands-on exercises that build on each other, culminating in larger projects that allow you to dive deep into data-driven investigative journalism.

The future of journalism is data-rich, and coding is no longer optional for journalists—it’s essential. As data continues to grow in volume and complexity, so too does our need to understand and interpret it. Coding allows us to transform data into a new kind of source—one that is vast, powerful, and waiting to be explored.

By the end of this book, you'll have not only the technical skills but also the investigative mindset needed to thrive in this data-rich environment. You’ll see how coding is more than a technical skill—it's a method of approaching stories, one that can unearth new angles, insights, and truths. Just as we trust our instincts when interviewing a human source, coding allows us to trust our instincts with data, digging deeper, asking questions, and shaping narratives that reveal what might otherwise remain hidden.

What You'll Learn in This Book

This book is a practical guide to coding for journalists, focusing on real-world applications and hands-on exercises. Throughout the chapters, you will:

- ➔ Learn the basics of Python and its applications in journalism.
- ➔ Explore techniques for scraping, cleaning, and analyzing data.
- ➔ Discover how to visualize data effectively using Python libraries.
- ➔ Apply coding skills to investigative projects, from tracking public spending to analyzing social media trends.

Each chapter builds on the last, culminating in larger projects that showcase how Python can elevate your reporting. By the end of this book, you will have not only the technical skills but also the investigative mindset needed to thrive in a data-rich environment.

Who Is This Book For?

This book is for every journalist—whether you're a seasoned reporter with years of experience or someone just starting out in the field. It's for the curious minds who want to explore how coding can elevate their storytelling, as well as for those who've never written a single line of code but are eager to learn.

If you've ever felt overwhelmed by spreadsheets, struggled to make sense of large datasets, or wondered how to uncover hidden stories buried in numbers, this book is for you. It's for journalists covering politics, health, education, business, or culture—any beat where data can add depth and clarity to your reporting.

You don't need a background in math, statistics, or computer science to get started. This book is designed with journalists in mind, breaking down complex concepts into simple, actionable steps. All you need is a willingness to learn, a passion for storytelling, and a desire to embrace the tools of the digital age.

Whether you're looking to automate repetitive tasks, analyze large datasets, or create compelling visualizations, this book will equip you with the skills to work smarter, dig deeper, and tell stories that resonate with your audience. If you're ready to take your journalism to the next level, this book is your guide.

Welcome to the Future of Journalism 💻

The future of journalism is data-rich, and coding is no longer optional—it's essential. As data continues to grow in volume and complexity, so too does our need to understand and interpret it. Coding allows us to transform data into a new kind of source—one that is vast, powerful, and waiting to be explored.

This book is your guide to mastering Python as a tool for storytelling. It's your invitation to take on the challenges of data journalism in the Big Data era. Together, we'll explore how coding empowers journalists to write the stories of tomorrow, today.

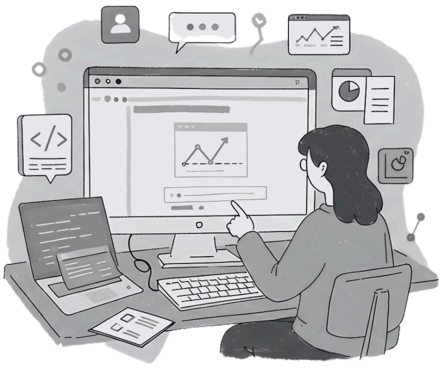
About the Author

Amr Eleraqi is a data scientist, educator, and journalism innovator with over a decade of experience turning data into impactful stories. He teaches at Toronto Metropolitan University in Canada and previously led InfoTimes, the award-winning Arabic data journalism team. Amr has also served as a Knight Fellow with the International Center for Journalists and as regional coordinator for the Facebook Journalism Project in the Middle East. His work automating media monitoring for the Egyptian Cabinet's Information Center advanced data-driven decision-making at the highest levels. Amr is passionate about empowering journalists to harness the power of data in storytelling.

https://www.linkedin.com/in/amreleraqi

To **Rasha and Yusuf**, my heart and my joy—thank you for your endless love and support. This book is for you.

Chapter 1: Intro to Python and Google Colab – A Byte of Python



🔍 1.1 Introduction to Python

In today's fast-paced media landscape, journalists are expected to do more than just write compelling stories—they need to uncover truths hidden in data, whether it's election results, public health trends, or corporate filings. But you don't have to be a "data journalist" to benefit from data.

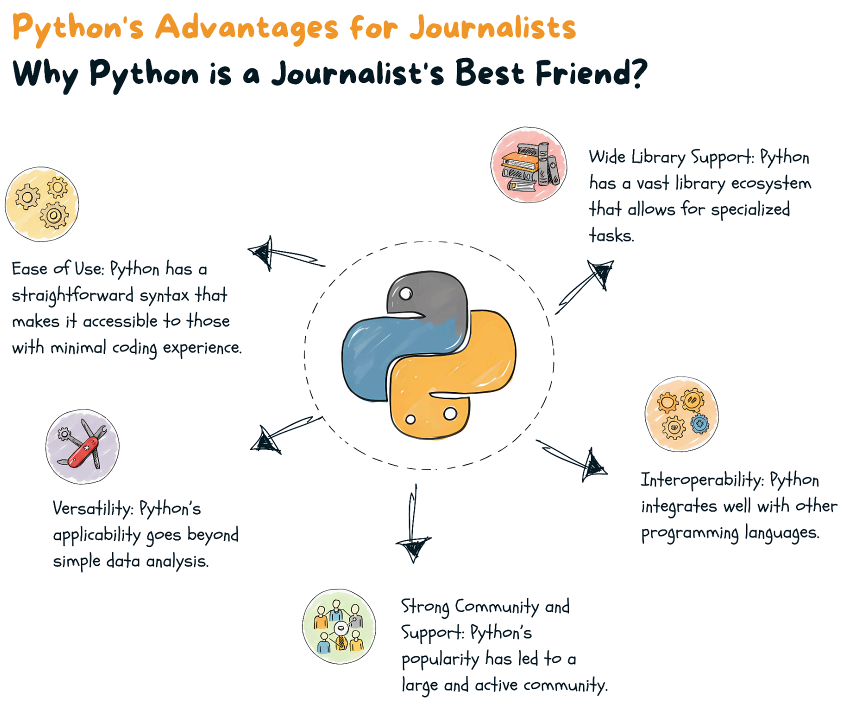
This chapter is for every journalist—whether you're a reporter, editor, or freelancer. It's for those who want to work smarter, dig deeper, and tell stories that stand out. Python, a powerful yet beginner-friendly programming language, can help you do just that. From automating tedious tasks to analyzing large datasets and creating eye-catching visuals, Python is a tool that can enhance your reporting, no matter your beat or experience level.

Think of this as your first step into a new way of working—one where data becomes your ally, not your obstacle. Let's get started.

🐍 1.2 What is Python?

Python is a high-level programming language created by Guido van Rossum in 1991. Python's design emphasizes readability and simplicity, making it accessible for beginners and highly efficient for complex applications. Today, Python is one of the most widely used languages in fields ranging from data science and web development to machine learning and automation.

Why Choose Python?


Python's Advantages for Journalists
Why Python is a Journalist's Best Friend?
Ease of Use: Python has a straightforward syntax that makes it accessible to those with minimal coding experience.
Wide Library Support: Python has a vast library ecosystem that allows for specialized tasks.
Versatility: Python's applicability goes beyond simple data analysis.
Interoperability: Python integrates well with other programming languages.
Strong Community and Support: Python's popularity has led to a large and active community.

1. **Ease of Use**: Python has a straightforward syntax that makes it accessible to those with minimal coding experience.

