MASTER'S COURSE · LECTURE 2

Principles and Methods of Data Analysis in the Energy Industry

Lecture 2 — Plotting, DataFrames, and Distributions

Information Technologies for Sustainable Energy Engineering

Sources: A. Downey, *Think Stats 2e*, Ch. 2 · A. Downey, *Elements of Data Science*, Ch. 6–7

ROADMAP

What we'll cover today

1

Part A

Visualizing data with Pyplot

Keyword arguments, line plots, decorating axes, markers for categorical data, Zipf's law, and logarithmic scales.

2

Part B

DataFrames and Series

Reading real data into pandas, validating it, computing summaries, Series arithmetic, Boolean filtering, and weighted means.

3

Part C

Distributions

Histograms, reading their shape, why raw counts mislead when comparing groups, central tendency and spread, and effect size.

Logical order: we learn to plot and to hold real data in a DataFrame first — then histograms make immediate sense as things we can compute and draw. PMFs, CDFs, and modeling follow in Lecture 3.

PART A

Visualizing Data with Pyplot

Elements of Data Science, Chapter 6 — Plotting

A new feature first: keyword arguments

- Positional arguments are matched to a function's parameters by order: np.power(10, 6) is 10 to the 6th power, not the reverse.
- Keyword arguments are matched by name: int('21', base=8) tells int() explicitly which parameter gets 8.
- Keyword syntax looks like assignment, but it does not create a variable — the name is fixed by the function's own definition, so an unrecognized keyword raises a TypeError.
- Almost every plotting function we use today (plt.plot, .hist, .legend) leans on keyword arguments for labels, colors, markers, and bins.

```
>>> np.power(10, 6)
1000000

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

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

Line plots with plt.plot

- matplotlib.pyplot (imported as plt) is Python's core plotting library.
- plt.plot(x, y) draws a line through paired x/y values; call it twice in one cell to overlay two lines on the same axes.
- Pyplot auto-scales the y-axis — which can mislead. plt.ylim(0, 80) forces the axis to start at zero, so two lines aren't visually exaggerated.
- The identical pattern plots any energy time series: monthly capacity factor, daily peak demand, or renewable-share trends over years.

```
import matplotlib.pyplot as plt

year = [2009, ..., 2018]
christian = [77, 76, ..., 65]
unaffiliated = [17, 17, ..., 26]

plt.plot(year, christian)
plt.plot(year, unaffiliated)
plt.ylim(0, 80);
```

Textbook example: Pew Research data on U.S. religious affiliation, 2009–2018 — chosen because it needs exactly this two-line, mislabeled-axis fix.

Decorating axes: labels, title, legend

- plt.xlabel(...), plt.ylabel(...), and plt.title(...) each take a single string.
- A legend needs a label on each line first — pass label='...' inside plt.plot — then call plt.legend() once.
- An unlabeled, untitled chart is not a finished result: every figure handed to a colleague or included in a report needs axis units and a legend.
- For energy dashboards, this is the difference between a chart that looks nice and one a shift engineer can actually read at 3 a.m.

```
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')
plt.legend();
```

Markers, format strings, and categorical axes

- When categories (like sandwich names — or equipment IDs) sit on an axis, connecting them with lines implies intermediate values that don't exist. Use markers instead.
- linestyle='' with marker='o' removes the line; a format string like 'o' or 's' is shorthand for the same thing 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.
- Direct energy use: comparing per-unit maintenance cost or downtime across turbines or substations, one dot per site rather than a misleading connected line.

```
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();
```

Zipf's Law and logarithmic scales

- Zipf's law: in almost any body of text, word frequency is roughly inversely proportional to its rank — f = k / r.
- Taking logs of both sides gives log f = log k − log r: a straight line with slope −1 on a log-log plot.
- plt.xscale('log') and plt.yscale('log') switch either axis to a logarithmic scale — essential whenever data spans orders of magnitude.
- Energy parallel: outage-size distributions, earthquake magnitudes, and wind-speed extremes are also often plotted log-log or on log-log-like axes (Weibull probability paper) because they span huge ranges.

```
plt.plot(ranks, freq_list)
plt.xlabel('Rank');
plt.ylabel('Frequency')
plt.xscale('log'); plt.yscale('log')

rise = np.diff(np.log10(ys))
run  = np.diff(np.log10(xs))
>>> rise / run   # slope ~ -1 ?
```

PART B

DataFrames and Series

Elements of Data Science, Chapter 7 — pandas fundamentals

Reading real data into a DataFrame

- A DataFrame is pandas' primary structure: rows are records, columns are variables — exactly the shape previewed in Lecture 1.
- Real government/survey data often arrives fixed-width, with a separate data dictionary specifying column names and character positions — pd.read_fwf() reads it directly.
- df.shape returns (rows, columns); df.columns lists variable names; df.head() previews the first five rows.
- df.head() shows a pregnancy-file NSFG extract: pounds a respondent's first three rows share one CASEID, meaning three pregnancies from the same person.

