Data Science – Khoa Học Dữ Liệu

Nguyễn Quản Bá Hồng*

Ngày 13 tháng 2 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Data Science – Khoa Học Dữ Liệu.*
PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_science/NQBH_data_science.pdf.
TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_science/NQBH_data_science.tex.

Mục lục

1 Basic Data Science – Khoa Học Dữ Liệu Cơ Bản

1.1 [Gru19]. JOEL GRUS. Data Science from Scratch: 1st Principles with Python. 2e

[731 Amazon ratings]

Amazon review. To really learn DS, should not only master tools – DS libraries, frameworks, modules, & toolkits – but also understand ideas & principles underlying them. Updated for Python 3.6, this 2e of DS from Scratch shows how these tools & algorithms work by implementing them from scratch.

If have an aptitude for mathematics & some programming skills, JOEL GRUS will help get comfortable with math & statistics at core of DS, & with hacking skills need to get started as a data scientist. Packed with new material of DL, statistics, & natural language processing, this updated book shows how to find gems in today's messy glut (dữ liệu hỗn độn) of data.

- Get a crash course in Python
- Learn basics of linear algebra, statistics, & probability – & how & when they're used in DS
- Collect, explore, clean, munge, & manipulate DS
- Dive into fundamentals of ML
- Implement models e.g. k-nearest neighbors, Naive Bayes, linear & logistic regression, decision trees, neural networks, & clustering
- Explore recommender systems, natural language processing, network analysis, MapReduce, & databases

Reviews.

- "GRUS takes you on a journey from being data-curious to getting a thorough understanding of bread-&-butter algorithms that every data scientist should know." – ROHIT SIVAPRASAD, Engineer, Facebook
- "I've recommended *DS from Scratch* to analysts & engineers wanting to make jump into ML. It's best tool for understanding fundamentals of discipline." – TOM MARTHALER, Engineering Manager, Amazon
- "Translating DS concepts into code is hard. GRUS's book makes it much easier." – WILLIAM COX, ML Engineer, Grubhub

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

Author Author. Joel Grus is a research engineer at Allen Institute for Artificial Intelligence. Previously he worked as a software engineer at Google & a data scientist at several startups. He lives in Seattle, where he regularly attends DS happy hours.

- Preface to 2e. 1e of *DS from Scratch* turned out very much book wanted it to be. But several years of developments in DS, of progress in Python ecosystem, & of personal growth as a developer & educator have *changed* what think a 1st book in DS should look life. In life, there are no do-overs. In writing, however, there are 2e.

  Rewritten all code & examples using Python 3.6 (& many of its newly introduced features, like type annotations). Woven into (Dệt thành) book an emphasis on writing clean code. Replaced some of 1e's toy examples with more realistic ones using "real" datasets. Added new material on topics e.g. DL, statistics, & natural language processing, corresponding to things that today's data scientists are likely to be working with. (Also removed some material that seems less relevant.) & gone over book with a fine-toothed comb, fixing bugs, rewriting explanations that are less clear than they could be, & freshening up some of jokes.

  Using Code Examples. Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/joelgrus/data-science-from-scratch. The only thing harder than writing a book is living with someone who's writing a book, & couldn't have pulled it off without their support.

- Preface to 1e.

  - DS. Data scientist has been called "sexiest job of 21st century", presumably by someone who has never visited a fire station. Nonetheless, DS is a hot & growing field, & it doesn't take a great deal of sleuthing (thám tử) to find analysts breathlessly prognosticating that over next 10 years, need billions & billions more data scientists than currently have.

    But what is DS? After all, can't produce data scientists if don't know what DS is. According to a Venn diagram that is somewhat famous in industry, DS lies at intersection of:

    * Hacking skills
    * Math & statistics knowledge
    * Substantive expertise

    Although originally intended to write a book covering all 3, quickly realize: a thorough treatment of "substantive expertise" (chuyên môn thực chất) would require 10s of thousands of pages. At that point, decided to focus on 1st 2. Goal: help develop hacking skills need to get started doing DS. Goal: help get comfortable with mathematics & statistics that are at core of DS.

    This is a somewhat heavy aspiration for a book. Best way to learn hacking skills is by hacking on things. By reading this book, will get a good understanding of way hack on things, which may not necessarily be best way for you to hack on things. Get a good understanding of some of tools used, which will not necessarily be best tools for you to use. Get a good understanding of way Grus approach data problems, which may not necessarily be best way for you to approach data problems. Intent (& hope): examples will inspire to try things own way. All code & data from book is available on GitHub to get started.

    Similarly, best way to learn mathematics is by doing mathematics. This is emphatically not a math book, & for most part, won't be "doing mathematics". However, can't really do DS without *some* understanding of probability & statistics & linear algebra. I.e., where appropriate, dive into mathematical equations, mathematical intuition, mathematical axioms, & cartoon versions of big mathematical ideas. Hope: won't be afraid to dive in with Grus.

    Throughout it all, also hope to give a sense that playing with data is fun, because playing with data is fun! (Especially compared to some of alternative, like tax preparation or coal mining.)

  - From Scratch. There are lots & lots of DS libraries, frameworks, modules, & toolkits that efficiently implement most common (as well as least common) DS algorithms & techniques. If become a data scientist, will become intimately familiar with NumPy, with scikit-learn, with pandas, & with a panoply of other libraries. They are great for doing DS. But they are also a good way to start doing DS without actually understanding DS.

    In this book, approach DS from scratch. I.e., build tools & implement algorithms by hand in order to better understand them. Put a lot of thought into creating implementations & examples that are clear, well commented, & readable. In most cases, tools built will be illuminating but impractical. They will work well on small toy datasets but fall over on "web-scale" ones.

    Throughout book, point to libraries might use to apply these techniques to larger datasets. But won't be using them here.

    There is a healthy debate raging over best language for learning DS. Many people believe it's statistical programming language R. (Call those people *wrong*.) A few people suggest Java or Scala. However, Python is obvious choice.

    Python has several features that make it well suited for learning (& doing) DS:

    * Free
    * Relatively simple to code in (&, in particular, to understand)
    * Has lots of useful DS-related libraries.

    Hesitant to call Python favorite programming language. There are other languages find more pleasant, better designed, or just more fun to code in. & yet pretty much every time start a new DS project, end up using Python. Every time need to quickly prototype sth that just works, end up using Python. & every time want to demonstrate DS concepts in a clear, easy-to-understand way, end up using Python. Accordingly, this book uses Python.

Goal of this book is not to teach Python. (Although nearly certain: by reading this book will learn some Python.) Take through a chapter-long crash course that highlights features that are most important for our purposes, but if know nothing about programming in Python (or about programming at all), then might want to supplement this book with some sort of "Python for Beginners" tutorial.

Remainder of introduction to DS will take this same approach – going into detail where going into detail seems crucial or illuminating, at other times leaving details for you to figure out yourself (or look up on Wikipedia).

Over years, trained a number of data scientists. While not all of them have gone on to become world-changing data ninja rockstars, left them all better data scientists than found them. Grown to believe: anyone who has some amount of mathematical aptitude & some amount of programming skill has necessary raw material to do DS. All she needs is an inquisitive mind, a willingness to work hared, & this book. Hence this book.

- 1. Introduction.

  "Data! Data! Data!" he cried impatiently. "I can't make bricks without clay." (Tôi không thể làm gạch nếu không có đất sét) – Arthur Conan Doyle

  - Ascendance of Data. Live in a world that's drowning in data. Websites track every user's every click. Your smartphone is building up a record of your location & speed every sec of every day. "Quantified selfers" wear pedometers-on-steroids that are always recording their heart rates, movement habits, diet, & sleep patterns. Smart cars collect driving habits, smart homes collect living habits, & smart marketers collect purchasing habits. Internet itself represents a huge graph of knowledge that contains (among other things), an enormous cross-referenced encyclopedia; domain-specific databases about movies, music, sports results, pinball machines, memes, & cocktails; & too many government statistics (some of them nearly true!) from too many governments to wrap your head around.
    Buried in these data are answers to countless question that no one's ever thought to ask. In this book, learn how to find them.
  - What is DS? There's a joke that says a data scientist is someone who knows more statistics than a computer scientist & more CS than a statistician. (didn't say it was a good joke.) In fact, some data scientists are – for all practical purposes – statisticians, while others are fairly indistinguishable from software engineers. Some are ML experts, while others couldn't machine-learn their way out of kindergarten. Some are PhDs with impressive publication records, while others have never read an academic paper (shame on them, though). In short, pretty much no matter how you define DS, find practitioners for whom definition is totally, absolutely wrong.
    Nonetheless, won't let that stop us from trying. Say: a data scientist is someone who extracts insights from messy data. Today's world is full of people trying to turn data into insight.
    E.g., dating site OkCupid asks its members to answer thousands of questions in order to find most appropriate matches for them. But it also analyzes these results to figure out innocuous-sounding questions can ask someone to fin out how likely someone is to sleep with you on 1st date.
    Facebook asks to list hometown & current location, ostensibly (bề ngoài) to make it easier for your friends to find & connect with you. But it also analyzes these locations to identify global migration patterns & where fanbases of different football teams live.
    As a large retailer, Target tracks your purchases & interactions, both online & in store. & it uses data to predictively model which of its customers are pregnant, to better market baby-related purchases to them.
    In 2012, Obama campaign employed dozens of data scientists who data-mined & experimented their way to identifying voters who needed extra attention, choosing optimal donor-specific fundraising appeals & programs, & focusing get-out-vote efforts where they were most likely to be useful. In 2016 Trump campaign tested a staggering varieyt of online ads & analyzed data to find what worked & what didn't.
    Now, before start feeling too jaded: some data scientists also occasionally use their skills for good – using data to make government more effecive, to help homeless, & to improve public health. But it certainly won't hurt your career if like figuring out best way to get people to click on advertisements.
  - Motivating Hypothetical: DataSciencester. Just been hired to lead DS efforts at DataSciencester, social network for data scientists.

    Note 1. *When wrote 1e of this book, thought: "a social network for data scientists" was a fun, silly hypothetical. Since then people have actually created social networks for data scientists, & have raised much more money from venture capitalists than made from his book. Most likely there is a valuable lesson here about silly DS hypotheticals &/ or book publishing.*

    Despite being *for* data scientists, DataSciencester has never actually invested in building its own DS practice. (In fairness, DataSciencester has never really invested in building its product either.) That will be your job! Through book, learn about DS concepts by solving problems encountered at work. Sometimes look at data explicitly supplied by users, sometimes look at data generated through their interactions with site, & sometimes seen look at data from experiments that we will design.
    & because DataSciencester has a strong "not-invented-here" mentality, build our own tools from scratch. At end, have a pretty solid understanding of fundamentals of DS. & ready to apply skills at a company with a less shaky premise, or to any other problems that happen to interest you.

- 2. A Crash Course in Python.

- 3. Visualizing Data.
- 4. Linear Algebra.
- 5. Statistics.
- 6. Probability.
- 7. Hypothesis & Inference.
- 8. Gradient Descent.
- 9. Getting Data.
- 10. Working with Data.
- 11. ML.

> "I am always ready to learn although I do not always like being taught." – Winston Churchill

Many people imagine: DS is mostly ML & data scientists mostly build & train & tweak ML models all day long. (Then again, many of those people don't actually know what ML *is*.) In fact, DS is mostly turning business problems into data problems & collecting data & understanding data & cleaning data & formatting data, after which ML is almost an afterthought. Even so, it's an interesting & essential afterthought that you pretty much have to know about in order to do DS.

- Modeling. Before can talk about ML, need to talk about *models*. What is a model? Simply a specification of a mathematical (or probabilistic) relationship that exists between different variables.

  E.g., if trying to raise money for your social networking site, might build a *business model* (likely in a spreadsheet) that takes inputs like "number of users", "ad revenue per user", & "number of employees" & outputs your annual profit for next several years. A cookbook recipe entails a model that relates inputs like "number of eaters" & "hungriness" to quantities of ingredients needed. & if you've ever watched poker on television, know: each player's win probability" is estimated in real time based on a model that takes into account cards that have been revealed so far & distribution of cards in deck.

  Business model is probably based on simple mathematical relationships: profit is revenue minus expenses, revenue is units sold times average price, & so on. Recipe model is probably based on trial & error – someone went in a kitchen & tried different combinations of ingredients until they found one they liked. & poker model is based on probability theory, rules of poker, & some reasonably innocuous assumptions about random process by which cards are dealt.

  – Mô hình kinh doanh có lẽ dựa trên các mối quan hệ toán học đơn giản: lợi nhuận là doanh thu trừ đi chi phí, doanh thu là số lượng bán ra nhân với giá trung bình, & vân vân. Mô hình công thức có lẽ dựa trên thử nghiệm & lỗi – ai đó vào bếp & thử nhiều cách kết hợp nguyên liệu khác nhau cho đến khi tìm được cách họ thích. & mô hình poker dựa trên lý thuyết xác suất, luật chơi poker, & 1 số giả định khá vô hại về quá trình ngẫu nhiên khi chia bài.

- What Is ML? Everyone has her own exact definition, but use *ML* to refer to creating & using models that are *learned from data*. In other contexts this might be called *predictive modeling* or *data mining*, but will stick with ML. Typically, goal: use existing data to develop models that can use to *predict* various outcomes for new data, e.g.:

  * Whether an email message is spam or not
  * Whether a credit card transaction is fraudulent (gian lận)
  * Which advertisement a shopper is most likely to click on
  * Which football team is going to win Super Bowl

  Look at both *supervised* models (in which there is a set of data labeled with correct answers to learn from) & *unsupervised* models (in which there are no such labels). There are various other types, like *semisupervised* (in which only some of data are labeled), *online* (in which model needs to continuously adjust to newly arriving data), & *reinforcement* (in which, after making a series of predictions, model gets a signal indicating how well it did) that won't cover in this book.

  Now, in even simplest situation there are entire universes of models that might describe relationship we're interested in. In most cases will ourselves choose a *parameterized* family of models & then use data to learn parameters that are in some way optimal.

  E.g., might assume: a person's height is (roughly) a linear function of his weight & then use data to learn what that linear function is. Or might assume: a decision tree is a good way to diagnose what diseases our patients have & then use data to learn "optimal" such tree. Throughout rest of book, investigate different families of models that we can learn.

  But before can do that, need to better understand fundamentals of ML. For rest of chap, discuss some of those basic concepts, before move on to models themselves.

- Overfitting & Underfitting. A common danger in ML is *overfitting* – producing a model that performs well on data you train it on but generalizes poorly to any ne data. This could involve learning *noise* in data. Or it could involve learning to identify specific inputs rather than whatever factors are actually predictive for desired output.

  Other side of this is *underfitting* – producing a model that doesn't perform well even on training data, although typically when this happens you decide your model isn't good enough & keep looking for a better one.

  In Fig. 11.1: Overfitting & underfitting, fit 3 polynomials to a sample of data.

Horizontal line shows best fig degree 0 (i.e., constant) polynomial. It severely *underfits* training data. Best fit degree 9 (i.e., 10-parameter) polynomial goes through every training data point exactly, but it very severely *overfits*; if were to pick a few more data points, it would quite likely miss them by a lot. & degree 1 line strikes a nice balance; it's pretty close to every point, & – if these data are representative – the line will likely to be close to new data points as well.

Clearly, models that are too complex lead to overfitting & don't generalize well beyond data they were trained on. So how do we make sure our models aren't too complex? Most fundamental approach involves using different data to train model & to test model.

Simplest way to do this: split dataset, so that e.g. $\frac{2}{3}$ of it is used to train model, after which measure model's performance on remaining $\frac{1}{3}$.

```
import random
from typing import TypeVar, List, Tuple
X = TypeVar('X') # generic type to represent a data point

# split dataset
def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """Split data into fractions [prob, 1 - prob]"""
    data = data[:] # make a shallow copy
    random.shuffle(data) # because shuffle modifies list
    cut = int(len(data) * prob) # use prob to find a cutoff
    return data[:cut], data[cut:] # split shuffled list there

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)

# proportions should be correct
assert len(train) == 750
assert len(test) == 250

# original data should be preserved (in some order)
assert sorted(train + test) == data
```

Often, have paired input variables & output variables. In that case, need to make sure to put corresponding values together in either training data or test data:

```
Y = TypeVar('Y') # generic type to represent output variables
def train_test_split(xs: List[X], ys: List[Y], test_pct: float) -> Tuple[List[X], List[X], List[Y], List[
    # generate indices & split them
    idxs = [i for i in range(len(xs))]
    train_idxs, test_idxs = split_data(idxs, 1 - test_pct)

    return ([xs[i] for i in train_idxs], [xs[i] for i in test_idxs], [ys[i] for i in train_idxs], [ys[i]
```

As always, want to make sure our code works right:

```
xs = [x for x in range(1000)] # xs are 1 ... 1000
ys = [2 * x for x in xs] # each y_i is twice x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)

# Check that the proportions are correct
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250

# Check that the corresponding data points are paired correctly
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

After which can do sth like:

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

If model was overfit to training data, then it will hopefully perform really poorly on (complete separate) test data. Said differently, if it performs well on test data, then can be more confident that it's *fitting* rather than *overfitting*.

However, there are a couple of ways this can go wrong.

The 1st is if there are common patterns in test & training data that wouldn't generalize to a larger dataset.

E.g., imagine: your dataset consists of user activity, with 1 row per user per week. In such a case, most users will appear in both training data & test data, & certain models might learn to *identify* users rather than discover relationships involving *attributes*. This isn't a huge worry, although it did happen to me once.

A bigger problem is if use test/train split not just to judge a model but also to *choose* from among many models. In that case, although each individual model may not be overfit, "choosing a model that performs best on test set" is a meta-training that makes test set function as a 2nd training set. (Of course model that performed best on test set is going to perform well on test set.)

– 1 vấn đề lớn hơn là nếu sử dụng test/train split không chỉ để đánh giá 1 mô hình mà còn để *chọn* trong số nhiều mô hình. Trong trường hợp đó, mặc dù mỗi mô hình riêng lẻ có thể không quá phù hợp, "việc chọn 1 mô hình hoạt động tốt nhất trên tập kiểm tra" là 1 siêu đào tạo khiến tập kiểm tra hoạt động như 1 tập đào tạo thứ 2. (Tất nhiên mô hình hoạt động tốt nhất trên tập kiểm tra sẽ hoạt động tốt trên tập kiểm tra.)

In such a situation, should split data into 3 parts: a training set for building models, a *validation* set for choosing among trained models, & a test set for judging final model.

- Correctness. When Grus is not doing DS, he dabble in medicine (chơi bời trong thuốc? WTF?). & in his spare time he's come up with a cheap, noninvasive test that can be given to a newborn baby that predicts – with $> 98\%$ accuracy – whether newborn will ever develop leukemia (bệnh bạch cầu). Lawyer has convinced him test is unpatentable, so share details here: predict leukemia iff baby is named Luke (which sounds sort of like "leukemia").

  This test is indeed $> 98\%$ accurate. Nonetheless, an incredibly stupid test, & a good illustration of why don't typically use "accuracy" to measure how good a (binary classification) model is.

  Imagine building a model to make a *binary* judgment. Is this email spam? Should hire this candidate? Is this air traveler secretly a terrorist?

  Given a set of labeled data & such a predictive model, every data point lies in 1 of 4 categories:

  * *True positive*: "This message is spam, & correctly predicted spam."
  * *False positive (Type 1 error)*: "This message is not spam, but predicted spam."
  * *False negative (Type 2 error)*: "This message is spam, but predicted not spam."
  * *True negative*: "This message is not spam, & correctly predicted not spam."

  Often represent these as counts in a Table: confusion matrix (ma trận nhầm lẫn). See how leukemia test fits into this framework. These days approximately 5 babies out of 1000 are named Luke. & lifetime prevalence of leukemia is $\approx 1.4\%$, or 14 out of every 1000 people.

  If believe these 2 factors are independent & apply "Luke is for leukemia" test to 1 million people, expect to see a confusion matrix like Fig. Can then use these to compute various statistics about model performance. E.g., *accuracy* is defined as fraction of correct predictions:

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total
    assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

  That seems like a pretty impressive number. But clearly this is not a good test, i.e., probably shouldn't put a lot of credence in raw accuracy (có lẽ không nên đặt nhiều niềm tin vào độ chính xác thô).

  Common to look at combination of *precision & recall*. Precision measures how accurate *positive* predictions were:

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fp)
assert precision(70, 4930, 13930, 981070) == 0.014
```

  Recall measures what fraction of positives our model identified:

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fn)
assert recall(70, 4930, 13930, 981070) == 0.005
```

  These are both terrible numbers, reflecting: this is a terrible model.

  Sometimes precision & recall are combined into *F1 score*, which is defined as:

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)
    return 2 * p * r / (p + r)
```

This is harmonic mean of precision & recall & necessary lies between them.

Usually choice of a model involves a tradeoff between precision & recall. A model that predict "yes" when it's even a little bit confident will probably have a high recall but a low precision; a model that predicts "yes" only when it's extremely confident is likely to have a low recall & a high precision.

– Thông thường, việc lựa chọn 1 mô hình liên quan đến sự đánh đổi giữa độ chính xác & độ thu hồi. Một mô hình dự đoán "có" khi nó thậm chí còn tự tin 1 chút thì có thể có độ thu hồi cao nhưng độ chính xác thấp; 1 mô hình chỉ dự đoán "có" khi nó cực kỳ tự tin thì có thể có độ thu hồi thấp & độ chính xác cao.

Alternatively, can think of this as a tradeoff between false positives & false negatives. Saying "yes" too often will give lots of false positives; saying "no" too often will give lots of false negatives.

– Ngoài ra, có thể coi đây là sự đánh đổi giữa kết quả dương tính giả & kết quả âm tính giả. Nói "có" quá thường xuyên sẽ đưa ra nhiều kết quả dương tính giả; nói "không" quá thường xuyên sẽ đưa ra nhiều kết quả âm tính giả.

Imagine: there were 10 risk factors for leukemia, & more of them you had more likely you were to develop leukemia. In that case can imagine a continuum of tests: "predict leukemia if at least 1 risk factor", "predict leukemia if at least 2 risk factors", & so on. As increase threshold, increase test's precision (since people with more risk factors are more likely to develop disease), & decrease test's recall (since fewer & fewer of eventual disease-sufferers will meet threshold). In cases like this, choosing right threshold is a matter of finding right tradeoff.

- Tưởng tượng: có 10 yếu tố nguy cơ mắc bệnh bạch cầu, & càng có nhiều yếu tố nguy cơ thì bạn càng có nhiều khả năng mắc bệnh bạch cầu. Trong trường hợp đó, hãy tưởng tượng một chuỗi các xét nghiệm: "dự đoán bệnh bạch cầu nếu có ít nhất 1 yếu tố nguy cơ", "dự đoán bệnh bạch cầu nếu có ít nhất 2 yếu tố nguy cơ", & cứ như vậy. Khi ngưỡng tăng, độ chính xác của xét nghiệm sẽ tăng (vì những người có nhiều yếu tố nguy cơ hơn có nhiều khả năng mắc bệnh hơn), & giảm độ nhớ lại của xét nghiệm (vì ít & ít người mắc bệnh cuối cùng sẽ đạt ngưỡng). Trong những trường hợp như thế này, việc chọn đúng ngưỡng là vấn đề tìm ra sự đánh đổi phù hợp.

- Bias-Variance Tradeoff. Another way of thinking about overfitting problem is as a tradeoff between bias & variance.

  Both are measures of what would happen if were to retrain your model many times on different sets of training data (from same larger population).

  E.g., degree 0 model in "Overfitting & Underfitting" will make a lot of mistakes for pretty much any training set (drawn from same population), i.e., it has a high *bias*. However, any 2 randomly chosen training sets should give pretty similar models (since any 2 randomly chosen training sets should have pretty similar average values). So say: it has a low *variance*. $\boxed{\text{High bias \& low variance typically correspond to underfitting}}$.

  On other hand, degree 9 model fit training set perfectly. It has very low bias but very high variance (since any 2 training sets would likely give rise to very different models). This corresponds to overfitting.

  Thinking about model problems this way can help figure out what to do when your model doesn't work so well.

  If model has high bias (i.e., perform poorly even on training data), 1 thing to try is adding more features. Going from degree 0 model to degree 1 model was a big improvement.

  If model was high variance, can similarly *remove* features. But another solution is to obtain more data (if can).

  In Fig. 11.2: Reducing variance with more data, fit a degree 9 polynomial to different size samples. Model fit based on 10 data points is all over place. If instead train on 100 data points, there's much less overfitting. & model trained from 1000 data points looks very similar to degree 1 model. Holding model complexity constant, more data you have, harder it is to overfit. On other hand, more data won't help with bias. If model doesn't use enough features to capture regularities in data, throwing more data at it won't help.

- Feature Extraction & Selection. When data doesn't have enough features, model is likely to underfit. & when data has too many features, easy to overfit. But what are features, & where do they come from?

  Features are whatever inputs we provide to model.

  In simplest case, features are simply given to you. If want to predict someone's salary based on years of experience, then years of experience is only feature you have. (Although, as saw in Overfitting & Underfitting, might also consider adding years of experience squared, cubed, & so on if that helps build a better model.)

  Things become more interesting as your data becomes more complicated. Imagine trying to build a spam filter to predict whether an email is junk or not. Most models won't know what to do with a raw email, which is just a collection of text. Have to extract features. E.g.:

  * Does email contain words *Viagra*?
  * How many times does letter *d* appear?
  * What was domain of sender?

