DS105W – Data for Data Science

Week 02

Python Fundamentals, Collections and First Steps with APIs

Dr Jon Cardoso-Silva
LSE Data Science Institute

🗓️ 29 Jan 2026

16:00 – 16:20

1 DataQuest Debrief & Markdown

Let’s start by seeing where you are with the DataQuest lessons and then connect them to what you experienced in Week 01.

Quick Completion Poll

Through Mentimeter we'll see how far you got with the 📝 **W02 Formative** DataQuest lessons:

- **Programming in Python** ✅ / ⏳ / ❌
- **Programming Python Variables** ✅ / ⏳ / ❌
- **Python Data Types: Integers, Floats, Strings** ✅ / ⏳ / ❌
- **Python Lists** ✅ / ⏳ / ❌
- **Python Dictionaries** ✅ / ⏳ / ❌

> 💡 You feel behind tomorrow in the lab if you haven't done all of these!.

Markdown Concepts

We use Markdown to format **Slack messages**, in our **Jupyter Notebooks**, these slides, and all the course webpages you see on Moodle.

If you type this:

```
This is a **bold** text.
```

```
This is an _italic_ text.
```

```
[This is a link](https://lse.ac.uk/dsi)
```

```
`print("Hello, World!")`
```

````
```python
# This is a code block
print("Hello, World!")
```
````

You get this:

This is a **bold** text.

This is an *italic* text.

This is a link

`print("Hello, World!")`

```python
# This is a code block
print("Hello, World!")
```

The code blocks are just to **represent** code, not to execute it.

Markdown Concepts (Headings)

There are also **headings** in Markdown. They help structure content — not just make text big!
(Those won't work in Slack, by the way.)

If you type this:	You get this:
`# Title (H1)`	# Title (H1)
`## Section (H2)`	## Section (H2)
`### Sub-section (H3)`	### Sub-section (H3)
`#### Sub-sub-section (H4)`	#### Sub-sub-section (H4)

⚠️ **Do not use `#` just to make text bigger!** It's not what it represents.
Use it to create hierarchical demarcations of sections instead.

Group Discussion: DataQuest Concepts

Let's connect what you learned in DataQuest to the data work you've done. Form groups of 3-4 people around you.

Your task (5 minutes total):

1. **Share your "aha moment"** (2 minutes): Each person briefly shares one concept from DataQuest that clicked for them this week. Could be variables, data types, lists, or anything else.
2. **Connect to Week 01** (2 minutes): Discuss together: How might these Python concepts relate to the DataFrame work you did last week?
3. **Pick one insight** (1 minute): Choose one connection your group found interesting to share in Slack.

> 💡 **Post to Slack's `#social` channel:** Drop your group's key insight in the thread. I'll synthesise common patterns in a moment.

16:20 – 16:40

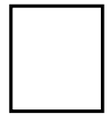
2 Binary & Memory Foundations

To understand why `my_list[0]` works, we need to understand how computers actually think.

Computers Only Understand 0s and 1s

Numbers, text, images, and sounds are all stored as sequences of 0s and 1s in your computer's memory. Each 0 or 1 is called a **bit**.

Think of a bit as a tiny box:



Computers Only Understand 0s and 1s

Numbers, text, images, and sounds are all stored as sequences of 0s and 1s in your computer's memory. Each 0 or 1 is called a **bit**.

Think of a bit as a tiny box:

$$\boxed{0} \quad \leftarrow \quad \text{a bit can have a value of } 0$$

Computers Only Understand 0s and 1s

Numbers, text, images, and sounds are all stored as sequences of 0s and 1s in your computer's memory. Each 0 or 1 is called a **bit**.

Think of a bit as a tiny box:

1 ← OR it can have a value of 1

but nothing else!

How Numbers are Stored

With more bits, we can represent more numbers. Here's how 4 bits can represent 16 different numbers:

0	0	0	0	→ 0	1	0	0	0	→ 8
0	0	0	1	→ 1	1	0	0	1	→ 9
0	0	1	0	→ 2	1	0	1	0	→ 10
0	0	1	1	→ 3	1	0	1	1	→ 11
0	1	0	0	→ 4	1	1	0	0	→ 12
0	1	0	1	→ 5	1	1	0	1	→ 13
0	1	1	0	→ 6	1	1	1	0	→ 14
0	1	1	1	→ 7	1	1	1	1	→ 15

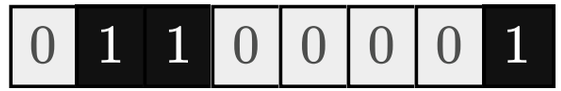
Connection to the DataQuest lessons (📝 W02 Practice): when you create an **integer** in Python, you are telling your computer to reserve a fixed-size space in memory for a series of 0s and 1s.

