MASTER'S COURSE · LECTURE 1

Principles and Methods of Data Analysis in the Energy Industry

Lecture 1 — Exploratory Data Analysis & Python Foundations

Information Technologies for Sustainable Energy Engineering

Sources: A. Downey, *Think Stats 2e*, Ch. 1 · A. Downey, *Elements of Data Science*, Ch. 1–5

ROADMAP

What we'll cover today

1

Part A

Thinking like a data analyst

Why anecdotes mislead, the five steps of a statistical approach, study design, data cleaning and validation, interpretation and ethics.

2

Part B

Python foundations for data work

Numbers and variables, text/dates/locations, lists and NumPy arrays, loops and files, dictionaries — the building blocks every later lecture depends on.

Why this order? Every technique in this course — from PMFs to regression on load curves — assumes fluency with these Python basics and this analytical mindset. This lecture builds the foundation the other eight lectures and nine labs stand on.

PART A

Thinking Like a Data Analyst

Think Stats 2e, Chapter 1 — Exploratory Data Analysis

Energy engineering runs on data you didn't choose

- Smart meters, SCADA systems, weather stations, and IoT sensors generate continuous streams of measurements — far more than any spreadsheet formula alone can make sense of.
- Questions engineers actually face: Does rooftop solar output really drop faster in humid regions? Did the retrofit reduce building energy use, or did the weather just get milder? Is a turbine's vibration trend a warning sign or normal noise?
- Answering these well requires the same discipline used across all data-driven fields: collect data carefully, describe it honestly, explore it before modeling it, and quantify uncertainty before drawing conclusions.

```
# a taste of what's ahead — 9 lectures
from here:
df['capacity_factor'].describe()
df.groupby('season')['load_kw'].mean()
```

Today: the concepts and the raw Python — not pandas yet.

THE CLASSIC CASE STUDY

“Do first babies tend to arrive late?”

Downey opens Think Stats with a question people argue about constantly, using only personal stories:

> *“My two friends who had their first babies BOTH went almost two weeks overdue.”*

> *“My first one came two weeks late — I think the second will come early!”*

> *“That can't be true — my sister was my mother's first, and she was early.”*

This is anecdotal evidence: unpublished, personal, and — as we'll see — usually unreliable as proof.

Swap the topic and the pattern repeats in engineering: “Our inverters always fail in year 3”, “Panels never survive the first hot summer here.” Same trap, different domain.

DIAGNOSIS

Four reasons anecdotal evidence fails

1

Small numbers

A handful of stories can't reveal an effect that's small relative to natural variation.

2

Selection bias

People who show up to argue about a topic aren't a random sample of everyone affected by it.

3

Confirmation bias

Believers repeat confirming stories; skeptics repeat counterexamples — neither counts cases fairly.

4

Inaccuracy

Personal stories get misremembered and exaggerated with every retelling.

In casual conversation, none of this is wrong — but it fails the higher bar we need before recommending capital spending, policy changes, or safety measures.

The statistical approach: five steps

1	**Data collection**	Use data gathered by a study designed for valid inference — not whoever happened to comment online.
2	**Descriptive statistics**	Summarize the data concisely and choose visualizations that reveal, not hide, its structure.
3	**Exploratory data analysis**	Look for patterns, differences, and inconsistencies before committing to a model.
4	**Estimation**	Use a sample to estimate characteristics of the full population it was drawn from.
5	**Hypothesis testing**	For an apparent effect, ask whether it could plausibly have happened by chance alone.

STUDY DESIGN

Vocabulary from the NSFG — translated to energy data

Term	Definition	Energy-industry analogue
Population	Everyone / everything the study wants to describe.	*All residential customers on a utility's grid.*
Sample	The subset we actually collect data from.	*The 5,000 smart meters selected for a demand-response pilot.*
Respondent / record	One unit of observation.	*One meter-reading, one turbine, one weather-station-hour.*
Representative	Every population member has an equal chance of being sampled.	*A meter sample matching the grid's real mix of house sizes and heating types.*

Cross-sectional vs. longitudinal studies

Cross-sectional

A snapshot of a group at one point in time. The NSFG's Cycle 6 ran Jan 2002 – Mar 2003, one “cycle” of a repeated cross-sectional design.