  Answer to a question like 1st question here is simply a yes or no, which typically encode as a 1 or 0. 2nd is a number. 3rd is a choice from a discrete set of options.

Pretty much always, extract features from our data that fall into 1 of these 3 categories. What's more, types of features have constrain types of models can use.

* Naive Bayes classifier built in Chap. 13 is suited to yes-to-no features, like 1st one in preceding list.
* Regression models, studied in Chaps. 14 & 16, require numeric features (which could include dummy variables that are 0s & 1s).
* Decision trees in Chap. 17 can deal with numeric or categorical data.

Although in spam filter example looked for ways to create features, sometimes instead look for ways to remove features.

E.g., your inputs might be vectors of several hundred numbers. Depending on situation, it might be appropriate to distill these down to a handful of important dimensions & use only that small number of features. Or it might be appropriate to use a technique (like regularization) that penalizes models more features they use.

How choose features? That's where a combination of *experience* & *domain expertise* comes into play. If received lots of emails, then probably have a sense: presence of certain words might be a good indicator of spamminess. & might also get sense: number of *d*s is likely not a good indicator of spamminess. But in general have to try different things, which is part of fun.

- For Further Exploration.
  * Next several chaps are about different families of ML models.
  * Coursera https://www.coursera.org/learn/machine-learning course is original MOOC & is a good place to get a deeper understanding of basics of ML.
  * The Elements of Statistical Learning, by Jerome H. Friedman, Robert Tibshirani, & Trevor Hastie (Springer), is a somewhat canonical textbook that can be downloaded online for free. But be warned: it's very mathy.

- 12. k-Nearest Neighbors.
- 13. Naive Bayes.
- 14. Simple Linear Regression.
- 15. Multiple Regression.
- 16. Logistic Regression.
- 17. Decision Trees.
- 18. Neural Networks.
- 19. DL.
- 20. Clustering.
- 21. Natural Language Processing.
- 22. Network Analysis.
- 23. Recommender Systems.
- 24. Databases & SQL.
- 25. MapReduce.
- 26. Data Ethics.
- 27. Go Forth & Do DS.

1.2 [McK22]. Wes McKinney. Python for Data Analysis: Data Wrangling with pandas, NumPy & Jupyter

[356 Amazon ratings][25357 Goodreads ratings]

Amazon review. Get definitive handbook for manipulating, processing, cleaning, & crunching datasets in Python. Updated for Python 3.10 & `pandas` 1.4, 3e of this hand-on guide is packed with practical case studies that show you how to solve a broad set of data analysis problems effectively. Learn latest versions of pandas, NumPy, & Jupyter in process.

Written by Wes McKinney, creator of Python `pandas` project, this book is a practical, modern introduction to data science tools in Python. Ideal for analysts new to Python & for Python programmers new to data science & scientific computing. Data files & related material are available on GitHub.

- use Jupyter notebook & IPython shell for exploratory computing
- Learn basic & advanced features in NumPy

- Get started with data analysis tools in `pandas` library
- Use flexible tools to load, clean, transform, merge, & reshape data
- Create informative visualizations with matplotlib
- Apply `pandas` groupby facility to slice, dice, & summarize datasets
- Analyze & manipulative regular & irregular time series data
- Learn how to solve real-world data analysis problems with thorough, detailed examples

About the Author. WES MCKINNEY is a Nashville-based software developer & entrepreneur. After finishing his undergraduate degree in mathematics at MIT in 2007, he went on to do quantitative finance work at AQR Capital Management in Greenwich, CT. Frustrated by cumbersome data analysis tools, he learned Python & started building what would later become `pandas` project. He's now an active member of Python data community & is an advocate for Python use in data analysis, finance, & statistical computing applications.

WES was later cofounder & CEO of DataPad, whose technology assets & team were acquired by Cloudera in 2014. He has since become involved in big data technology, joining Project Management Committees for Apache Arrow & Apache Parquet projects in Apache Software Foundation. In 2018, he founded Ursa Labs, a not-for-profit organization focused Apache Arrow development, in partnership with RStudio & 2 Sigma Investments. In 2021, he cofounded technology startup Voltron Data, where he currently works as Chief Technology Officer.

"With this new edition, WES has updated his book to ensure it remains go-to resource for all things related to data analysis with Python & pandas. I cannot recommend this book highly enough." – PAUL BARRY, Lecturer & author of O'Reiley; *Head 1st Python*

WES MCKINNEY, cofounder & chief technology officer of Voltron Data, is an active member of Python data community & an advocate for Python use in data analysis, finance, & statistical computing applications. A graduate of MIT, he's also a member of project management committees for Apache Software Foundation's Apache Arrow & Apache Parquet projects.

Preface. 1e of this book was published in 2012, during a time when open source data analysis libraries for Python, especially pandas, were very new & developing rapidly. When time came to write 2e in 2016–2017, needed to update book not only for Python 3.6 (1e used Python 2.7) but also for many changes in `pandas` that had occurred over previous 5 years. 2022, there are fewer Python language changes (now at Python 3.10, with 3.11 coming out at end of 2022), but `pandas` has continued to evolve.

In 3e, goal: bring content up to date with current versions of Python, NumPy, pandas, & other projects, while also remaining relatively conservative about discussing newer Python projects having appeared in last few years. Since this book has become an important resource for many university courses & working professionals, try to avoid topics that are at risk of falling out of date within 1–2 year. That way paper copies won't be too difficult to follow in 2023 or 2024 or beyond.

A new feature of 3e: open access online version hosted on website `https://wesmckinney.com/book`, to serve as a resource & convenience for owners of print & digital editions. Intend to keep content reasonably up to date there, so if you paper paper book & run into sth that doesn't work properly, should check there for latest content changes.

Using Code Examples. Can find data files & related material for each chap in this book's GitHub repository at `https://github.com/wesm/pydata-book`, which is mirrored to Gitee (for those who cannot access GitHub) at `https://gitee.com/wesmckinn/pydata-book`.

This book is here to help get job done. In general, if example code is offered with this book, may use it in your programs & documentation. Do not need to contact for permission unless you're reproducing a significant portion of code. E.g., writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does not require permission. Answering a question by citing this book & quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

Acknowledgments for 3e (2022). > 1 decade since started writing 1e of this book & > 15 years since originally started journey as a Python programmer. A lot has changed since then! Python has evolved from a relatively niche (ngách) language for data analysis to most popular & most widely used language powering plurality (if not majority!) of DS, ML, & AI work.

Have not been an active contributor to `pandas` open source project since 2013, but its worldwide developer community has continued to thrive, serving as a model of community-centric open source software development. Many "next-generation" Python projects that deal with tabular data are modeling their user interfaces directly after pandas, so project has proved to have an enduring influence on future trajectory of Python DS ecosystem.

Acknowledgments for 2e (2017). 5 years almost to day since completed manuscript for this book's 1e in Jul 2012. A lot has changed. Python community has grown immensely, & ecosystem of open source software around it has flourished.

This new edition of book would not exist if for tireless efforts of `pandas` core developers, who have grown project & its user community into 1 of cornerstones of Python DS ecosystem.

With open source software projects more thinly resourced than ever relative to size of user bases, it is becoming increasingly important for businesses to provide support for development of key open source projects. It's the right thing to do.

- 1. Preliminaries.
  - 1.1. What Is This Book About? This book is concerned with nuts & bolts of manipulating, processing, cleaning, & crunching (nhai giòn tan) data in Python. Goal: offer a guide to parts of Python programming language & its data-oriented library ecosystem & tools that will equip you to become an effective data analyst. While "data analysis" is in title of book, focus is

specifically on Python programming, libraries, & tools as opposed to data analysis methodology. This is Python programming you need *for* data analysis.

Sometime after WES originally published this book in 2012, people started using term *data science* as an umbrella description for everything from simple descriptive statistics to more advanced statistical analysis & ML. Python open source ecosystem for doing data analysis (or DS) has also expanded significantly since then. There are now many other books which focus specifically on these more advanced methodologies. Hope: this book serves as adequate preparation to enable you to move on to a more domain-specific resource.

Remark 1. *Some might characterize much of content of book as "data manipulation" as opposed to "data analysis." Also use terms* wrangling *or* munging *to refer to data manipulation.*

What Kinds of Data? Primary focus is on *structured data*, a deliberately vague term that encompasses many different common forms of data, e.g.:

* Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files.
* Multidimensional arrays (matrices).
* Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user).
* Evenly or unevenly spaced time series.

This is by no means a complete list. Even though it may not always be obvious, a large percentage of datasets can be transformed into a structured form that is more suitable for analysis & modeling. If not, it may be possible to extract features from a dataset into a structured form. E.g., a collection of news articles could be processed into a word frequency table, which could then be used to perform sentiment analysis.

Most users of spreadsheet programs like Microsoft Excel, perhaps most widely used data analysis tool in the world, will not be strangers to these kinds of data.

- 1.2. Why Python for Data Analysis? For many people, Python programming language has strong appeal. Since its 1st appearance in 1991, Python has become 1 of most popular interpreted programming languages, along with Perl, Ruby, & others. Python & Ruby have become especially popular since 2005 or so for building websites using their numerous web frameworks, like Rails (Ruby) & Django (Python). Such languages are often called *scripting* languages, as they can be used to quickly write small programs, or *scripts* to automate other tasks. I don't like term "scripting languages," as it carries a connotation that they cannot be used for building serious software. Among interpreted languages, for various historical & cultural reasons, Python has developed a large & active scientific computing & data analysis community. In last 20 years, Python has gone from a bleeding-edge or "at your own risk" scientific computing language to 1 of most important languages for DS, ML, & general software development in academia & industry.

For data analysis & interactive computing & data visualization, Python will inevitably draw comparisons with other open source & commercial programming languages & tools in wide use, e.g. R, MATLAB, SAS, Stata, & others. In recent years, Python's improved open source libraries (e.g. `pandas` & scikit-learn) have made it a popular choice for data analysis tasks. Combined with Python's overall strength for general-purpose software engineering, it is an excellent option as a primary language for building data applications.

* Python as Glue. Part of Python's success in scientific computing: ease of integrating C, C++, & FORTRAN code - -1 phần thành công của Python trong điện toán khoa học: dễ dàng tích hợp mã C, C++, & FORTRAN. Most modern computing environments share a similar set of legacy FORTRAN & C libraries for doing linear algebra, optimization, integration, fast Fourier transforms, & other such algorithms. Same story has held true for many companies & national labs that have used Python to glue together decades' worth of legacy software.

  Many programs consist of small portions of code where most of time is spent, with large amounts of "glue code" that doesn't run often. In many cases, execution time of glue code is significant; effort is most fruitfully invested in optimizing computational bottlenecks, sometimes by moving code to a lower-level language like C.
* Solving "2-Language" Problem. In many organizations, common to research, prototype, & test new ideas using a more specialized computing language like SAS or R & then later port those ideas to be part of a larger production system written in, say, Java, C#, or C++. What people are increasingly finding: Python is a suitable language not only for doing research & prototyping but also for building production systems. *Why maintain 2 development environments when one will suffice?* Believe more & more companies will go down this path, as there are often significant organizational benefits to having both researchers & software engineers using same set of programming tools.

  Over last decade some new approaches to solving "2-language" problem have appeared, e.g. Julia programming language. Getting most out of Python in many cases *will* require programming in a low-level language like C or C++ & creating Python bindings to that code. I.e., "just-in-time" (JIT) compiler technology provided by libraries like Numba have provided a way to achieve excellent performance in many computational algorithms without having to leave Python programming environment.
* **Why Not Python?** While Python is an excellent environment for building many kinds of analytical applications & general-purpose systems, there are a number of uses for which Python may be less suitable.

  As Python is an interpreted programming language, in general most Python code will run substantially slower than code written in a compiled language like Java or C++. As *programmer time* is often more valuable than *CPU time*, many are

happy to make this trade-off. However, in an application with very low latency or demanding resource utilization requirements (e.g., a high-frequency trading systems), time spent programming in a lower-level (but also lower-productivity) language like C++ to achieve maximum possible performance might be time well spent.
– Vì Python là ngôn ngữ lập trình được thông dịch, nhìn chung hầu hết mã Python sẽ chạy chậm hơn đáng kể so với mã được viết bằng ngôn ngữ biên dịch như Java hoặc C++. Vì *thời gian lập trình* thường có giá trị hơn *thời gian CPU*, nhiều người vui vẻ chấp nhận sự đánh đổi này. Tuy nhiên, trong 1 ứng dụng có độ trễ rất thấp hoặc yêu cầu sử dụng tài nguyên khắt khe (ví dụ: hệ thống giao dịch tần suất cao), thời gian dành cho việc lập trình bằng ngôn ngữ cấp thấp hơn (nhưng cũng có năng suất thấp hơn) như C++ để đạt được hiệu suất tối đa có thể là thời gian được sử dụng hợp lý.
Python can be a challenging language for building highly concurrent, multithreaded applications, particularly applications with many CPU-bound threads. Reason for this: it has what is known as *global interpreter lock* (GIL), a mechanism that prevents interpreter from executing > 1 Python instruction at a time. Technical reasons for why GIL exists are beyond scope of this book. While it is true that in many big data processing applications, a cluster of computers may be required to process a dataset in a reasonable amount of time, there are still situations where a single-process, multithreaded system is desirable.
This is not to say: Python cannot execute truly multithreaded, parallel code. Python C extensions that use native mutithreading (in C or C++) can run code in parallel without being impacted by GIL, as long as they do not need to regularly interact with Python objects.

- 1.3. Essential Python Libraries. For those who are less familiar with Python data ecosystem & libraries used throughout book, a brief overview of some of them:
  - NumPy. NumPy, short for Numerical Python, has long been a cornerstone of numerical computing in Python. It provides data structures, algorithms, & library glue needed for most scientific applications involving numerical data in Python. `NumPy` contains, among other things:
    - A fast & efficient multidimensional array object `ndarray`
    - Functions for performing element-wise computations with arrays or mathematical operations between arrays
    - Tools for reading & writing array-based datasets to disk
    - Linear algebra operations, Fourier transform, & random number generation
    - A mature C API to enable Python extensions & native C or C++ code to access NumPy's data structures & computational facilities

    Beyond fast array-processing capabilities that `NumPy` adds to Python, 1 of its primary uses in data analysis is as a container for data to be passed between algorithms & libraries. For numerical data, `NumPy` arrays are more efficient for storing & manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, e.g. C or FORTRAN, can operate on data stored in a `NumPy` array without copying data into some other memory representation. Thus, many numerical computing tools for Python either assume `NumPy` arrays as a primary data structure or else target interoperability with NumPy.
  - `pandas`. pandas provides high-level data structures & functions designed to make working with structured or tabular data intuitive & flexible. Since its emergence in 2010, it has helped enable Python to be a powerful & productive data analysis environment. Primary objects in `pandas` that will be used in this book are `DataFrame`, a tabular, column-oriented data structure with both row & column labels, & `Series`, a 1D labeled array object.
    `pandas` blends array-computing ideas of `NumPy` with kinds of data manipulation capabilities found in spreadsheets & relationship databases (e.g. SQL). It provides convenient indexing functionality to enable you to reshape, slice & dice, perform aggregations (thực hiện tổng hợp), & select subsets of data. Since data manipulation, preparation, & cleaning are such important skills in data analysis, `pandas` is 1 of primary focuses of this book.
    As a bit of background, McKinney started building `pandas` in early 2008 during his tenure at AQR Capital Management, a quantitative investment management firm. At time, McKinney had a distinct set of requirements that were not addressed by any single tool at his disposal:
    - Data structures with labeled axes supporting automatic or explicit data alignment – this prevents common errors resulting from misaligned data & working with differently indexed data coming from different sources
    - Integrated time series functionality
    - Same data structures handle both time series data & non-time series data
    - Arithmetic operations & reductions that preserve metadata
    - Flexible handling of missing data
    - Merge & other relational operations found in popular databases (e.g., SQL-based)

    Wanted to be able to do all of these things in 1 place, preferably in a language well suited to general-purpose software development. Python was a good candidate language for this, but at that time an integrated set of data structures & tools providing this functionality did not exist. As a result of having been built initially to solve finance & business analytics problems, `pandas` features especially deep time series functionality & tools well suited for working with time-indexed data generated by business processes.
    McKinney spent a large part of 2011 & 2012 expanding pandas's capabilities with some of former AQR colleagues, Adam Klein, Chang She. In 2013, stopped being as involved in day-to-day project development, & `pandas` has since become a fully community-owned & community-maintained project with well > 2000 unique contributors around world.

For users of R language for statistical computing, `DataFrame` name will be familiar, as object was named after similar R `data.frame` object. Unlike Python, data frames are built into R programming language & its standard library. As a result, many features found in `pandas` are typically either part of R core implementation or provided by add-on packages. pandas name itself is derived from *panel data*, an econometrics term for multidimensional structured datasets, & a play on phrase *Python data analysis*.

* matplotlib. matplotlib is most popular Python library for producing plots & other 2D data visualizations. It was originally created by John D. Hunter & is now maintained by a large team of developers. It is designed for creating plots suitable for publication. While there are other visualization libraries available to Python programmers, matplotlib is still widely used & integrates reasonably well with rest of ecosystem. Think it is a safe choice as a default visualization tool.
* `IPython & Jupyter.` IPython project began in 2001 as Fernando Pérez's side project to make a better interactive Python interpreter. Over subsequent 20 years it has become 1 of most important tools in modern Python data stack. While it does not provide any computational or data analytical tools by itself, IPython is designed for both interactive computing & software development work. It encourages an *execute-explore* workflow instead of typical *edit-compile-run* workflow of many other programming languages. It also provides integrated access to OS's shell & filesystem; this reduces need to switch between a terminal window & a Python session in many cases. Since much of data analysis coding involves exploration, trial & error, & iteration, IPython can help you get job done faster.

  In 2014, Fernando & IPython team announced Jupyter project, a broader initiative to design language-agnostic interactive computing tools. IPython web notebook became Jupyter notebook, with support now for > 40 programming languages. IPython system can now be used as a *kernel* (a programming language mode) for using Python with Jupyter. IPython itself has become a component of much broader Jupyter open source project, which provides a productive environment for interactive & exploratory computing. Its oldest & simplest "mode" is as an enhanced Python shell designed to accelerate writing, testing, & debugging of Python code. You can also use IPython system through Jupyter notebook.

  Jupyter notebook system also allows you to author content in Markdown & HTML, providing you a means to create rich documents with code & text.

  McKinney personally uses IPython & Jupyter regularly in Python work, whether running, debugging, or testing code. In accompanying book materials on GitHub, you will find Jupyter notebooks containing all code examples from each chap. If cannot access GitHub where you are, can try mirror on Gitee.
* `SciPy.` SciPy is a collection of packages addressing a number of foundational problems in scientific computing. Some of tools it contains in its various modules:
  · `scipy.integrate`: Numerical integration routines & differential equation solvers
  · `scipy.linalg`: Linear algebra routines & matrix decompositions extending beyond those provided in `numpy.linalg`
  · `scipy.optimize`: Function optimizers (minimizers) & root finding algorithms
  · `scipy.signal`: Signal processing tools
  · `scipy.sparse`: Sparse matrices & sparse linear system solvers
  · `scipy.special`: Wrapper around SPECFUN, a FORTRAN library implementing many common mathematical functions, e.g. `gamma` function
  · `scipy.stats`: Standard continuous & discrete probability distributions (density functions, samplers, continuous distribution functions), various statistical tests, & more descriptive statistics

  Together, `NumPy` & SciPy form a reasonably complete & mature computational foundation for many traditional scientific computing applications.
* `scikit-learn`: Since project's inception in 2007, scikit-learn has become premier general-purpose ML toolkit for Python programmers. As of this writing, > 2000 different individuals have contributed code to project. It includes submodels for such models as:
  · Classification: SVM, nearest neighbors, random forest, logistic regression, etc.
  · Regression: Lasso, ridge regression, etc.
  · Clustering: k-means, spectral clustering, etc.
  · Dimensionality reduction: PCA, feature selection, matrix factorization, etc.
  · Model selection: Grid search, cross-validation, metrics
  · Preprocessing: Feature extraction, normalization

  Along with pandas, statsmodels, & IPython, scikit-learn has been critical for enabling Python to be a productive DS programming language. While I won't be able to include a comprehensive guide to scikit-learn in this book, I will give a brief introduction to some of its models & how to use them with other tools presented in book.
* `statsmodels` is a statistical analysis package that was seeded by work from Stanford University statistics professor Jonathan Taylor, who implemented a number of regression analysis models popular in R programming language. Skipper Seabold & Josef Perktold formally created new statsmodels project in 2010 & since then have grown project to a critical mass of engaged users & contributors. Nathaniel Smith developed Patsy project, which provides a formula or model specification framework for statsmodels inspired by R's formula system.

  Compared with scikit-learn, statsmodels contains algorithms for classical (primarily frequentist) statistics & econometrics. This includes such submodules as:
  · Regression models: linear regression, generalized linear models, robust linear models, linear mixed effect models, etc.

· Analysis of variance (ANOVA)
· Time series analysis: AR, ARMA, ARIMA, VAR, & other models
· Nonparametric methods: Kernel density estimation, kernel regression
· Visualization of statistical model results

statsmodels is more focused on statistical inference, providing uncertainty estimates & p-values for parameters. scikit-learn, by contrast, is more prediction focused.

As with scikit-learn, give a brief introduction to statsmodels & how to use it with `NumPy` & pandas.

* `Other Packages.` In 2022, there are many other Python libraries which might be discussed in a book about DS. This includes some newer projects like TensorFlow or PyTorch, which have become popular for ML or AI work. Now that there are other books out there that focus more specifically on those projects, recommend using this book to build a foundation in general-purpose Python data wrangling. Then, you should be well prepared to move on to a more advanced resource that may assume a certain level of expertise.

◦ 1.4. Installation & Setup. Since everyone uses Python for different applications, there is no single solution for setting up Python & obtaining necessary add-on packages. Many readers will not have a complete Python development environment suitable for following along with this book, so here give detailed instructions to get set up on each OS. Use Miniconda, a minimal installation of conda package manager, along with conda-forge, a community-maintained software distribution based on conda. This book uses Python 3.10 throughout, but if read in future, welcome to install a newer version of Python.

If for some reason these instructions become out-of-date by time you are reading this, can check website for book which I will endeavor to keep up to date with latest installation instructions.

* `Miniconda on Windows.`
* `GNU/Linux.` Linux details will vary a bit depending on Linux distribution type, but here give details for such distributions as Debian, Ubuntu, CentOS, & Fedora. Setup is similar to macOS with exception of how Miniconda is installed. Most readers will want to download default 64-bit installer file, which is for x86 architecture (but possible in future more users will have aarch64-based Linux machines). Installer is a shell script that must be executed in terminal. Then have a file named sth similar to `Miniconda3-latest-Linux-x86_64.sh`. To install it, execute this script with `bash`:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

Remark 2. *Some Linux distributions have all required Python packages (although outdated versions, in some cases) in their package managers & can be installed using a tool like* `apt`*. Setup described here uses Miniconda, as it's both easily reproducible across distributions & simpler to upgrade packages to their latest versions.*

Will have a choice of where to put Miniconda files. Recommend installing files in default location in home directory; e.g., `/home/$USER/miniconda` (with your username, naturally).

Installer will ask if wish to modify shell scripts to automatically activate Miniconda. Recommend doing this (select "yes") as a matter of convenience.