2. **Wide Library Support**: Python has a vast library ecosystem that allows for specialized tasks. Libraries (pre-built tools) like Pandas, BeautifulSoup, and Matplotlib make data manipulation, web scraping, and visualization straightforward, while more advanced libraries like NLTK and spaCy enable natural language processing and text analysis.

3. **Versatility**: Python's applicability goes beyond simple data analysis. Journalists can use it for automation (automating routine tasks), web scraping (gathering data from the web), data visualization

(creating charts and graphs), and machine learning (predictive analysis and insights from large data sets).

4. **Interoperability**: Python integrates well with other programming languages and can be combined with tools like SQL for database queries, making it easy to work with data from different sources.

5. **Strong Community and Support**: Python's popularity has led to a large and active community, with numerous tutorials, forums, and resources. This makes it easier to find solutions to problems, learn new techniques, and stay updated on the latest trends and tools.

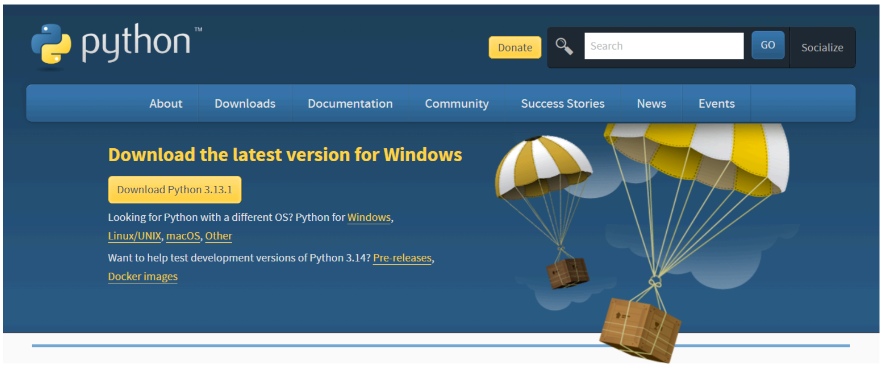
Advantages and Disadvantages of Python in Data Journalism

Advantages	Disadvantages
Broader Applicability: data analysis, automation, web scraping, and machine learning	Fewer specialized packages for specific statistical needs
Easy for beginners to learn	Visualization options can be more limited than R
Works well with other programming languages	Multiple setup options (environments) can be confusing
Large community and lots of tutorials available	

How to Install Python on Your Computer (Locally)

If you plan to use Google Colab as your environment, you don't need to install Python locally. Google Colab provides a pre-installed Python environment in the cloud. However, if you plan to run Jupyter Notebook for local projects, you will need Python installed on your computer.

If you want to run Python locally on your computer (outside of Google Colab), here's a quick installation guide:



Download the latest version for Windows

Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python 3.14? Pre-releases, Docker images

The Python download for windows

1. Download Python:
    - Visit the official Python website: https://www.python.org/downloads/
    - Select the latest version (Python 3.x) and download the installer for your operating system (Windows, macOS, or Linux).
2. Install Python:
    - Open the downloaded installer.
    - On the setup screen, check the box that says "Add Python to PATH" (this makes it easier to run Python from the command line).
    - Click Install Now and follow the installation prompts.
3. Verify the Installation:
    - Open your command prompt (Windows) or terminal (macOS/Linux).

    - Type `python --version or python3 --version` and press Enter. You should see the installed Python version displayed.

4. Install a Package Manager (pip):
    - `pip` is Python's package manager, which helps you install additional libraries (e.g., Pandas, BeautifulSoup).
    - `pip` is included with Python by default, but you can verify it by running `pip --version` in your terminal.

💻 1.3 Setting Up Your Workspace on Google Colab

Google Colab is a free, cloud-based platform developed by Google that lets you write and run Python code in your browser. Colab's simplicity and online accessibility make it an ideal environment for journalists, who may not want to install software on their computers.

1.3.1 Accessing Google Colab

1. Open Google Colab http://colab.new
2. Log in with your Google account.
3. Create a new notebook by selecting File > New notebook.

💡 Tip: You'll need a Google account to access Google Colab. This allows you to save your work directly to Google Drive, making it easy to access and share your projects from any device.

First Task: Print "Hello, World!"

Let's start with the classic first program:

1. In the first cell of your new notebook, type the following code:

```
x = "Hello World!"
print (x)
```

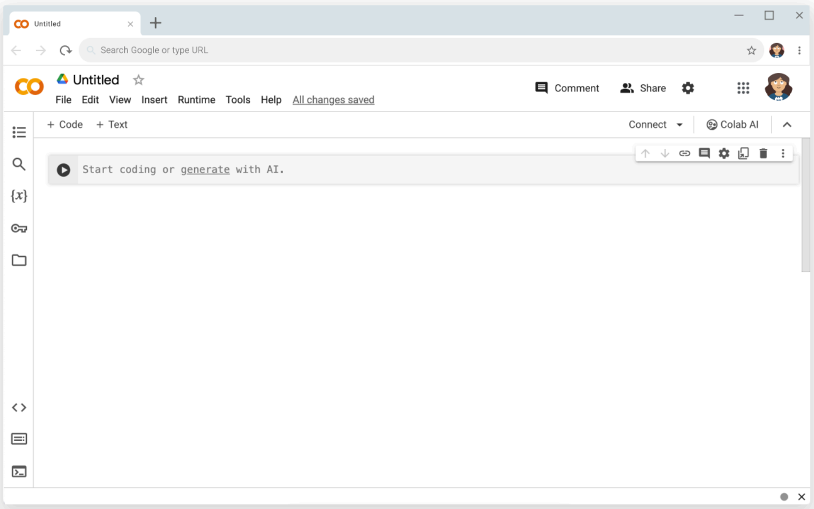
2. Run the cell by pressing **Shift + Enter** or clicking the **Run** button.
3. You should see the output:

```
Hello, World!
```

Congratulations! You've written and executed your first Python program on Google Colab. 🎉

1.3.2 Exploring the Colab Interface

Google Colab's interface is organized into cells where you can write and execute code or add explanatory text. Here's a breakdown of the main components and tabs in the interface:


Untitled
Search Google or type URL
Untitled
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
Connect
Colab AI
Start coding or generate with AI.

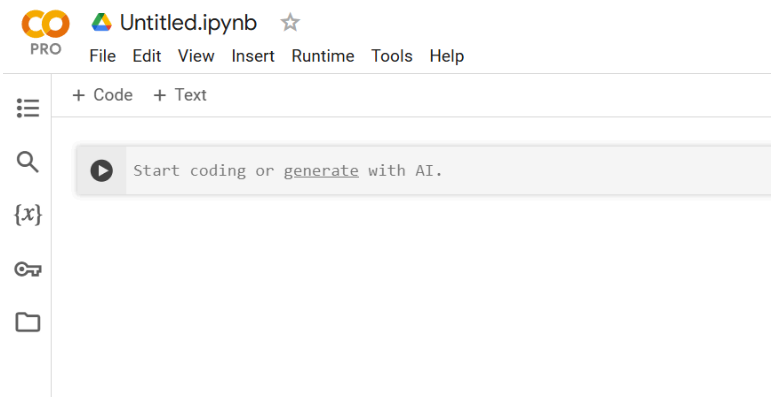
➔ *Main Tabs and Their Features*

Main Tabs	Features
File	Create, open, and save notebooks.
Edit	Undo, redo, cut, copy, and paste cells.
View	Access the table of contents and organize your notebook.
Insert	Add code cells (for Python code) or text cells (for notes).
Runtime	Run, manage, and configure the notebook's runtime (e.g., CPU, GPU).
Tools	Access settings, keyboard shortcuts, and the command palette.
Help	Documentation and resources for assistance.