Energy example

An annual energy-audit campaign across 200 buildings, done once.

Longitudinal

The same group observed repeatedly over time.

Energy example

A SCADA system logging the same turbines' vibration every minute for years.

SAMPLING DESIGN

Representative sampling — and when we deliberately break it

- A representative sample gives every population member an equal chance of being included; results generalize cleanly to the whole population.
- The NSFG is deliberately not representative: it oversamples Hispanics, African-Americans, and teenagers so each subgroup is large enough for valid statistics.
- The trade-off: oversampled data needs extra care (weighting) before it can describe the general population.
- Energy parallel: a utility studying EV-charging load might oversample neighborhoods with high EV adoption to get enough charging events to analyze — but must reweight before claiming the result describes the whole city.

DataFrames and Series — the shape of things to come

Think Stats reads survey data into a pandas DataFrame: rows are records, columns are variables.

```
>>> df.shape
(13593, 244)

>>> df.columns[1]
'pregordr'

>>> col = df['pregordr']   # a Series
>>> type(col)
<class 'pandas.core.series.Series'>
```

- A column pulled from a DataFrame is a Series — like a list, with an index and extra features.
- We will build DataFrames properly from Lecture/Lab 7 onward. Today's goal is the raw Python underneath: variables, sequences, loops, dictionaries.
- Keep this picture in mind: every dataset we analyze this semester — load curves, weather logs, turbine telemetry — eventually becomes rows and columns like this.

Data cleaning and validation

- Cleaning: handle special/missing codes, fix units, and derive new columns — e.g. converting pounds+ounces to a single weight, or centiyears to years.
- Special codes are dangerous if untreated: 97/98/99 (“not ascertained / refused / don't know”) can silently produce a “99-pound baby” if treated as real numbers.
- Validation: cross-check computed summaries (value_counts) against a trusted reference — the codebook, a prior report, a calibration certificate.
- Energy analogue: a sensor code of -1 or 9999 for “no reading” must become NaN, not be averaged into your load curve; a reported capacity factor over 100% is a red flag, not a record.

TWO LEVELS AT ONCE

Interpretation: statistics and context together

- A number without context can mislead: a respondent's pregnancy-outcome sequence (4 4 4 4 4 4 1) is “not unusual” statistically, but represents six miscarriages endured by one person.
- Every dataset represents real people, equipment, or communities — energy data included. A spike in outage duration is also a household without heat.
- We have an obligation to use data competently and respectfully — this applies just as much to smart-meter records of real households as to health surveys.

PART B

Python Foundations

Elements of Data Science, Chapters 1–5 — from Variables to Dictionaries

Two kinds of numbers: int and float

- int — integer values, typically used for counts: number of turbines, number of outages.
- float — numbers with a fractional part, used for measurements: 3.14159, a voltage reading, an efficiency ratio.
- Floats can be written in scientific notation: 1.2345e3 means $1.2345 \times 10^3 = 1234.5$.
- Floating-point numbers are finite approximations — 0.1 + 0.1 + 0.1 gives 0.30000000000000004, not exactly 0.3. Never test floats for exact equality.

```
>>> 3
3
>>> 3.14159
3.14159
>>> 1.2345e3
1234.5
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Arithmetic operators and precedence

`+  -`	addition, subtraction
`*  /`	multiplication, division
`**`	exponentiation
`( )`	override precedence, exactly as in math

Python does NOT allow implicit multiplication: 3(2+1) is a TypeError, not 3×(2+1). PEMDAS still applies: parentheses, exponents, multiply/divide, add/subtract.

```
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
>>> 2 ** 3
8
```

```
# Exercise: raise 1+2 to the power 3*4.
# Expected result: 531441
```

Libraries: importing NumPy for math functions

- sin, cos, exp, log are not built into Python — they live in a library, a reusable collection of values and functions.
- NumPy (“Numerical Python”) is the library we import as np — the standard short name used everywhere in data science code, including every lab this semester.
- Names are case-sensitive: import NumPy as np fails; it must be numpy.
- np.log is the natural logarithm; np.exp raises e to a power — both reappear constantly for growth, decay, and efficiency calculations.