After completing installation, start a new terminal process & verify that you are picking up new Miniconda installation:

```
(base) nqbh@nqbh-dell:~/advanced_STEM_beyond/data_science$ python
Python 3.12.7 | packaged by Anaconda, Inc. | (main, Oct  4 2024, 13:27:36) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To exit Python shell, type `exit()` & press Enter or press Ctrl-D.

* Miniconda on macOS.
* Installing Necessary Packages. Have set up Miniconda on system, time to install main packages will be using in this book. 1st step: configure `conda-forge` as default package channel by running commands in a shell:

```
(base) $ conda config --add channels conda-forge
(base) $ conda config --set channel_priority strict
```

Now create a new conda "environment" with `conda create` command using Python 3.10:

```
(base) $ conda create -y -n pydata-book python=3.10

(base) nqbh@nqbh-dell:~$ conda create -y -n pydata-book python=3.12.7
Retrieving notices: done
Channels:
- conda-forge
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##
```

```
environment location: /home/nqbh/anaconda3/envs/pydata-book

added / updated specs:
- python=3.12.7


The following packages will be downloaded:

package                    |            build
---------------------------|-----------------
_libgcc_mutex-0.1          |      conda_forge           3 KB  conda-forge
_openmp_mutex-4.5          |            2_gnu          23 KB  conda-forge
bzip2-1.0.8                |       h4bc722e_7         247 KB  conda-forge
ca-certificates-2024.12.14 |       hbcca054_0         153 KB  conda-forge
ld_impl_linux-64-2.43      |       h712a8e2_2         654 KB  conda-forge
libexpat-2.6.4             |       h5888daf_0          72 KB  conda-forge
libffi-3.4.2               |       h7f98852_5          57 KB  conda-forge
libgcc-14.2.0              |       h77fa898_1         829 KB  conda-forge
libgcc-ng-14.2.0           |       h69a702a_1          53 KB  conda-forge
libgomp-14.2.0             |       h77fa898_1         450 KB  conda-forge
liblzma-5.6.3              |       hb9d3cd8_1         109 KB  conda-forge
liblzma-devel-5.6.3        |       hb9d3cd8_1         368 KB  conda-forge
libnsl-2.0.1               |       hd590300_0          33 KB  conda-forge
libsqlite-3.47.2           |       hee588c1_0         853 KB  conda-forge
libuuid-2.38.1             |       h0b41bf4_0          33 KB  conda-forge
libxcrypt-4.4.36           |       hd590300_1          98 KB  conda-forge
libzlib-1.3.1              |       hb9d3cd8_2          60 KB  conda-forge
ncurses-6.5                |       he02047a_1         868 KB  conda-forge
openssl-3.4.0              |       h7b32b05_1         2.8 MB  conda-forge
pip-24.3.1                 |     pyh8b19718_2         1.2 MB  conda-forge
python-3.12.7              |hc5c86c4_0_cpython        30.1 MB  conda-forge
readline-8.2               |       h8228510_1         275 KB  conda-forge
setuptools-75.7.0          |     pyhff2d567_0         756 KB  conda-forge
tk-8.6.13                  |noxft_h4845f30_101         3.2 MB  conda-forge
tzdata-2024b               |       hc8b5060_0         119 KB  conda-forge
wheel-0.45.1               |     pyhd8ed1ab_1          61 KB  conda-forge
xz-5.6.3                   |       hbcc6ac9_1          23 KB  conda-forge
xz-gpl-tools-5.6.3         |       hbcc6ac9_1          33 KB  conda-forge
xz-tools-5.6.3             |       hb9d3cd8_1          88 KB  conda-forge
------------------------------------------------------------
Total:        43.4 MB

The following NEW packages will be INSTALLED:

_libgcc_mutex      conda-forge/linux-64::_libgcc_mutex-0.1-conda_forge
_openmp_mutex      conda-forge/linux-64::_openmp_mutex-4.5-2_gnu
bzip2              conda-forge/linux-64::bzip2-1.0.8-h4bc722e_7
ca-certificates    conda-forge/linux-64::ca-certificates-2024.12.14-hbcca054_0
ld_impl_linux-64   conda-forge/linux-64::ld_impl_linux-64-2.43-h712a8e2_2
libexpat           conda-forge/linux-64::libexpat-2.6.4-h5888daf_0
libffi             conda-forge/linux-64::libffi-3.4.2-h7f98852_5
libgcc             conda-forge/linux-64::libgcc-14.2.0-h77fa898_1
libgcc-ng          conda-forge/linux-64::libgcc-ng-14.2.0-h69a702a_1
libgomp            conda-forge/linux-64::libgomp-14.2.0-h77fa898_1
liblzma            conda-forge/linux-64::liblzma-5.6.3-hb9d3cd8_1
liblzma-devel      conda-forge/linux-64::liblzma-devel-5.6.3-hb9d3cd8_1
libnsl             conda-forge/linux-64::libnsl-2.0.1-hd590300_0
libsqlite          conda-forge/linux-64::libsqlite-3.47.2-hee588c1_0
libuuid            conda-forge/linux-64::libuuid-2.38.1-h0b41bf4_0
libxcrypt          conda-forge/linux-64::libxcrypt-4.4.36-hd590300_1
libzlib            conda-forge/linux-64::libzlib-1.3.1-hb9d3cd8_2
ncurses            conda-forge/linux-64::ncurses-6.5-he02047a_1
openssl            conda-forge/linux-64::openssl-3.4.0-h7b32b05_1
pip                conda-forge/noarch::pip-24.3.1-pyh8b19718_2
```

```
python              conda-forge/linux-64::python-3.12.7-hc5c86c4_0_cpython
readline            conda-forge/linux-64::readline-8.2-h8228510_1
setuptools          conda-forge/noarch::setuptools-75.7.0-pyhff2d567_0
tk                  conda-forge/linux-64::tk-8.6.13-noxft_h4845f30_101
tzdata              conda-forge/noarch::tzdata-2024b-hc8b5060_0
wheel               conda-forge/noarch::wheel-0.45.1-pyhd8ed1ab_1
xz                  conda-forge/linux-64::xz-5.6.3-hbcc6ac9_1
xz-gpl-tools        conda-forge/linux-64::xz-gpl-tools-5.6.3-hbcc6ac9_1
xz-tools            conda-forge/linux-64::xz-tools-5.6.3-hb9d3cd8_1



Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate pydata-book
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

After installation completes, activate environment with `conda activate`:

```
(base) nqbh@nqbh-dell:~$ conda activate pydata-book
(pydata-book) nqbh@nqbh-dell:~$
```

Remark 3. *Necessary to use* `conda activate` *to activate your environment each time you open a new terminal. Can see information about active conda environment at any time from terminal by running* `conda info`*.*