1. File
    - New notebook: Creates a new Colab notebook.
    - Open notebook: Opens a saved notebook from Google Drive, GitHub, or local files.
    - Save a copy in Drive: Saves a copy of the current notebook in your Google Drive.
    - Download: Allows you to download the notebook in various formats, such as `.ipynb` or `.py` for local use.
2. Edit
    - Undo/Redo: Quickly undo or redo recent actions.
    - Cut, Copy, Paste Cells: Copy or move cells within the notebook.
    - Find and Replace: Searches for text within the notebook, helpful for larger projects.
3. View
    - Table of contents: Displays a collapsible table of contents, making navigation easier for longer notebooks.
    - Mode: Switch between edit and command mode for different cell interactions.

    - Code snippets: Provides a library of pre-written code snippets to help with common tasks, accessible directly in the notebook.
4. Insert
    - Code cell: Adds a new code cell to the notebook for Python code.
    - Text cell: Adds a new text cell where you can write markdown text for explanations, titles, or instructions.
5. Runtime
    - Run all: Executes all cells in the notebook.
    - Change runtime type: Enables you to select CPU, GPU, or TPU for processing. GPU or TPU accelerators are useful for tasks requiring extra computing power, like machine learning.
    - Manage sessions: Shows the memory usage and allows restarting or disconnecting from the runtime to free up resources.
6. Tools
    - Settings: Configure the notebook's appearance, including font size, theme, and other display options.
    - Keyboard shortcuts: Lists and customizes shortcuts to enhance productivity.
    - Command palette: A searchable command center to access various actions quickly.
7. Help
    - Documentation: Access Google Colab and Python library documentation for quick references.
    - Report a bug: Submit bug reports or give feedback on the Colab interface.

➔ *Cells: The Building Blocks of Colab*


PRO
Untitled.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Start coding or generate with AI.

- **Code Cells:** Where you write Python code. Each code cell can be run independently, allowing for incremental testing and debugging.

  Each code cell can be run independently, allowing for incremental testing and debugging.

  How to Use:

  - ☐ Click inside a code cell to edit it.
  - ☐ Press **Shift + Enter** or click the **Run** button to execute the code.

  💡 **Tip:** Outputs from code cells (e.g., text, graphs, or errors) appear directly below the cell, keeping everything organized and easy to follow.

- **Text Cells:** Use markdown to create formatted text for explanations, instructions, or notes, making it easier to document your workflow and results.

  How to Use:

  - ☐ Double-click a text cell to edit it.
  - ☐ Use Markdown syntax for formatting (e.g., # for headings, * for italics).

☐ Press Shift + Enter to render the formatted text.

💡 **Tip:** Document your workflow with text cells to make your notebook clear and shareable. Use headings, bullet points, and links for better readability.

1.3.3 Useful Shortcuts in Colab

Shortcut	Function
`Shift + Enter`	Run the current cell and move to the next
`Ctrl + Enter`	Run the current cell and stay in place
`Alt + Enter`	Run the current cell and insert a new one below
`Ctrl + M + M`	Convert a cell to markdown
`Ctrl + M + Y`	Convert a cell to code
`Ctrl + M + D`	Delete the current cell
`Ctrl + /`	Comment or uncomment selected lines

💡 Tips:

- Use commenting (# This is a comment) to explain code to yourself or collaborators.
- Experiment with Markdown in text cells to add structure and explanations to your notebook.

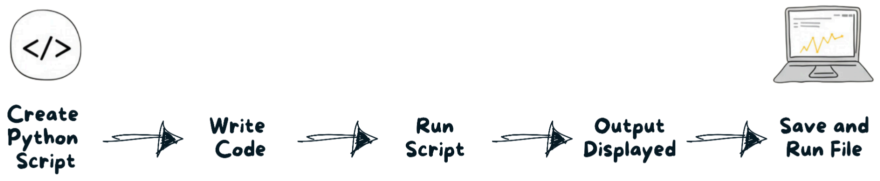
📝 1.4 Python Basics

With Google Colab set up, let's dive into Python fundamentals essential for data journalism. These basics will cover data types, variables, how Python interprets whitespace and indentation, and adding comments to code.

1.4.1 The Beginning: Running Your First Python Script

As a journalist, you often need to verify the phone numbers of your sources. Let's write a simple Python script to validate a single phone number based on a standard format (e.g., starting with a + and having at least 10 digits). This time, we'll make it interactive by asking the user to input the phone number.

Creating `phone_validator.py`



1. Open Google Colab

   Visit http://colab.new to open a new notebook.

2. Name Your Notebook

   - Click on the default notebook name (usually "Untitled" at the top left).
   - Rename it to phone_validator and press Enter.

3. Insert a Text Cell

   - Click on + Text (located at the top left of the toolbar).
   - In the new text cell, type: `# Validating a Human Source Phone Number`

- `Press` Enter to finish editing.

4. Insert a Code Cell

    - Click on + Code (also in the toolbar).
    - In the new code cell, write:

```
# Ask the user to input a phone number

phone_number = input("Enter a phone number to
validate: ")


# Check if the phone number is valid

if phone_number.startswith("+") and
len(phone_number) >= 10:

    print(f"{phone_number} is valid.")

else:

    print(f"{phone_number} is NOT valid.")
```

    - Press Shift + Enter or click the Run button (the play icon on the left of the cell).

Explanation:

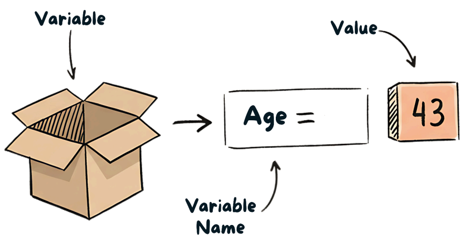
- User Input: The input() function prompts the user to enter a phone number.
- Validation: The script checks if the phone number starts with a + (indicating a country code) and has at least 10 digits.
- Output: The print() function displays whether the phone number is valid or not.

💡 **Tip:**

This script is interactive—you can run it multiple times and test different phone numbers. For example, try entering "+1234567890", "123-456-7890", or "+44 20 7946 0958" to see the results.

1.4.2 Data Types and Variables

In Python, **variables** are used to store data, and data types define the kind of data a variable can hold. Understanding these is essential for organizing and analyzing information in your journalism projects.

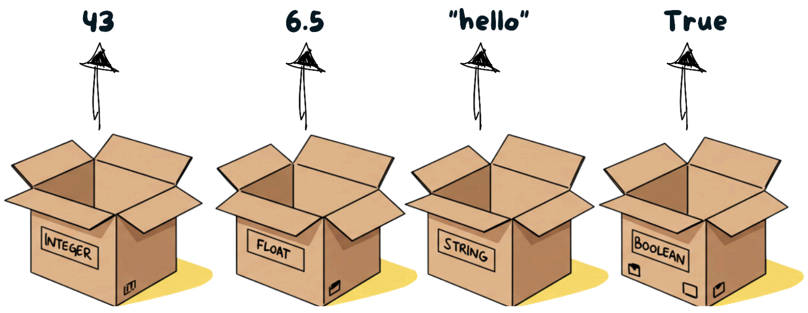


```
age = 43
```

Here's a breakdown of the key data types you'll encounter:

- **Integer**: Integers are whole numbers, positive or negative, without decimals. They're useful for counting or representing discrete values.
- **Float**: Floats are decimal numbers, useful for representing precise measurements or averages.
- **String**: Strings are sequences of characters, used to store text data like names, headlines, or descriptions.

- **Boolean**: Booleans represent True or False values. They're often used for decision-making in your code.



Examples of Python data types: Integer (43), Float (6.5), String ("hello"), Boolean (True).