```
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)
```

Energy analogue: reading a SCADA export or a utility's fixed-width interval-data file with pd.read_fwf() or pd.read_csv() the same way.

Selecting a Series, then validating it

- A DataFrame behaves like a dictionary of columns: df['col'] returns a Series — one column, with an index.
- NaN marks missing/inapplicable data — e.g., birth weight is NaN when the pregnancy didn't end in live birth.
- Validation step 1: value_counts(dropna=False).sort_index() lists every value and its count, low to high — compare this against the codebook / sensor spec before trusting the data.
- Special codes (98 = refused, 99 = don't know) show up as ordinary-looking numbers until you check — exactly like Lecture 1's sensor error codes.

```
pounds = nsfg['BIRTHWGT_LB1']
>>> pounds.head()
0    7.0
1    NaN
2    9.0

pounds.value_counts(dropna=False)
        .sort_index()
# ... 98.0  2   99.0  89   NaN  2863
```

describe(), and cleaning special codes

- Series.describe() gives count, mean, std, min, 25/50/75th percentiles, and max in one call — a fast second validation pass.
- Before cleaning, mean = 8.05 lb — inflated by the 98/99 'refused/don't know' 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.
- After cleaning, mean drops to 6.75 lb and std drops from 10.8 to 1.4 — a dramatic reminder that a few bad values can distort every downstream statistic.

```
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
```

Series arithmetic, then a first histogram

- Series arithmetic is elementwise, just like NumPy arrays: pounds * 16 + ounces converts and combines two columns in one line.
- A distribution is the set of values in a dataset and their frequencies. Series.hist(bins=30) plots one directly from a cleaned Series.
- The birth-weight histogram is roughly bell-shaped but skewed left — more light (preterm) babies than heavy ones. The same asymmetric-tail pattern shows up in many energy quantities: outage durations, time-to-failure.

```
birth_weight = pounds_clean +
ounces_clean/16

birth_weight.hist(bins=30)
plt.xlabel('Birth weight (lb)')
plt.ylabel('Number of live births')
plt.title('Distribution of birth weight');
```

Boolean Series and logical operators

- Comparing a Series to a value produces a Boolean Series — True/False for every row: preterm = (nsfg['PRGLNGTH'] < 37).
- .sum() on a Boolean Series counts the True values; .mean() gives the fraction that are True (here, about 38% of all pregnancy outcomes).
- Combine conditions with & (AND), | (OR), ~ (NOT) — not the Python keywords and/or/not, which don't work elementwise on a Series.
- Caution: NaN behaves like False under these operators, so ~condition can silently include 'unknown' rows as if they passed.

```
preterm = (nsfg['PRGLNGTH'] < 37)
live = (nsfg['OUTCOME'] == 1)

>>> preterm.mean()
0.3847
>>> (live & preterm).mean()
0.0893   # ~9% of ALL pregnancies
```

Filtering data with a Boolean Series

- A Boolean Series can index another Series: birth_weight[preterm] keeps only the rows where preterm is True.
- This is the standard pandas idiom for 'give me the subset that meets a condition' — no explicit loop required.
- Full-term babies (7.43 lb) are heavier on average than preterm babies (5.48 lb) — exactly the kind of grouped comparison we'll formalize later with effect size.

```
preterm_weight = birth_weight[preterm]
>>> preterm_weight.mean()
5.480958781362007

fullterm = (nsfg['PRGLNGTH'] >= 37)
full_term_weight = birth_weight[fullterm]
>>> full_term_weight.mean()
7.429609416096791
```

Energy analogue: turbine_output[icing_flag] vs. turbine_output[~icing_flag] — isolate readings under a specific operating condition.

Weighted means: correcting for oversampling

- Recall Lecture 1: the NSFG deliberately oversamples some groups. Each record carries a sampling weight (here, WGT2015_2017) representing how many population members it stands for.
- A weighted mean multiplies each value by its weight, sums, and divides by the sum of weights — rather than treating every row equally.
- isna() / notna() build a validity mask; combine it with your other Boolean Series (live & single & fullterm) before weighting.
- The weighted mean birth weight comes out slightly higher than the unweighted one — the oversampled groups tend to have lighter babies, so ignoring weights biases the estimate.

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

Saving a cleaned extract

- Large fixed-width files are slow to re-read every session. Select only the columns you need into a smaller DataFrame, then save it in an efficient binary format.
- df.to_hdf(filename, key=..., complevel=...) writes a compressed HDF5 file; pd.read_hdf() reads it back — much faster than re-parsing raw text.
- The everyday equivalent for this course: df.to_csv() or df.to_parquet() after cleaning a large sensor export, so Lab work starts from a small, trustworthy file instead of redoing cleaning every time.

