PRINCIPLES AND METHODS OF DATA ANALYSIS IN THE ENERGY INDUSTRY

Lecture 2 — Lecture Notes (Conspectus)

Plotting, DataFrames, and Distributions

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

Sources: A. B. Downey, *Think Stats: Exploratory Data Analysis in Python*, 2nd ed., Chapter 2 (greenteapress.com/thinkstats2) · A. B. Downey, *Elements of Data Science*, Chapters 6–7 (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:

- Produce a labeled, legible line or marker plot with Matplotlib/Pyplot, including axis limits, a legend, and (where useful) logarithmic scales.
- Read real tabular data into a pandas DataFrame, validate it with value_counts() and describe(), and clean special/missing codes.
- Use Boolean Series to filter data, and compute weighted means to correct for oversampling.
- Explain what a distribution is, summarize it with mean/variance/standard deviation, and quantify a difference between groups with Cohen's d.

Logical order for this lecture: we learn to plot, and to hold real data in a DataFrame, before returning to distributions — so histograms are introduced as things we can immediately compute and draw, not abstract formulas. PMFs, CDFs, and modeling distributions follow in Lecture 3.

Part A — Visualizing Data with Pyplot

Based on: A. Downey, *Elements of Data Science*, Chapter 6, “Plotting.”

A.1 A prerequisite: keyword arguments

Most of the plotting functions in this lecture rely on **keyword arguments** — arguments identified by name rather than position. Compare `np.power(10, 6)` (a **positional** argument: 10 to the 6th power, not the reverse) with `int('21', base=8)`, where `base=8` is a keyword argument. Keyword syntax looks like assignment but does not create a variable — the parameter name is fixed by the function itself, and an unrecognized keyword raises a `TypeError`.

```
>>> int('21', base=8)
17

>>> int('123', bass=11)
TypeError: 'bass' is an invalid keyword argument for int()
```

A.2 Line plots and misleading axes

matplotlib.pyplot (imported as `plt`) is Python's core plotting library. `plt.plot(x, y)` draws a line through paired x/y values; calling it more than once in the same cell overlays multiple lines on one set of axes.

Pyplot chooses axis ranges automatically, and this can mislead. If two lines are compared but the y-axis doesn't start at zero, the visual gap between them can look larger than it really is. `plt.ylim(lower, upper)` fixes this by setting explicit bounds.

```
import matplotlib.pyplot as plt
plt.plot(year, christian)
plt.plot(year, unaffiliated)
plt.ylim(0, 80)   # force the axis to start at zero
```

Energy application: the identical pattern plots any time series — monthly capacity factor, daily peak demand, renewable-energy share over years — and the same axis-scaling caution applies whenever two series are compared visually.

A.3 Decorating axes: labels, title, legend

`plt.xlabel(...)`, `plt.ylabel(...)`, and `plt.title(...)` each take a single string. To add a legend, first give each line a `label=...` keyword inside `plt.plot()`, then call `plt.legend()` once. An unlabeled, untitled chart is not a finished deliverable — every figure shared with a colleague or included in a report needs units on its axes and a legend identifying each series.

```
plt.plot(year, christian, label='Christian')
plt.plot(year, unaffiliated, label='Unaffiliated')
plt.ylim(0, 80)
plt.xlabel('Year'); plt.ylabel('% of adults')
plt.title('Religious affiliation of U.S. adults')
plt.legend();
```

A.4 Markers for categorical data

When the values on an axis are discrete categories — sandwich names, equipment IDs, turbine serial numbers — connecting them with a line implies intermediate values that don't exist. The fix is to remove the line and plot markers instead, using linestyle='' with marker='o', or the shorthand format-string form 'o' as a positional argument. plt.hlines(labels, x1, x2) draws a horizontal connector between two values per category, useful for before/after or two-city comparisons.

```
plt.hlines(name_list, london_price_list, boston_price_list, color='gray')
plt.plot(boston_price_list, name_list, 'o', color='C3', label='Boston')
plt.plot(london_price_list, name_list, 's', color='C0', label='London')
plt.xlabel('Price in USD'); plt.legend();
```

Energy analogue

Comparing per-unit maintenance cost or downtime across turbines or substations: one marker per site, optionally connected across two time periods with `hlines` — never a plain line through unrelated categories.

