PRINCIPLES AND METHODS OF DATA ANALYSIS IN THE ENERGY INDUSTRY

Lecture 1 — Lecture Notes (Conspectus)

Exploratory Data Analysis & Python Foundations

Programme: Information Technologies for Sustainable Energy Engineering (Master's level)

Sources: A. B. Downey, *Think Stats: Exploratory Data Analysis in Python*, 2nd ed., Chapter 1 (greenteapress.com/thinkstats2) · A. B. Downey, *Elements of Data Science*, Chapters 1–5 (allendowney.github.io/ElementsOfDataScience). Both texts are distributed under CC BY-NC-SA.

Learning objectives

By the end of this lecture, students will be able to:

- Explain why anecdotal evidence is insufficient for engineering decisions, and describe the five steps of a statistical approach to a data question.
- Use the vocabulary of study design — population, sample, respondent, record, representative sample, oversampling, cross-sectional vs. longitudinal study — and map each term onto an energy-industry example.
- Explain, at a conceptual level, why data must be cleaned and validated before being trusted, and what a DataFrame/Series is.
- Write basic Python: numeric and string expressions, variables, function calls, tuples/lists/NumPy arrays, for-loops with conditionals, file reading, and dictionaries.
- Compute and correctly interpret absolute, relative, and percent differences between two datasets.

Part A — Thinking Like a Data Analyst

Based on: A. Downey, *Think Stats 2e*, Chapter 1, “Exploratory data analysis.”

A.1 Motivation: why energy engineering needs this course

Modern energy systems are instrumented at a scale that did not exist a generation ago: smart meters record consumption at 15-minute or hourly resolution for millions of customers; SCADA systems log turbine, inverter, and substation telemetry continuously; weather stations and satellite products provide irradiance and wind data at fine spatial and temporal resolution. None of this data is useful until it has been collected thoughtfully, cleaned honestly, explored before modeling, and interpreted with appropriate uncertainty.

Typical questions an energy engineer might face are structurally identical to the questions social scientists ask: Does rooftop solar output really degrade faster in humid climates? Did a retrofit reduce a building's energy use, or did the weather simply get milder that year? Is a rising vibration trend on a turbine bearing a genuine warning sign, or normal noise? Answering these responsibly requires exactly the discipline covered in this lecture.

A.2 The anecdotal-evidence trap

Downey opens Think Stats with a widely argued question: "do first babies tend to arrive late?" People on both sides support their view with personal stories such as:

> **Anecdotes from online discussion (Think Stats 2e)**
>
> *"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."*

Reports like these are called anecdotal evidence: unpublished, personal, and usually collected casually. There is nothing wrong with anecdotes in ordinary conversation, but they generally fail as proof, for four reasons:

- **Small number of observations.** If a real effect exists, it is likely small relative to natural variation — a handful of stories cannot reliably detect it.
- **Selection bias.** People who join a discussion about a topic are not a random sample of everyone affected by it; they may have joined precisely because their own experience was unusual.
- **Confirmation bias.** People who already believe a claim are more likely to contribute confirming examples; skeptics are more likely to cite counterexamples.
- **Inaccuracy.** Anecdotes are personal stories, frequently misremembered or exaggerated with each retelling.

The same pattern recurs constantly in engineering discussions — "our inverters always fail in year three," "panels never survive their first hot summer here" — and deserves the same scrutiny.

A.3 The statistical approach: five steps

To do better than anecdote, Downey proposes five components of a statistical approach to a question:

- **1. Data collection.** Use data from a study designed explicitly to support valid inference about the population of interest, not opportunistic contributions from whoever chose to comment.
- **2. Descriptive statistics.** Summarize the data concisely, and choose visualizations that reveal its structure rather than obscure it.
- **3. Exploratory data analysis (EDA).** Look for patterns, differences, and other features relevant to the question, while checking for inconsistencies and limitations in the data itself.
- **4. Estimation.** Use a sample to estimate characteristics of the broader population from which it was drawn.
- **5. Hypothesis testing.** Where an apparent effect is observed (e.g., a difference between two groups), evaluate whether it could plausibly have arisen by chance.

This course devotes roughly one lecture to each of these ideas in turn, beginning with EDA and descriptive statistics.