Examples

# Integer

```
number_of_articles = 10  # Stores the number of articles
written
```

# Float

```
average_rating = 4.7  # Stores the average rating of a news
article
```

# String

```
headline = "Breaking News: Election Results Announced"  #
Stores a news headline
```

# Boolean

```
is_published = True  # Indicates whether an article has
been published
```

Printing in One Line

```
print("Headline:", headline, "Word Count:", word_count,
"Average Reading Time (minutes):", average_reading_time,
"Published:", is_published)
```

Each value is printed on the same line, separated by a space.

Output: Headline: Climate Change Summit Yields New Agreements Word Count: 850 Average Reading Time (minutes): 3.5 Published: True

💡 Tips:

- Strings in Python should always be enclosed in quotes, either double (`"  "`) or single (`'  '`). Both work the same, but it's best practice to use them consistently.
- When creating a variable, use the assignment operator (`=`) to assign a value to it.

Variable Naming Conventions in Python

When naming variables in Python, follow these simple rules to make your code clear and easy to understand:

1. Use Descriptive Names
2. Use Snake Case

For multi-word variable names, use underscores (_) to separate words. This is called snake_case.

3. Start with a Letter or Underscore

Variable names must start with a letter (a-z, A-Z) or an underscore (_). They cannot start with a number.

4. Avoid Reserved Words

Do not use Python keywords like print, len, if, etc., as variable names, as this can cause errors in your code.

Example: Good vs. Bad

👍 headline, view_count, is_published are good names because they reflect the variable's content.

👎 a, x, data1 are poor names because they do not convey the variable's purpose.

👍 total_views, author_name

👎 totalviews, AuthorName

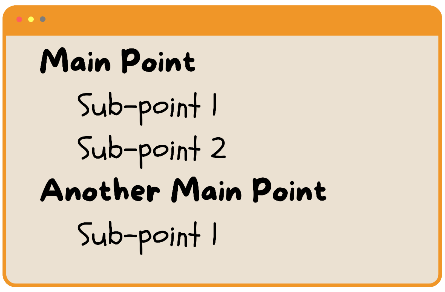
1.4.3 Whitespaces and Indentation

In Python, whitespace (spaces and tabs) and indentation (how far you move code to the right) are super important. They help Python understand how your code is organized, almost like how paragraphs and bullet points help organize a document.

In Python, indentation is used to group lines of code together. Think of it like this:

- Indented code is like a sub-point under a main point.
- Non-indented code is like the main point itself.

Python doesn't use curly braces {} (like some other languages) to group code. Instead, it uses indentation to figure out which lines of code belong together.



Main Point
    Sub-point 1
    Sub-point 2
Another Main Point
    Sub-point 1

Example of Proper Indentation

```
views = 1500
if views > 1000:
    print("This article is
popular!")
else:
    print("This article is less popular.")
```

What's Happening Here?

- The first line (views = 1500) is not indented—it's the main point.
- The if and else lines are like questions:
- If the article has more than 1000 views, Python runs the indented line under if.
- If not, it runs the indented line under else.

💡 Tips:

- Python requires consistent indentation, typically four spaces or a single tab. Mixing spaces and tabs can cause errors, so stick to one method throughout your code.
- Use 4 spaces or 1 tab for indentation.
- Don't mix spaces and tabs—it confuses Python and can cause errors.

1.4.4 Comments

Comments are like sticky notes you add to your code to explain what it does or why you wrote it. Python ignores these notes when running your code—they're just for humans to read.

Comments begin with a # symbol.

➔ **Single-Line Comments**

Start a comment with the # symbol. Use single-line comments to quickly explain what a specific line of code does.

```
# This is a comment explaining the next line of code
```

➔ **Multi-Line Comments**

Python doesn't have a built-in multi-line comment feature. However, you can use triple quotes " " " to create block comments that span multiple lines.

```
" " "
This section of the code initializes
variables and prints a welcome message.
" " "
```

💡 **Tip: Write Helpful Comments**

- Explain why, not what: Don't just repeat what the code does—explain why it's there.
- Keep it simple: Write clear, concise comments.
- Update them: If you change your code, update the comments too.

By adding comments to your code, you'll make it more readable and maintainable. It's like leaving a trail of breadcrumbs for yourself and others to follow.

1.4.5 Lists, Dictionaries, and DataFrames

➔ Lists

Lists are like containers that let you store multiple items in one place. They're super useful for organizing data, especially when the order of items matters—like a list of news headlines or article titles. You can easily add items to a list and access them by their position, or **index**.

Creating a List

You create a list by putting items inside square brackets [] and separating them with commas.

Example:

```
headlines = ["Climate Change Report", "Election Results"]
```

Adding to a List

You can add elements to a list using the `.append()` method (to add one item) or `.extend()` method (to add multiple items).

Example:

```
headlines = ["Climate Change Report", "Election Results"]

# Add a single item
headlines.append("New Policy Updates")

print(headlines)  # Output: ["Climate Change Report",
"Election Results", "New Policy Updates"]

# Add multiple items
headlines.extend(["Economy News", "Sports Highlights"])

print(headlines)  # Output: ["Climate Change Report",
"Election Results", "New Policy Updates", "Economy News",
"Sports Highlights"]
```

Indexing in a List

Each item in a list has a position, called an index. Python starts counting from 0, so the first item is at index 0, the second at index 1, and so on. You can also use negative numbers to count from the end of the list.

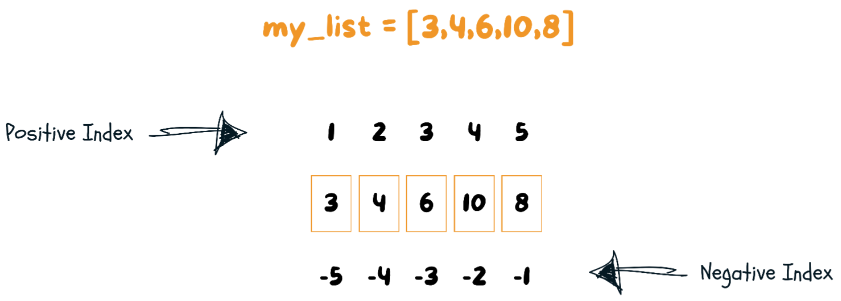


Illustration of list indexing in Python: Positive indices (0 to 4) and negative indices (-5 to -1) for the list my_list = [3, 4, 6, 10, 8]

```
# Access the first item
print(headlines[0])  # Output: "Climate Change Report"

# Access the last item
print(headlines[-1])  # Output: "Sports Highlights"
```

💡 **Tip**: Lists are ideal for ordered collections where the sequence of items is important, such as a list of news headlines sorted by publish time.

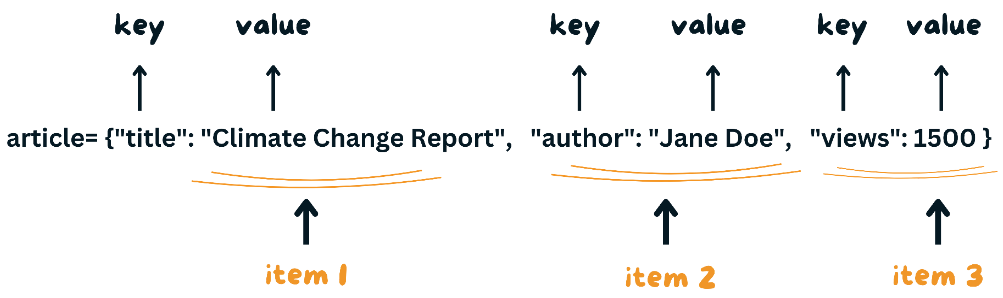
➔ Dictionaries

Dictionaries store data as **key-value pairs**, allowing you to label each item, which is useful for more structured data. You can add new key-value pairs or update existing ones, and you access values by their keys rather than by position.