The ASCII Table: How Text Becomes Binary

In the early days of computing, text was represented using the ASCII table. ASCII uses 7-8 bits to represent each individual character. Here are some examples:

The letter '`A`' is represented by the number 65 encoded in binary as:

The letter '`a`' (lowercase) is represented by the number 97 encoded in binary as:



The linebreak character '`\n`' is represented by the number 10 encoded in binary as:

Click here to see the full table. Modern Python uses UTF-8, which includes emojis! 🤗

🗂️ Variables as Memory Addresses

When you create a variable in Python, you're not creating a container for data. You're creating a **reference** (a pointer) to a **location** in memory where the data lives.

```
temperature = 25.3
```

What actually happens:

1. Python converts `25.3` to binary
2. Finds available space in memory
3. Stores the binary representation there
4. Creates label `temperature` pointing to that location

When you reassign:

```
temperature = 28.7
```

Python doesn't change the old value. It stores `28.7` in a new location and updates the reference.

Memory Addresses (simplified):

Address	Variable	Value
0x1000	temperature →	25.3
…	…	…
0x1048	temperature →	28.7

🔍 You can try it yourself:

```
temperature = 25.3
print(id(temperature))  # Memory address

temperature = 28.7
print(id(temperature))  # Different address!
```


Data Types and Memory Storage

> 💡 Although you can't change the size of the data types in 'pure Python', we will enforce this when we start working with numpy and pandas in the next few weeks.

Integer Storage:

- `int32`: Uses 32 bits (4 bytes)

  Can store: -2,147,483,648 to 2,147,483,647
- `int64`: Uses 64 bits (8 bytes)

  Can store: much larger numbers!

Why this matters:

A DataFrame with 1 million integers:

- `int32`: ~4 MB of memory
- `int64`: ~8 MB of memory

Choosing the right type saves memory!

Float Precision:

Python floats use 64 bits by default.

This is why sometimes:

```
0.1 + 0.2 == 0.3  # False!
# Actually gives: 0.30000000000000004
```

Odly, the binary representation of numbers in programming isn't always exact.

> 🤯 Click here to read more about this.

📦 Collections Store References Too

Lists and dictionaries don't store the actual data. They, too, store references to where the data lives in memory.

```
temps = [18, 22, 19]
```

Memory layout:

temps →	[0x2000, 0x2004, 0x2008]
0x2000	18
0x2004	22
0x2008	19

Why this matters:

1. Lists can resize because they just add more references
2. When you pass a list to a function, you're passing the reference, not copying all the data
3. This is why operations on large datasets can be fast

Collections are **containers of references**, not containers of actual values.

Why bother? Understanding this memory model helps you understand why some operations are fast (just changing a reference) and others are slow (copying actual data), and why some data structures work better for certain tasks.

16:40 – 17:05

3 Collections In Practice

Most of the time in data science, we work with **collections** of values.

In ‘pure Python’, the two most important collections are **lists** and **dictionaries** and they are the foundation for the more complex data structures we will learn about later in Weeks 03 and 04.

⚠️ The Problem with Separate Lists

> Think about the data you worked with in the (📝 W01 Practice) at the end. You saw **DataFrames** (tables) with multiple columns. What if we tried to store that data using separate lists?

```
# Weather data as separate lists
dates = ['2025-10-13', '2025-10-14', '2025-10-15']
temps = [18, 22, 19]
conditions = ['cloudy', 'sunny', 'rainy']

# To get info about the first day:
print(f"Date: {dates[0]}")
print(f"Temp: {temps[0]}°C")
print(f"Condition: {conditions[0]}")
```

This is why we need better data structures. More complex structures will let us connect related data with meaningful names instead of fragile positions.

Three critical problems:

1. **Coordination nightmare**: If you sort `temps`, the other lists don't follow. Your data is now corrupted.
2. **Fragile connections**: The relationship between `dates[0]`, `temps[0]`, and `conditions[0]` exists only in your mind, not in the code.
3. **Error-prone**: Add a temperature but forget to add a date? Your lists are now different lengths.