Now, install essential packages used throughout book (along with their dependencies) with `conda install`:

```
(pydata-book) $ conda install -y {\tt pandas} jupyter matplotlib

(pydata-book) nqbh@nqbh-dell:~$ conda install -y {\tt pandas} jupyter matplotlib
Channels:
- conda-forge
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

environment location: /home/nqbh/anaconda3/envs/pydata-book

added / updated specs:
- jupyter
- matplotlib
- pandas


The following packages will be downloaded:

package                    |            build
---------------------------|-----------------
alsa-lib-1.2.13            |       hb9d3cd8_0         547 KB  conda-forge
anyio-4.8.0                |     pyhd8ed1ab_0         113 KB  conda-forge
argon2-cffi-23.1.0         |     pyhd8ed1ab_1          18 KB  conda-forge
argon2-cffi-bindings-21.2.0|  py312h66e93f0_5          34 KB  conda-forge
arrow-1.3.0                |     pyhd8ed1ab_1          98 KB  conda-forge
asttokens-3.0.0            |     pyhd8ed1ab_1          28 KB  conda-forge
```

```
async-lru-2.0.4            |     pyhd8ed1ab_1          15 KB  conda-forge
attrs-24.3.0               |     pyh71513ae_0          55 KB  conda-forge
babel-2.16.0               |     pyhd8ed1ab_1         6.2 MB  conda-forge
beautifulsoup4-4.12.3      |     pyha770c72_1         115 KB  conda-forge
bleach-6.2.0               |     pyhd8ed1ab_3         129 KB  conda-forge
bleach-with-css-6.2.0      |       hd8ed1ab_3           6 KB  conda-forge
brotli-1.1.0               |       hb9d3cd8_2          19 KB  conda-forge
brotli-bin-1.1.0           |       hb9d3cd8_2          18 KB  conda-forge
brotli-python-1.1.0        |  py312h2ec8cdc_2         342 KB  conda-forge
cached-property-1.5.2      |       hd8ed1ab_1           4 KB  conda-forge
cached_property-1.5.2      |     pyha770c72_1          11 KB  conda-forge
cairo-1.18.2               |       h3394656_1         956 KB  conda-forge
certifi-2024.12.14         |     pyhd8ed1ab_0         158 KB  conda-forge
cffi-1.17.1                |  py312h06ac9bb_0         288 KB  conda-forge
charset-normalizer-3.4.1   |     pyhd8ed1ab_0          46 KB  conda-forge
comm-0.2.2                 |     pyhd8ed1ab_1          12 KB  conda-forge
contourpy-1.3.1            |  py312h68727a3_0         270 KB  conda-forge
cycler-0.12.1              |     pyhd8ed1ab_1          13 KB  conda-forge
cyrus-sasl-2.1.27          |       h54b06d7_7         214 KB  conda-forge
dbus-1.13.6                |       h5008d03_3         604 KB  conda-forge
debugpy-1.8.11             |  py312h2ec8cdc_0         2.5 MB  conda-forge
decorator-5.1.1            |     pyhd8ed1ab_1          14 KB  conda-forge
defusedxml-0.7.1           |     pyhd8ed1ab_0          23 KB  conda-forge
double-conversion-3.3.0    |       h59595ed_0          77 KB  conda-forge
entrypoints-0.4            |     pyhd8ed1ab_1          11 KB  conda-forge
exceptiongroup-1.2.2       |     pyhd8ed1ab_1          20 KB  conda-forge
executing-2.1.0            |     pyhd8ed1ab_1          28 KB  conda-forge
expat-2.6.4                |       h5888daf_0         135 KB  conda-forge
font-ttf-dejavu-sans-mono-2.37|       hab24e00_0         388 KB  conda-forge
font-ttf-inconsolata-3.000 |       h77eed37_0          94 KB  conda-forge
font-ttf-source-code-pro-2.038|       h77eed37_0         684 KB  conda-forge
font-ttf-ubuntu-0.83       |       h77eed37_3         1.5 MB  conda-forge
fontconfig-2.15.0          |       h7e30c49_1         259 KB  conda-forge
fonts-conda-ecosystem-1    |                0           4 KB  conda-forge
fonts-conda-forge-1        |                0           4 KB  conda-forge
fonttools-4.55.3           |  py312h178313f_1         2.7 MB  conda-forge
fqdn-1.5.1                 |     pyhd8ed1ab_1          16 KB  conda-forge
freetype-2.12.1            |       h267a509_2         620 KB  conda-forge
graphite2-1.3.13           |    h59595ed_1003          95 KB  conda-forge
h11-0.14.0                 |     pyhd8ed1ab_1          51 KB  conda-forge
h2-4.1.0                   |     pyhd8ed1ab_1          51 KB  conda-forge
harfbuzz-10.1.0            |       h0b3b770_0         1.5 MB  conda-forge
hpack-4.0.0                |     pyhd8ed1ab_1          29 KB  conda-forge
httpcore-1.0.7             |     pyh29332c3_1          48 KB  conda-forge
httpx-0.28.1               |     pyhd8ed1ab_0          62 KB  conda-forge
hyperframe-6.0.1           |     pyhd8ed1ab_1          17 KB  conda-forge
icu-75.1                   |       he02047a_0        11.6 MB  conda-forge
idna-3.10                  |     pyhd8ed1ab_1          49 KB  conda-forge
importlib-metadata-8.5.0   |     pyha770c72_1          28 KB  conda-forge
importlib_resources-6.5.2  |     pyhd8ed1ab_0          33 KB  conda-forge
ipykernel-6.29.5           |     pyh3099207_0         116 KB  conda-forge
ipython-8.31.0             |     pyh707e725_0         587 KB  conda-forge
ipywidgets-8.1.5           |     pyhd8ed1ab_1         111 KB  conda-forge
isoduration-20.11.0        |     pyhd8ed1ab_1          19 KB  conda-forge
jedi-0.19.2                |     pyhd8ed1ab_1         824 KB  conda-forge
jinja2-3.1.5               |     pyhd8ed1ab_0         110 KB  conda-forge
json5-0.10.0               |     pyhd8ed1ab_1          31 KB  conda-forge
jsonpointer-3.0.0          |  py312h7900ff3_1          17 KB  conda-forge
jsonschema-4.23.0          |     pyhd8ed1ab_1          73 KB  conda-forge
jsonschema-specifications-2024.10.1|     pyhd8ed1ab_1          16 KB  conda-forge
jsonschema-with-format-nongpl-4.23.0|       hd8ed1ab_1           7 KB  conda-forge
jupyter-1.1.1              |     pyhd8ed1ab_1           9 KB  conda-forge
jupyter-lsp-2.2.5          |     pyhd8ed1ab_1          54 KB  conda-forge
jupyter_client-8.6.3       |     pyhd8ed1ab_1         104 KB  conda-forge
```

```
jupyter_console-6.6.3       |     pyhd8ed1ab_1          26 KB  conda-forge
jupyter_core-5.7.2          |     pyh31011fe_1          56 KB  conda-forge
jupyter_events-0.11.0       |     pyhd8ed1ab_0          22 KB  conda-forge
jupyter_server-2.15.0       |     pyhd8ed1ab_0         320 KB  conda-forge
jupyter_server_terminals-0.5.3|     pyhd8ed1ab_1          19 KB  conda-forge
jupyterlab-4.3.4            |     pyhd8ed1ab_0         6.9 MB  conda-forge
jupyterlab_pygments-0.3.0   |     pyhd8ed1ab_2          18 KB  conda-forge
jupyterlab_server-2.27.3    |     pyhd8ed1ab_1          48 KB  conda-forge
jupyterlab_widgets-3.0.13   |     pyhd8ed1ab_1         182 KB  conda-forge
keyutils-1.6.1              |       h166bdaf_0         115 KB  conda-forge
kiwisolver-1.4.7            |  py312h68727a3_0          69 KB  conda-forge
krb5-1.21.3                 |       h659f571_0         1.3 MB  conda-forge
lcms2-2.16                  |       hb7c19ff_0         239 KB  conda-forge
lerc-4.0.0                  |       h27087fc_0         275 KB  conda-forge
libblas-3.9.0               |26_linux64_openblas          16 KB  conda-forge
libbrotlicommon-1.1.0       |       hb9d3cd8_2          67 KB  conda-forge
libbrotlidec-1.1.0          |       hb9d3cd8_2          32 KB  conda-forge
libbrotlienc-1.1.0          |       hb9d3cd8_2         275 KB  conda-forge
libcblas-3.9.0              |26_linux64_openblas          16 KB  conda-forge
libclang-cpp19.1-19.1.6     |default_hb5137d0_0        19.6 MB  conda-forge
libclang13-19.1.6           |default_h9c6a7e4_0        11.3 MB  conda-forge
libcups-2.3.3               |       h4637d8d_4         4.3 MB  conda-forge
libdeflate-1.23             |       h4ddbbb0_0          71 KB  conda-forge
libdrm-2.4.124              |       hb9d3cd8_0         237 KB  conda-forge
libedit-3.1.20240808        | pl5321h7949ede_0         132 KB  conda-forge
libegl-1.7.0                |       ha4b6fd6_2          44 KB  conda-forge
libgfortran-14.2.0          |       h69a702a_1          53 KB  conda-forge
libgfortran5-14.2.0         |       hd5240d6_1         1.4 MB  conda-forge
libgl-1.7.0                 |       ha4b6fd6_2         132 KB  conda-forge
libglib-2.82.2              |       h2ff4ddf_0         3.7 MB  conda-forge
libglvnd-1.7.0              |       ha4b6fd6_2         129 KB  conda-forge
libglx-1.7.0                |       ha4b6fd6_2          74 KB  conda-forge
libiconv-1.17               |       hd590300_2         689 KB  conda-forge
libjpeg-turbo-3.0.0         |       hd590300_1         604 KB  conda-forge
liblapack-3.9.0             |26_linux64_openblas          16 KB  conda-forge
libllvm19-19.1.6            |       ha7bfdaf_0        38.3 MB  conda-forge
libntlm-1.8                 |       hb9d3cd8_0          33 KB  conda-forge
libopenblas-0.3.28          |pthreads_h94d23a6_1         5.3 MB  conda-forge
libopengl-1.7.0             |       ha4b6fd6_2          50 KB  conda-forge
libpciaccess-0.18           |       hd590300_0          28 KB  conda-forge
libpng-1.6.45               |       h943b412_0         283 KB  conda-forge
libpq-17.2                  |       h3b95a9b_1         2.5 MB  conda-forge
libsodium-1.0.20            |       h4ab18f5_0         201 KB  conda-forge
libstdcxx-14.2.0            |       hc0a3c3a_1         3.7 MB  conda-forge
libstdcxx-ng-14.2.0         |       h4852527_1          53 KB  conda-forge
libtiff-4.7.0               |       hd9ff511_3         418 KB  conda-forge
libwebp-base-1.5.0          |       h851e524_0         420 KB  conda-forge
libxcb-1.17.0               |       h8a09558_0         387 KB  conda-forge
libxkbcommon-1.7.0          |       h2c5496b_1         579 KB  conda-forge
libxml2-2.13.5              |       h8d12d68_1         674 KB  conda-forge
libxslt-1.1.39              |       h76b75d6_0         248 KB  conda-forge
markupsafe-3.0.2            |  py312h178313f_1          24 KB  conda-forge
matplotlib-3.10.0           |  py312h7900ff3_0          16 KB  conda-forge
matplotlib-base-3.10.0      |  py312hd3ec401_0         7.8 MB  conda-forge
matplotlib-inline-0.1.7     |     pyhd8ed1ab_1          14 KB  conda-forge
mistune-3.1.0               |     pyhd8ed1ab_0          67 KB  conda-forge
munkres-1.1.4               |     pyh9f0ad1d_0          12 KB  conda-forge
mysql-common-9.0.1          |       h266115a_4         605 KB  conda-forge
mysql-libs-9.0.1            |       he0572af_4         1.3 MB  conda-forge
nbclient-0.10.2             |     pyhd8ed1ab_0          27 KB  conda-forge
nbconvert-core-7.16.5       |     pyhd8ed1ab_1         185 KB  conda-forge
nbformat-5.10.4             |     pyhd8ed1ab_1          99 KB  conda-forge
nest-asyncio-1.6.0          |     pyhd8ed1ab_1          11 KB  conda-forge
notebook-7.3.2              |     pyhd8ed1ab_0         8.6 MB  conda-forge
```

```
notebook-shim-0.2.4         |     pyhd8ed1ab_1          16 KB  conda-forge
numpy-2.2.1                 |  py312h7e784f5_0         8.1 MB  conda-forge
openjpeg-2.5.3              |       h5fbd93e_0         335 KB  conda-forge
openldap-2.6.9              |       he970967_0         766 KB  conda-forge
overrides-7.7.0             |     pyhd8ed1ab_1          29 KB  conda-forge
packaging-24.2              |     pyhd8ed1ab_2          59 KB  conda-forge
pandas-2.2.3                |  py312hf9745cd_1        14.7 MB  conda-forge
pandocfilters-1.5.0         |     pyhd8ed1ab_0          11 KB  conda-forge
parso-0.8.4                 |     pyhd8ed1ab_1          74 KB  conda-forge
pcre2-10.44                 |       hba22ea6_2         930 KB  conda-forge
pexpect-4.9.0               |     pyhd8ed1ab_1          52 KB  conda-forge
pickleshare-0.7.5           |  pyhd8ed1ab_1004          11 KB  conda-forge
pillow-11.1.0               |  py312h80c1187_0        40.8 MB  conda-forge
pixman-0.44.2               |       h29eaf8c_0         372 KB  conda-forge
pkgutil-resolve-name-1.3.10|     pyhd8ed1ab_2          10 KB  conda-forge
platformdirs-4.3.6          |     pyhd8ed1ab_1          20 KB  conda-forge
prometheus_client-0.21.1    |     pyhd8ed1ab_0          48 KB  conda-forge
prompt-toolkit-3.0.48       |     pyha770c72_1         264 KB  conda-forge
prompt_toolkit-3.0.48       |       hd8ed1ab_1           6 KB  conda-forge
psutil-6.1.1                |  py312h66e93f0_0         476 KB  conda-forge
pthread-stubs-0.4           |    hb9d3cd8_1002           8 KB  conda-forge
ptyprocess-0.7.0            |     pyhd8ed1ab_1          19 KB  conda-forge
pure_eval-0.2.3             |     pyhd8ed1ab_1          16 KB  conda-forge
pycparser-2.22              |     pyh29332c3_1         108 KB  conda-forge
pygments-2.19.1             |     pyhd8ed1ab_0         868 KB  conda-forge
pyparsing-3.2.1             |     pyhd8ed1ab_0          91 KB  conda-forge
pyside6-6.8.1               |  py312h91f0f75_0        10.4 MB  conda-forge
pysocks-1.7.1               |     pyha55dd90_7          21 KB  conda-forge
python-dateutil-2.9.0.post0|     pyhff2d567_1         217 KB  conda-forge
python-fastjsonschema-2.21.1|     pyhd8ed1ab_0         221 KB  conda-forge
python-json-logger-2.0.7    |     pyhd8ed1ab_0          13 KB  conda-forge
python-tzdata-2024.2        |     pyhd8ed1ab_1         139 KB  conda-forge
python_abi-3.12             |          5_cp312           6 KB  conda-forge
pytz-2024.1                 |     pyhd8ed1ab_0         184 KB  conda-forge
pyyaml-6.0.2                |  py312h66e93f0_1         202 KB  conda-forge
pyzmq-26.2.0                |  py312hbf22597_3         369 KB  conda-forge
qhull-2020.2                |       h434a139_5         540 KB  conda-forge
qt6-main-6.8.1              |       h588cce1_2        49.2 MB  conda-forge
referencing-0.35.1          |     pyhd8ed1ab_1          41 KB  conda-forge
requests-2.32.3             |     pyhd8ed1ab_1          57 KB  conda-forge
rfc3339-validator-0.1.4     |     pyhd8ed1ab_1          10 KB  conda-forge
rfc3986-validator-0.1.1     |     pyh9f0ad1d_0           8 KB  conda-forge
rpds-py-0.22.3              |  py312h12e396e_0         346 KB  conda-forge
send2trash-1.8.3            |     pyh0d859eb_1          22 KB  conda-forge
six-1.17.0                  |     pyhd8ed1ab_0          16 KB  conda-forge
sniffio-1.3.1               |     pyhd8ed1ab_1          15 KB  conda-forge
soupsieve-2.5               |     pyhd8ed1ab_1          36 KB  conda-forge
stack_data-0.6.3            |     pyhd8ed1ab_1          26 KB  conda-forge
terminado-0.18.1            |     pyh0d859eb_0          22 KB  conda-forge
tinycss2-1.4.0              |     pyhd8ed1ab_0          28 KB  conda-forge
tomli-2.2.1                 |     pyhd8ed1ab_1          19 KB  conda-forge
tornado-6.4.2               |  py312h66e93f0_0         821 KB  conda-forge
traitlets-5.14.3            |     pyhd8ed1ab_1         107 KB  conda-forge
types-python-dateutil-2.9.0.20241206|     pyhd8ed1ab_0          22 KB  conda-forge
typing-extensions-4.12.2    |       hd8ed1ab_1          10 KB  conda-forge
typing_extensions-4.12.2    |     pyha770c72_1          39 KB  conda-forge
typing_utils-0.1.0          |     pyhd8ed1ab_1          15 KB  conda-forge
unicodedata2-15.1.0         |  py312h66e93f0_1         360 KB  conda-forge
uri-template-1.3.0          |     pyhd8ed1ab_1          23 KB  conda-forge
urllib3-2.3.0               |     pyhd8ed1ab_0          98 KB  conda-forge
wayland-1.23.1              |       h3e06ad9_0         314 KB  conda-forge
wcwidth-0.2.13              |     pyhd8ed1ab_1          32 KB  conda-forge
webcolors-24.11.1           |     pyhd8ed1ab_0          18 KB  conda-forge
webencodings-0.5.1          |     pyhd8ed1ab_3          15 KB  conda-forge
```

```
websocket-client-1.8.0     |     pyhd8ed1ab_1          46 KB  conda-forge
widgetsnbextension-4.0.13  |     pyhd8ed1ab_1         877 KB  conda-forge
xcb-util-0.4.1             |       hb711507_2          19 KB  conda-forge
xcb-util-cursor-0.1.5      |       hb9d3cd8_0          20 KB  conda-forge
xcb-util-image-0.4.0       |       hb711507_2          24 KB  conda-forge
xcb-util-keysyms-0.4.1     |       hb711507_0          14 KB  conda-forge
xcb-util-renderutil-0.3.10 |       hb711507_0          17 KB  conda-forge
xcb-util-wm-0.4.2          |       hb711507_0          50 KB  conda-forge
xkeyboard-config-2.43      |       hb9d3cd8_0         380 KB  conda-forge
xorg-libice-1.1.2          |       hb9d3cd8_0          57 KB  conda-forge
xorg-libsm-1.2.5           |       he73a12e_0          27 KB  conda-forge
xorg-libx11-1.8.10         |       h4f16b4b_1         818 KB  conda-forge
xorg-libxau-1.0.12         |       hb9d3cd8_0          14 KB  conda-forge
xorg-libxcomposite-0.4.6   |       hb9d3cd8_2          13 KB  conda-forge
xorg-libxcursor-1.2.3      |       hb9d3cd8_0          32 KB  conda-forge
xorg-libxdamage-1.1.6      |       hb9d3cd8_0          13 KB  conda-forge
xorg-libxdmcp-1.1.5        |       hb9d3cd8_0          19 KB  conda-forge
xorg-libxext-1.3.6         |       hb9d3cd8_0          49 KB  conda-forge
xorg-libxfixes-6.0.1       |       hb9d3cd8_0          19 KB  conda-forge
xorg-libxi-1.8.2           |       hb9d3cd8_0          46 KB  conda-forge
xorg-libxrandr-1.5.4       |       hb9d3cd8_0          29 KB  conda-forge
xorg-libxrender-0.9.12     |       hb9d3cd8_0          32 KB  conda-forge
xorg-libxtst-1.2.5         |       hb9d3cd8_3          32 KB  conda-forge
xorg-libxxf86vm-1.1.6      |       hb9d3cd8_0          17 KB  conda-forge
yaml-0.2.5                 |       h7f98852_2          87 KB  conda-forge
zeromq-4.3.5               |       h3b0a872_7         328 KB  conda-forge
zipp-3.21.0                |     pyhd8ed1ab_1          21 KB  conda-forge
zstandard-0.23.0           |  py312hef9b889_1         410 KB  conda-forge
zstd-1.5.6                 |       ha6fb4c9_0         542 KB  conda-forge
------------------------------------------------------------
                                           Total:       295.3 MB

The following NEW packages will be INSTALLED:

alsa-lib           conda-forge/linux-64::alsa-lib-1.2.13-hb9d3cd8_0
anyio              conda-forge/noarch::anyio-4.8.0-pyhd8ed1ab_0
argon2-cffi        conda-forge/noarch::argon2-cffi-23.1.0-pyhd8ed1ab_1
argon2-cffi-bindi~ conda-forge/linux-64::argon2-cffi-bindings-21.2.0-py312h66e93f0_5
arrow              conda-forge/noarch::arrow-1.3.0-pyhd8ed1ab_1
asttokens          conda-forge/noarch::asttokens-3.0.0-pyhd8ed1ab_1
async-lru          conda-forge/noarch::async-lru-2.0.4-pyhd8ed1ab_1
attrs              conda-forge/noarch::attrs-24.3.0-pyh71513ae_0
babel              conda-forge/noarch::babel-2.16.0-pyhd8ed1ab_1
beautifulsoup4     conda-forge/noarch::beautifulsoup4-4.12.3-pyha770c72_1
bleach             conda-forge/noarch::bleach-6.2.0-pyhd8ed1ab_3
bleach-with-css    conda-forge/noarch::bleach-with-css-6.2.0-hd8ed1ab_3
brotli             conda-forge/linux-64::brotli-1.1.0-hb9d3cd8_2
brotli-bin         conda-forge/linux-64::brotli-bin-1.1.0-hb9d3cd8_2
brotli-python      conda-forge/linux-64::brotli-python-1.1.0-py312h2ec8cdc_2
cached-property    conda-forge/noarch::cached-property-1.5.2-hd8ed1ab_1
cached_property    conda-forge/noarch::cached_property-1.5.2-pyha770c72_1
cairo              conda-forge/linux-64::cairo-1.18.2-h3394656_1
certifi            conda-forge/noarch::certifi-2024.12.14-pyhd8ed1ab_0
cffi               conda-forge/linux-64::cffi-1.17.1-py312h06ac9bb_0
charset-normalizer conda-forge/noarch::charset-normalizer-3.4.1-pyhd8ed1ab_0
comm               conda-forge/noarch::comm-0.2.2-pyhd8ed1ab_1
contourpy          conda-forge/linux-64::contourpy-1.3.1-py312h68727a3_0
cycler             conda-forge/noarch::cycler-0.12.1-pyhd8ed1ab_1
cyrus-sasl         conda-forge/linux-64::cyrus-sasl-2.1.27-h54b06d7_7
dbus               conda-forge/linux-64::dbus-1.13.6-h5008d03_3
debugpy            conda-forge/linux-64::debugpy-1.8.11-py312h2ec8cdc_0
decorator          conda-forge/noarch::decorator-5.1.1-pyhd8ed1ab_1
defusedxml         conda-forge/noarch::defusedxml-0.7.1-pyhd8ed1ab_0
double-conversion  conda-forge/linux-64::double-conversion-3.3.0-h59595ed_0
```

```
  entrypoints        conda-forge/noarch::entrypoints-0.4-pyhd8ed1ab_1
  exceptiongroup     conda-forge/noarch::exceptiongroup-1.2.2-pyhd8ed1ab_1
  executing          conda-forge/noarch::executing-2.1.0-pyhd8ed1ab_1
  expat              conda-forge/linux-64::expat-2.6.4-h5888daf_0
  font-ttf-dejavu-s~ conda-forge/noarch::font-ttf-dejavu-sans-mono-2.37-hab24e00_0
  font-ttf-inconsol~ conda-forge/noarch::font-ttf-inconsolata-3.000-h77eed37_0
  font-ttf-source-c~ conda-forge/noarch::font-ttf-source-code-pro-2.038-h77eed37_0
  font-ttf-ubuntu    conda-forge/noarch::font-ttf-ubuntu-0.83-h77eed37_3
  fontconfig         conda-forge/linux-64::fontconfig-2.15.0-h7e30c49_1
  fonts-conda-ecosy~ conda-forge/noarch::fonts-conda-ecosystem-1-0
  fonts-conda-forge  conda-forge/noarch::fonts-conda-forge-1-0
  fonttools          conda-forge/linux-64::fonttools-4.55.3-py312h178313f_1
  fqdn               conda-forge/noarch::fqdn-1.5.1-pyhd8ed1ab_1
  freetype           conda-forge/linux-64::freetype-2.12.1-h267a509_2
  graphite2          conda-forge/linux-64::graphite2-1.3.13-h59595ed_1003
  h11                conda-forge/noarch::h11-0.14.0-pyhd8ed1ab_1
  h2                 conda-forge/noarch::h2-4.1.0-pyhd8ed1ab_1
  harfbuzz           conda-forge/linux-64::harfbuzz-10.1.0-h0b3b770_0
  hpack              conda-forge/noarch::hpack-4.0.0-pyhd8ed1ab_1
  httpcore           conda-forge/noarch::httpcore-1.0.7-pyh29332c3_1
  httpx              conda-forge/noarch::httpx-0.28.1-pyhd8ed1ab_0
  hyperframe         conda-forge/noarch::hyperframe-6.0.1-pyhd8ed1ab_1
  icu                conda-forge/linux-64::icu-75.1-he02047a_0
  idna               conda-forge/noarch::idna-3.10-pyhd8ed1ab_1
  importlib-metadata conda-forge/noarch::importlib-metadata-8.5.0-pyha770c72_1
  importlib_resourc~ conda-forge/noarch::importlib_resources-6.5.2-pyhd8ed1ab_0
  ipykernel          conda-forge/noarch::ipykernel-6.29.5-pyh3099207_0
  ipython            conda-forge/noarch::ipython-8.31.0-pyh707e725_0
  ipywidgets         conda-forge/noarch::ipywidgets-8.1.5-pyhd8ed1ab_1
  isoduration        conda-forge/noarch::isoduration-20.11.0-pyhd8ed1ab_1
  jedi               conda-forge/noarch::jedi-0.19.2-pyhd8ed1ab_1
  jinja2             conda-forge/noarch::jinja2-3.1.5-pyhd8ed1ab_0
  json5              conda-forge/noarch::json5-0.10.0-pyhd8ed1ab_1
  jsonpointer        conda-forge/linux-64::jsonpointer-3.0.0-py312h7900ff3_1
  jsonschema         conda-forge/noarch::jsonschema-4.23.0-pyhd8ed1ab_1
  jsonschema-specif~ conda-forge/noarch::jsonschema-specifications-2024.10.1-pyhd8ed1ab_1
  jsonschema-with-f~ conda-forge/noarch::jsonschema-with-format-nongpl-4.23.0-hd8ed1ab_1
  jupyter            conda-forge/noarch::jupyter-1.1.1-pyhd8ed1ab_1
  jupyter-lsp        conda-forge/noarch::jupyter-lsp-2.2.5-pyhd8ed1ab_1
  jupyter_client     conda-forge/noarch::jupyter_client-8.6.3-pyhd8ed1ab_1
  jupyter_console    conda-forge/noarch::jupyter_console-6.6.3-pyhd8ed1ab_1
  jupyter_core       conda-forge/noarch::jupyter_core-5.7.2-pyh31011fe_1
  jupyter_events     conda-forge/noarch::jupyter_events-0.11.0-pyhd8ed1ab_0
  jupyter_server     conda-forge/noarch::jupyter_server-2.15.0-pyhd8ed1ab_0
  jupyter_server_te~ conda-forge/noarch::jupyter_server_terminals-0.5.3-pyhd8ed1ab_1
  jupyterlab         conda-forge/noarch::jupyterlab-4.3.4-pyhd8ed1ab_0
  jupyterlab_pygmen~ conda-forge/noarch::jupyterlab_pygments-0.3.0-pyhd8ed1ab_2
  jupyterlab_server  conda-forge/noarch::jupyterlab_server-2.27.3-pyhd8ed1ab_1
  jupyterlab_widgets conda-forge/noarch::jupyterlab_widgets-3.0.13-pyhd8ed1ab_1
  keyutils           conda-forge/linux-64::keyutils-1.6.1-h166bdaf_0
  kiwisolver         conda-forge/linux-64::kiwisolver-1.4.7-py312h68727a3_0
  krb5               conda-forge/linux-64::krb5-1.21.3-h659f571_0
  lcms2              conda-forge/linux-64::lcms2-2.16-hb7c19ff_0
  lerc               conda-forge/linux-64::lerc-4.0.0-h27087fc_0
  libblas            conda-forge/linux-64::libblas-3.9.0-26_linux64_openblas
  libbrotlicommon    conda-forge/linux-64::libbrotlicommon-1.1.0-hb9d3cd8_2
  libbrotlidec       conda-forge/linux-64::libbrotlidec-1.1.0-hb9d3cd8_2
  libbrotlienc       conda-forge/linux-64::libbrotlienc-1.1.0-hb9d3cd8_2
  libcblas           conda-forge/linux-64::libcblas-3.9.0-26_linux64_openblas
  libclang-cpp19.1   conda-forge/linux-64::libclang-cpp19.1-19.1.6-default_hb5137d0_0
  libclang13         conda-forge/linux-64::libclang13-19.1.6-default_h9c6a7e4_0
  libcups            conda-forge/linux-64::libcups-2.3.3-h4637d8d_4
  libdeflate         conda-forge/linux-64::libdeflate-1.23-h4ddbbb0_0
  libdrm             conda-forge/linux-64::libdrm-2.4.124-hb9d3cd8_0
```

```
libedit            conda-forge/linux-64::libedit-3.1.20240808-pl5321h7949ede_0
libegl             conda-forge/linux-64::libegl-1.7.0-ha4b6fd6_2
libgfortran        conda-forge/linux-64::libgfortran-14.2.0-h69a702a_1
libgfortran5       conda-forge/linux-64::libgfortran5-14.2.0-hd5240d6_1
libgl              conda-forge/linux-64::libgl-1.7.0-ha4b6fd6_2
libglib            conda-forge/linux-64::libglib-2.82.2-h2ff4ddf_0
libglvnd           conda-forge/linux-64::libglvnd-1.7.0-ha4b6fd6_2
libglx             conda-forge/linux-64::libglx-1.7.0-ha4b6fd6_2
libiconv           conda-forge/linux-64::libiconv-1.17-hd590300_2
libjpeg-turbo      conda-forge/linux-64::libjpeg-turbo-3.0.0-hd590300_1
liblapack          conda-forge/linux-64::liblapack-3.9.0-26_linux64_openblas
libllvm19          conda-forge/linux-64::libllvm19-19.1.6-ha7bfdaf_0
libntlm            conda-forge/linux-64::libntlm-1.8-hb9d3cd8_0
libopenblas        conda-forge/linux-64::libopenblas-0.3.28-pthreads_h94d23a6_1
libopengl          conda-forge/linux-64::libopengl-1.7.0-ha4b6fd6_2
libpciaccess       conda-forge/linux-64::libpciaccess-0.18-hd590300_0
libpng             conda-forge/linux-64::libpng-1.6.45-h943b412_0
libpq              conda-forge/linux-64::libpq-17.2-h3b95a9b_1
libsodium          conda-forge/linux-64::libsodium-1.0.20-h4ab18f5_0
libstdcxx          conda-forge/linux-64::libstdcxx-14.2.0-hc0a3c3a_1
libstdcxx-ng       conda-forge/linux-64::libstdcxx-ng-14.2.0-h4852527_1
libtiff            conda-forge/linux-64::libtiff-4.7.0-hd9ff511_3
libwebp-base       conda-forge/linux-64::libwebp-base-1.5.0-h851e524_0
libxcb             conda-forge/linux-64::libxcb-1.17.0-h8a09558_0
libxkbcommon       conda-forge/linux-64::libxkbcommon-1.7.0-h2c5496b_1
libxml2            conda-forge/linux-64::libxml2-2.13.5-h8d12d68_1
libxslt            conda-forge/linux-64::libxslt-1.1.39-h76b75d6_0
markupsafe         conda-forge/linux-64::markupsafe-3.0.2-py312h178313f_1
matplotlib         conda-forge/linux-64::matplotlib-3.10.0-py312h7900ff3_0
matplotlib-base    conda-forge/linux-64::matplotlib-base-3.10.0-py312hd3ec401_0
matplotlib-inline  conda-forge/noarch::matplotlib-inline-0.1.7-pyhd8ed1ab_1
mistune            conda-forge/noarch::mistune-3.1.0-pyhd8ed1ab_0
munkres            conda-forge/noarch::munkres-1.1.4-pyh9f0ad1d_0
mysql-common       conda-forge/linux-64::mysql-common-9.0.1-h266115a_4
mysql-libs         conda-forge/linux-64::mysql-libs-9.0.1-he0572af_4
nbclient           conda-forge/noarch::nbclient-0.10.2-pyhd8ed1ab_0
nbconvert-core     conda-forge/noarch::nbconvert-core-7.16.5-pyhd8ed1ab_1
nbformat           conda-forge/noarch::nbformat-5.10.4-pyhd8ed1ab_1
nest-asyncio       conda-forge/noarch::nest-asyncio-1.6.0-pyhd8ed1ab_1
notebook           conda-forge/noarch::notebook-7.3.2-pyhd8ed1ab_0
notebook-shim      conda-forge/noarch::notebook-shim-0.2.4-pyhd8ed1ab_1
numpy              conda-forge/linux-64::numpy-2.2.1-py312h7e784f5_0
openjpeg           conda-forge/linux-64::openjpeg-2.5.3-h5fbd93e_0
openldap           conda-forge/linux-64::openldap-2.6.9-he970967_0
overrides          conda-forge/noarch::overrides-7.7.0-pyhd8ed1ab_1
packaging          conda-forge/noarch::packaging-24.2-pyhd8ed1ab_2
pandas             conda-forge/linux-64::pandas-2.2.3-py312hf9745cd_1
pandocfilters      conda-forge/noarch::pandocfilters-1.5.0-pyhd8ed1ab_0
parso              conda-forge/noarch::parso-0.8.4-pyhd8ed1ab_1
pcre2              conda-forge/linux-64::pcre2-10.44-hba22ea6_2
pexpect            conda-forge/noarch::pexpect-4.9.0-pyhd8ed1ab_1
pickleshare        conda-forge/noarch::pickleshare-0.7.5-pyhd8ed1ab_1004
pillow             conda-forge/linux-64::pillow-11.1.0-py312h80c1187_0
pixman             conda-forge/linux-64::pixman-0.44.2-h29eaf8c_0
pkgutil-resolve-n~ conda-forge/noarch::pkgutil-resolve-name-1.3.10-pyhd8ed1ab_2
platformdirs       conda-forge/noarch::platformdirs-4.3.6-pyhd8ed1ab_1
prometheus_client  conda-forge/noarch::prometheus_client-0.21.1-pyhd8ed1ab_0
prompt-toolkit     conda-forge/noarch::prompt-toolkit-3.0.48-pyha770c72_1
prompt_toolkit     conda-forge/noarch::prompt_toolkit-3.0.48-hd8ed1ab_1
psutil             conda-forge/linux-64::psutil-6.1.1-py312h66e93f0_0
pthread-stubs      conda-forge/linux-64::pthread-stubs-0.4-hb9d3cd8_1002
ptyprocess         conda-forge/noarch::ptyprocess-0.7.0-pyhd8ed1ab_1
pure_eval          conda-forge/noarch::pure_eval-0.2.3-pyhd8ed1ab_1
pycparser          conda-forge/noarch::pycparser-2.22-pyh29332c3_1
```

```
pygments           conda-forge/noarch::pygments-2.19.1-pyhd8ed1ab_0
pyparsing          conda-forge/noarch::pyparsing-3.2.1-pyhd8ed1ab_0
pyside6            conda-forge/linux-64::pyside6-6.8.1-py312h91f0f75_0
pysocks            conda-forge/noarch::pysocks-1.7.1-pyha55dd90_7
python-dateutil    conda-forge/noarch::python-dateutil-2.9.0.post0-pyhff2d567_1
python-fastjsonsc~ conda-forge/noarch::python-fastjsonschema-2.21.1-pyhd8ed1ab_0
python-json-logger conda-forge/noarch::python-json-logger-2.0.7-pyhd8ed1ab_0
python-tzdata      conda-forge/noarch::python-tzdata-2024.2-pyhd8ed1ab_1
python_abi         conda-forge/linux-64::python_abi-3.12-5_cp312
pytz               conda-forge/noarch::pytz-2024.1-pyhd8ed1ab_0
pyyaml             conda-forge/linux-64::pyyaml-6.0.2-py312h66e93f0_1
pyzmq              conda-forge/linux-64::pyzmq-26.2.0-py312hbf22597_3
qhull              conda-forge/linux-64::qhull-2020.2-h434a139_5
qt6-main           conda-forge/linux-64::qt6-main-6.8.1-h588cce1_2
referencing        conda-forge/noarch::referencing-0.35.1-pyhd8ed1ab_1
requests           conda-forge/noarch::requests-2.32.3-pyhd8ed1ab_1
rfc3339-validator  conda-forge/noarch::rfc3339-validator-0.1.4-pyhd8ed1ab_1
rfc3986-validator  conda-forge/noarch::rfc3986-validator-0.1.1-pyh9f0ad1d_0
rpds-py            conda-forge/linux-64::rpds-py-0.22.3-py312h12e396e_0
send2trash         conda-forge/noarch::send2trash-1.8.3-pyh0d859eb_1
six                conda-forge/noarch::six-1.17.0-pyhd8ed1ab_0
sniffio            conda-forge/noarch::sniffio-1.3.1-pyhd8ed1ab_1
soupsieve          conda-forge/noarch::soupsieve-2.5-pyhd8ed1ab_1
stack_data         conda-forge/noarch::stack_data-0.6.3-pyhd8ed1ab_1
terminado          conda-forge/noarch::terminado-0.18.1-pyh0d859eb_0
tinycss2           conda-forge/noarch::tinycss2-1.4.0-pyhd8ed1ab_0
tomli              conda-forge/noarch::tomli-2.2.1-pyhd8ed1ab_1
tornado            conda-forge/linux-64::tornado-6.4.2-py312h66e93f0_0
traitlets          conda-forge/noarch::traitlets-5.14.3-pyhd8ed1ab_1
types-python-date~ conda-forge/noarch::types-python-dateutil-2.9.0.20241206-pyhd8ed1ab_0
typing-extensions  conda-forge/noarch::typing-extensions-4.12.2-hd8ed1ab_1
typing_extensions  conda-forge/noarch::typing_extensions-4.12.2-pyha770c72_1
typing_utils       conda-forge/noarch::typing_utils-0.1.0-pyhd8ed1ab_1
unicodedata2       conda-forge/linux-64::unicodedata2-15.1.0-py312h66e93f0_1
uri-template       conda-forge/noarch::uri-template-1.3.0-pyhd8ed1ab_1
urllib3            conda-forge/noarch::urllib3-2.3.0-pyhd8ed1ab_0
wayland            conda-forge/linux-64::wayland-1.23.1-h3e06ad9_0
wcwidth            conda-forge/noarch::wcwidth-0.2.13-pyhd8ed1ab_1
webcolors          conda-forge/noarch::webcolors-24.11.1-pyhd8ed1ab_0
webencodings       conda-forge/noarch::webencodings-0.5.1-pyhd8ed1ab_3
websocket-client   conda-forge/noarch::websocket-client-1.8.0-pyhd8ed1ab_1
widgetsnbextension conda-forge/noarch::widgetsnbextension-4.0.13-pyhd8ed1ab_1
xcb-util           conda-forge/linux-64::xcb-util-0.4.1-hb711507_2
xcb-util-cursor    conda-forge/linux-64::xcb-util-cursor-0.1.5-hb9d3cd8_0
xcb-util-image     conda-forge/linux-64::xcb-util-image-0.4.0-hb711507_2
xcb-util-keysyms   conda-forge/linux-64::xcb-util-keysyms-0.4.1-hb711507_0
xcb-util-renderut~ conda-forge/linux-64::xcb-util-renderutil-0.3.10-hb711507_0
xcb-util-wm        conda-forge/linux-64::xcb-util-wm-0.4.2-hb711507_0
xkeyboard-config   conda-forge/linux-64::xkeyboard-config-2.43-hb9d3cd8_0
xorg-libice        conda-forge/linux-64::xorg-libice-1.1.2-hb9d3cd8_0
xorg-libsm         conda-forge/linux-64::xorg-libsm-1.2.5-he73a12e_0
xorg-libx11        conda-forge/linux-64::xorg-libx11-1.8.10-h4f16b4b_1
xorg-libxau        conda-forge/linux-64::xorg-libxau-1.0.12-hb9d3cd8_0
xorg-libxcomposite conda-forge/linux-64::xorg-libxcomposite-0.4.6-hb9d3cd8_2
xorg-libxcursor    conda-forge/linux-64::xorg-libxcursor-1.2.3-hb9d3cd8_0
xorg-libxdamage    conda-forge/linux-64::xorg-libxdamage-1.1.6-hb9d3cd8_0
xorg-libxdmcp      conda-forge/linux-64::xorg-libxdmcp-1.1.5-hb9d3cd8_0
xorg-libxext       conda-forge/linux-64::xorg-libxext-1.3.6-hb9d3cd8_0
xorg-libxfixes     conda-forge/linux-64::xorg-libxfixes-6.0.1-hb9d3cd8_0
xorg-libxi         conda-forge/linux-64::xorg-libxi-1.8.2-hb9d3cd8_0
xorg-libxrandr     conda-forge/linux-64::xorg-libxrandr-1.5.4-hb9d3cd8_0
xorg-libxrender    conda-forge/linux-64::xorg-libxrender-0.9.12-hb9d3cd8_0
xorg-libxtst       conda-forge/linux-64::xorg-libxtst-1.2.5-hb9d3cd8_3
xorg-libxxf86vm    conda-forge/linux-64::xorg-libxxf86vm-1.1.6-hb9d3cd8_0
```

```
yaml               conda-forge/linux-64::yaml-0.2.5-h7f98852_2
zeromq             conda-forge/linux-64::zeromq-4.3.5-h3b0a872_7
zipp               conda-forge/noarch::zipp-3.21.0-pyhd8ed1ab_1
zstandard          conda-forge/linux-64::zstandard-0.23.0-py312hef9b889_1
zstd               conda-forge/linux-64::zstd-1.5.6-ha6fb4c9_0



Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Will be using some other packages, too, but these can be installed later once they are needed. There are 2 ways to install packages: with `conda install` & with `pip install`. `conda install` should always be preferred when using Miniconda, but some packages are not available through conda, so if `conda install $package_name` fails, try `pip install $package_name`.

Remark 4. *If want to install all of packages used in rest of book, can do that now by running:*

```
(pydata-book) nqbh@nqbh-dell:~$ conda install lxml beautifulsoup4 html5lib openpyxl \
requests sqlalchemy seaborn scipy statsmodels \
patsy scikit-learn pyarrow pytables numba
Channels:
- conda-forge
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

environment location: /home/nqbh/anaconda3/envs/pydata-book

added / updated specs:
- beautifulsoup4
- html5lib
- lxml
- numba
- openpyxl
- patsy
- pyarrow
- pytables
- requests
- scikit-learn
- scipy
- seaborn
- sqlalchemy
- statsmodels


The following packages will be downloaded:

package                    |            build
---------------------------|-----------------
aws-c-auth-0.8.0           |      hb921021_15         105 KB  conda-forge
aws-c-cal-0.8.1            |       h1a47875_3          46 KB  conda-forge
aws-c-common-0.10.6        |       hb9d3cd8_0         231 KB  conda-forge
aws-c-compression-0.3.0    |       h4e1184b_5          19 KB  conda-forge
aws-c-event-stream-0.5.0   |      h7959bf6_11          53 KB  conda-forge
aws-c-http-0.9.2           |       hefd7a92_4         193 KB  conda-forge
aws-c-io-0.15.3            |       h831e299_5         154 KB  conda-forge
aws-c-mqtt-0.11.0          |      h11f4f37_12         190 KB  conda-forge
aws-c-s3-0.7.7             |       hf454442_0         111 KB  conda-forge
```