A.4 Study design vocabulary

Downey's case study uses data from the U.S. National Survey of Family Growth (NSFG), a study conducted by the CDC since 1973. Working with any survey or sensor dataset requires the same vocabulary of study design:

Term	Definition	Energy-industry analogue
Population	Everyone or everything the study aims to describe.	All residential customers connected to a utility's grid.
Sample	The subset of the population from which data is actually collected.	The 5,000 smart meters enrolled in a demand-response pilot.
Respondent / record	A single unit of observation.	One meter-reading, one turbine, one weather-station-hour.
Codebook	Documentation of a study's variables, encodings, and survey questions.	A sensor's datasheet and register map / units documentation.
Representative sample	A sample in which every population member has an equal chance of inclusion.	A meter sample that mirrors the grid's true mix of house sizes and heating types.
Oversampling	Deliberately increasing the share of a subgroup in the sample to ensure enough observations for valid inference about that subgroup.	Oversampling neighborhoods with high EV adoption to gather enough charging events to analyze.

A.5 Cross-sectional vs. longitudinal studies

Cross-sectional study: captures a snapshot of a population at a single point in time. The NSFG has been run seven times; each repetition is called a cycle — Think Stats uses Cycle 6 (Jan 2002 – Mar 2003). **Energy example:** *an annual energy-audit campaign across 200 buildings, conducted once.*

Longitudinal study: observes the same group repeatedly over time. **Energy example:** *a SCADA system logging the same set of turbines' vibration every minute, continuously, for years.*

Neither design is universally better — the choice depends on the question. Cross-sectional studies are cheaper and faster; longitudinal studies can detect trends and changes within the same units over time, which cross-sectional snapshots cannot.

A.6 DataFrames and Series — a first look

Think Stats reads the NSFG data into a pandas DataFrame — a table with one row per record (here, one pregnancy) and one column per variable. A single column pulled out of a DataFrame (e.g., df['pregordr']) is a pandas Series, which behaves much like a Python list but carries an index and additional functionality.

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

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

We will build and manipulate DataFrames properly starting around Lecture/Lab 7, once the underlying Python (variables, sequences, loops, dictionaries) is solid. For now, keep this picture in mind: every dataset analyzed this semester — load curves, weather logs, turbine telemetry — eventually takes this rows-and-columns shape.

A.7 Data cleaning and validation

Raw data almost always needs cleaning before analysis: handling special/missing codes, fixing units, and deriving new variables. In the NSFG data, ages are stored as integer "centiyears" and must be divided by 100; birth weight is split across pounds and ounces columns and must be combined. Special codes such as 97/98/99 ("not ascertained / refused / don't know") are dangerous if treated as ordinary numbers — Downey shows how failing to recode them can silently produce a "51-pound baby" in the data.

Validation means cross-checking computed summaries — e.g., value_counts() tallies — against a trusted reference such as a codebook or a calibration certificate, to catch import errors or misunderstandings before they propagate into an analysis.

> **Energy analogue**
>
> A sensor code of −1 or 9999 meaning "no reading" must be recoded to NaN before averaging, or it will silently corrupt a load curve. A reported capacity factor above 100% is a validation red flag, not a valid record.

A.8 Interpretation: statistics and context together

Working with data effectively requires thinking on two levels simultaneously: the statistical level and the human/contextual level. Downey illustrates this with a respondent whose sequence of pregnancy outcomes was six miscarriages followed by one live birth — statistically unremarkable, but a deeply significant personal history. Every dataset represents real people, equipment, or communities, and energy data is no exception: a spike in outage duration in a dataset is also a household without heat. Analysts have an obligation to use data competently and with respect for what — and whom — it represents.

Part B — Python Foundations

Based on: A. Downey, *Elements of Data Science*, Chapters 1–5.

B.1 Numbers: int and float (Ch. 1)

Python's two most common numeric types are int (integers, typically used for counts) and float (numbers with a fractional part, used for measurements). Floats can be written in scientific notation: 1.2345e3 means 1.2345×10^3.

```
>>> 3
3
>>> 3.14159
3.14159
>>> 1.2345e3
1234.5
```

Floating-point numbers are finite approximations: `0.1 + 0.1 + 0.1` evaluates to `0.30000000000000004`, not exactly 0.3. Never test floats for exact equality; this matters whenever sensor data is compared against thresholds.

B.2 Arithmetic operators and precedence (Ch. 1)

Operators: `+ -` (addition, subtraction), `* /` (multiplication, division), `**` (exponentiation). Precedence follows the familiar PEMDAS rule; parentheses override it. Python does **not** support implicit multiplication — writing `3 (2 + 1)` raises a `TypeError`, unlike math notation.