A.5 Zipf's Law and logarithmic scales

Zipf's law: in almost any body of text, the frequency of a word is roughly inversely proportional to its rank — f = k / r. Taking logarithms of both sides gives log f = log k − log r, which implies that plotting frequency against rank on a **log-log** scale should produce a straight line with slope −1. `plt.xscale('log')` and `plt.yscale('log')` switch either axis to a logarithmic scale — essential whenever data spans several orders of magnitude.

```
plt.plot(ranks, freq_list)
plt.xscale('log'); plt.yscale('log')

rise = np.diff(np.log10(ys))   # difference on the log y-axis
run  = np.diff(np.log10(xs))   # difference on the log x-axis
>>> rise / run                 # slope of the line — close to -1?
```

Energy application: outage-size distributions, wind-speed extremes, and other heavy-tailed quantities are often plotted on log-log or log-probability axes for exactly this reason — a straight line reveals a power-law-like relationship that a linear-scale plot would compress into an unreadable spike near zero.

Part B — DataFrames and Series

Based on: A. Downey, *Elements of Data Science*, Chapter 7, “DataFrames and Series.”

B.1 Reading real data into a DataFrame

A `DataFrame` is pandas' primary structure: one row per record, one column per variable — exactly the shape previewed in Lecture 1. Government and survey data is often distributed in **fixed-width format**, where every row has the same length and each column occupies a fixed range of characters, together with a separate **data dictionary** specifying column names and positions. `pd.read_fwf()` reads such a file directly once the dictionary has been parsed (here, with the `statadict` library).

```
from statadict import parse_stata_dict
stata_dict = parse_stata_dict(dict_file)

nsfg = pd.read_fwf(data_file, names=stata_dict.names,
colspecs=stata_dict.colspecs)
>>> nsfg.shape
(9553, 248)
```

`DataFrame` attributes worth knowing immediately: `.shape` (rows, columns), `.columns` (variable names), and the method `.head()` (first five rows, for a quick look). **Reading the codebook carefully matters:** misinterpreting a column silently produces nonsense results that are easy to miss.

Energy application: reading a SCADA export or a utility's fixed-width interval-data file with `pd.read_fwf()` or the more common `pd.read_csv()` follows exactly this pattern.

B.2 Series, and a first validation pass

A DataFrame behaves like a dictionary of columns: `df['col']` returns a `Series` — a single column with an index, carrying forward the Series concept previewed in Lecture 1. `NaN` marks missing or inapplicable data (e.g., birth weight is `NaN` when a pregnancy did not end in live birth).

Validation, step 1: `value_counts(dropna=False).sort_index()` lists every value in a Series and its count, sorted from lowest to highest — compare this directly against the codebook (or a sensor's documented value range) before trusting the data. Special codes such as 98 ("refused") and 99 ("don't know") look like ordinary numbers until checked this way — the same caution from Lecture 1's data-cleaning discussion.

```
pounds = nsfg['BIRTHWGT_LB1']
pounds.value_counts(dropna=False).sort_index()
# ... 8.0  1287   9.0  396  ...  98.0  2   99.0  89   NaN  2863
```

B.3 describe(), and cleaning special codes

`Series.describe()` gives count, mean, standard deviation, minimum, the 25th/50th/75th percentiles, and maximum in one call — a fast second validation pass. Before cleaning, `pounds.describe()` reports a mean of about 8.05 lb — inflated by the 98/99 codes hiding among real weights. `Series.replace([98, 99], np.nan)` swaps the special codes for `NaN`, exactly as in Lecture 1's data-cleaning discussion.

```
pounds.describe()                          # mean 8.008819  std 10.771360  max
99.0
pounds_clean = pounds.replace([98, 99], np.nan)
pounds_clean.describe()                    # mean 6.754357  std  1.383268  max
14.0
```

A few bad values distort every downstream statistic: removing the special codes drops the mean from 8.05 to 6.75 lb and the standard deviation from 10.8 to 1.4.

B.4 Series arithmetic and a first histogram

Series arithmetic is elementwise, exactly like NumPy arrays: `pounds * 16 + ounces` converts pounds to ounces and adds the ounces column in a single expression, combining two columns into one derived quantity. A **distribution** is the set of values in a dataset together with their frequencies — visualized directly from a cleaned Series with `.hist(bins=30)`.

```
birth_weight = pounds_clean + ounces_clean / 16
birth_weight.hist(bins=30)
plt.xlabel('Birth weight (lb)'); plt.ylabel('Number of live births')
```

The birth-weight histogram is roughly bell-shaped but skewed left — more light (preterm) babies than heavy ones. The same asymmetric-tail pattern recurs in many energy quantities: time-to-failure and outage-duration distributions are also typically skewed, with a long tail of rare, extreme values.