```
aws-c-sdkutils-0.2.1       |       h4e1184b_4          55 KB  conda-forge
aws-checksums-0.2.2        |       h4e1184b_4          71 KB  conda-forge
aws-crt-cpp-0.29.7         |       hd92328a_7         346 KB  conda-forge
aws-sdk-cpp-1.11.458       |       hc430e4a_4         2.9 MB  conda-forge
azure-core-cpp-1.14.0      |       h5cfcd09_0         337 KB  conda-forge
azure-identity-cpp-1.10.0  |       h113e628_0         227 KB  conda-forge
azure-storage-blobs-cpp-12.13.0|       h3cf044e_1         536 KB  conda-forge
azure-storage-common-cpp-12.8.0|       h736e048_1         146 KB  conda-forge
azure-storage-files-datalake-cpp-12.12.0|       ha633028_1         281 KB  conda-forge
blosc-1.21.6               |       he440d0b_1          47 KB  conda-forge
c-ares-1.34.4              |       hb9d3cd8_0         201 KB  conda-forge
c-blosc2-2.15.2            |       h3122c55_1         334 KB  conda-forge
et_xmlfile-2.0.0           |     pyhd8ed1ab_1          21 KB  conda-forge
gflags-2.2.2               |    h5888daf_1005         117 KB  conda-forge
glog-0.7.1                 |       hbabe93e_0         140 KB  conda-forge
greenlet-3.1.1             |  py312h2ec8cdc_1         232 KB  conda-forge
hdf5-1.14.4                |nompi_h2d575fe_105         3.8 MB  conda-forge
html5lib-1.1               |     pyhd8ed1ab_2          93 KB  conda-forge
joblib-1.4.2               |     pyhd8ed1ab_1         215 KB  conda-forge
libabseil-20240722.0       | cxx17_hbbce691_4         1.3 MB  conda-forge
libaec-1.1.3               |       h59595ed_0          35 KB  conda-forge
libarrow-18.1.0            |   hd595efa_7_cpu         8.4 MB  conda-forge
libarrow-acero-18.1.0      |   hcb10f89_7_cpu         598 KB  conda-forge
libarrow-dataset-18.1.0    |   hcb10f89_7_cpu         574 KB  conda-forge
libarrow-substrait-18.1.0  |   h08228c5_7_cpu         510 KB  conda-forge
libcrc32c-1.1.2            |       h9c3ff4c_0          20 KB  conda-forge
libcurl-8.11.1             |       h332b0f4_0         413 KB  conda-forge
libev-4.33                 |       hd590300_2         110 KB  conda-forge
libevent-2.1.12            |       hf998b51_1         417 KB  conda-forge
libgoogle-cloud-2.33.0     |       h2b5623c_1         1.2 MB  conda-forge
libgoogle-cloud-storage-2.33.0|       h0121fbd_1         766 KB  conda-forge
libgrpc-1.67.1             |       h25350d4_1         7.4 MB  conda-forge
libllvm14-14.0.6           |       hcd5def8_4        30.0 MB  conda-forge
libnghttp2-1.64.0          |       h161d5f1_0         632 KB  conda-forge
libparquet-18.1.0          |   h081d1f1_7_cpu         1.1 MB  conda-forge
libprotobuf-5.28.3         |       h6128344_1         2.8 MB  conda-forge
libre2-11-2024.07.02       |       hbbce691_2         205 KB  conda-forge
libssh2-1.11.1             |       hf672d98_0         297 KB  conda-forge
libthrift-0.21.0           |       h0e7cc3e_0         416 KB  conda-forge
libutf8proc-2.9.0          |       hb9d3cd8_1          80 KB  conda-forge
llvmlite-0.43.0            |  py312h374181b_1         3.3 MB  conda-forge
lxml-5.3.0                 |  py312he28fd5a_2         1.3 MB  conda-forge
lz4-c-1.10.0               |       h5888daf_1         163 KB  conda-forge
nomkl-1.0                  |       h5ca1d4c_0           4 KB  conda-forge
numba-0.60.0               |  py312h83e6fd3_0         5.4 MB  conda-forge
numexpr-2.10.2             |py312h6a710ac_100         191 KB  conda-forge
numpy-2.0.2                |  py312h58c1407_1         8.1 MB  conda-forge
openpyxl-3.1.5             |  py312h710cb58_1         680 KB  conda-forge
orc-2.0.3                  |       h12ee42a_2         1.1 MB  conda-forge
patsy-1.0.1                |     pyhd8ed1ab_1         182 KB  conda-forge
py-cpuinfo-9.0.0           |     pyhd8ed1ab_1          25 KB  conda-forge
pyarrow-18.1.0             |  py312h7900ff3_0          25 KB  conda-forge
pyarrow-core-18.1.0        |py312h01725c0_0_cpu         4.4 MB  conda-forge
pytables-3.10.2            |  py312hf8651a9_0         1.6 MB  conda-forge
re2-2024.07.02             |       h9925aae_2          26 KB  conda-forge
s2n-1.5.10                 |       hb5b8611_0         347 KB  conda-forge
scikit-learn-1.6.0         |  py312h7a48858_0        10.0 MB  conda-forge
scipy-1.15.0               |  py312h180e4f1_1        18.2 MB  conda-forge
seaborn-0.13.2             |       hd8ed1ab_3           7 KB  conda-forge
seaborn-base-0.13.2        |     pyhd8ed1ab_3         223 KB  conda-forge
snappy-1.2.1               |       h8bd8927_1          42 KB  conda-forge
sqlalchemy-2.0.36          |  py312h66e93f0_0         3.3 MB  conda-forge
statsmodels-0.14.4         |  py312hc0a28a1_0        11.5 MB  conda-forge
threadpoolctl-3.5.0        |     pyhc1e730c_0          23 KB  conda-forge
```

```
zlib-ng-2.2.3              |      h7955e40_0         106 KB  conda-forge
------------------------------------------------------------
Total:       138.7 MB

The following NEW packages will be INSTALLED:

aws-c-auth          conda-forge/linux-64::aws-c-auth-0.8.0-hb921021_15
aws-c-cal           conda-forge/linux-64::aws-c-cal-0.8.1-h1a47875_3
aws-c-common        conda-forge/linux-64::aws-c-common-0.10.6-hb9d3cd8_0
aws-c-compression   conda-forge/linux-64::aws-c-compression-0.3.0-h4e1184b_5
aws-c-event-stream conda-forge/linux-64::aws-c-event-stream-0.5.0-h7959bf6_11
aws-c-http          conda-forge/linux-64::aws-c-http-0.9.2-hefd7a92_4
aws-c-io            conda-forge/linux-64::aws-c-io-0.15.3-h831e299_5
aws-c-mqtt          conda-forge/linux-64::aws-c-mqtt-0.11.0-h11f4f37_12
aws-c-s3            conda-forge/linux-64::aws-c-s3-0.7.7-hf454442_0
aws-c-sdkutils      conda-forge/linux-64::aws-c-sdkutils-0.2.1-h4e1184b_4
aws-checksums       conda-forge/linux-64::aws-checksums-0.2.2-h4e1184b_4
aws-crt-cpp         conda-forge/linux-64::aws-crt-cpp-0.29.7-hd92328a_7
aws-sdk-cpp         conda-forge/linux-64::aws-sdk-cpp-1.11.458-hc430e4a_4
azure-core-cpp      conda-forge/linux-64::azure-core-cpp-1.14.0-h5cfcd09_0
azure-identity-cpp conda-forge/linux-64::azure-identity-cpp-1.10.0-h113e628_0
azure-storage-blo~ conda-forge/linux-64::azure-storage-blobs-cpp-12.13.0-h3cf044e_1
azure-storage-com~ conda-forge/linux-64::azure-storage-common-cpp-12.8.0-h736e048_1
azure-storage-fil~ conda-forge/linux-64::azure-storage-files-datalake-cpp-12.12.0-ha633028_1
blosc               conda-forge/linux-64::blosc-1.21.6-he440d0b_1
c-ares              conda-forge/linux-64::c-ares-1.34.4-hb9d3cd8_0
c-blosc2            conda-forge/linux-64::c-blosc2-2.15.2-h3122c55_1
et_xmlfile          conda-forge/noarch::et_xmlfile-2.0.0-pyhd8ed1ab_1
gflags              conda-forge/linux-64::gflags-2.2.2-h5888daf_1005
glog                conda-forge/linux-64::glog-0.7.1-hbabe93e_0
greenlet            conda-forge/linux-64::greenlet-3.1.1-py312h2ec8cdc_1
hdf5                conda-forge/linux-64::hdf5-1.14.4-nompi_h2d575fe_105
html5lib            conda-forge/noarch::html5lib-1.1-pyhd8ed1ab_2
joblib              conda-forge/noarch::joblib-1.4.2-pyhd8ed1ab_1
libabseil           conda-forge/linux-64::libabseil-20240722.0-cxx17_hbbce691_4
libaec              conda-forge/linux-64::libaec-1.1.3-h59595ed_0
libarrow            conda-forge/linux-64::libarrow-18.1.0-hd595efa_7_cpu
libarrow-acero      conda-forge/linux-64::libarrow-acero-18.1.0-hcb10f89_7_cpu
libarrow-dataset    conda-forge/linux-64::libarrow-dataset-18.1.0-hcb10f89_7_cpu
libarrow-substrait conda-forge/linux-64::libarrow-substrait-18.1.0-h08228c5_7_cpu
libcrc32c           conda-forge/linux-64::libcrc32c-1.1.2-h9c3ff4c_0
libcurl             conda-forge/linux-64::libcurl-8.11.1-h332b0f4_0
libev               conda-forge/linux-64::libev-4.33-hd590300_2
libevent            conda-forge/linux-64::libevent-2.1.12-hf998b51_1
libgoogle-cloud     conda-forge/linux-64::libgoogle-cloud-2.33.0-h2b5623c_1
libgoogle-cloud-s~ conda-forge/linux-64::libgoogle-cloud-storage-2.33.0-h0121fbd_1
libgrpc             conda-forge/linux-64::libgrpc-1.67.1-h25350d4_1
libllvm14           conda-forge/linux-64::libllvm14-14.0.6-hcd5def8_4
libnghttp2          conda-forge/linux-64::libnghttp2-1.64.0-h161d5f1_0
libparquet          conda-forge/linux-64::libparquet-18.1.0-h081d1f1_7_cpu
libprotobuf         conda-forge/linux-64::libprotobuf-5.28.3-h6128344_1
libre2-11           conda-forge/linux-64::libre2-11-2024.07.02-hbbce691_2
libssh2             conda-forge/linux-64::libssh2-1.11.1-hf672d98_0
libthrift           conda-forge/linux-64::libthrift-0.21.0-h0e7cc3e_0
libutf8proc         conda-forge/linux-64::libutf8proc-2.9.0-hb9d3cd8_1
llvmlite            conda-forge/linux-64::llvmlite-0.43.0-py312h374181b_1
lxml                conda-forge/linux-64::lxml-5.3.0-py312he28fd5a_2
lz4-c               conda-forge/linux-64::lz4-c-1.10.0-h5888daf_1
nomkl               conda-forge/noarch::nomkl-1.0-h5ca1d4c_0
numba               conda-forge/linux-64::numba-0.60.0-py312h83e6fd3_0
numexpr             conda-forge/linux-64::numexpr-2.10.2-py312h6a710ac_100
openpyxl            conda-forge/linux-64::openpyxl-3.1.5-py312h710cb58_1
orc                 conda-forge/linux-64::orc-2.0.3-h12ee42a_2
patsy               conda-forge/noarch::patsy-1.0.1-pyhd8ed1ab_1
```

```
py-cpuinfo         conda-forge/noarch::py-cpuinfo-9.0.0-pyhd8ed1ab_1
pyarrow            conda-forge/linux-64::pyarrow-18.1.0-py312h7900ff3_0
pyarrow-core       conda-forge/linux-64::pyarrow-core-18.1.0-py312h01725c0_0_cpu
pytables           conda-forge/linux-64::pytables-3.10.2-py312hf8651a9_0
re2                conda-forge/linux-64::re2-2024.07.02-h9925aae_2
s2n                conda-forge/linux-64::s2n-1.5.10-hb5b8611_0
scikit-learn       conda-forge/linux-64::scikit-learn-1.6.0-py312h7a48858_0
scipy              conda-forge/linux-64::scipy-1.15.0-py312h180e4f1_1
seaborn            conda-forge/noarch::seaborn-0.13.2-hd8ed1ab_3
seaborn-base       conda-forge/noarch::seaborn-base-0.13.2-pyhd8ed1ab_3
snappy             conda-forge/linux-64::snappy-1.2.1-h8bd8927_1
sqlalchemy         conda-forge/linux-64::sqlalchemy-2.0.36-py312h66e93f0_0
statsmodels        conda-forge/linux-64::statsmodels-0.14.4-py312hc0a28a1_0
threadpoolctl      conda-forge/noarch::threadpoolctl-3.5.0-pyhc1e730c_0
zlib-ng            conda-forge/linux-64::zlib-ng-2.2.3-h7955e40_0

The following packages will be DOWNGRADED:

numpy                             2.2.1-py312h7e784f5_0 --> 2.0.2-py312h58c1407_1


Proceed ([y]/n)? y


Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Question 1 (Downgrade NumPy)**.** *Why downgrade* `NumPy` *version?*

On Windows, substitute a carat ^ for line continuation \ used on Linux & macOS.

Can update packages by using `conda update` command:

```
conda update package_name
```

`pip` also supports upgrades using `-upgrade` flag:

```
pip install --upgrade package_name
```

Have several opportunities to try out these commands throughout book.

Remark 5. *While can use both* `conda` *&* `pip` *to install packages, should avoid updating packages originally installed with conda using pip (& vice versa), as doing so can lead to environment problems. Recommend sticking to conda if can & falling back on pip only for packages that are unavailable with* `conda install`*.*

* Integrated Development Environments & Text Editors. When asked about standard development environment, almost always says "IPython plus a text editor." Typically write a program & iteratively test & debug each piece of it in IPython or Jupyter notebooks. Also useful to be able to play around with data interactively & visually verify that a particular set of data manipulations is doing right thing. Libraries like `pandas` & `NumPy` are designed to be productive to use in shell.

  When building software, however, some users may prefer to use a more richly featured integrated development environment (IDE) & rather than an editor like Emacs or Vim which provide a more minimal environment out of box. Some that you can explore:

  · PyDev (free), an IDE built on Eclipse platform
  · PyCharm from JetBrains (subscription-based for commercial users, free for open source developers)
  · Python Tools for Visual Studio (for Windows users)
  · Spyder (free), an IDE currently shipped with Anaconda
  · Komodo IDE (commercial)

  Due to popularity of Python, most text editors, like VS Code & Sublime Text 2, have excellent Python support.

◦ 1.5. Community & Conferences. Outside of an internet search, various scientific & data-related Python mailing lists are generally helpful & responsive to questions. Some to take a look at include:

* pydata: A Google Group list for questions related to Python for data analysis & pandas
* pystatsmodels: For statsmodels or pandas-related questions
* Mailing list for scikit-learn `scikit-learn@python.org` & ML in Python, generally

* numpy-discussion: For NumPy-related questions
* scipy-user: For general SciPy or scientific Python questions

Deliberately did not post URLs for these in case they change. They can be easily located via an internet search.

Each year many conferences are held all over world for Python programmers. If would like to connect with other Python programmers who share interests, encourage to explore attending one, if possible. Many conferences have financial support available for those who cannot afford admission or travel to conference. Some to consider:

* PyCon & EuroPython: 2 main general Python conferences in North America & Europe, resp.
* SciPy & EuroSciPy: Scientific-computing-oriented conferences in North America & Europe, resp.
* SciPy & EuroSciPy: Scientific-computing-oriented conferences in North America & Europe, resp.
* PyData: A worldwide series of regional conferences a targeted at DS & data analysis use cases
* International & regional PyCon conferences (see https://pycon.org for a complete listing)

- 1.6. Navigating This Book. If have never programmed in Python before, will want to spend some time in Chaps. 2–3, where have placed a condensed tutorial on Python language features & IPython shell & Jupyter notebooks. These things are prerequisite knowledge for remainder of book. If have Python experience already, may instead choose to skim or skip these chaps.

  Next, give a short introduction to key features of NumPy, leaving more advanced `NumPy` use for Appendix A. Then, introduce `pandas` & devote rest of book to data analysis topics applying `pandas, NumPy, matplotlib` (for visualization). Have structured material in an incremental fashion, though there is occasionally some minor crossover between chaps, with a few cases where concepts are used that haven't been introduced yet.

  While readers may have many different end goals for their work, tasks required generally fall into a number of different broad groups:

  * *Interacting with outside world*: Reading & writing with a variety of file formats & data stores
  * *Preparation*: Cleaning, munging, combining, normalizing, reshaping, slicing & dicing, & transforming data for analysis
  * *Transformation*: Applying mathematical & statistical operations to groups of datasets to derive new datasets (e.g., aggregating a large table by group variables)
  * *Modeling & computation*: Connecting your data to statistical models, ML algorithms, or other computational tools
  * *Presentation*: Creating interactive or static graphical visualizations or textual summaries

  * Code Examples. Most of code examples in book are shown with input & output as it should appear executed in IPython shell or in Jupyter notebooks:

    ```
    In [5]: CODE EXAMPLE
    Out[5]: OUTPUT
    ```

    When see a code example like this, intent is for you to type example code in `In` block in your coding environment & execute it by pressing `Enter` key (or `Shift-Enter` in Jupyter). Should see output similar to what is shown in `Out` block. Changed default console output settings in `NumPy` & `pandas` to improve readability & brevity throughout book. E.g., may see more digits of precision printed in numeric data. To exactly match output shown in book, can execute following Python code before running code examples:

    ```
    import numpy as np
    import {\tt pandas} as pd
    pd.options.display.max_columns = 20
    pd.options.display.max_rows = 20
    pd.options.display.max_colwidth = 80
    np.set_printoptions(precision=4, suppress=True)
    ```

  * Data Examples. Datasets for examples in each chap are hosted in https://github.com/wesm/pydata-book (or in https://gitee.com/wesmckinn/pydata-book if cannot access GitHub). Can download this data either by using Git version control system on command line or by downloading a zip file of repository from website. If you run into problems, navigate to book website https://wesmckinney.com/book for up-to-date instructions about obtaining book materials.
    If download a zip file containing example datasets, must then fully extract contents of zip file to a directory & navigate to that directory from terminal before proceeding with running book's code examples:

    ```
    $ pwd
    /home/wesm/book-materials
    $ ls
    appa.ipynb ch05.ipynb ch09.ipynb ch13.ipynb  README.md
    ch02.ipynb ch06.ipynb ch10.ipynb COPYING     requirements.txt
    ch03.ipynb ch07.ipynb ch11.ipynb datasets
    ch04.ipynb ch08.ipynb ch12.ipynb examples
    ```

    Have made every effort to ensure: GitHub repository contains everything necessary to reproduce examples, but may have made some mistakes or omissions.

* Import Conventions. Python community has adopted a number of naming conventions for commonly used modules:

```
import numpy as np
import matplotlib.pyplot as plt
import {\tt pandas} as pd
import seaborn as sns
import statsmodels as sm
```

I.e., when see `np.arrange`, this is a reference to `arrange` function in NumPy. This is done because it's considered bad practice in Python software development to import everything `from numpy import *` from a large package like NumPy.

- 2. Python Language Basics, IPython, & Jupyter Notebooks. When wrote 1e of this book in 2011–2012, there were fewer resources available for learning about doing data analysis in Python. This was partially a chicken-&-egg problem; many libraries that we know take for granted, like `pandas, scikit-learn, statsmodels`, were comparatively immature back then. Now in 2022, there is now a growing literature on DS, data analysis, & ML, supplementing prior works on general-purpose scientific computing geared toward computational scientists, physicists, & professionals in other research fields. There are also excellent books about learning Python programming language itself & becoming an effective software engineer.

  As this book is intended as an introductory text in working with data in Python, feel valuable to have a self-contained overview of some of most important features of Python's built-in data structures & libraries from perspective of data manipulation. So, will only present roughly enough information in Chaps. 2–3 to enable you to follow along with rest of book.

  Much of this book focuses on table-based analytics & data preparation tools for working with datasets that are small enough to fit on your personal computer. to use these tools you must sometimes do some wrangling to arrange messy data into a more nicely tabular (or *structured*) form. Fortunately, Python is an ideal language for doing this. Greater your facility with Python language & its built-in data types, easier it will be for you to prepare new datasets for analysis.

  Some of tools in this book are best explored from a live IPython or Jupyter session. Once learn how to start up IPython & Jupyter session. Once learn how to start up IPython & Jupyter, recommend: follow along with examples so can experiment & try different things. As with any keyboard-driven console-like environment, developing familiarity with common commands is also part of learning curve.

  Remark 6. *There are introductory Python concepts that this chap does not cover, like classes & object-oriented programming, which may find useful in your foray ()cuộc đột kích, cướp phá, xâm lược) into data analysis in Python.*

  To deepen Python language knowledge, recommend: supplement this chap with official Python tutorial & potentially 1 of many excellent books on general-purpose Python programming. Some recommendations to get you started include:

  - *Python Cookbook, 3e, by* David Beazley, Brian K. Jones
  - *Fluent Python by* Luciano Ramalho
  - *Effective Python, 2e, by* Brett Slatkin

  - 2.1. Python Interpreter. Python is an *interpreted* language. Python interpreter runs a program by executing 1 statement at a time. Standard interactive Python interpreter can be invoked on command line with `python` command:

    ```
    (pydata-book) nqbh@nqbh-dell:~$ python
    Python 3.12.7 | packaged by conda-forge | (main, Oct  4 2024, 16:05:46) [GCC 13.3.0] on linux
    Type "help", "copyright", "credits" or "license" for more information.
    >>>
    ```

    `>>` is *prompt* after which you'll type code expressions. To exit Python interpreter, can either type `exit()` or press Ctrl-D (works on Linux & macOS only).

    Running Python programs is as simple as calling `python` with a `.py` file as its 1st argument.

    While some Python programmers execute all of their Python code in this way, those doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, web-based code notebooks originally created within IPython project. Give an introduction to using IPython & Jupyter in this chap & have included a deeper look at IPython functionality in Appendix A. When use `%run` command, IPython executes code in specified file in same process, enabling to explore results interactively when it's done:

    ```
    (pydata-book) nqbh@nqbh-dell:~$ ipython
    Python 3.12.7 | packaged by conda-forge | (main, Oct  4 2024, 16:05:46) [GCC 13.3.0]
    IPython 8.31.0 -- An enhanced Interactive Python. Type '?' for help.
    Hello world

    In [2]:
    ```

    Default IPython prompt adopts numbered `In [2]:` style, cf. standard `>>` prompt.

- 2.2. IPython Basics. Run with IPython shell & Jupyter notebook, & introduce to some of essential concepts.
  - Running IPython Shell. Can launch IPython shell on command line just like launching regular Python interpreter except with ipython command `ipython`. You can execute arbitrary Python statements by typing them & pressing Return (or Enter). When type just a variable into IPython, it renders a string representation of object:

```
In [5]: import numpy as np

In [6]: data = [np.random.standard_normal() for i in range(7)]

In [7]: data
Out[7]:
[-0.960515015233981,
0.29199995965351516,
0.656773965407049,
1.0319443387105414,
-0.15623460611892206,
-0.17214640580390445,
0.5260760636382895]
```

    1st 2 lines are Python code statements; 2nd statement creates a variable named `data` that refers to a newly created Python dictionary. Last line prints value of `data` in console.
    Many kinds of Python objects are formatted to be more readable, or *pretty-printed*, which is distinct from normal printing with `print`. If printed above `data` variable in standard Python interpreter, it would be much less readable:

```
>>> import numpy as np
>>> data = [np.random.standard_normal() for i in range(7)]
>>> print(data)
>>> data
[-0.5767699931966723, -0.1010317773535111, -1.7841005313329152,
-1.524392126408841, 0.22191374220117385, -1.9835710588082562,
-1.6081963964963528]
```

    IPython also provides facilities to execute arbitrary blocks of code (via a somewhat glorified copy-&-paste approach) & whole Python scripts. Can also use Jupyter notebook to work with larger blocks of code.
  - Running Jupyter Notebook. 1 of major components of Jupyter project is *notebook*, a type of interactive document for code, text (including Markdown), data visualizations, & other output. Jupyter notebook interacts with *kernels*, which are implementations of Jupyter interactive computing protocol specific to different programming languages. Python Jupyter kernel uses IPython system for its underlying behavior.
    To start up Jupyter, run command `jupyter notebook` in a terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4bb443a63f2d...
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
To access the notebook, open this file in a browser:
file:///home/wesm/.local/share/jupyter/runtime/nbserver-185259-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
or http://127.0.0.1:8888/?token=0a77b52fefe52ab83e3c35dff8de121e4...
```

    NQBH's:

```
(base) nqbh@nqbh-dell:~$ conda activate pydata-book
(pydata-book) nqbh@nqbh-dell:~$ jupyter notebook
[I 2025-01-10 15:13:58.486 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2025-01-10 15:13:58.488 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2025-01-10 15:13:58.491 ServerApp] jupyterlab | extension was successfully linked.
[I 2025-01-10 15:13:58.493 ServerApp] notebook | extension was successfully linked.
[I 2025-01-10 15:13:58.610 ServerApp] notebook_shim | extension was successfully linked.
[I 2025-01-10 15:13:58.622 ServerApp] notebook_shim | extension was successfully loaded.
[I 2025-01-10 15:13:58.623 ServerApp] jupyter_lsp | extension was successfully loaded.
```

```
[I 2025-01-10 15:13:58.624 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2025-01-10 15:13:58.624 LabApp] JupyterLab extension loaded from /home/nqbh/anaconda3/envs/pydata-b
[I 2025-01-10 15:13:58.624 LabApp] JupyterLab application directory is /home/nqbh/anaconda3/envs/pydat
[I 2025-01-10 15:13:58.624 LabApp] Extension Manager is 'pypi'.
[I 2025-01-10 15:13:58.661 ServerApp] jupyterlab | extension was successfully loaded.
[I 2025-01-10 15:13:58.663 ServerApp] notebook | extension was successfully loaded.
[I 2025-01-10 15:13:58.663 ServerApp] Serving notebooks from local directory: /home/nqbh
[I 2025-01-10 15:13:58.663 ServerApp] Jupyter Server 2.15.0 is running at:
[I 2025-01-10 15:13:58.663 ServerApp] http://localhost:8888/tree?token=6f7ecdf66d339bf97d3a73aa22ceff2
[I 2025-01-10 15:13:58.663 ServerApp]     http://127.0.0.1:8888/tree?token=6f7ecdf66d339bf97d3a73aa22c
[I 2025-01-10 15:13:58.663 ServerApp] Use Control-C to stop this server and shut down all kernels (twi
[C 2025-01-10 15:13:58.684 ServerApp]