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

Exercise: write a Python expression that raises 1+2 to the power 3×4. Expected result: 531441.

B.3 Libraries and math functions: NumPy (Ch. 1)

Functions such as `sin`, `cos`, `exp`, and `log` are not built into core Python — they come from a **library**, a reusable collection of values and functions. NumPy ("Numerical Python") is the library used throughout this course; it is conventionally imported as `np`. Library names are case-sensitive.

```
import numpy as np

>>> np.pi
3.141592653589793
>>> np.log(100)     # natural log
4.605170185988092
>>> np.exp(1)       # e^1
2.718281828459045
```

B.4 Variables and an exponential-growth example (Ch. 1)

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

Downey works through the compound-interest formula V = P(1 + r/n)^(nt) as a worked example that exercises variables, exponentiation, and (later) logarithms together:

```
P = 2100    # principal
r = 0.034   # annual interest rate
n = 4       # compounding frequency
t = 7       # years
V = P * (1 + r/n)**(n*t)
>>> V
2661.6108980682593
```

The identical exponential-growth pattern recurs throughout energy engineering: demand-growth projections, battery capacity fade, and continuous-compounding models of the form V = P·e^(rt).

B.5 Strings (Ch. 2)

A **string** is text in quotes: `'Data'` or `"Science"`. The `+` operator concatenates strings rather than adding them. Numbers imported as strings behave surprisingly under arithmetic: `'123' + 1` raises a `TypeError`, while `'123' * 3` gives `'123123123'` (string repetition, not multiplication of the number).

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

Practical note: this is a common source of silent bugs — a numeric-looking column imported from a CSV file may actually be stored as strings and must be explicitly converted with `int()` or `float()` before arithmetic.

B.6 Dates, times, and durations (Ch. 2)

Dates and times read from files are typically plain strings until converted. pandas provides `Timestamp` to represent a specific date/time, with attributes `.year`, `.month`, `.day`, `.day_name()`, and `.month_name()`. Subtracting two `Timestamp` values yields a `Timedelta`, with a `.days` attribute and finer-grained `.components`.

```
import pandas as pd
dob = pd.Timestamp('May 11, 1967')
now = pd.Timestamp.now()
age = now - dob
>>> age.days
20993
>>> dob.day_name()
'Thursday'
```

Timestamps also support comparison operators (>, <, ==), which produce bool values (True / False) — useful, for example, for checking whether a maintenance date has passed. This toolkit underlies every timestamped dataset used later in the course: hourly load readings, weather observations, and maintenance logs.

B.7 Representing location: tuples and distance (Ch. 2)

Latitude/longitude pairs are commonly bundled as a **tuple**: `boston = (42.3601, -71.0589)`. A tuple is an ordered, fixed (immutable) sequence of values; it can be unpacked with `y, x = boston`. The `def` keyword defines a reusable function — for example, `haversine_distance(coord1, coord2)` computes the great-circle distance between two lat-lon points.

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

boston = (42.3601, -71.0589)
london = (51.5074, -0.1278)
>>> haversine_distance(boston, london)
5265.656325981015   # kilometers, error < 1%
```

Energy application: computing the distance from a weather station to a solar or wind installation, or between substations, for spatial interpolation of weather inputs — precisely the pattern shown here.

B.8 Tuples, lists, and NumPy arrays (Ch. 3)

Python offers three closely related ways to represent a sequence of values:

Type	Example	Key property
Tuple	(9.99, 7.99, 7.49)	Immutable — cannot be modified after creation.
List	[9.99, 7.99, 7.49]	Mutable, general-purpose; the + operator concatenates rather than adding elementwise.
NumPy array	np.array([9.99, 7.99, 7.49])	All elements share one dtype; arithmetic is elementwise — the standard structure for numeric data work.

Downey illustrates this with sandwich-price data from Boston and London (from a 2019 *Economist* article). Subtracting two lists raises a `TypeError`; subtracting two NumPy arrays works elementwise and returns a new array — this is the key reason NumPy arrays, not lists, are the default for numeric data throughout this course.

```
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])
>>> boston - london
array([2.49, 2.99, 3.09, 2.  , 2.54, 2.74])
```

B.9 Statistical summaries (Ch. 3)

NumPy provides summary functions that work on both lists and arrays: `np.mean` (central tendency), `np.std` (spread), and `np.min` / `np.max` (range). A useful identity: the mean of the (plain) differences between two arrays equals the difference of their means.