```
variables = ['CASEID', 'OUTCOME',
  'BIRTHWGT_LB1', 'PRGLNGTH', ...]
subset = nsfg[variables]

subset.to_hdf('nsfg.hdf', key='nsfg',
              complevel=6)

read_back = pd.read_hdf('nsfg.hdf',
                        key='nsfg')
```

PART C

Distributions

Think Stats 2e, Chapter 2 — Histograms and summary statistics

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 (count of occurrences).
- In plain Python, a histogram is just a dictionary built with a loop — exactly the pattern from 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

>>> import thinkstats2
>>> h = thinkstats2.Hist([1,2,2,3,5])
>>> h.Freq(2)
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 (normal/Gaussian) yet asymmetric — birth weight's left tail (light babies) extends farther than its right tail.
- Outliers are extreme values that may be errors or genuine rare events. Hist.Smallest(n) / Largest(n) surface them for inspection.
- Pregnancy lengths below 10 weeks are almost certainly coding errors; a reported 50-week pregnancy is 'medically unlikely' and needs domain judgment, not blind trust.
- Energy translation: a turbine-RPM histogram's mode tells you normal operating speed; Smallest/Largest surface sensor glitches or genuine extreme events before you compute anything else.

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 comparison is distorted whenever the two groups have different sample sizes.
- There are fewer 'first babies' than 'others' in the data, 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.
- Same issue in energy reporting: comparing a fault-count histogram for 50 old turbines against 200 new ones needs normalizing by fleet size before any conclusion about reliability is valid.

Summarizing a distribution: mean, variance, std

- A histogram is a complete description of a sample — but often we want a few numbers instead. Summary statistics answer: central tendency, modes, spread, tails, outliers.
- Mean ≠ average: 'mean' is one specific formula; 'average' is any summary statistic chosen to describe central tendency. The mean can mislead when a distribution isn't well described by one typical value (Downey's pumpkin-weight example).
- Variance is the mean squared deviation from the mean; standard deviation is its square root, in the original units — easier to interpret than 'square weeks'.

```
mean = live.prglngth.mean()   # 38.6 weeks
std  = live.prglngth.std()    # 2.7 weeks
var  = live.prglngth.var()    # 7.3 wk²
```

A useful rule of thumb from this result: deviations of 2–3 weeks from the mean pregnancy length are common, not surprising.

Effect size: is a difference big enough to matter?

- An effect size summarizes the size of a difference between groups — e.g., the difference in mean pregnancy length: 38.601 vs. 38.523 weeks, just 0.078 weeks (≈13 hours).
- Cohen's d expresses that raw difference in units of the pooled standard deviation: d = (x̄1 − x̄2) / s.
- Here d ≈ 0.029 — tiny. For comparison, the height difference between men and women is about d ≈ 1.7.
- Effect size matters more than statistical significance alone: a difference can be 'real' yet practically irrelevant — always ask whether it's clinically/operationally significant, not just non-zero.

```
def cohen_effect_size(g1, g2):
    diff = g1.mean() - g2.mean()
    n1, n2 = len(g1), len(g2)
    pooled_var = (n1*g1.var()
                  + n2*g2.var()) / (n1+n2)
    return diff / np.sqrt(pooled_var)
```

Reporting results responsibly

- The same result can be reported many honest ways, and the right choice depends on the audience: a scientist wants any real effect; a doctor wants clinically significant effects; a patient wants results relevant to their own decision.
- Your goals shape which summary statistics you lead with — but the choice must stay honest: it's fine to be persuasive, not fine to be misleading.
- Always acknowledge uncertainty and limitations alongside the headline number — exactly the standard this course expects in every lab report and the energy-sector performance reports you'll eventually write professionally.

WRAP-UP

What you should walk away with

- Pyplot fluency: line plots, decorated axes, markers for categorical data, and log-log scales for wide-ranging quantities.
- Practical pandas: reading real data, validating with value_counts()/describe(), cleaning special codes, Boolean filtering, and weighted means.
- How to read a histogram's shape, spot outliers, and why raw counts can mislead when comparing groups of different sizes.
- Summarizing a distribution responsibly: mean/variance/std, effect size (Cohen's d), and honest, audience-aware reporting.

Coming up

Lab 2: load pandas data, clean it, and explore histograms for an energy dataset

Lecture 3: Probability mass, cumulative, and density functions — comparing and modeling distributions

READING & SOURCES

Required reading and sources for this lecture

- Allen B. Downey, Think Stats: Exploratory Data Analysis in Python, 2nd ed. — Chapter 2 (“Distributions”). Free at greenteapress.com/thinkstats2
- Allen B. Downey, Elements of Data Science — Chapters 6–7 (“Plotting,” “DataFrames and Series”). 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 2:

Make sure matplotlib and pandas are installed. Skim Chapters 6–7 of Elements of Data Science and try the inline exercises — we will build directly on them.