To access the server, open this file in a browser:
file:///home/nqbh/.local/share/jupyter/runtime/jpserver-73029-open.html
Or copy and paste one of these URLs:
http://localhost:8888/tree?token=6f7ecdf66d339bf97d3a73aa22ceff2f9eb3957d2aa3d4a6
http://127.0.0.1:8888/tree?token=6f7ecdf66d339bf97d3a73aa22ceff2f9eb3957d2aa3d4a6
[I 2025-01-10 15:13:58.695 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfil
Gtk-Message: 15:13:58.798: Not loading module "atk-bridge": The functionality is provided by GTK nativ
[73110, Main Thread] WARNING: GTK+ module /snap/firefox/5561/gnome-platform/usr/lib/gtk-2.0/modules/li
GTK+ 2.x symbols detected. Using GTK+ 2.x and GTK+ 3 in the same process is not supported.: 'glib warn

(firefox:73110): Gtk-WARNING **: 15:13:58.845: GTK+ module /snap/firefox/5561/gnome-platform/usr/lib/g
GTK+ 2.x symbols detected. Using GTK+ 2.x and GTK+ 3 in the same process is not supported.
Gtk-Message: 15:13:58.845: Failed to load module "canberra-gtk-module"
[73110, Main Thread] WARNING: GTK+ module /snap/firefox/5561/gnome-platform/usr/lib/gtk-2.0/modules/li
GTK+ 2.x symbols detected. Using GTK+ 2.x and GTK+ 3 in the same process is not supported.: 'glib warn

(firefox:73110): Gtk-WARNING **: 15:13:58.846: GTK+ module /snap/firefox/5561/gnome-platform/usr/lib/g
GTK+ 2.x symbols detected. Using GTK+ 2.x and GTK+ 3 in the same process is not supported.
Gtk-Message: 15:13:58.846: Failed to load module "canberra-gtk-module"
```

On many platforms, Jupyter will automatically open in default web browser (unless start it with `-no-browser`). Otherwise, can navigate to HTTP address printed when started notebook, here http://localhost:8888/?token=0a77b52fefe52ab83e3c35
See Fig. 2.1: Jupyter notebook landing page for what this looks like in Google Chrome.

Remark 7. *Many people use Jupyter as a local computing environment, but it can also be deployed on servers & accessed remotely. Won't cover those details here, but encourage to explore this topic on internet if it's relevant to your needs.*

To create a new notebook, click `New` button & select `Python 3` option. Should see sth like Fig. 2.2: Jupyter new notebook view. If this is 1st time, try clicking on empty code "cell" & entering a line of Python code. Then press Shift-Enter to execute it.
When save notebook (see `Save & Checkpoint` under notebook `File` menu), it creates a file with extension `.ipynb`: a self-contained file format containing all of content (including any evaluated code output) currently in notebook. These can be loaded & edited by other Jupyter users.
To rename an open notebook, click on notebook title at top of page & type new title, pressing `Enter` when finished.
To load an existing notebook, put file in same directory where started notebook process (or in a subfolder within it), then click name from landing page. Can try it out with notebooks from `wesm/pydata-book` repository on GitHub Fig. 2.3: Jupyter example view for an existing notebook.
When want to close a notebook, click File menu & select `Close & Halt`. If simply close browser tab, Python process associated with notebook will keep running in background.
While Jupyter notebook may feel like a distinct experience from IPython shell, nearly all of commands & tools in this chap can be used in either environment.

∗ Tab Completion. On surface, IPython shell looks like a cosmetically different version (phiên bản khác biệt về mặt thẩm mỹ) of standard terminal Python interpreter (invoked with `python`). 1 of major improvements over standard Python shell is *tab completion*, found in many IDEs or other interactive computing analysis environments. While entering expressions in shell, pressing `Tab` key will search namespace for any variables (objects, functions, etc.) matching characters you have typed so far & show results in a convenient drop-down menu:

```
In [1]: an_apple = 27
In [2]: an_example = 42
In [3]: an<Tab>
an_apple an_example  any
```

Note: IPython displayed both of 2 variables I defined, as well as built-in function `any`. Also, you can also complete methods

& attributes on any object after typing a period:

```
In [3]: b = [1, 2, 3]

In [4]:  b.<Tab>
append()  count()   insert()  reverse()
clear()   extend()  pop()     sort()
copy()    index()   remove()
```

Same is true for modules:

```
In [1]:  import datetime

In [2]: datetime.
date          MAXYEAR       timedelta     UTC
datetime      MINYEAR       timezone
datetime_CAPI time          tzinfo
```

Remark 8. *IPython by default hides methods & attributes starting with underscores, e.g. magic methods & internal "private" methods & attributes, in order to avoid cluttering display (& confusing novice users – gây bối rối cho người dùng mới!). These, too, can be tab-completed, but must 1st type an underscore to see them. If prefer to always see such methods in tab completion, can change this setting in IPython configuration. See IPython documentation to find out how to do this.*

Tab completion works in many contexts outside of searching interactive namespace & completng object or module attributes. When typing anything that looks like a file path (even in a Python string), pressing Tab key will complete anything on your computer's filesystem matching what you've typed.

Combined with `%run` command, this functionality can save you many keystrokes.

Another area where tab completion saves time is in completion of function keyword arguments (including = sign!) Fig. 2.4: Autocomplete function keywords in a Jupyter notebook.

Have a closer look at functions in a little bit:

* Introspection. Using a question mark `?` before or after a variable will display some general information about object:

```
In [12]: b?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

In [13]: ?b
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

In [14]: print?
Signature: print(*args, sep=' ', end='\n', file=None, flush=False)
Docstring:
Prints the values to a stream, or to sys.stdout by default.

sep
string inserted between values, default a space.
end
string appended after the last value, default a newline.
file
a file-like object (stream); defaults to the current sys.stdout.
flush
whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

This is referred to as *object introspection*. If object is a function or instance method, docstring, if defined, will also be shown. Suppose we'd written following function (which you can reproduce in IPython or Jupyter):

```
def add_numbers(a, b):
"""
Add two numbers together
Returns
-------
the_sum : type of arguments
"""
return a + b
```

Then using `?` shows us docstring:

```
In [6]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
-------
the_sum : type of arguments
File:    <ipython-input-9-6a548a216e27>
Type:    function
```

`?` has a final usage, which is for searching IPython namespace in a manner similar to standard Unix or Windows command line. A number of characters combined with wildcard `*` will show all names matching wildcard expression. E.g., could get a list of all functions in top-level NumPy namespace containing `load`: [Missing line: `np.loads` cf. book]

```
In [1]: import numpy as np

In [2]: np.*load*?
np.__loader__
np.load
np.loadtxt
```

- 2.3. Python Language Basics. Give an overview of essential Python programming concepts & language mechanics. In Chap. 3, go into more detail about Python data structures, functions, & other built-in tools.
  - Language Semantics. Python language design is distinguished by its emphasis on readability, simplicity, & explicitness. Some people go so far as to liken it to "executable pseudocode."
    - `Indentation, not braces.` Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, & Perl. Consider a `for` loop from a sorting algorithm:

      ```
      for x in array:
      if x < pivot:
      less.append(x)
      else:
      greater.append(x)
      ```

      A colon denotes start of an indented code block after which all of code must be indented by same amount until end of block.

      Love it or hate it, significant whitespace is a fact of life for Python programmers. While it may seem foreign at 1st, will hopefully grow accustomed to it in time.

      Remark 9. *Strong recommend using* 4 spaces *as your default indentation & replacing tabs with 4 spaces. Many text editors have a setting that will replace tab stops with spaces automatically insert 4 spaces on new lines following a colon & replace tabs by 4 spaces.*

      As you can see by now, Python statements also do not need to be terminated by semicolons. Semicolons can be used, however, to separate multiple statements on a single line:

      ```
      a = 5; b = 6; c = 7
      ```

      Putting multiple statements on 1 line is generally discouraged in Python as it can make code less readable.
    - Everything is an object. An important characteristic of Python language is consistency of its *object model*. Every number, string, data structure, function, class, module, & so on exists in Python interpreter in its own "box," which is referred to as a *Python object*. Each object has an associated *type* (e.g., *integer, string*, or *function*) & internal data. In practice this makes language very flexible, as even functions can be treated like any other object.

- Comments. Any text preceded by hash mark (pound sign) # is ignored by Python interpreter. Often used to add comments to code. At times may also want to exclude certain blocks of code without deleting them. 1 solution: *comment out* code:

```
results = []
for line in file_handle:
# keep the empty lines for now
# if len(line) == 0:
#   continue
results.append(line.replace("foo", "bar"))
```

Comments can also occur after a line of executed code. While some programmers prefer comments to be placed in line preceding a particular line of code, this can be useful at times:

```
print("Reached this line") # Simple status report
```

- Function & object method calls. Call functions using parentheses & passing 0 or more arguments, optionally assigning returned value to a variable:

```
result = f(x, y, z)
g()
```

Almost every object in Python has attached functions, known as *methods*, that have access to object's internal contents. Can call them using following syntax:

```
obj.some_method(x, y, z)
```

Functions can take both *positional & keyword* arguments:

```
result = f(a, b, c, d=5, e="foo")
```

- Variables & argument passing. When assigning a variable (or *name*) in Python, you are creating a *reference* to object shown on RHS of equals sign. In practical terms, consider a list of integers:

```
In [8]: a = [1, 2, 3]
```

Suppose: assign `a` to a new variable `b`:

```
In [9]: b = a

In [10]: b
Out[10]: [1, 2, 3]
```

In some languages, assignment if `b` will cause data `[1, 2, 3]` to be copied. In Python, `a` & `b` actually now refer to same object, original list `[1, 2, 3]` (see Fig. 2.5: 2 references for same object for a mock-up). Can prove this by appending an element to `a` & then examining `b`:

```
In [11]: a.append(4)

In [12]: b
Out[12]: [1, 2, 3, 4]
```

Understanding semantics of references in Python, & when, how, & why data is copied, is especially critical when you are working with larger datasets in Python.
– Hiểu được ngữ nghĩa của các tham chiếu trong Python, & khi nào, như thế nào, & lý do tại sao dữ liệu được sao chép, đặc biệt quan trọng khi bạn làm việc với các tập dữ liệu lớn hơn trong Python.

Remark 10. *Assignment is also referred to as* binding*, as we are binding a name to an object. Variable names that have been assigned may occasionally be referred to as bound variables.*

When pass objects as arguments to a function, new local variables are created referencing original objects without any copying. If bind a new object to a variable inside a function, that will not overwrite a variable of same name in "scope" outside of function ("parent scope"). Therefore possible to alter internals of a mutable argument. Suppose had following function:

```
In [13]: def append_element(some_list, element):
....:     some_list.append(element)
```

Then have:

```
In [14]: data = [1, 2, 3]

In [15]: append_element(data, 4)
```

```
In [16]: data
Out[16]: [1, 2, 3, 4]
```

- Dynamic references, strong types. Variables in Python have no inherent type associated with them; a variable can refer to a different type of object simply by doing an assignment. There is no problem with following:

```
In [17]: a = 5

In [18]: type(a)
Out[18]: int

In [19]: a = "foo"

In [20]: type(a)
Out[20]: str
```

Variables are names for objects within a particular namespace; type information is stored in object itself. Some observers might hastily conclude: Python is not a "typed language." Wrong: consider:

```
In [21]: "5" + 5
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-21-7fe5aa79f268> in <module>
----> 1 "5" + 5
TypeError: can only concatenate str (not "int") to str
```

In some languages, string `'5'` might get implicitly converted (or *cast*) to an integer, thus yielding 10. In other languages integer `5` might be cast to a string, yielding concatenated string `'55'`. In Python, such implicit casts are not allowed. In this regard, say: Python is a *strongly typed* language, i.e., every object has a specific type (or *class*), & implicit conversions will occur only in certain permitted circumstances, e.g.:

```
In [22]: a = 4.5

In [23]: b = 2

# String formatting, to be visited later
In [24]: print(f"a is {type(a)}, b is {type(b)}")
a is <class 'float'>, b is <class 'int'>

In [25]: a / b
Out[25]: 2.25
```

Here, even though `b` is an integer, it is implicitly converted to a float for division operation.
Knowing type of an object is important, & useful to be able to write functions that can handle many different kinds of input. Can check: an object is an instance of a particular type using `isinstance` function:

```
In [26]: a = 5

In [27]: isinstance(a, int)
Out[27]: True
```

`isinstance` can accept a type of types if want to check: an object's type is among those present in tuple:

```
In [28]: a = 5; b = 4.5

In [29]: isinstance(a, (int, float))
Out[29]: True

In [30]: isinstance(b, (int, float))
Out[30]: True
```

- Attributes & methods. Objects n Python typically have both attributes (other Python objects stored "inside" object) & methods (functions associated with an object that can have access to object's internal data). Both of them are accessed via syntax `obj.attribute_name`:

```
In [1]: a = "foo"
```

```
In [2]: a.<Press Tab>
capitalize()   encode()       format()       isalpha()      isidentifier() isspace()      ljust()
casefold()     endswith()     format_map()   isascii()      islower()      istitle()      lower()
center()       expandtabs()   index()        isdecimal()    isnumeric()    isupper()      lstrip()
count()        find()         isalnum()      isdigit()      isprintable()  join()         maketrans(
```

Attributes & methods can also be accessed by name via `getattr` function:

```
In [32]: getattr(a, "split")
Out[32]: <function str.split(sep=None, maxsplit=-1)>
```

While will not extensively use functions `getattr` & related functions `hasattr` & `setattr` in this book, they can be used very effectively to write generic, reusable code.

- Duck typing. Often may not care about type of an object but rather only whether it has certain methods or behavior. This is sometimes called *duck typing*, after saying "If it walks like a duck & quacks like a duck, then it's a duck." E.g., you can verify: an object is iterable if it implements *iterator protocol*. For many objects, this means it has an `__iter__` "magic method," though an alternative & better way to check is to try using `iter` function:

```
In [33]: def isiterable(obj):
....:     try:
....:         iter(obj)
....:         return True
....:     except TypeError: # not iterable
....:         return False
```

This function would return `True` for strings as well as most Python collection types:

```
In [34]: isiterable("a string")
Out[34]: True

In [35]: isiterable([1, 2, 3])
Out[35]: True

In [36]: isiterable(5)
Out[36]: False
```

- Imports. In Python, a *module* is simply a file with `.py` extension containing Python code. Suppose had following module:

```
# some_module.py
PI = 3.14159

def f(x):
return x + 2

def g(a, b):
return a + b
```

If wanted to access variables & functions defined in `some_module.py`, from another file in same directory, could do:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Or alternately:

```
from some_module import g, PI
result = g(5, PI)
```

By using `as` keyword, can give imports different variable names:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

- Binary operators & comparisons. Most of binary math operations & comparisons use familiar mathematical syntax used in other programming languages:

```
In [37]: 5 - 7
Out[37]: -2

In [38]: 12 + 21.5
Out[38]: 33.5

In [39]: 5 <= 2
Out[39]: False
```

See Table 2.1: Binary operators for all of available binary operators.
To check if 2 variables refer to same object, use `is` keyword. Use `is not` to check that 2 objects are not the same:

```
In [40]: a = [1, 2, 3]

In [41]: b = a

In [42]: c = list(a)

In [43]: a is b
Out[43]: True

In [44]: a is not c
Out[44]: True
```

Since `list` function always creates a new Python list (i.e., a copy), can be sure: `c` is distinct from `a`. Comparing with `is` is not same as `==` operator, because in this case have:

```
In [45]: a == c
Out[45]: True
```

A common use of `is` & `is not` is to check if a variable is `None`, since there is only 1 instance of `None`:

```
In [46]: a = None

In [47]: a is None
Out[47]: True
```

· Mutable & immutable objects. Many objects in Python, e.g. lists, dictionaries, NumPy arrays, & most user-defined types (classes), are *mutable* (có thể thay đổi). I.e., object or values that they contain can be modified:

```
In [48]: a_list = ["foo", 2, [4, 5]]

In [49]: a_list[2] = (3, 4)

In [50]: a_list
Out[50]: ['foo', 2, (3, 4)]
```

Others, like strings & tuples, are immutable, which means their internal data cannot be changed:

```
In [51]: a_tuple = (3, 5, (4, 5))
In [52]: a_tuple[1] = "four"
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-52-cd2a018a7529> in <module>
----> 1 a_tuple[1] = "four"
TypeError: 'tuple' object does not support item assignment
```

Remember: just because you *can* mutate an object does not mean that you always *should*. Such actions are known as *side effects*. E.g., when writing a function, any side effects should be explicitly communicated to user in function's documentation or comments. If possible, recommend trying to avoid side effects & *favor immutability*, even though there may be mutable objects involved.

∗ Scalar Types. Python has a small set of built-in types for handling numerical data, strings, Boolean (`True` or `False`) values, & dates & time. These "single value" types are sometimes called *scalar types*, & refer to them in this book as *scalars*. See Table 2.2: Standard Python scalar types

· `None`: Python "null" value (only 1 instance of `None` object exists)
· `str`: String type; holds Unicode strings

- `bytes`: Raw binary data
- `float`: Double-precision floating-point number (note there is no separate `double` type)
- `bool`: A Boolean `True` or `False` value
- `int`: Arbitrary precision integer

for a list of main scalar types. Date & time handling will be discussed separately, as these are provided by `datetime` module in standard library.

- Numeric types. Primary Python types for numbers are `int` & `float`. An `int` can store arbitrarily large numbers:

```
In [53]: ival = 17239871

In [54]: ival ** 6
Out[54]: 26254519291092456596965462913230729701102721
```

  Floating-point numbers are represented with Python `float` type. Under hood, each one is a double-precision value. They can also be expressed with scientific notation:

```
In [55]: fval = 7.243

In [56]: fval2 = 6.78e-5
```

  Integer division not resulting in a whole number will always yield a floating-point number:

```
In [57]: 3 / 2
Out[57]: 1.5
```

  To get C-style integer division (which drops fractional part if result is not a whole number), use floor division operator `//`:

```
In [58]: 3 // 2
Out[58]: 1
```

- Strings. Many people use Python for its built-in string handling capabilities. Can write *string literals* using either single quotes `'` or double quotes `"` (double quotes are generally favored):

```
a = 'one way of writing a string'
b = "another way"
```

  Python string type is `str`.
  For multiline strings with line breaks, can use triple quotes, either `'''` or `"""`:

```
c = """
This is a longer string that
spans multiple lines
"""
```

  It may surprise: this string `c` actually contains 4 lines of text; line breaks after `"""` & after `lines` are included in string. Can count new line characters with `count` method on `c`:

```
In [60]: c.count("\n")
Out[60]: 3
```

  Python strings are immutable; you cannot modify a string:

```
In [61]: a = "this is a string"

In [62]: a[10] = "f"
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-62-3b2d95f10db4> in <module>
----> 1 a[10] = "f"
TypeError: 'str' object does not support item assignment
```

  To interpret this error message, read from bottom up. Tried to replace character ("item") at position 10 with letter `"f"`, but this is not allowed for string objects. If need to modify a string, have to use a function or method that creates a new string, e.g. string `replace` method:

```
In [63]: b = a.replace("string", "longer string")
```

```
In [64]: b
Out[64]: 'this is a longer string'
```

After this operation, variable `a` is unmodified:

```
In [65]: a
Out[65]: 'this is a string'
```

Many Python objects can be converted to a string using `str` function:

```
In [66]: a = 5.6
In [67]: s = str(a)
In [68]: print(s)
5.6
```

Strings are a sequence of Unicode characters & therefore can be treated like other sequences, e.g. lists & tuples:

```
In [69]: s = "python"

In [70]: list(s)
Out[70]: ['p', 'y', 't', 'h', 'o', 'n']

In [71]: s[:3]
Out[71]: 'pyt'
```

Syntax `s[:3]` is called *slicing* & is implemented for many kinds of Python sequences. This will be explained in more detail later on, as it is used extensively in this book.
Backslash character `\` is an *escape character*, i.e., it is used to specify special characters like newline `\n` or Unicode characters. To write a string literal with backslashes, need to escape them:

```
In [72]: s = "12\\34"

In [73]: print(s)
12\34
```

If have a string with a lot of backslashes & no special characters, might find this a bit annoying. Fortunately, can preface leading quote of string with `r`, i.e., characters should be interpreted as is:

```
In [74]: s = r"this\has\no\special\characters"

In [75]: s
Out[75]: 'this\\has\\no\\special\\characters'
```

`r` stands for *raw*.
Adding 2 strings together concatenates them & produces a new string:

```
In [76]: a = "this is the first half "

In [77]: b = "and this is the second half"

In [78]: a + b
Out[78]: 'this is the first half and this is the second half'
```

String templating or formatting is another important topic. Number of ways to do so has expanded with advent of Python 3, & here briefly describe mechanics of 1 of main interfaces. String objects have a `format` method that can be used to substitute formatted arguments into string, producing a new string:

```
In [79]: template = "{0:.2f} {1:s} are worth US${2:d}"
```

In this string: `{0:.2f}` means to format 1st argument as a floating-point number with no decimal places. `{1:s}` means to format 2nd argument as a string. `{2:d}` means to format 3rd argument as an exact integer.
To substitute arguments for these format parameters, pass a sequence of arguments to `format` method:

```
In [80]: template.format(88.46, "Argentine Pesos", 1)
Out[80]: '88.46 Argentine Pesos are worth US$1'
```

Python 3.6 introduced a new feature called *f-strings* (short for *formatted string literals*) which can make creating formatted strings even more convenient. To create an f-string, write character `f` immediately preceding a string literal. Within string, enclose Python expressions in curly braces to substitute value of expression into formatted string:

```
In [81]: amount = 10

In [82]: rate = 88.46

In [83]: currency = "Pesos"

In [84]: result = f"{amount} {currency} is worth US${amount / rate}"
```

Format specifiers can be added after each expression using same syntax as with string templates above:

```
In [85]: f"{amount} {currency} is worth US${amount / rate:.2f}"
Out[85]: '10 Pesos is worth US$0.11'
```

String formatting is a deep topic; there are multiple methods & numerous options & tweaks available to control how values are formatted in resulting string. To learn more, consult https://docs.python.org/3/library/string.html.

- Bytes & Unicode. In modern Python (i.e., Python 3.0 & up), Unicode has become 1st-class string type to enable more consistent handling of ASCII & non-ASCII text. In older versions of Python, strings were all bytes without any explicit Unicode encoding. Could convert to Unicode assuming knew character encoding. An example Unicode string with non-ASCII characters:

```
In [86]: val = "espa■ol"

In [87]: val
Out[87]: 'espa■ol'
```

Can convert this Unicode string to its UTF-8 bytes representation using `encode` method:

```
In [88]: val_utf8 = val.encode("utf-8")

In [89]: val_utf8
Out[89]: b'espa\xc3\xb1ol'

In [90]: type(val_utf8)
Out[90]: bytes
```

Assuming know Unicode encoding of a `bytes` object, can go back using `decode` method:

```
In [91]: val_utf8.decode("utf-8")
Out[91]: 'espa■ol'
```

While now preferable to use UTF-8 for any encoding, for historical reasons, may encounter data in any number of different encodings:

```
In [92]: val.encode("latin1")
Out[92]: b'espa\xf1ol'

In [93]: val.encode("utf-16")
Out[93]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [94]: val.encode("utf-16le")
Out[94]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

Most common to encounter `bytes` object in context of working with files, where implicitly decoding all data to Unicode strings may not be desired.

- Booleans. 2 Boolean values in Python are written as `True, False`. Comparisons & other conditional expressions evaluate to either `True` or `False`. Boolean values are combined with `and` & `or` keywords:

```
In [95]: True and True
Out[95]: True

In [96]: False or True
Out[96]: True
```

When converted to numbers, `False` becomes `0` & `True` becomes `1`:

```
In [97]: int(False)
Out[97]: 0
```

```
In [98]: int(True)
Out[98]: 1
```

Keyword `not` flips a Boolean value from `True` to `False` or vice versa:

```
In [99]: a = True

In [100]: b = False

In [101]: not a
Out[101]: False

In [102]: not b
Out[102]: True
```

- Type casting. `str`, `bool`, `int`, `float` types are also functions that can be used to cast values to those types:

```
In [103]: s = "3.14159"

In [104]: fval = float(s)

In [105]: type(fval)
Out[105]: float

In [106]: int(fval)
Out[106]: 3

In [107]: bool(fval)
Out[107]: True

In [108]: bool(0)
Out[108]: False
```

Note: most nonzero values when cast to `bool` become `True`.

- None. `None` is Python null value type:

```
In [109]: a = None

In [110]: a is None
Out[110]: True

In [111]: b = 5

In [112]: b is not None
Out[112]: True
```

`None` is also a common default value for function arguments:

```
def add_and_maybe_multiply(a, b, c = None):
result = a + b

if c is not None:
result = result * c

return result
```

- Dates & times. Built-in Python `datetime` module provides `datetime, date, time` types. `datetime` type combines information stored in `date` & `time` & is most commonly used:

```
In [113]: from datetime import datetime, date, time

In [114]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [115]: dt.day
Out[115]: 29

In [116]: dt.minute
```

```
Out[116]: 30
```

Given a `datetime` instance, can extract equivalent `date` & `time` objects by calling methods on `datetime` of same name:

```
In [117]: dt.date()
Out[117]: datetime.date(2011, 10, 29)

In [118]: dt.time()
Out[118]: datetime.time(20, 30, 21)
```

`strftime` method formats a `datetime` as a string:

```
In [119]: dt.strftime("%Y-%m-%d %H:%M")
Out[119]: '2011-10-29 20:30'
```

Strings can be converted (parsed) into `datetime` objects with `strptime` function:

```
In [120]: datetime.strptime("20091031", "%Y%m%d")
Out[120]: datetime.datetime(2009, 10, 31, 0, 0)
```

See Table 11.2: `datetime` format specification (ISO C89 compatible) for a full list of format specifications.
When aggregating or otherwise group time series data, it will occasionally be useful to replace time fields of a series of `datetimes` – e.g., replacing `minute, second` fields with 0:

```
In [121]: dt_hour = dt.replace(minute=0, second=0)

In [122]: dt_hour
Out[122]: datetime.datetime(2011, 10, 29, 20, 0)
```

Since `datetime.datetime` is an immutable type, methods like these always produce new objects. So in previous example, `dt` is not modified by `replace`:

```
In [123]: dt
Out[123]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

Difference of 2 `datetime` objects produces a `datetime.timedelta` type:

```
In [124]: dt2 = datetime(2011, 11, 15, 22, 30)

In [125]: delta = dt2 - dt

In [126]: delta
Out[126]: datetime.timedelta(days=17, seconds=7179)

In [127]: type(delta)
Out[127]: datetime.timedelta
```

Output `timedelta(17, 7179)` indicates: `timedelta` encodes an offset of 17 days & 7178 seconds.
Adding a `timedelta` to a `datetime` produces a new shifted `datetime`:

```
In [128]: dt
Out[128]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [129]: dt + delta
Out[129]: datetime.datetime(2011, 11, 15, 22, 30)
```

∗ Control Flow. Python has several built-in keywords for conditional logic, loops, & other standard *control flow* concepts found in other programming languages.