🧠 Memory: Lists vs Dictionaries

Still, it's important to understand how lists and dictionaries organise those references differently as these are the foundations for everything that is to come.

Lists: Sequential Memory

Lists store references in order. Python can quickly access any position because it knows exactly where each item lives in memory.

```
temps = [18, 22, 19]
# Memory: [ref→18, ref→22, ref→19]
```

Fast: Access by position (`temps[0]`)
Limitation: No meaningful names

Dictionaries: Named Memory

Dictionaries use a hash table to map keys to memory locations. The key becomes a meaningful label for the data.

```
weather = {'temp': 18, 'city': 'London'}
# Memory: 'temp' → ref→18, 'city' → ref→'London'
```

Fast: Access by name (`weather['temp']`)
Advantage: Self-documenting code

🤓 **If you want to know more:** Python dicts and memory usage

📋 Lists vs Dictionaries: Concept and Syntax

Most of the data you will work with will already come in a collection, but if you need to create one, here's how you do it.

List

```
# List of temperatures in Celsius
temperatures = [22, 21, 19, 23, 20]
```

Or

```
temperatures = [
    22,
    21,
    19,
    23,
    20
]
```

Dictionary

```
# Dictionary of temperatures
weather_data = {'09:00': 22, '12:00': 21, '15:00': 19,
```

Or

```
weather_data = {
    '09:00': 22,
    '12:00': 21,
    '15:00': 19,
    '18:00': 23,
    '21:00': 20
}
```

> 💡 You can use multiple lines for readability.

🔍 Accessing Elements

Accessing data differs between lists and dictionaries.

List

```
# Accessing the first temperature
first_temp = temperatures[0]

# Accessing the last temperature
last_temp = temperatures[-1]
```

Lists are indexed by integers, starting at 0.

To get an element, you need to know its position in the list.

Dictionaries

```
# Accessing temperature at 12:00
temp_at_noon = weather_data['12:00']

# Accessing temperature at 18:00
temp_at_evening = weather_data['18:00']
```

Dictionaries are accessed by keys, not by position.

You need to know the key to retrieve the value.

🌤️ Dictionary of Lists: The DataFrame Pattern

Let's see how real weather data might look using a dictionary of lists:

```
# A dictionary of lists (like a DataFrame!)
weekly_weather = {
    'date': ['2025-10-13', '2025-10-14', '2025-10-15'],
    'temp': [18, 22, 19],
    'humidity': [65, 58, 72],
    'conditions': ['cloudy', 'sunny', 'rainy']
}

# Accessing data - column first, then row
all_temps = weekly_weather['temp']  # Gets [18, 22, 19]
monday_temp = weekly_weather['temp'][0]  # Gets 18
tuesday_conditions = weekly_weather['conditions'][1]  # Gets 'sunny'
```

This is very similar to how pandas DataFrames work! Each key is a column name, and each value is a list of data for that column. Notice the access pattern: `dict['column'][row]` - same as `df['column'][0]` from Week 01.

Quick Poll: When to Use Each Structure?

Let's check your understanding. Go to the `#social` channel on Slack and vote:

Scenario: You're storing hourly temperature readings for London, and you want to look up the temperature at a specific time (like "14:00").

Which structure would you use?

A. A simple list: `[18, 22, 19, 23, 20]`
B. A dictionary: `{'09:00': 18, '12:00': 22, '15:00': 19, ...}`
C. Separate lists for times and temperatures
D. A list of dictionaries

17:05 – 17:15

Break

After the break:

- Live data collection from APIs
- Converting JSON to Python dictionaries
- Connecting API data to the pandas DataFrames you used in Week 01

4 APIs & Live Data Collection

17:15 – 17:50

The Problem with Manual Data Entry

So far, we've created our own lists and dictionaries manually, mostly to demonstrate the concepts. But in practice, we will collect it from data sources.

❌ **Problem:**
Typing weather data manually is tedious and error-prone.

✅ **Solution:**
In this course, we will use **APIs** (Application Programming Interfaces) to fetch live real data dynamically.

What is an API?

An API is like a **vending machine**:

1. You **make a request** (insert a coin & press a button).
2. The **API processes it** (retrieves your snack).
3. You **get a response** (your snack comes out).

In Python, we use a package called `requests` to "talk" to APIs.