```
>>> np.mean(boston - london)
2.6416666666666666
>>> np.mean(boston) - np.mean(london)
2.6416666666666675
```

B.10 Absolute, relative, and percent differences (Ch. 3)

Three ways to compare two numbers a and b:

- **Absolute difference:** a − b — same units as the data, but hides the scale of the comparison.
- **Relative difference:** (a − b) / reference — a dimensionless fraction; the choice of reference matters.
- **Percent difference:** the relative difference × 100.

The choice of denominator changes the headline: (9.99−7.5)/7.5 = 33% "more expensive in Boston," but (9.99−7.5)/9.99 = 25% "cheaper in London" — the same two prices, two different-sounding claims. Downey also shows that the **mean of percent differences is not the same as the percent difference of the means** — the two summaries answer subtly different questions, and the analyst must choose and disclose which one is being reported.

> **Why this matters for energy reporting**
>
> Every efficiency or performance claim of the form "15% more efficient" or "20% lower losses" implicitly depends on a baseline. Always identify the denominator before accepting — or publishing — such a figure.

B.11 Loops and counting (Ch. 4)

`for x in sequence:` repeats an indented block once per element of `sequence`, creating (or reusing) the loop variable `x` each time. Incrementing a counter, `count += 1`, is shorthand for `count = count + 1`.

```
readings = (12.4, 13.1, 11.9)
count = 0
for x in readings:
    print(x)
    count += 1
>>> count
3
```

Loops are the mechanism by which we will process every row of a sensor log, every hour of a load profile, and every reading in a weather file.

B.12 Files and conditionals (Ch. 4)

`open('file.txt')` returns a file pointer; iterating over it with a `for` loop yields one line at a time — essential for large log files that should not be loaded entirely into memory. `if condition:` runs its indented block only when the condition evaluates to `True`; indentation (four spaces, never tabs) defines the block. `break` exits a loop immediately. `line.split()` splits a line of text into a list of words on whitespace.

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

This loop-and-counting pattern is the standard way to process a large text or log file line by line in Python — a useful, low-stakes way to practice before applying the same structure to a real sensor log or fault-code file.

B.13 Dictionaries (Ch. 5)

A **dictionary** maps keys (almost any type) to values: `d = {'one': 1, 'two': 2}`. Look up a value with `d[key]`, test membership with `key in d`, and iterate key-value pairs with `for key, value in d.items():`. Each key is unique — assigning to an existing key updates its value rather than duplicating the entry.

```
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)]
```

`collections.Counter` extends this pattern with `.most_common(n)`, directly useful for ranking the most frequent fault codes, meter models, or outage causes in a maintenance log — the energy-domain analogue of Downey's word-frequency example.

Summary and preparation for the next session

Key terms introduced in this lecture

anecdotal evidence · population · sample · respondent / record · codebook · representative sample · oversampling · cross-sectional study · longitudinal study · cycle · DataFrame · Series · recode · data cleaning · validation · int · float · variable · library · tuple · list · NumPy array · elementwise · absolute / relative / percent difference · for loop · loop variable · file pointer · if statement · break · dictionary · key · value · Counter

What you should be able to do after this lecture

- Critique an engineering claim that rests only on anecdotal evidence, and name which of the four failure modes applies.

- Describe a dataset using the vocabulary of study design (population, sample, representativeness).
- Write short Python snippets using numbers, variables, strings, dates, tuples/lists/arrays, loops, file reads, conditionals, and dictionaries.
- Correctly compute and interpret an absolute, relative, and percent difference between two measurements — and state which denominator was used.

Preparation for Lab 1

Install Python via the Anaconda distribution, or set up a free Google Colab account. Skim Chapters 1–5 of Elements of Data Science and attempt the inline exercises — Lab 1 builds directly on them, using small energy-flavored datasets in place of the book's own examples.

Preview of Lecture 2

Distributions — histograms, probability mass functions (PMFs), and ways to describe and compare the shape of a dataset (Think Stats 2e, Ch. 2; Elements of Data Science, Ch. 6–8).

References

- Downey, A. B. *Think Stats: Exploratory Data Analysis in Python*, 2nd ed. Green Tea Press. https://greenteapress.com/thinkstats2/html/index.html
- Downey, A. B. *Elements of Data Science*. https://allendowney.github.io/ElementsOfDataScience/