· if, elif, & else. `if` statement is 1 of most well-known control flow statement types. It checks a condition that, if `True`, evaluates code in block that follows:

```
x = -5
if x < 0:
print("It's negative")
```

& `if` statement can be optionally followed by 1 or more `elif` blocks & a catchall `else` block if all of conditions are `False`:

```
if x < 0:
print("It's negative")
```

```
elif x == 0:
print("Equal to zero")
elif 0 < x < 5:
print("Positive but smaller than 5")
else:
print("Positive and larger than or equal to 5")
```

If any of conditions are `True`, no further `elif` or `else` blocks will be reached. With a compound condition using `and` or `or`, conditions are evaluated left to right & will short-circuit:

```
In [130]: a = 5; b = 7

In [131]: c = 8; d = 4

In [132]: if a < b or c > d:
.....:
print("Made it")
Made it
```

In this example, comparison `c > d` never gets evaluated because 1st comparison was `True`.
Also possible to chain comparisons:

```
In [133]: 4 > 3 > 2 > 1
Out[133]: True
```

- for loops. `for` loops are for iterating over a collection (like a list or tuple) or an iterater. Standard syntax for a `for` loop is:

```
for value in collection:
# do something with value
```

Can advance a `for` loop to next iteration, skipping remainder of block, using `continue` keyword. Consider this code, which sums up integers in a list & skips `None` values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
if value is None:
continue
total += value
```

A `for` loop can be exited altogether with `break` keyword. This code sums elements of list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
if value == 5:
break
total_until_5 += value
```

`break` keyword only terminates innermost `for` loop; any outer `for` loops will continue to run:

```
In [134]: for i in range(4):
.....:     for j in range(4):
.....:         if j > i:
.....:             break
.....:         print((i, j))
.....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

If elements in collection or iterator are sequences (tuples or lists, say), they can be conveniently *unpacked* into variables in `for` loop statement:

```
for a, b, c in iterator:
# do something
```

- while loops. A `while` loop specifies a condition & a block of code that is to be executed until condition evaluates to `False` or loop is explicitly ended with `break`:

```
x = 256
total = 0
while x > 0:
if total > 500:
break
total += x
x = x // 2
```

- pass. `pass` is "no-op" (or "do nothing") statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is required only because Python uses whitespace to delimit blocks:

```
if x < 0:
print("negative!")
elif x == 0:
# TODO: put something smart here
pass
else:
print("positive!")
```

- range. `range` function generates a sequence of evenly spaced integers:

```
In [135]: range(10)
Out[135]: range(0, 10)

In [136]: list(range(10))
Out[136]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A start, end, & step (which may be negative) can be given:

```
In [137]: list(range(0, 20, 2))
Out[137]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [138]: list(range(5, 0, -1))
Out[138]: [5, 4, 3, 2, 1]
```

`range` produces integers up to but not including endpoint. A common use of `range` is for iterating through sequences by index:

```
In [139]: seq = [1, 2, 3, 4]

In [140]: for i in range(len(seq)):
.....:     print(f"element {i}: {seq[i]}")
element 0: 1
element 1: 2
element 2: 3
element 3: 4
```

While can use functions like `list` to store all integers generated by `range` in some other data structure, often default iterator form will be what you want. This snippet sums all numbers from 0 to 99999 that are multiples of 3 or 5:

```
In [141]: total = 0
In [142]: for i in range(100_000):
.....:     # % is the modulo operator
.....:     if i % 3 == 0 or i % 5 == 0:
.....:         total += i

In [143]: print(total)
2333316668
```

While range generated can be arbitrarily large, memory use at any given time may be very small.

- 2.4. Conclusion. This chap provided a brief introduction to some basic Python language concepts & IPython & Jupyter programming environments. In Chap. 3, discuss many built-in data types, functions, & input-output utilities that will be used continuously throughout rest of book.

- 3. Built-In Data Structures, Functions, & Files. This chap discusses capabilities built into Python language that will be used ubiquitously throughout book. While add-on libraries like `pandas` & NumPy add advanced computational functionality for larger datasets, they are designed to be used together with Python's built-in data manipulation tools.

  Start with PYthon's workhorse data structures: tuples, lists, dictionaries, & sets. Discuss creating your own reusable Python functions. Look at mechanics of Python file objects & interacting with local hard drive.

  - 3.1. Data Structures & Sequences. Python's data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer. Start with tuple, list, & dictionary, which are some of most frequently used *sequence* types.

    - Tuple. A *tuple* is a fixed-length, immutable sequence of Python objects which, once assigned, cannot be changed. Easiest way to create one is with a comma-separated sequence of values wrapped in parentheses:

```
In [2]: tup = (4, 5, 6)

In [3]: tup
Out[3]: (4, 5, 6)
```

In many contexts, parentheses can be omitted, so here could also have written:

```
In [4]: tup = 4, 5, 6

In [5]: tup
Out[5]: (4, 5, 6)
```

Can convert any sequence or iterator to a tuple by invoking `tuple`:

```
In [6]: tuple([4, 0, 2])
Out[6]: (4, 0, 2)

In [7]: tup = tuple('string')

In [8]: tup
Out[8]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets `[]` as with most other sequence types. As in C, C++, Java, & many other languages, sequences are 0-indexed in Python:

```
In [9]: tup[0]
Out[9]: 's'
```

When defining tuples within more complicated expressions, often necessary to enclose values in parentheses, as in this example of creating a tuple of tuples:

```
In [10]: nested_tup = (4, 5, 6), (7, 8)

In [11]: nested_tup
Out[11]: ((4, 5, 6), (7, 8))

In [12]: nested_tup[0]
Out[12]: (4, 5, 6)

In [13]: nested_tup[1]
Out[13]: (7, 8)
```

While objects stored in a tuple may be mutable themselves, once tuple is created, impossible to modify which object is stored in each slot:

```
In [14]: tup = tuple(['foo', [1, 2], True])
In [15]: tup[2] = False
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-15-b89d0c4ae599> in <module>
```

```
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

If an object inside a tuple is mutable, e.g. a list, can modify it in place:

```
In [16]: tup[1].append(3)

In [17]: tup
Out[17]: ('foo', [1, 2, 3], True)
```

Can concatenate tuples using + operator to produce longer tuples:

```
In [18]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[18]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has effect of concatenating that many copies of tuple:

```
In [19]: ('foo', 'bar') * 4
Out[19]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note: objects themselves are not copied, only references to them.

· Unpacking tuples. If try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* value on RHS of equals sign:

```
In [20]: tup = (4, 5, 6)

In [21]: a, b, c = tup

In [22]: b
Out[22]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [23]: tup = 4, 5, (6, 7)

In [24]: a, b, (c, d) = tup

In [25]: d
Out[25]: 7
```

Using this functionality you can easily swap variable names, a task that in many languages might look like:

```
tmp = a
a = b
b = tmp
```

But, in Python, swap can be done like this:

```
In [26]: a, b = 1, 2
In [27]: a
Out[27]: 1
In [28]: b
Out[28]: 2
In [29]: b, a = a, b
In [30]: a
Out[30]: 2
In [31]: b
Out[31]: 1
```

A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [32]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [33]: for a, b, c in seq:
....:     print(f'a={a}, b={b}, c={c}')
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

Another common use is returning multiple values from a function.

There are some situations where may want to "pluck" (nhổ) a few elements from beginning of a tuple. There is a special syntax that can do this, `*rest`, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

```
In [34]: values = 1, 2, 3, 4, 5
In [35]: a, b, *rest = values
In [36]: a
Out[36]: 1
In [37]: b
Out[37]: 2
In [38]: rest
Out[38]: [3, 4, 5]
```

This `reset` bit is sometimes something you want to discard; there is nothing special about `rest` name. As a matter of convention, many Python programmers will use underscore `_` for unwanted variables:

```
In [39]: a, b, *_ = values
```

· Tuple methods. Since size & contents of a tuple cannot be modified, very light on instance methods. A particularly useful one (also available on lists) is `count`, which counts number of occurrences of a value:

```
In [40]: a = (1, 2, 2, 2, 3, 4, 2)
In [41]: a.count(2)
Out[41]: 4
```

∗ List. In contrast with tuples, lists are variable length & their contents can be modified in place. Lists are mutable. Can define them using square brackets `[]` or using `list` type function:

```
In [42]: a_list = [2, 3, 7, None]
In [43]: tup = ("foo", "bar", "baz")
In [44]: b_list = list(tup)
In [45]: b_list
Out[45]: ['foo', 'bar', 'baz']
In [46]: b_list[1] = "peekaboo"
In [47]: b_list
Out[47]: ['foo', 'peekaboo', 'baz']
```

Lists & tuples are semantically similar (though tuples cannot be modified) & can be used interchangeably in many functions.

`list` built-in function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
In [48]: gen = range(10)
In [49]: gen
Out[49]: range(0, 10)
In [50]: list(gen)
Out[50]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

· Adding & removing elements. Elements can be appended to end of list with `append` method:

```
In [51]: b_list.append("dwarf")
In [52]: b_list
Out[52]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Using `insert`, can insert an element at a specific location in list:

```
In [53]: b_list.insert(1, "red")
In [54]: b_list
Out[54]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

Insertion index must be between 0 & length of list, inclusive.

Remark 11. `insert` *is computationally expensive compared with* `append`*, because references to subsequent elements have to be shifted internally to make room for new element. If need to insert elements at both beginning & end of a sequence, may wish to explore* `collections.deque`*, a double-ended queue, which is optimized for this purpose & found in Python Standard Library.*

Inverse operation to `insert` is `pop`, which removes & returns an element at a particular index:

```
In [55]: b_list.pop(2)
Out[55]: 'peekaboo'
In [56]: b_list
```

```
Out[56]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value with `remove`, which locates 1st such value & removes it from list:

```
In [57]: b_list.append("foo")
In [58]: b_list
Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']
In [59]: b_list.remove("foo")
In [60]: b_list
Out[60]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` & `remove`, can use a Python list as a set-like data structure (although Python has actual set objects).
Check if a list contains a value using `in` keyword:

```
In [61]: "dwarf" in b_list
Out[61]: True
```

Keyword `not` can be used to negate `in`:

```
In [62]: "dwarf" not in b_list
Out[62]: False
```

Checking whether a list contains a value is a lot slower than doing so with dictionaries & sets, as Python makes a linear scan across values of list, whereas it can check others (based on hash tables) in constant time.

- Concatenating & combining lists. Similar to tuples, adding 2 lists together with `+` concatenates them:

```
In [63]: [4, None, "foo"] + [7, 8, (2, 3)]
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```

If have a list already defined, can append multiple elements to it using `extend` method:

```
In [64]: x = [4, None, "foo"]
In [65]: x.extend([7, 8, (2, 3)])
In [66]: x
Out[66]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note: list concatenation by addition is a comparatively expensive operation since a new list must be created & objects copied over. using `extend` to append elements to an existing list, especially if building up a large list, is usually preferable. Thus

```
everything = []
for chunk in list_of_lists:
everything.extend(chunk)
```

is faster than concatenative alternative:

```
everything = []
for chunk in list_of_lists:
everything = everything + chunk
```

- Sorting. Can sort a list in place (without creating a new object) by calling its `sort` function:

```
In [67]: a = [7, 2, 5, 1, 3]
In [68]: a.sort()
In [69]: a
Out[69]: [1, 2, 3, 5, 7]
```

`sort` has a few options that will occasionally come in handy. One is ability to pass a secondary *sort key* – i.e., a function that produces a value to use to sort objects. E.g., could sort a collection of strings by their lengths:

```
In [70]: b = ["saw", "small", "He", "foxes", "six"]
In [71]: b.sort(key=len)
In [72]: b
Out[72]: ['He', 'saw', 'six', 'small', 'foxes']
```

`sorted` function can produce a sorted copy of a general sequence.

- Slicing. Can select sections of most sequence types by using slice notation, which in its basic form consists of `start:stop` passed to indexing operator `[]`:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

Slices can also be assigned with a sequence:

```
In [75]: seq[3:5] = [6, 3]
In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 6, 0, 1]
```

While element at `start` index is included, `stop` index is *not included*, so that number of elements in result is `stop - start`.
Either `start` or `stop` can be omitted, in which case they default to start of sequence & end of sequence, resp.:

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]
In [78]: seq[3:]
Out[78]: [6, 3, 6, 0, 1]
```

Negative indices slide sequence relative to end:

```
In [79]: seq[-4:]
Out[79]: [3, 6, 0, 1]
In [80]: seq[-6:-2]
Out[80]: [3, 6, 3, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See Fig. 3.1: Illustration of Python slicing conventions for a helpful illustration of slicing with positive & negative integers. In figure, indices are shown at "bin edges" to help show where slice selections start & stop using positive or negative indices.
A `step` can also be used after a 2nd colon to, say, take every other element:

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 0]
```

A clever use of this is to pass `-1`, which has useful effect of reversing a list or tuple:

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 3, 6, 3, 2, 7]
```

* Dictionary. Dictionary or `dict` may be most important built-in Python data structure. In other programming languages, dictionaries are sometimes called *hash maps* or *associative arrays*. A dictionary stores a collection of *key-value* pairs, where *key & value* are Python objects. Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key. 1 approach for creating a dictionary is to use curly braces `{}` & colons to separate keys & values:

```
In [83]: empty_dict = {}
In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}
In [85]: d1
Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Can access, insert, or set elements using same syntax as for accessing elements of a list or tuple:

```
In [86]: d1[7] = "an integer"
In [87]: d1
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
In [88]: d1["b"]
Out[88]: [1, 2, 3, 4]
```

Can check if a dictionary contains a key using same syntax used for checking whether a list or tuple contains a value:

```
In [89]: "b" in d1
Out[89]: True
```

Can delete values using either `del` keyword or `pop` method (which simultaneously returns value & deletes key):

```
In [90]: d1[5] = "some value"
In [91]: d1
Out[91]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
```

```
 7: 'an integer',
 5: 'some value'}
In [92]: d1["dummy"] = "another value"
In [93]: d1
Out[93]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
In [94]: del d1[5]
In [95]: d1
Out[95]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}
In [96]: ret = d1.pop("dummy")
In [97]: ret
Out[97]: 'another value'
In [98]: d1
Out[98]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

`keys & values` method gives you iterators of dictionary's keys & values, resp. Order of keys depends on order of their insertion, & these functions output keys & values in same respective order:

```
In [99]: list(d1.keys())
Out[99]: ['a', 'b', 7]
In [100]: list(d1.values())
Out[100]: ['some value', [1, 2, 3, 4], 'an integer']
```

If need to iterate over both keys & values, can use `items` method to iterate over keys & values as 2-tuples:

```
In [101]: list(d1.items())
Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

Can merge 1 dictionary into another using `update` method:

```
In [102]: d1.update({"b": "foo", "c": 12})
In [103]: d1
Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

`update` method changes dictionaries in place, so any existing keys in data passed to `update` will have their old values discarded.

- Creating dictionaries from sequences. Common to occasionally end up with 2 sequences that you want to pair up element-wise in a dictionary. As a 1st cut, might write code like this:

  ```
  mapping = {}
  for key, value in zip(key_list, value_list):
  mapping[key] = value
  ```

  Since a dictionary is essentially a collection of 2-tuples, `dict` function accepts a list of 2-tuples:

  ```
  In [104]: tuples = zip(range(5), reversed(range(5)))
  In [105]: tuples
  Out[105]: <zip at 0x7fefe4553a00>
  In [106]: mapping = dict(tuples)
  In [107]: mapping
  Out[107]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
  ```

  Talk about *dictionary comprehensions*, which are another way to construct dictionaries.
- Default values. Common to have logic like:

  ```
  if key in some_dict:
  value = some_dict[key]
  else:
  value = default_value
  ```

Thus, dictionary methods get & pop can take a default value to be returned, so that above if-else block can be written simply as:

```
value = some_dict.get(key, default_value)
```

get by default will return None if key is not present, while pop will raise an exception. With *setting* values, it may be that values in a dictionary are another kind of collection, like a list. E.g., could imagine categorizing a list of words by their 1st letters as a dictionary of lists:

```
In [108]: words = ["apple", "bat", "bar", "atom", "book"]
In [109]: by_letter = {}
In [110]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:    else:
.....:        by_letter[letter].append(word)
.....:
In [111]: by_letter
Out[111]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

setdefault dictionary method can be used to simplify this workflow. Preceding for loop can be rewritten as:

```
In [112]: by_letter = {}
In [113]: for word in words:
.....:     letter = word[0]
.....:     by_letter.setdefault(letter, []).append(word)
.....:
In [114]: by_letter
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

Built-in collections module has a useful class, defaultdict, which makes this even easier. To create one, pass a type or function for generating default value for each slot in dictionary:

```
In [115]: from collections import defaultdict
In [116]: by_letter = defaultdict(list)
In [117]: for word in words:
.....:     by_letter[word[0]].append(word)
```

· Valid dictionary key types. While values of a dictionary can be any Python object, keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all objects in tuple need to be immutable, too). Technical term here is *hashability*. Can check whether an object is hashable (can be used as a key in a dictionary) with hash function:

```
In [118]: hash("string")
Out[118]: 3634226001988967898
In [119]: hash((1, 2, (2, 3)))
Out[119]: -9209053662355515447
In [120]: hash((1, 2, [2, 3])) # fails because lists are mutable
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-120-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

Hash values you see when using hash function in general will depend on Python version you are using.
To use a list as a key, 1 option: convert it to a tuple, which can be hashed as long as its elements also can be:

```
In [121]: d = {}
In [122]: d[tuple([1, 2, 3])] = 5
In [123]: d
Out[123]: {(1, 2, 3): 5}
```

* Set. A *set* is an unordered collection of unique elements. A set can be created in 2 ways: via set function or via a *set literal* with curly braces:

```
In [124]: set([2, 2, 2, 1, 3, 3])
Out[124]: {1, 2, 3}
In [125]: {2, 2, 2, 1, 3, 3}
```

```
Out[125]: {1, 2, 3}
```

Sets support mathematical set *operations* like union, intersection, difference, & symmetric difference. Consider these 2 example sets:

```
In [126]: a = {1, 2, 3, 4, 5}
In [127]: b = {3, 4, 5, 6, 7, 8}
```

Union of these 2 sets is set of distinct elements occurring in either set. This can be computed with either `union` method or `|` binary operator:

```
In [128]: a.union(b)
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}
In [129]: a | b
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```

Intersection contains elements occurring in both sets. `&` operator or `intersection` method can be used:

```
In [130]: a.intersection(b)
Out[130]: {3, 4, 5}
In [131]: a & b
Out[131]: {3, 4, 5}
```

See Table 3.1: Python set operations for a list of commonly used set methods.

Remark 12. *If you pass an input that is not a set to methods like* `union` *&* `intersection`*, Python will convert input to a set before executing operation. When using binary operators, both objects must already be sets.*

All of logical set operations have in-place counterparts, which enable you to replace contents of set on left side of operation with result. For very large sets, this may be more efficient:

```
In [132]: c = a.copy()
In [133]: c |= b
In [134]: c
Out[134]: {1, 2, 3, 4, 5, 6, 7, 8}
In [135]: d = a.copy()
In [136]: d &= b
In [137]: d
Out[137]: {3, 4, 5}
```

Like dictionary keys, set elements generally must be immutable, & they must be *hashable* (i.e., calling `hash` on a value does not raise an exception). In order to store list-like elements (or other mutable sequences) in a set, can convert them to tuples:

```
In [138]: my_data = [1, 2, 3, 4]
In [139]: my_set = {tuple(my_data)}
In [140]: my_set
Out[140]: {(1, 2, 3, 4)}
```

Can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [141]: a_set = {1, 2, 3, 4, 5}
In [142]: {1, 2, 3}.issubset(a_set)
Out[142]: True
In [143]: a_set.issuperset({1, 2, 3})
Out[143]: True
```

Sets are equal iff their contents are equal:

```
In [144]: {1, 2, 3} == {3, 2, 1}
Out[144]: True
```

* Built-In Sequence Functions. Python has a handful of useful sequence functions that you should familiarize yourself with & use at any opportunity.
  - enumerate. Common when iterating over a sequence to want to keep track of index of current item. A do-it-yourself approach would look like:

```
index = 0
for value in collection:
# do something with value
index += 1
```

Since this is so common, Python has a built-in function, `enumerate`, which returns a sequence of `(i, value)` tuples:

```
for index, value in enumerate(collection):
# do something with value
```

- sorted. `sorted` function returns a new sorted list from elements of any sequence:

```
In [145]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[145]: [0, 1, 2, 2, 3, 6, 7]
In [146]: sorted("horse race")
Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

  `sorted` function accepts same arguments as `sort` method on lists.
- zip. `zip` "pairs" up elements of a number of lists, tuples, or other sequences to create a list of tuples:

```
In [147]: seq1 = ["foo", "bar", "baz"]
In [148]: seq2 = ["one", "two", "three"]
In [149]: zipped = zip(seq1, seq2)
In [150]: list(zipped)
Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

  `zip` can take an arbitrary number of sequences, & number of elements it produces is determined by *shortest* sequence:

```
In [151]: seq3 = [False, True]
In [152]: list(zip(seq1, seq2, seq3))
Out[152]: [('foo', 'one', False), ('bar', 'two', True)]
```

  A common use of `zip` is simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [153]: for index, (a, b) in enumerate(zip(seq1, seq2)):
.....:     printf(f"{index}: {a}, {b}")
.....:
0: foo, one
1: bar, two
2: baz, three
```

- reversed. `reversed` iterates over elements of a sequence in reverse order:

```
In [154]: list(reversed(range(10)))
Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

  Keep in mind: `reversed` is a generator, so it does not create reversed sequence until materialized (e.g., with `list` or a `for` loop).

* List, Set, & Dictionary Comprehensions. *List comprehensions* are a convenient & widely used Python language feature. They allow you to concisely form a new list by filtering elements of a collection, transforming elements passing filter into 1 concise expression. They take basic form:

```
[expr for value in collection if condition]
```

$\Leftrightarrow$ following `for` loop:

```
result = []
for value in collection:
if condition:
result.append(expr)
```

Filter condition can be omitted, leaving only expression. E.g., given a list of strings, could filter out strings with length 2 or less & convert them to uppercase like this:

```
In [155]: strings = ["a", "as", "bat", "car", "dove", "python"]
In [156]: [x.upper() for x in strings if len(x) > 2]
Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set & dictionary comprehensions are a natural extension, producing sets & dictionaries in an idiomatically similar way instead of lists.
A dictionary comprehension looks like this:

```
dict_comp = {key-expr: value-expr for value in collection if condition}
```

A set comprehension looks like equivalent list comprehension except with curly braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```

Like list comprehensions, set & dictionary comprehensions are mostly conveniences, but they similarly can make code both easier to write & read. Consider list of strings from before. Suppose wanted a set containing just lengths of strings contained in collection; could easily compute this using a set comprehension:

```
In [157]: unique_lengths = {len(x) for x in strings}
In [158]: unique_lengths
Out[158]: {1, 2, 3, 4, 6}
```

Could also express this more functionally using `map` function, introduced shortly:

```
In [159]: set(map(len, strings))
Out[159]: {1, 2, 3, 4, 6}
```

As a simple dictionary comprehension example, could create a lookup map of these strings for their locations in list:

```
In [160]: loc_mapping = {value: index for index, value in enumerate(strings)}
In [161]: loc_mapping
Out[161]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

* Nested list comprehensions. Suppose have a list of lists containing some English & Spanish names:

```
In [162]: all_data = [["John", "Emily", "Michael", "Mary", "Steven"],
.....:              ["Maria", "Juan", "Javier", "Natalia", "Pilar"]]
```

Suppose wanted to get a single list containing all names with 2 or more `a`'s in them. Could certainly do this with a simple `for` loop:

```
In [163]: names_of_interest = []
In [164]: for names in all_data:
.....:     enough_as = [name for name in names if name.count("a") >= 2]
.....:     names_of_interest.extend(enough_as)
.....:
In [165]: names_of_interest
Out[165]: ['Maria', 'Natalia']
```

Can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [166]: result = [name for names in all_data for name in names
.....:            if name.count("a") >= 2]
In [167]: result
Out[167]: ['Maria', 'Natalia']
```