Creating a Dictionary

You create a dictionary by putting key-value pairs inside curly braces `{}`, with each key separated from its value by a colon `:`

```
article = {
    "title": "Climate Change Report",
    "author": "Jane Doe",
    "views": 1500
}
```


key
value
key
value
key
value
article= {"title": "Climate Change Report", "author": "Jane Doe", "views": 1500 }
item 1
item 2
item 3

article dictionary, where each key (e.g., "title", "author", "views") maps to a corresponding value (e.g., "Climate Change Report", "Jane Doe", 1500).

Adding to a Dictionary

To add or update an entry in a dictionary, assign a value to a new or existing key.

# Add a new key-value pair

```
article["date"] = "2024-01-01"

print(article)  # Output: {'title': 'Climate Change
Report', 'author': 'Jane Doe', 'views': 1500, 'date':
'2024-01-01'}
```

# Update an existing key-value pair

```
article["views"] = 1600

print(article)  # Output: {'title': 'Climate Change
Report', 'author': 'Jane Doe', 'views': 1600, 'date':
'2024-01-01'}
```

Indexing in a Dictionary

In dictionaries, data is accessed by key rather than by position.
You can access values by their keys. Using .get() is safer because it won't cause an error if the key doesn't exist—it just returns None.

# Access the value for a specific key

```
print(article["title"])  # Output: "Climate Change Report"

print(article.get("author"))  # Output: "Jane Doe"
```

💡 **Tip**: Using `.get()` is safer for accessing dictionary values, as it returns `None` if the key doesn't exist rather than causing an error.

➔ DataFrames

DataFrames, provided by the Pandas library, are 2-dimensional, tabular structures with labeled axes (rows and columns). They're ideal for handling larger datasets with multiple types of data. You can add new columns or rows and access data by index or label.

Structure of DataFrame

1. **Rows**: Each row represents a single record or entry.
2. **Columns**: Each column represents a different feature or attribute of the data.

Consider a simple DataFrame of news articles:

Index	Title	Author	Views	Date
0	Climate Change Report	Jane Doe	1500	2024-01-01
1	Election Results	John Smith	1200	2024-01-02
2	New Policy Updates	Alice Brown	800	2024-01-03

Key Parts of DataFrame

- **Index**: The first column on the left (often labeled “Index” by default) represents the row numbers, which uniquely identify each row.
- **Columns**: The top row lists the column names, each representing a specific type of information (e.g., Title, Author, Views, Date).
- **Values**: The cells within the DataFrame contain values for each column and row combination.

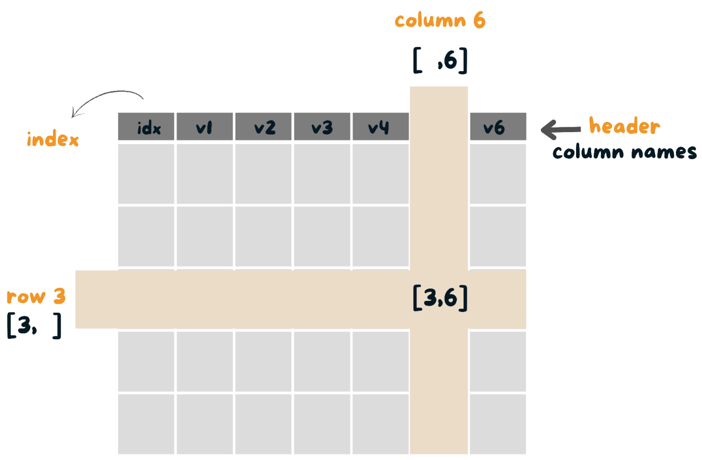


Illustration of a DataFrame structure: Rows (index) and columns (header) with labeled axes, showing how data is organized in a tabular format with values (v1, v2, etc.) in each cell.

Basic Operations on DataFrames

Adding Column

To add a new column, specify the new column name and assign values to it. For example:

```
df["Category"] = ["Environment", "Politics", "Policy"]
```

Adding Row

To add a row, you can use the .append() method (note that .append() is deprecated in Pandas and will be replaced by pd.concat() in future updates):

```
new_row = {"Title": "Sports Update", "Author": "Bob White",
"Views": 900, "Date": "2024-01-04", "Category": "Sports"}

df = df.append(new_row, ignore_index=True)
```

Accessing Data by Column or Row

Column Access: Access a column by name, such as `df["Title"]`, which would give:

Row Access by Index (iloc): access a row by its integer index using `.iloc[]`

# Access the first row

```
print(df.iloc[0])
```

Row and Column Access (loc): access specific rows and columns by label.
# Access the 'author' column for the first row

```
print(df.loc[0, "author"])
```

# Output: "Jane Doe"

Accessing Multiple Rows and Columns: access multiple rows and columns at once using slicing or lists.

# Access the first two rows

```
print(df.iloc[0:2])
```

Accessing Multiple Columns:

# Access the "Title" and "Views" columns

```
print(df[["Title", "Views"]])
```

Summary

Here's a quick reference for adding and indexing in each data structure:

Data Structure	Adding Data	Indexing Data
List	`.append(item)`, `.extend([item1, item2])`	`list[index]` for specific items by position
Dictionary	`dict[key] = value`	`dict[key]` or `dict.get(key)`
DataFrame	`df["new_column"] = [values]`, `.append(row)`	`df["column"]`, `df.iloc[index]`, `df.loc[]`

By mastering these basics, you can efficiently manage data in lists, dictionaries, and DataFrames, which are essential for organizing and analyzing data in data journalism projects.

1.4.6 Loops and Conditionals

Imagine you have a basket of fruits 🍎🍌🍇, and you want to check what type of fruit each one is. Instead of picking up each fruit and saying, "This is an apple, this is a banana, this is a grape," one by one, you can just say:

"For each fruit in the basket, check its name!"

This is what a loop does—it lets you repeat the same action for every item in a group!

Now, imagine you also want to separate the fruits into two baskets: one for apples and one for everything else. You'd say:

"If the fruit is an apple, put it in the apple basket. Otherwise, put it in the other basket."

This is what a conditional (or if statement) does—it lets you make decisions based on certain conditions.

- **Loops:** Tools that let you repeat an action for every item in a group (like a list of fruits or headlines).
- **Conditionals:** Tools that let you check a condition and decide what to do next (like filtering popular articles).

For Loop: Iterates over a list of items, allowing you to perform actions on each item. Useful for analyzing or displaying data for multiple articles, e.g., printing each headline in a list.

Example:

```
headlines = ["Climate Change Report", "Election Results",
"New Policy Updates"]

for headline in headlines:

    print(headline)
```

The loop goes through each headline and prints it.

If Statement: Checks a condition and executes code if the condition is true. Essential for filtering data, such as identifying popular articles.

Example:

```
if views > 1000:

    print("This article is popular!")
```

The code checks if views are greater than 1,000.

💡 **Tip**: Combining loops and conditionals allows you to apply filters or conditions to data sets. For example, if you have a list of headlines, you can use a loop to iterate through each and print only the headlines that mention "Election."

Example:

```
headlines = ["Climate Change Report", "Election Results",
"New Policy Updates"]


for headline in headlines:

    if "Election" in headline:

        print("Election-related headline:", headline)
```

The loop goes through each headline. If the headline contains “Election,” it prints it.