```
import numpy as np

>>> np.pi
3.141592653589793
>>> np.log(100)
4.605170185988092
>>> np.exp(1)
2.718281828459045
```

Variables, and an exponential-growth example

A variable is a name that refers to a value: x = 5 creates x with value 5. Assignment (=) is not mathematical equality — it stores a value under a name.

```
# Compound interest: V = P (1 + r/n)^(nt)
P = 2100   # principal
r = 0.034  # annual rate
n = 4      # compounding frequency
t = 7      # years

V = P * (1 + r/n)**(n*t)
>>> V
2661.6108980682593
```

- The same exponential-growth pattern shows up throughout energy engineering: compounding demand growth, battery capacity fade, and continuous-compounding models V = P e^(rt).
- Exercise: change n to 2 (semi-annual compounding) and predict whether V goes up or down before running it.

Strings: representing text data

- A string is text in quotes: 'Data' or "Science". The + operator concatenates strings rather than adding them.
- Numbers stored as strings behave surprisingly: '123' + 1 is a TypeError, but '123' * 3 gives '123123123'.
- Convert between types explicitly: int('123'), float('12.3'), str(123) — but int('12.3') fails because of the decimal point.
- Real datasets are full of numbers-as-strings — a sensor reading imported from CSV as text is a common source of silent bugs.

```
>>> first = 'Data'
>>> last = 'Science'
>>> first + ' ' + last
'Data Science'

>>> int('123')
123
>>> str(12.3)
'12.3'
```

Dates, times, and durations with pandas

- Dates/times read from files often arrive as plain strings — not useful for computation until converted.
- pandas.Timestamp represents a specific date and time; pd.Timestamp.now() gives the current moment.
- A Timestamp exposes .year, .month, .day, .day_name(), .month_name() directly.
- Subtracting two Timestamps gives a Timedelta, with .days, .components, and comparison operators (>, <, ==).
- Every hourly load reading, every weather observation, every maintenance log entry in this course will be timestamped — this is the toolkit for all of it.

```
import pandas as pd

dob = pd.Timestamp('May 11, 1967')
now = pd.Timestamp.now()
age = now - dob

>>> age.days
20993
>>> dob.day_name()
'Thursday'
```

Representing location: tuples and distance

- Latitude/longitude pairs are commonly bundled as a tuple: boston = (42.3601, -71.0589).
- A tuple is an ordered, fixed sequence of values of any type — unpack it with y, x = boston.
- def defines a reusable function; here, haversine_distance(coord1, coord2) computes great-circle distance between two lat-lon points.
- Direct energy use: distance from a weather station to a solar farm, or between substations, for spatial interpolation of weather inputs.

```
def haversine(theta):
    return np.sin(theta / 2) ** 2

boston = (42.3601, -71.0589)
london = (51.5074, -0.1278)

>>> haversine_distance(boston, london)
5265.66   # kilometers
```

Three ways to hold a sequence of values

Tuple	`(9.99, 7.99, 7.49)`	Immutable — cannot be changed after creation.
List	`[9.99, 7.99, 7.49]`	Mutable, general-purpose; + concatenates rather than adding elementwise.
NumPy array	`np.array([9.99, 7.99, 7.49])`	Same type throughout; arithmetic is elementwise — the workhorse for data.

boston_prices - london_prices fails on lists (TypeError) but works elementwise on arrays — this is why we use NumPy arrays for almost all numeric data work.

Statistical summaries with NumPy

- np.mean — central tendency; np.std — spread/variability; np.min / np.max — the range.
- These functions accept lists or arrays interchangeably, but arithmetic between two sequences requires arrays.
- The mean of differences equals the difference of means (a useful check, and it always holds for plain differences).