B.5 Boolean Series, logical operators, and filtering

Comparing a Series to a value produces a **Boolean Series**: `preterm = (nsfg['PRGLNGTH'] < 37)` is `True`/`False` for every row. `.sum()` on a Boolean Series counts the `True` values; `.mean()` gives the **fraction** that are `True`.

Combine conditions with `&` (AND), `|` (OR), and `~` (NOT) — not Python's `and`/`or`/`not`, which do not work elementwise on a Series. **Caution:** these operators treat `NaN` the same as `False`, so `~condition` can silently include rows with unknown status as if they had passed the condition.

```
preterm = (nsfg['PRGLNGTH'] < 37)
live = (nsfg['OUTCOME'] == 1)
>>> (live & preterm).mean()
0.0893   # ~9% of ALL pregnancies were live, preterm births

preterm_weight = birth_weight[preterm]       # filter with the Boolean Series
itself
>>> preterm_weight.mean()
5.480958781362007
```

Energy application: `turbine_output[icing_flag]` vs. `turbine_output[~icing_flag]` isolates readings under a specific operating condition — the same bracket-indexing idiom used throughout.

B.6 Weighted means: correcting for oversampling

Recall from Lecture 1 that the NSFG deliberately oversamples certain groups. Each record carries a sampling weight (here, WGT2015_2017) indicating how many population members it represents. A weighted mean multiplies each value by its weight, sums the products, and divides by the sum of the weights — rather than treating every row as equally representative. isna() and notna() build a validity mask that can be combined with other Boolean Series before weighting.

```
valid = birth_weight.notna()
selected = valid & live & single & fullterm
weighted_mean = (birth_weight[selected] * sampling_weight[selected]).sum() /
sampling_weight[selected].sum()
```

In this dataset the weighted mean birth weight comes out slightly higher than the unweighted mean, because the oversampled groups tend to have lighter babies on average — a concrete illustration of why oversampling (introduced conceptually in Lecture 1) must be corrected for before reporting a population-level statistic.

B.7 Saving a cleaned extract

Large fixed-width files are slow to re-parse every session. Once cleaned, select only the needed columns into a smaller DataFrame and save it in an efficient binary format: `df.to_hdf(filename, key=..., complevel=...)` writes a compressed HDF5 file; `pd.read_hdf()` reads it back far faster than reparsing raw text. The everyday equivalent for this course is `df.to_csv()` or `df.to_parquet()` after cleaning a large sensor export, so lab work starts from a small, trustworthy file instead of repeating the cleaning step every time.

Part C — Distributions

Based on: A. Downey, *Think Stats 2e*, Chapter 2, "Distributions," and A. Downey, *Elements of Data Science*, Chapter 8, "Distributions."

C.1 What is a distribution?

The **distribution** of a variable is the set of values that appear in a dataset and how often each one appears. A **histogram** is the most common representation: a mapping from value to frequency. In plain Python, a histogram is simply a dictionary built with a loop — the same pattern introduced in Lecture 1's dictionaries chapter: `hist[x] = hist.get(x, 0) + 1`. Think Stats' `Hist` class wraps this idea with convenience methods: `Freq(value)`, `Values()`, and `Items()` for iterating value-frequency pairs.

```
hist = {}
for x in t:
    hist[x] = hist.get(x, 0) + 1

>>> h = thinkstats2.Hist([1, 2, 2, 3, 5])
>>> h.Freq(2)
2
```

C.2 Reading a histogram's shape — and its outliers

The **mode** is the most frequent value: NSFG birth weight peaks at 7 lb; pregnancy length peaks sharply at 39 weeks. A distribution can be roughly bell-shaped (resembling the **normal**, or Gaussian, distribution) yet still **asymmetric** — birth weight's left tail (light babies) extends farther than its right tail.

Outliers are extreme values that may be measurement/recording errors or genuine rare events. `Hist.Smallest(n)` / `Largest(n)` surface them for inspection before any further analysis. In the NSFG, pregnancy lengths below 10 weeks are almost certainly coding errors, while a reported 50-week pregnancy is "medically unlikely" and needs domain judgment, not blind trust — deciding how to handle an outlier depends on domain knowledge about where the data came from and what the analysis is for.

> **Energy analogue**
>
> A turbine-RPM histogram's mode tells you normal operating speed; `Smallest`/`Largest` surface sensor glitches or genuine extreme events before any statistic is computed from the data.

C.3 Why histograms mislead when comparing groups

Overlaying two histograms — e.g., pregnancy length for first babies vs. others — makes the most frequent values obvious, but the comparison is distorted whenever the two groups have different sample sizes: there are fewer "first babies" than "others" in the NSFG, so raw-count bars aren't directly comparable bar-for-bar. The fix — normalizing counts into probabilities, a PMF — is the starting point of Lecture 3.