Key Takeaways:

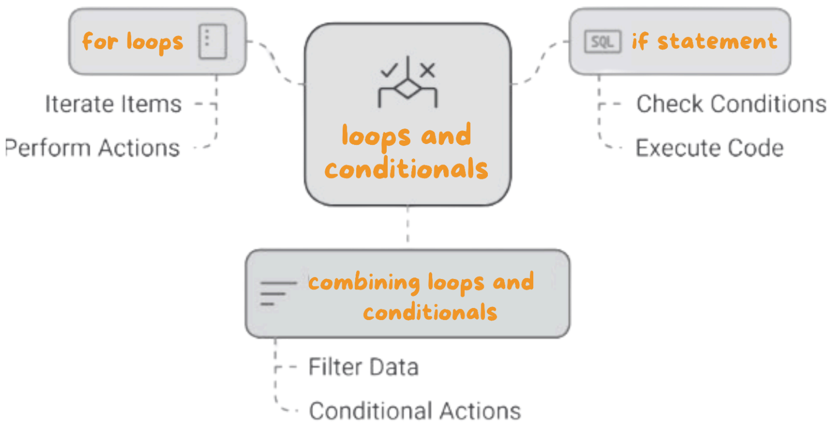


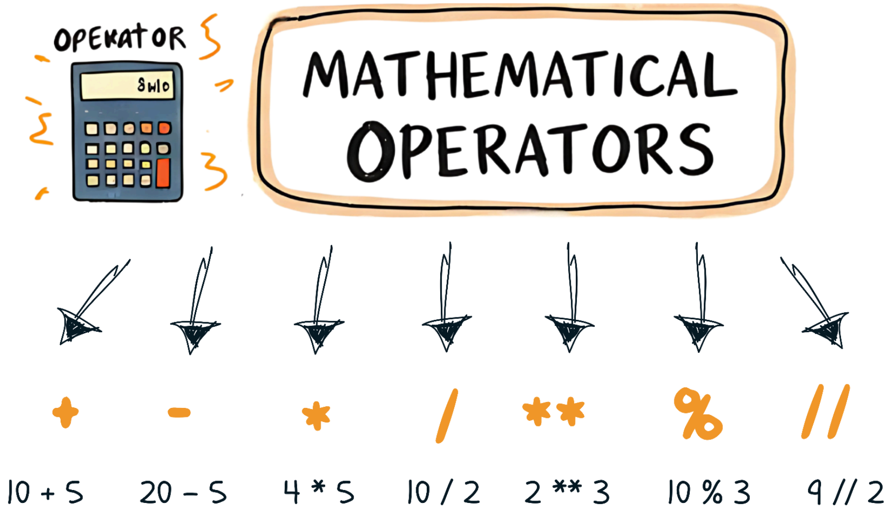
Illustration of loops and conditionals in programming:

- For Loops: Use them to repeat an action for every item in a list.
- If Statements: Use them to check conditions and run actions only when the condition is true.
- Combining Loops and Conditionals: Lets you filter data while going through a list, making it a powerful tool for tasks like analyzing articles or datasets.

🧮 1.5 Mathematical Operations

Python provides built-in operators for mathematical calculations, which are often needed in data analysis.

Operator	Operation	Example	Result
`+`	Addition	`10 + 5`	`15`
`-`	Subtraction	`20 - 5`	`15`
`*`	Multiplication	`4 * 5`	`20`
`/`	Division	`10 / 2`	`5.0`
`**`	Exponentiation	`2 ** 3`	`8`
`%`	Modulus (remainder)	`10 % 3`	`1`
`//`	Floor Division	`9 // 2`	`4`



Overview of Python's mathematical operators

🧰 Practical Exercise:

Imagine you're a journalist working on a data-driven story about the performance of your newsroom's articles. Your editor-in-chief has asked you to analyze key metrics—such as total engagement, projected growth, and the equitable impact of each article—to help the team make informed decisions about future content.

This case study walks you through the process of using Python to analyze article performance. We'll calculate total engagement, project future views, and determine the share of impact for each article. By the end, you'll have a clear understanding of how to use Python to solve real-world journalism problems and present your findings in a professional report.

You've been given the following data for three recently published articles:

- Total views: 1,500
- Number of articles: 3
- Total comments: 275

Your task is to answer the following questions:

1. What is the total engagement (views + comments) across all articles?
2. What is the expected number of views next month if they double?
3. What is the average engagement (views + comments) per article?
4. What is the share of views for each article if we assume equal impact?

```
# data
views = 1500
articles = 3
comments = 275

# 1. Total engagement (views + comments)
total_engagement = views + comments

print("Total engagement (views + comments):",
total_engagement)
Output: Total engagement (views + comments): 1775

# 2. Projected views for next month (double current views)
projected_views_next_month = views * 2

print("Projected views for next month:",
projected_views_next_month)
Output: Projected views for next month: 3000

# 3. Average engagement per article
average_engagement_per_article = (views + comments) /
articles

print("Average engagement per article:",
average_engagement_per_article)
Output: Average engagement per article: 591.67

# 4. Share of views per article (assuming equal impact)
views_per_article = views / articles

print("Share of views per article (assuming equal
impact):", views_per_article)
Output: Share of views per article (assuming equal impact): 500.0
```

⚖️ 1.6 Comparison Operators

Comparison operators are used to compare two values or variables. They return a **Boolean value**: `True` if the comparison is correct, and `False` if it isn't.

Operator	Name	Example	Result
==	Equal to	`5 == 5`	`True`
!=	Not equal to	`5 != 3`	`True`
>	Greater than	`5 > 3`	`True`
<	Less than	`5 < 3`	`False`
>=	Greater than or equal to	`5 >= 5`	`True`
<=	Less than or equal to	`5 <= 4`	`False`

🧩 1.7 Logical Operators

Logical operators allow you to combine conditions, useful for filtering data based on multiple criteria.

Operator	Description	Example	Result
`and`	True if both conditions are true	`(views > 1000) and (year == 2024)`	`True`
`or`	True if at least one condition is true	`(views > 2000) or (year == 2024)`	`True`
`not`	Reverses the condition	`not (views > 2000)`	`True`

🧰 Practical Exercise:

Imagine you're analyzing a dataset of news articles to identify high-performing content. You want to filter articles based on specific conditions, such as view count, publication year, and publication status. Logical operators like and, or, and not can help you create complex filters to extract the exact data you need.

This case study walks you through the process of using logical operators in Python to analyze article data. By the end, you'll understand how to combine conditions to filter data effectively and make data-driven decisions for your newsroom.

You've been given the following data for a specific article:

- Views: 1,500
- Year: 2024
- Author: "Jane Doe"
- Publication Status: True (published)

Your task is to answer the following questions:

5. Is this a popular article from the current year?
6. Is this article either highly viewed or published?
7. Does this article have fewer than 2,000 views?

```
# data
views = 1500  # Number of views
year = 2024   # Publication year
author = "Jane Doe"  # Author name
is_published = True  # Publication status

# 1. Check if the article is popular and from the current year

if (views > 1000) and (year == 2024):
    print("This is a popular article from the current
year.")
```

output: This is a popular article from the current year.

```
# 2. Check if the article is either highly viewed or published

if (views > 2000) or (is_published):

    print("This article is either highly viewed or it is
published.")
```

output: This article is either highly viewed or it is published.

```
# 3. Check if the article has fewer than 2000 views

if not (views > 2000):

    print("This article has fewer than 2000 views.")
```

output: This article has fewer than 2000 views.

🧰 Practical Exercise:

Analyzing News Views Data in Colab

Imagine you're a journalist analyzing the performance of a news article over two days. You've been given the number of views for today and yesterday, and your task is to calculate the difference, total views, and determine whether views have increased or decreased. This case study will guide you through the process of using Python in Google Colab to analyze this data.

Here's the data you've been given:

- Views Today: 1,200
- Views Yesterday: 900

Your task is to:

1. Calculate the difference in views between today and yesterday.
2. Calculate the total views over the two days.
3. Determine if views have increased or decreased.