```
boston = np.array([9.99, 7.99, 7.49, 7.0,
6.29, 4.99])
london = np.array([7.5, 5, 4.4, 5, 3.75,
2.25])

diff = boston - london
>>> np.mean(diff)
2.6416666666666666
>>> np.std(boston), np.std(london)
(1.64, 1.72)
```

Absolute, relative, and percent differences

- Absolute difference: boston - london — same units as the data, but hides scale.
- Relative difference: divide by a reference value; percent difference multiplies that by 100.
- The choice of denominator changes the story: (9.99-7.5)/7.5 = 33% more in Boston, but (9.99-7.5)/9.99 = 25% cheaper in London — same two numbers.
- The mean of percent differences is NOT the same as the percent difference of the means — pick the summary deliberately and say which one you used.
- This exact ambiguity appears in every energy KPI report: “15% more efficient” always begs the question — 15% of what baseline?

The for loop, and counting

- for x in sequence: repeats a block once per element, creating (or reusing) the loop variable x each time.
- Increment/decrement shorthand: count += 1 reads count, adds 1, and reassigns it — same idea as count = count + 1.
- Loops are how we will process every row of a sensor log, every hour of a load profile, every reading in a weather file.

```
readings = (12.4, 13.1, 11.9)
count = 0
for x in readings:
    print(x)
    count += 1

>>> count
3
```

Reading files, if statements, and break

- open('file.txt') returns a file pointer; looping over it yields one line at a time — essential for large sensor/log files that don't fit conveniently in memory.
- if condition: runs an indented block only when the condition is True; indentation (4 spaces, never tabs) defines the block.
- break exits a loop immediately — e.g. stop reading a log file once you've seen enough lines, without scanning the whole file.
- line.split() breaks a line of text into a list of words on whitespace — the same idea used to parse delimited sensor-log lines.

```
fp = open('meter_log.txt')
count = 0
for line in fp:
    if line.startswith('#'):   # skip comments
        continue
    count += len(line.split())
    if count > 100_000:
        break
```

Dictionaries: key → value mappings

- A dictionary maps keys (almost any type) to values: d = {'one': 1, 'two': 2}.
- Look up with d[key]; test membership with key in d; loop over key-value pairs with for key, value in d.items().
- Each key is unique — reassigning an existing key updates its value rather than duplicating it.
- collections.Counter builds on this idea to rank the most common items: counter.most_common(10).
- Energy use case: counting how often each fault code, meter model, or outage cause appears in a maintenance log — exactly the word-frequency pattern shown here.

```
fault_counts = {}
for code in fault_log:
    if code in fault_counts:
        fault_counts[code] += 1
    else:
        fault_counts[code] = 1

import collections
top = collections.Counter(fault_counts)
>>> top.most_common(3)
[('OVERTEMP', 41), ('LOWVOLT', 27),
('COMMS', 15)]
```

What you should walk away with

- A statistical mindset: prefer designed data collection and honest summaries over anecdotes; keep interpretation grounded in context.
- The vocabulary of study design: population, sample, cross-sectional vs. longitudinal, representative sampling, oversampling.
- Fluency with Python's core building blocks: numbers, variables, strings, Timestamps, tuples/lists/arrays, loops, files, and dictionaries.
- The habit of debugging incrementally — write a little code, test it, then add more.

Coming up

Lab 1: hands-on Python + NumPy warm-up using energy datasets

Lecture 2: Distributions — histograms, PMFs, and describing a dataset's shape

READING & SOURCES

Required reading and sources for this lecture

- Allen B. Downey, Think Stats: Exploratory Data Analysis in Python, 2nd ed. — Chapter 1 (“Exploratory data analysis”). Free at greenteapress.com/thinkstats2
- Allen B. Downey, Elements of Data Science — Chapters 1–5 (“Variables and Values” through “Dictionaries”). Free at allendowney.github.io/ElementsOfDataScience
- Both texts are released under Creative Commons licenses (CC BY-NC-SA) — full attribution in the course syllabus.

Before Lab 1:

Install Python (Anaconda distribution recommended) or set up a free Google Colab account. Skim Chapters 1–5 of Elements of Data Science and try the exercises inline — we will build directly on them.