The `requests` package does not come pre-installed with Python. You need to install it using `pip`.

- On **VS Code**, click on the **Menu** icon then navigate to **Terminal** > **New Terminal**.
- A window will pop up at the bottom of the screen.
- In the terminal window, type `pip install requests` and press `Enter`.
- Wait for the installation to complete.
- The `requests` package is now installed and ready to use on Jupyter Notebooks.

The 🌤 Open-Meteo API for Weather Data

Visit Open-Meteo

🔗 https://open-meteo.com/

Explore their API documentation.

🚀 We will:

- Request hourly temperature for London
- Receive structured weather data (it comes back in a format called JSON).

Constructing the Request

```
import requests

url = "https://api.open-meteo.com/v1/forecast"
params = {
    "latitude": 51.5085,
    "longitude": -0.1257,
    "hourly": "temperature_2m",
    "timezone": "Europe/London"
}

response = requests.get(url, params=params)
```

✅ **We now have real-time weather data!**

⏭ Let’s inspect the response (live demo).

📦 Converting API Data to a Python Dictionary

The `response` you get from the API is just pure text, just a **string** that looks like a Python dictionary.

```
print(response)
```

> 💡 **TIP:** Just because something **looks like** a dictionary or a list, it doesn't mean it is.

To convert it into a Python dictionary, we use the `json()` method:

```
weather_data = response.json()
type(weather_data)
```

Now `weather_data` is a Python dictionary!

We can then access its values just like any other dictionary:

```
weather_data["hourly"]["temperature_2m"]
```


🎯 Live Demo: Open-Meteo API

Let me show you this API in action. We'll:

1. **Make the request** and examine the response structure
2. **Extract temperature data** into a Python list
3. **See how this data could become a pandas DataFrame column**

During this demo, think about: How is this different from the clean CSV files you worked with in Week 01? What are the advantages and challenges of live data?

5 Synthesis & W03 Preview

17:50 – 18:00

🔗 Connecting Dictionaries to DataFrames

Here's what is coming next. Next week, code-wise, we will be focused on transforming JSON data into clean and nicely tabular DataFrames.

Your knowledge of dictionaries and lists will be put to the test!

Dictionary structure:

```
weather_dict = {
    'time': ['09:00', '12:00', '15:00'],
    'temp': [18, 22, 19],
    'humidity': [65, 58, 72]
}
```

Access a "column":

```
temperatures = weather_dict['temp']
# Result: [18, 22, 19]
```

DataFrame equivalent:

```
import pandas as pd

df = pd.DataFrame(weather_dict)
print(df)
```

Access a column:

```
temperatures = df['temp']
# Result: pandas Series with [18, 22, 19]
```


🔮 JSON Preview: What's Coming in Week 03

JSON objects are a mixture of dictionaries and lists, nested inside each other. Your goal will be to transform them into nice tables!

```
# This is what Open-Meteo actually returns:
api_response = {
    "hourly": {
        "time": ["2025-10-16T00:00", "2025-10-16T01:00", "2025-10-16T02:00"],
        "temperature_2m": [15.2, 14.8, 14.5]
    },
    "hourly_units": {
        "temperature_2m": "°C"
    }
}
```

Practice your lists and dictionaries skills! You need to be able to understand if a structure is a list or a dictionary, and how to **manipulate them**.

You practice some of it in the 💻 W02 Lab tomorrow but keep practicing further!

➡️ SOON What's Next?

💻 W02 Lab (Friday)

- Practice extracting nested data structures

📝 W03 Formative (next week)

Lots of new cool stuff awaits you!

- Terminal basics: file systems, paths, and directories
- More DataQuest: `for` loops and `if-else` conditionals
- Git basics: version control for your data projects

🎯 The Big Picture

- **Week 01**: Explored data that was already clean and structured
- **Week 02**: Learned how computers work and how to get live data
- **Week 03**: Bridge the gap by transforming messy JSON into clean tables

🤔 If you want to go deeper

Key Questions to Reflect On:

- How do the Python fundamentals from DataQuest connect to the data analysis you did in the (📝 W02 Practice)?
- What's the relationship between memory and DataFrame operations?
- How might APIs change the way you think about data collection?

Next week: We'll transform complex JSON data into the clean DataFrames you're already comfortable working with.

💬 **Remember:** Use the `#help` channel on Slack for any questions that come up as you work through the lab tomorrow!