Exercise Steps

1. **Create Variables**: Define two variables, `views_today` and `views_yesterday`, representing the views for an article on two different days.
2. **Mathematical Operations**: Perform basic arithmetic operations to analyze the difference and total views.
3. **Logical Operations**: Use conditional statements to check if the views have increased or decreased.
4. **Print and Explore Data**: Use basic functions to display the length, type, and results of operations.

💡 Pro Tips:

- Use `type()` to verify the data type of each variable, which is helpful for debugging or working with different types of data.
- Using `len()` on strings or lists is a quick way to check their length, useful when dealing with text data or article titles in journalism.
- Try adding more variables (e.g., `views_last_week`) and use additional operations, like calculating weekly averages, for further practice.

📋 1.8 Common Commands and Functions

Python provides built-in functions to help you work more effectively with data:

`sum()`: Adds all elements in a list, useful for summing view counts or totals.

```
total_views = sum([1500, 1200, 800])
```

`sorted()`: Sorts lists (alphabetically or numerically).

```
sorted_headlines = sorted(headlines)
```

`max()` and **`min()`** functions return the highest and lowest values in a list, respectively. They're useful for finding the most and least viewed articles.

`count()` function counts the number of times a specific value appears in a list or string. It's useful for analyzing frequency, such as counting how often a keyword appears in a dataset.

`round()` function rounds a number to a specified number of decimal places. It's useful for cleaning up numerical data for reporting.

`find()` function returns the index of the first occurrence of a substring in a string. It's useful for searching text data, such as finding keywords in headlines.

1.9 Hands-On Exercises:

This section provides a series of practical exercises to help you master working with DataFrames in Python using the Pandas library. By completing these exercises, you'll gain hands-on experience in creating, manipulating, and analyzing tabular data—essential skills for journalists working with datasets.

Each exercise builds on the previous one, guiding you through:

1. Creating DataFrames
2. Adding columns and rows
3. Accessing specific values
4. Performing calculations
5. Filtering data

By the end of this section, you'll be able to confidently use DataFrames to organize, analyze, and present data effectively. Let's dive in!

Exercise 1: Creating a DataFrame:

You are working with a dataset of articles and their engagement statistics. Create a DataFrame using the following data:

Article	Views	Likes
Climate Report	1500	200
Election Update	2000	350
Tech Trends	1200	150

1. Name the DataFrame `df`.
2. Print the DataFrame to verify it was created correctly.

Steps to Solve:

Import the Pandas library:

```
import pandas as pd
```

Define the data: Use a dictionary to define the data for the columns:

```
data = {

    "Article": ["Climate Report", "Election Update", "Tech Trends"],

    "Views": [1500, 2000, 1200],

    "Likes": [200, 350, 150]

}
```

Create the DataFrame: Use `pd.DataFrame()` to create the DataFrame:

```
df = pd.DataFrame(data)
```

Print the DataFrame: Verify the data by printing the DataFrame:

```
print(df)
```

Explanation:

- The `pd.DataFrame()` function converts the dictionary into a table-like structure.
- Each key in the dictionary represents a column, and its values represent the rows.

Tips:

- Always check for typos in your column names to avoid errors later.
- Printing the DataFrame helps verify its structure.

Exercise 2: Adding a New Column

Add a new column called `Comments` to the `df` DataFrame with the following values: `50, 120, 30`

Print the updated DataFrame to confirm the new column was added.
Steps to Solve:

Add the new column:
```
df["Comments"] = [50, 120, 30]
```

Print the updated DataFrame:
```
print(df)
```

Explanation:

- The syntax `df["Comments"] = [...]` adds a new column to the DataFrame.
- The list `[50, 120, 30]` specifies the values for each row.

Tips:

- Ensure the number of values matches the number of rows in the DataFrame.
- Use descriptive column names for clarity.

Exercise 3: Adding a New Row

Add a new row to the `df` DataFrame with the following data:

- `Article`: "Sports Highlights"
- `Views`: 1800
- `Likes`: 250
- `Comments`: 80

Steps to Solve:

Add the new row using **`loc`**:

```
df.loc[len(df)] = ["Sports Highlights", 1800, 250, 80]
```

Print the updated DataFrame:

```
print(df)
```

Explanation:

- `df.loc[len(df)]` specifies the position for the new row (after the last row).
- The list of values represents the new row.

Tips:

- Use `len(df)` to ensure the new row is added at the correct position.
- Always check the updated DataFrame to confirm the new row was added correctly.

Exercise 4: Accessing a Specific Value:

Use `iloc` and `loc` to access specific values:

1. Find the value of `Likes` for the second row ("Election Update") using `iloc`.
2. Find the value of `Comments` for the last row using `iloc`.
3. Find the value of `Views` for the row where the `Article` is "Tech Trends" using `loc`.

Steps to Solve:

Access the `Likes` value for the second row:

```
print("Likes for 'Election Update':", df.iloc[1]["Likes"])
```

Access the `Comments` value for the last row:

```
print("Comments for last row:", df.iloc[-1]["Comments"])
```

Access the `Views` value for "Tech Trends":

```
print("Views for 'Tech Trends':", df.loc[df["Article"] ==
"Tech Trends", "Views"].values[0])
```

Explanation:

- `iloc` accesses rows and columns by their position (index).
- `loc` accesses rows and columns by their labels (name or condition).

Tips:

- Use `iloc[-1]` to access the last row dynamically.
- When using `loc` with conditions, always verify the `.values[0]` to extract the value.

Exercise 5: Adding a Calculated Column

Add a new column `Engagement Rate (%)` with the formula:

```
(Likes + Comments) / Views * 100
```

Steps to Solve:

Calculate and add the column:

```
df["Engagement Rate (%)"] = (df["Likes"] + df["Comments"])
/ df["Views"] * 100
```

Print the updated DataFrame:

```
print(df)
```

Explanation:

- The formula calculates the percentage of engagement based on `Likes` and `Comments`.
- The result is stored in a new column.

Tips:

- Use parentheses to ensure the correct order of operations.
- Confirm that the calculation is applied to all rows.

Exercise 6: Filtering Data

Filter the DataFrame to include only rows where `Views` are greater than or equal to 1500. Save the filtered data in a new DataFrame called `df_filtered`.

Steps to Solve:

Filter the rows:

```
df_filtered = df[df["Views"] >= 1500]
```

Print the filtered DataFrame:

```
print(df_filtered)
```

Explanation:

- The condition `df["Views"] >= 1500` creates a filter for the rows.
- Only rows that meet the condition are included in `df_filtered`.

Tips:

- Use descriptive names like `df_filtered` for the new DataFrame.
- Always double-check the filtered data for accuracy.

Bonus Challenge: Combining Everything

1. Create a new DataFrame with 4 rows and columns: `Article`, `Views`, `Likes`, `Comments`.
2. Add a column `Engagement Rate (%)`.
3. Filter rows where `Likes` are greater than or equal to 300, and save them in a new DataFrame.

Steps to Solve:

Create the DataFrame:

```
data = {

    "Article": ["Article A", "Article B", "Article C",
"Article D"],

    "Views": [1000, 2000, 1500, 1800],

    "Likes": [200, 400, 300, 250],

    "Comments": [50, 100, 70, 90]

}

df = pd.DataFrame(data)
```

Add the **`Engagement Rate (%)`**:

```
df["Engagement Rate (%)"] = (df["Likes"] + df["Comments"])
/ df["Views"] * 100
```

Filter rows by **`Likes`**:

```
df_filtered = df[df["Likes"] >= 300]

print(df_filtered)
```

Expected Skills After Completing:

- Creating and manipulating DataFrames.
- Accessing and filtering data dynamically.
- Performing calculations to derive insights.