Energy application: comparing a fault-count histogram for 50 old turbines against 200 new ones needs normalizing by fleet size before any conclusion about relative reliability is valid.

C.4 Summarizing a distribution: mean, variance, standard deviation

A histogram is a complete description of a sample, but often a few summary numbers are more useful. Characteristics worth reporting include central tendency, number of modes, spread, tail behavior, and outliers.

Mean is not the same word as average: "mean" refers specifically to the sum of values divided by their count; "average" is any summary statistic chosen to describe central tendency. The mean can mislead when no single value is typical — Downey's example: three 1-lb decorative pumpkins, two 3-lb pie pumpkins, and one 591-lb giant pumpkin have a mean of 100 lb, which describes no actual pumpkin in the garden.

Variance is the mean squared deviation from the mean; **standard deviation** is its square root, expressed in the original units (so it is easier to interpret than variance's "square weeks" or "square kilowatts").

```
mean = live.prglngth.mean()   # 38.6 weeks
std  = live.prglngth.std()    # 2.7 weeks
var  = live.prglngth.var()    # 7.3 week²
```

A useful rule of thumb from this result: deviations of 2-3 weeks from the mean pregnancy length are common, not surprising — knowing the standard deviation calibrates what counts as an unusual observation.

C.5 Effect size: is a difference big enough to matter?

An **effect size** summarizes the size of a difference between groups. The raw difference in mean pregnancy length between first babies and others is 38.601 vs. 38.523 weeks — just 0.078 weeks, about 13 hours.

Cohen's d expresses that raw difference in units of the pooled standard deviation: $d = (\bar{x}_1 - \bar{x}_2) / s$. In this example $d \approx 0.029$ — tiny. For comparison, the height difference between men and women is about $d \approx 1.7$.

```
def cohen_effect_size(group1, group2):
    diff = group1.mean() - group2.mean()
    n1, n2 = len(group1), len(group2)
    pooled_var = (n1 * group1.var() + n2 * group2.var()) / (n1 + n2)
    return diff / math.sqrt(pooled_var)
```

Effect size matters more than statistical significance alone: a difference can be "real" (i.e., not due to chance) yet practically irrelevant. Always ask whether a result is clinically or operationally significant, not just whether it is non-zero.

C.6 Reporting results responsibly

The same underlying result can be reported many honest ways, and the right choice depends on the audience: a scientist wants to know about any real effect, however small; a doctor cares about effects large enough to change a treatment decision; a patient wants results relevant to their own situation. Your goals shape which summary statistics you lead with, but the choice

must remain honest — it is acceptable to be persuasive; it is not acceptable to be misleading. Every result should acknowledge its uncertainty and limitations alongside the headline number, the same standard expected in every lab report this semester and in professional energy-sector performance reporting.

Summary and preparation for the next session

Key terms introduced in this lecture

keyword argument · positional argument · Pyplot · plt.plot / xlabel / ylabel / title / legend / ylim · format string · log-log scale · Zipf's law · fixed-width format · data dictionary · DataFrame attributes (shape, columns, head) · Series · value_counts · describe · special code cleaning · Series arithmetic · Boolean Series · logical operators (&, |, ~) · filtering · sampling weight · weighted mean · to_hdf/read_hdf · distribution · histogram · Hist · mode · outlier · central tendency · variance · standard deviation · effect size · Cohen's d

What you should be able to do after this lecture

- Produce a Pyplot figure with correctly scaled, labeled axes, a legend, and markers where the data is categorical.
- Read a real dataset into a DataFrame, validate it, clean special codes, and filter it with Boolean Series.
- Compute a weighted mean and explain why oversampled survey data requires one.
- Read a histogram's shape, identify its mode and outliers, and explain why raw counts can mislead when comparing groups of different sizes.
- Summarize a distribution with mean/std and quantify a between-group difference with Cohen's d.

Preparation for Lab 2

Make sure `matplotlib` and `pandas` are installed. Skim Chapters 6–7 of *Elements of Data Science* and attempt the inline exercises — Lab 2 builds directly on them, using an energy dataset in place of the NSFG examples.

Preview of Lecture 3

Probability mass, cumulative, and density functions — normalizing histograms into PMFs, reading and comparing CDFs, modeling a distribution with a theoretical curve, and kernel density estimation (Think Stats 2e, Ch. 3–6; Elements of Data Science, Ch. 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/