📊 1.10 Essential Libraries

Python libraries are crucial in data journalism, simplifying tasks like data collection, manipulation, analysis, and visualization. Here's an organized list of libraries by role, detailing their use, installation command, and import command.

1.10.1 Data Collection and Web Scraping

Library	Use	Installation	Import Command
Requests	Making HTTP requests for APIs or web scraping	`!pip install requests`	`import requests`
BeautifulSoup	Parsing HTML and XML documents for scraping	`!pip install beautifulsoup4`	`from bs4 import BeautifulSoup`
Scrapy	Advanced web scraping framework	`!pip install scrapy`	`import scrapy`
Newspaper3k	Extracting text from news articles	`!pip install newspaper3k`	`from newspaper import Article`

Tweepy	Collecting tweets through the Twitter API	`!pip install tweepy`	`import tweepy`

💡 **Tip**: Use `Requests` for quick HTTP requests and `Scrapy` for complex scraping projects that require spidering and asynchronous requests.

1.10.2 Data Manipulation and Analysis

Library	Use	Installation	Import Command
Pandas	Data analysis and manipulation	`!pip install pandas`	`import pandas as pd`
Numpy	Numerical computations, array manipulation	`!pip install numpy`	`import numpy as np`
Dask	Parallel computing with large datasets	`!pip install dask`	`import dask.dataframe as dd`
SQLite3	Managing relational databases	*Included with Python*	`import sqlite3`
Openpyxl	Reading/writing Excel files	`!pip install openpyxl`	`import openpyxl`

💡 **Tip**: Use **Pandas** for general data manipulation and **Dask** for larger datasets that don't fit into memory.

1.10.3 Data Visualization

Library	Use	Installation	Import Command
Matplotlib	Basic data visualization (charts, plots)	`!pip install matplotlib`	`import matplotlib.pyplot as plt`
Seaborn	Statistical data visualization with aesthetics	`!pip install seaborn`	`import seaborn as sns`
Plotly	Interactive data visualizations	`!pip install plotly`	`import plotly.express as px`
Altair	Declarative statistical visualizations	`!pip install altair`	`import altair as alt`
Geopandas	Geospatial data visualization	`!pip install geopandas`	`import geopandas as gpd`

💡 Tip: Checking if a Library is Installed

If you're not sure whether a library is installed in your Python environment, you can:

1. Run `!pip show <library_name>` in a cell (e.g., `!pip show pandas`) to check if the library is installed and see details like version and location.
2. Use `pip list` in a cell to view a full list of installed packages along with their versions:

```
!pip list
```

📰 1.11 Project: Building and Analyzing a DataFrame of Articles

In this project, you'll create a DataFrame containing article information, then explore basic operations to analyze the data.

Step 1: Set Up Your Google Colab Environment

1. Open Google Colab.
2. Create a new notebook (`File > New Notebook`).

Step 2: Import the Pandas Library

Pandas is the library we'll use to create and manipulate DataFrames.

```
import pandas as pd
```

Step 3: Create a DataFrame

Let's start by creating a list of dictionaries, where each dictionary represents an article with information such as `title`, `author`, `views`, and `date`. Then, we'll convert this list into a DataFrame.

```
# Define article data as a list of dictionaries

articles = [

    {"title": "Climate Change Report", "author": "Jane Doe", "views": 1500, "date": "2024-01-01"},

    {"title": "Election Results", "author": "John Smith", "views": 1200, "date": "2024-01-02"},

    {"title": "Policy Update", "author": "Alice Brown", "views": 800, "date": "2024-01-03"},

    {"title": "Sports Highlights", "author": "Bob White", "views": 900, "date": "2024-01-04"},
    {"title": "Economic Analysis", "author": "Emily Davis",
"views": 2000, "date": "2024-01-05"}

]

# Convert the list of dictionaries into a DataFrame

df = pd.DataFrame(articles)

# Display the DataFrame

df
```

Output:

The `df` variable now contains a DataFrame that looks like this:

	title	author	views	date
0	Climate Change Report	Jane Doe	1500	2024-01-01
1	Election Results	John Smith	1200	2024-01-02
2	Policy Update	Alice Brown	800	2024-01-03
3	Sports Highlights	Bob White	900	2024-01-04
4	Economic Analysis	Emily Davis	2000	2024-01-05

Step 4: Basic DataFrame Operations

View the First Few Rows Use the `.head()` function to display the first few rows of the DataFrame.

```
df.head()
```

Check Data Types View the data types of each column with `.dtypes`.

```
df.dtypes
```

Get Summary Statistics Use `.describe()` to get basic statistics like mean, min, and max for numeric columns.

```
df.describe()
```

Step 5: Adding and Modifying Columns

Add a "Category" Column Add a new column called "category" with some sample data.

```
df["category"] = ["Environment", "Politics", "Policy",
"Sports", "Economics"]
```

Calculate Average Views Calculate the average number of views across all articles and add it as a new column called "average_views".

```
df["average_views"] = df["views"].mean()

df
```

Step 6: Filtering Data

Filter Popular Articles Select articles with views greater than 1000 and create a new DataFrame `popular_articles`.

```
popular_articles = df[df["views"] > 1000]

popular_articles
```

Filter by Category Find articles in a specific category, such as "Politics".

```
politics_articles = df[df["category"] == "Politics"]

politics_articles
```

Step 7: Sorting the Data

Sort by Views Sort the DataFrame by views in descending order to find the most viewed articles.

```
df_sorted = df.sort_values(by="views", ascending=False)

df_sorted
```

🧰 Check Your Knowledge:

Test your understanding of the concepts covered in this project:

What does the .head() function do in Pandas?
a) Displays the last few rows of a DataFrame.
b) Displays the first few rows of a DataFrame.
c) Deletes the first few rows of a DataFrame.
d) None of the above.

Answer: b) Displays the first few rows of a DataFrame.

How do you add a new column to a DataFrame?
a) df.add_column("new_column", values)
b) df["new_column"] = values
c) df.insert("new_column", values)
d) df.create_column("new_column", values)

Answer: b) df["new_column"] = values

What does the .describe() function provide?
a) A list of column names.
b) Summary statistics for numeric columns.
c) A list of unique values in each column.

d) A filtered DataFrame.

Answer: b) Summary statistics for numeric columns.

How do you filter rows where the "views" column is greater than 1000?
a) df[df["views"] > 1000]
b) df.filter("views" > 1000)
c) df.select("views" > 1000)
d) df[df.views > 1000]

Answer: a) df[df["views"] > 1000]

What does the .sort_values() function do?
a) Deletes rows based on a condition.
b) Sorts the DataFrame by one or more columns.
c) Adds a new column to the DataFrame.
d) Filters rows based on a condition.

Answer: b) Sorts the DataFrame by one or more columns.

🔍 1.12 Additional Tips

1. **Code Readability**: Use comments to make your code easy to understand and revisit.

2. **Markdown for Context**: Use Markdown cells in Colab to add structure and explanations, guiding readers through your analysis.

3. **Clear Variable Names**: Choose descriptive variable names that make your code self-explanatory.

4. **Incremental Testing**: Break tasks into smaller steps and test frequently to catch errors early.

📌 1.13 Chapter Summary

In this chapter, we covered:

- **Google Colab:** A beginner-friendly, cloud-based coding platform for writing and running Python code without setup.
- **Python Basics**: Key concepts like data types, lists, dictionaries, and loops.
- **Essential Libraries**: Pandas, Matplotlib, Seaborn, BeautifulSoup, and Requests for data manipulation, visualization, and web scraping.
- **Building and Analyzing a DataFrame:** Applying your skills to a real-world scenario with article data.g.

These skills will help you start exploring data and creating powerful visual narratives! Practice these exercises to solidify your understanding and start building your data journalism toolkit.

ISBN 9798309733132

90000

9 798309 733132