At 1st, nested list comprehensions are a bit hard to wrap your head around. `for` parts of list comprehension are arranged according to order of nesting, & any filter condition is put at end as before. Another example where "flatten" a list of tuples of integers into a simple list of integers:

```
In [168]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [169]: flattened = [x for tup in some_tuples for x in tup]
In [170]: flattened
Out[170]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind: order of `for` expressions would be same if wrote a nested `for` loop instead of a list comprehension:

```
flattened = []
for tup in some_tuples:
for x in tup:
flattened.append(x)
```

Can have arbitrarily many levels of nesting, though if have >2 or 3 levels of nesting, should probably start to question whether this makes sense from a code readability standpoint. Important to distinguish syntax just shown from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [172]: [[x for x in tup] for tup in some_tuples]
Out[172]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This produces a list of lists, rather than a flattened list of all of inner elements.

- 3.2. Functions. *Functions* are primary & most important method of code organization & reuse in Python. As a rule of thumb, if anticipate needing to repeat same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

Functions are declared with `def` keyword. A function contains a block of code with an optional use of `return` keyword:

```
In [173]: def my_function(x, y):
.....:      return x + y
```

When a line with `return` is reached, value or expression after `return` is sent to context where function was called, e.g.:

```
In [174]: my_function(1, 2)
Out[174]: 3
In [175]: result = my_function(1, 2)
In [176]: result
Out[176]: 3
```

There is no issue with having multiple `return` statements. If Python reaches end of a function without encountering a `return` statement, `None` is returned automatically. E.g.:

```
In [177]: def function_without_return(x):
.....:     print(x)
In [178]: result = function_without_return("hello!")
hello!
In [179]: print(result)
None
```

Each function can have *positional* arguments & *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. Here define a function with an optional `z` argument with default value `1.5`:

```
def my_function2(x, y, z=1.5):
if z > 1:
return z * (x + y)
else:
return z / (x + y)
```

While keyword arguments are optional, all positional arguments must be specified when calling a function.
Can pass values to `z` argument with or without keyword provided, though using keyword is encouraged:

```
In [181]: my_function2(5, 6, z=0.7)
Out[181]: 0.06363636363636363
In [182]: my_function2(3.14, 7, 3.5)
Out[182]: 35.49
In [183]: my_function2(10, 20)
Out[183]: 45.0
```

Main restriction on function arguments: keyword arguments *must* follow positional arguments (if any). Can specify keyword arguments in any order. This frees you from having to remember order in which function arguments were specified. Need to remember only what their names are.

∗ Namespaces, Scope, & Local Functions. Functions can access variables created inside function as well as those outside function in higher (or even *global*) scopes. An alternative & more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to local namespace. Local namespace is created when function is called & is immediately populated by function's arguments. After function is finished, local namespace is destroyed (with some exceptions that are outside purview (phạm vi) of this chap). Consider following function:

  ```
  def func():
  a = []
  for i in range(5):
  a.append(i)
  ```

  When `func()` is called, empty list `a` is created, 5 elements are appended, & then `a` is destroyed when function exists. Suppose instead had declared `a` as follows:

  ```
  In [184]: a = []
  In [185]: def func():
  .....:     for i in range(5):
  .....:         a.append(i)
  ```

Each call to `func` will modify list `a`:

```
In [186]: func()
In [187]: a
Out[187]: [0, 1, 2, 3, 4]
In [188]: func()
In [189]: a
Out[189]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

Assigning variables outside of function's scope is possible, but those variables must be declared explicitly using either `global` or `nonlocal` keywords:

```
In [190]: a = None
In [191]: def bind_a_variable():
.....:     global a
.....:     a = []
.....: bind_a_variable(a)
.....:
In [192]: print(a)
[]
```

`nonlocal` allows a function to modify variables defined in a higher-level scope that is not global. Since its use is somewhat esoteric (bí truyền, bí mật) (I never use it in this book), refer to Python documentation to learn more about it.

Remark 13. *Generally discourage use of* `global` *keyword. Typically, global variables are used to store some kind of state in a system. If fiond yourself using a lot of them, it may indicate a need for object-oriented programming (using classes).*

* Returning Multiple Values. When 1st programmed in Python after having programmed in Java & C++, 1 of favorite features was ability to return multiple values from a function with simple syntax. E.g.:

```
def f():
a = 5
b = 6
c = 7
return a, b, c

a, b, c = f()
```

In data analysis & other scientific applications, may find yourself doing this often. What's happening here: function is actually just returning *1* object, a tuple, which is then being unpacked into result variables. In preceding example, could have done this instead:

```
return_value = f()
```

In this case, `return_value` would be a 3-tuple with 3 returned variables. A potentially attractive alternative to returning multiple values like before might be to return a dictionary instead:

```
def f():
a = 5
b = 6
c = 7
return {"a" : a, "b" : b, "c" : c}
```

This alternative technique can be useful depending on what you are trying to do.

* Functions Are Objects. Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose were doing some data cleaning & needed to apply a bunch of transformations to following list of strings:

```
In [193]: states = ["Alabama ", "Georgia!", "Georgia", "georgia", "FlOrIda",
"south carolina##", "West virginia?"]
```

Anyone who has ever worked with user-submitted survey data has seen messy results like these. Lots of things need to happen to make this list of strings uniform & ready for analysis: stripping whitespace, removing punctuation symbols, & standardizing proper capitalization. 1 way to do this: use built-in string methods along with `re` standard library module for regular expressions:

```
import re

def clean_strings(strings):
result = []
```

```
for value in strings:
value = value.strip()
value = re.sub("[!#?]", "", value)
value = value.title()
result.append(value)
return result
```

Result looks like this:

```
In [195]: clean_strings(states)
Out[195]:
['Alabama',
'Georgia',
'Georgia',
'Georgia',
'Florida',
'South    Carolina',
'West Virginia']
```

An alternative approach that you may find useful: make a list of operations you want to apply to a particular set of strings:

```
def remove_punctuation(value):
return re.sub("[!#?]", "", value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
result = []
for value in strings:
for func in ops:
value = func(value)
result.append(value)
return result
```

Then have following:

```
In [197]: clean_strings(states, clean_ops)
Out[197]:
['Alabama',
'Georgia',
'Georgia',
'Georgia',
'Florida',
'South   Carolina',
'West Virginia']
```

A more *functional* pattern like this enables you to easily modify how strings are transformed at a very high level. `clean_strings` function is also now more reusable & generic.
Can use functions as arguments to other functions like built-in `map` function, which applies a function to a sequence of some kind:

```
In [198]: for x in map(remove_punctuation, states):
.....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

`map` can be used as an alternative to list comprehensions without any filter.

* Anonymous (Lambda) Functions. Python has support for *anonymous* or*lambda* functions, which are a way of writing functions consisting of a single statement, result of which is return of writing functions consisting of a single statement, result of which is return value. They are defined with `lambda` keyword, which has no meaning other than "we are declaring an anonymous function":

```
In [199]: def short_function(x):
.....:     return x * 2
In [200]: equiv_anon = lambda x: x * 2
```

Usually refer to these as lambda functions in rest of book. They are especially convenient in data analysis because there are many cases where data transformation functions will take functions as arguments. Often less typing (& clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning lambda function to a local variable. E.g.:

```
In [201]: def apply_to_list(some_list, f):
.....:     return [f(x) for x in some_list]
In [202]: ints = [4, 0, 1, 5, 6]
In [203]: apply_to_list(ints, lambda x: x * 2)
Out[203]: [8, 0, 2, 10, 12]
```

Could also have written `[x * 2 for x in ints]`, but here were able to succinctly pass a custom operator to `apply_to_list` function.
As another example, suppose wanted to sort a collection of strings by number of distinct letters in each string:

```
In [204]: strings = ["foo", "card", "bar", "aaaa", "abab"]
```

Here could pass a lambda function to list's `sort` method:

```
In [205]: strings.sort(key=lambda x: len(set(x)))
In [206]: strings
Out[206]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

* Generators. Many objects in Python support iteration, e.g. over objects in a list or lines in a file. This is accomplished by means of *iterator protocol*, a generic way to make objects iterable. E.g., iterating over a dictionary yields dictionary keys:

```
In [207]: some_dict = {"a": 1, "b": 2, "c": 3}
In [208]: for key in some_dict:
.....:     print(key)
a
b
c
```

When write `for key in some_dict`, Python interpreter 1st attempts to create an iterator out of `some_dict`:

```
In [209]: dict_iterator = iter(some_dict)
In [210]: dict_iterator
Out[210]: <dict_keyiterator at 0x7fefe45465c0>
```

An iterator is any object that will yield objects to Python interpreter when used in a context like a `for loop`. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods e.g. `min, max, sum` & type constructors like `list, tuple`:

```
In [211]: list(dict_iterator)
Out[211]: ['a', 'b', 'c']
```

A *generator* is a convenient way, similar to writing a normal function, to construct a new iterable object. Whereas normal functions execute & return a single result at a time, generators can return a sequence of multiple values by pausing & resuming execution each time generator is used. To create a generator, use `yield` keyword instead of `return` in a function:

```
def squares(n=10):
print(f"Generating squares from 1 to {n ** 2}")
for i in range(1, n + 1):
yield i ** 2
```

When actually call generator, no code is immediately executed:

```
In [213]: gen = squares()
In [214]: gen
Out[214]: <generator object squares at 0x7fefe437d620>
```

Not until you request elements from generator that it begins executing its code:

```
In [215]: for x in gen:
.....:     print(x, end=" ")
Generating squares from 1 to 100
```

```
1 4 9 16 25 36 49 64 81 100
```

Remark 14. *Since generators produce output 1 element at a time vs. an entire list all at once, it can help your program use less memory.*

- Generator expressions. Another way to make a generator is by using a *generator expression*. This is a generator analogue to list, dictionary, & set comprehensions. To create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

```
In [216]: gen = (x ** 2 for x in range(100))
In [217]: gen
Out[217]: <generator object <genexpr> at 0x7fefe437d000>
```

⇔ following verbose generator:

```
def _make_gen():
for x in range(100):
yield  ** 2
gen = _make_gen()
```

Generator expressions can be used instead of list comprehensions as function arguments in some cases:

```
In [218]: sum(x ** 2 for x in range(100))
Out[218]: 328350
In [219]: dict((i, i ** 2) for i in range(5))
Out[219]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Depending on number of elements produced by comprehension expression, generator version can sometimes by meaningfully faster.

- itertools module. Standard library `itertools` module has a collection of generators for many common data algorithms. E.g., `groupby` takes any sequence & a function, grouping consecutive elements in sequence by return value of function. E.g.:

```
In [220]: import itertools
In [221]: def first_letter(x):
.....:     return x[0]
In [222]: names = ["Alan", "Adam", "Wes", "Will", "Albert", "Steven"]
In [223]: for letter, names in itertools.groupby(names, first_letter):
.....:     print(letter, list(names)) # names is a generator
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

See Table 3.2: Some useful `itertools` functions for a list of a few other `itertools` functions frequently found helpful. Check out https://docs.python.org/3/library/itertools.html for more on this useful built-in utility module. Function: Description

1. `chain(*iterables)`: Generates a sequence by chaining iterators together. Once elements from 1st iterator are exhausted, elements from next iterator are returned, & so on.
2. `combinations(iterable, k)`: Generates a sequence of all possible k-tuples of elements in iterable, ignoring order & without replacement (see also companion function `combinations_with_replacement`).
3. `permutations(iterate, k)`: Generates a sequence of all possible k-tuples of elements in iterable, respecting order.
4. `groupby(iterable[, keyfunc])`: Generates `(key, sub-iterator)` for each unique key.
5. `product(*iterables, repeat=1)`: Generates Cartesian product of input iterables as tuples, similar to a nested `for` loop.

* Errors & Exception Handling. Handling Python errors or *exceptions* gracefully is an important part of building robust programs. In data analysis applications, many functions work only on certain kinds of input. E.g., Python's `float` function is capable of casting a string to a floating-point number, but it fails with `ValueError` on improper inputs:

```
In [224]: float("1.2345")
Out[224]: 1.2345
In [225]: float("something")
---------------------------------------------------------------------------
ValueError
Traceback (most recent call last)
<ipython-input-225-5ccfe07933f4> in <module>
----> 1 float("something")
```

```
ValueError: could not convert string to float: 'something'
```

Suppose wanted a version of `float` that fails gracefully, returning input argument. Can do this by writing a function that encloses call to `float` in a `try/except` block (execute this code in IPython):

```
def attempt_float(x):
try:
return float(x)
except:
return x
```

Code in `except` part of block will only be executed if `float(x)` raises an exception:

```
In [227]: attempt_float("1.2345")
Out[227]: 1.2345
In [228]: attempt_float("something")
Out[228]: 'something'
```

Might notice: `float` can raise exceptions other than `ValueError`:

```
In [229]: float((1, 2))
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-229-82f777b0e564> in <module>
----> 1 float((1, 2))
TypeError: float() argument must be a string or a real number, not 'tuple'
```

Might want to suppress only `ValueError`, since a `TypeError` (input was not a string or numeric value) might indicate a legitimate bug in your program. To do that, write exception type after `except`:

```
def attempt_float(x):
try:
return float(x)
except ValueError:
return x
```

Have then:

```
In [231]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-231-8b0026e9e6b7> in <module>
----> 1 attempt_float((1, 2))
<ipython-input-230-6209ddecd2b5> in attempt_float(x)
1 def attempt_float(x):
2
try:
----> 3
return float(x)
4
except ValueError:
5
return x
TypeError: float() argument must be a string or a real number, not 'tuple'
```

Can catch multiple exception types by writing a tuple of exception types instead (parentheses are required):

```
def attempt_float(x):
try:
return float(x)
except (TypeError, ValueError):
return x
```

In some cases, may not want to suppress an exception, but want some code to be executed regardless of whether or not code in `try` block succeeds. To do this, use `finally`:

```
f = open(path, mode="w")
```

```
try:
write_to_file(f)
finally:
f.close()
```

Here, file object `f` will *always* get closed. Similarly, can have code that executes only if `try:` block succeeds using `else`:

```
f = open(path, mode="w")

try:
write_to_file(f)
except:
print("Failed")
else:
print("Succeeded")
finally:
f.close()
```

- Exceptions in IPython. If an exception is raised while you are `%run`-ing a script or executing any statement, IPython will by default print a full call stack trace (traceback) with a few lines of context around position at each point in stack:

  ```
  In [10]: %run examples/ipython_bug.py
  ---------------------------------------------------------------------------
  AssertionError
  Traceback (most recent call last)
  /home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
  13
  throws_an_exception()
  14
  ---> 15 calling_things()
  /home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
  11 def calling_things():
  12
  works_fine()
  ---> 13
  throws_an_exception()
  14
  15 calling_things()
  /home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
  7
  a = 5
  8
  b = 6
  ----> 9
  assert(a + b == 10)
  10
  11 def calling_things():
  AssertionError:
  ```

  Having additional context by itself is a big advantage over standard Python interpreter (which does not provide any additional context). Can control amount of context shown using `%xmode` magic command, from `Plain` (same as standard Python interpreter) to `Verbose` (which inlines function argument values & more). See in Appendix B, can step *into stack* (using `% debug` ("địt em béo ú gay" or "dái em béo ú ghê" or "đi ẻ bị u gan") or `%pdb` ("phở/phân đầu/đuôi bò/buồi" or "phở/phân đặc biệt") magics) after an error has occurred for interactive postmortem debugging.

- 3.3. Files & OS. Most of this book uses high-level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, important to understand basics of how to work with files in Python. Fortunately, relatively straightforward, which is 1 reason Python is so popular for text & file munging.

  To open a file for reading or writing, use built-in `open` function with either a relative or absolute file path & an optional file encoding:

  ```
  In [233]: path = "examples/segismundo.txt"
  In [234]: f = open(path, encoding="utf-8")
  ```

Here, pass `encoding="utf-8"` as a best practice because default Unicode encoding for reading files varies from platform to platform.

By default, file is opened in read-only model `"r"`. Can then treat file object `f` like a list & iterate over lines like so:

```
for line in f:
print(line)
```

Lines come out of file with end-of-line (EOL) markers intact, often see code to get an EOL-free list of lines in a file like:

```
In [235]: lines = [x.rstrip() for x in open(path, encoding="utf-8")]
In [236]: lines
Out[236]:
['Sue■a el rico en su riqueza,',
'que más cuidados le ofrece;',
'',
'sue■a el pobre que padece',
'su miseria y su pobreza;',
'',
'sue■a el que a medrar empieza,',
'sue■a el que afana y pretende,',
'sue■a el que agravia y ofende,',
'',
'y en el mundo, en conclusión,',
'todos sue■an lo que son,',
'aunque ninguno lo entiende.',
'']
```

When use `open` to create file objects, recommended to close file when finished with it. Closing file releases its resources back to OS:

```
In [237]: f.close()
```

1 of ways to make it easier to clean up open files: use `with` statement:

```
In [238]: with open(path, encoding="utf-8") as f:
.....:     lines = [x.rstrip() for x in f]
```

This will automatically close file `f` when exiting `with` block. Failing to ensure that files are closed will not cause problems in many small programs or scripts, but it can be an issue in programs that need to interact with a large number of files.

If had typed `f = open(path, "w")`, a *new file* at `examples/segismundo.txt` would have been created (be careful!), overwriting any file in its place. There is also `"x"` file mode, which creates a writable file but fails if file path already exists. See Table 3.3: Python file modes for a list of all valid file read/write modes.

For readable files, some of most commonly used methods are `read, seek, tell`. `read` returns a certain number of characters from file. What constitutes a "character" is determined by file encoding or simply raw bytes if file is opened in binary mode:

```
In [239]: f1 = open(path)
In [240]: f1.read(10)
Out[240]: 'Sue■a el r'
In [241]: f2 = open(path, mode="rb") # Binary mode
In [242]: f2.read(10)
Out[242]: b'Sue\xc3\xb1a el '
```

`read` method advances file object position by number of bytes read. `tell` gives current position:

```
In [243]: f1.tell()
Out[243]: 11
In [244]: f2.tell()
Out[244]: 10
```

Even though read 10 characters from file `f1` opened in text mode, position is 11 because it took that many bytes to decode 10 characters using default encoding. Can check default encoding in `sys` module:

```
In [245]: import sys
In [246]: sys.getdefaultencoding()
Out[246]: 'utf-8'
```

To get consistent behavior across platforms, best to pass an encoding (e.g. `encoding="utf-8"`, which is widely used) when opening files.
`seek` changes file position to indicated byte in file:

```
In [247]: f1.seek(3)
Out[247]: 3
In [248]: f1.read(1)
Out[248]: '■'
In [249]: f1.tell()
Out[249]: 5
```

Lastly, remember to close files:

```
In [250]: f1.close()
In [251]: f2.close()
```

To write text to a file, can use file's `write` or `writelines` methods. E.g., could create a version of `examples/segismundo.txt` with no blank lines like so:

```
In [252]: path
Out[252]: 'examples/segismundo.txt'
In [253]: with open("tmp.txt", mode="w") as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)
In [254]: with open("tmp.txt") as f:
.....:     lines = f.readlines()
In [255]: lines
Out[255]:
['Sue■a el rico en su riqueza,\n',
'que más cuidados le ofrece;\n',
'sue■a el pobre que padece\n',
'su miseria y su pobreza;\n',
'sue■a el que a medrar empieza,\n',
'sue■a el que afana y pretende,\n',
'sue■a el que agravia y ofende,\n',
'y en el mundo, en conclusión,\n',
'todos sue■an lo que son,\n',
'aunque ninguno lo entiende.\n']
```

See Table 3.4: Important Python file methods or attributes for many of most commonly used file methods.

∗ Bytes & Unicode with Files. Default behavior for Python files (whether readable or writable) is *text mode*, i.e., intend to work with Python strings (i.e., Unicode). This contrasts with *binary mode*, which can obtain by appending `b` to file mode. Revisiting file (which contains non-ASCII characters with UTF-8 encoding) from previous sect, have:

```
In [258]: with open(path) as f:
.....:     chars = f.read(10)
In [259]: chars
Out[259]: 'Sue■a el r'
In [260]: len(chars)
Out[260]: 10
```

UTF-8 is a variable-length Unicode encoding, so when request some number of characters from file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from file to decode that many characters. If open file in `"rb"` mode instead, `read` requests that exact number of bytes:

```
In [261]: with open(path, mode="rb") as f:
.....:      data = f.read(10)
In [262]: data
Out[262]: b'Sue\xc3\xb1a el '
```

Depending on text encoding, may be able to decode bytes to a `str` object yourself, but only if each of encoded Unicode characters is fully formed:

```
In [263]: data.decode("utf-8")
Out[263]: 'Sue■a el '
In [264]: data[:4].decode("utf-8")
---------------------------------------------------------------------------
UnicodeDecodeError
Traceback (most recent call last)
<ipython-input-264-846a5c2fed34> in <module>
----> 1 data[:4].decode("utf-8")
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpecte
d end of data
```

Text mode, combined with `encoding` option of `open`, provides a convenient way to convert from 1 Unicode encoding to another:

```
In [265]: sink_path = "sink.txt"
In [266]: with open(path) as source:
.....:    with open(sink_path, "x", encoding="iso-8859-1") as sink:
.....:        sink.write(source.read())
In [267]: with open(sink_path, encoding="iso-8859-1") as f:
.....:     print(f.read(10))
Sue■a el r
```

Beware using `seek` when opening files in any mode other than binary. If file position falls in middle of bytes defining a Unicode character, then subsequent reads will result in an error:

```
In [269]: f = open(path, encoding='utf-8')
In [270]: f.read(5)
Out[270]: 'Sue■a'
In [271]: f.seek(4)
Out[271]: 4
In [272]: f.read(1)
---------------------------------------------------------------------------
UnicodeDecodeError
Traceback (most recent call last)
<ipython-input-272-5a354f952aa4> in <module>
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.10/codecs.py in decode(self, input, final)
320
# decode input (taking the buffer into account)
321
data = self.buffer + input
--> 322
(result, consumed) = self._buffer_decode(data, self.errors, final
)
323
# keep undecoded input until the next call
324
self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid s
tart byte
In [273]: f.close()
```

If find yourself regularly doing data analysis on non-ASCII text data, mastering Python's Unicode functionality will prove valuable. See https://docs.python.org for much more.

- 3.4. Conclusion. With some of basics of Python environment & language now under your belt, time to move on & learn about NumPy & array-oriented computing in Python.

- 4. NumPy Basics: Arrays & Vectorized Computation. NumPy, short for Numerical Python, is 1 of most important foundational packages for numerical computing in Python. Many computational packages providing scientific functionality use NumPy's array objects as 1 of standard interface *lingua francas* for data exchange. Much of knowledge about NumPy covered is transferable to `pandas` as well.

  Some of things you'll find in Numpy:

  - `ndarray`, an efficient multidimensional array providing fast array-oriented arithmetic operations & flexible *broadcasting* capabilities

- Mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading/writing array data to disk & working with memory-mapped files
- Linear algebra, random number generation, & Fourier transform capabilities
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN

Because NumPy provides a comprehensive & well-documented C API, straightforward to pass data to external libraries written in a low-level language, & for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C, C++, or FORTRAN codebases & giving them a dynamic & accessible interface.

While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays & array-oriented computing will help you use tools with array computing semantics, like `pandas`, much more effectively. Since NumPy is a large topic, will cover many advanced NumPy features like broadcasting in more depth later (see Appendix A). Many of these advanced features are not needed to follow rest of book, but they may help you as you go deeper into scientific computing in Python.

For most data analysis applications, main areas of functionality focused on are:

- Fast array-based operations for data munging & cleaning, subsetting & filtering, transformation, & any other kind of computation
- Common array algorithms like sorting, unique, & set operations
- Efficient descriptive statistics & aggregating/summarizing data
- Data alignment & relational data manipulations for merging & joining heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, & function application)

While NumPy provides a computational foundation for general numerical data processing, many readers will want to use `pandas` as basis for most kinds of statistics or analytics, especially on tabular data. Also, `pandas` provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

Remark 15. *Array-oriented computing in Python traces its roots back to 1995, when* JIM HUGUNIN *created Numeric library. Over next 10 years, many scientific programming communities began doing array programming in Python, but library ecosystem had became fragmented in early 2000s. In 2005,* TRAVIS OLIPHANT *was able to forge NumPy project from then Numeric & Numarray projects to bring community together around a single array computing framework.*

1 of reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
  – NumPy lưu trữ dữ liệu nội bộ trong 1 khối bộ nhớ liền kề, độc lập với các đối tượng Python tích hợp khác. Thư viện thuật toán của NumPy được viết bằng ngôn ngữ C có thể hoạt động trên bộ nhớ này mà không cần kiểm tra kiểu hoặc bất kỳ chi phí nào khác. Mảng NumPy cũng sử dụng ít bộ nhớ hơn nhiều so với chuỗi Python tích hợp.
- NumPy operations perform complex computations on entire arrays without need for Python `for` loops, which can be slow for large sequences. NumPy is faster than regular Python code because its C-based algorithms avoid overhead present with regular interpreted Python code.
  – Các hoạt động NumPy thực hiện các phép tính phức tạp trên toàn bộ mảng mà không cần vòng lặp Python `for`, có thể chậm đối với các chuỗi lớn. NumPy nhanh hơn mã Python thông thường vì các thuật toán dựa trên C của nó tránh được chi phí phát sinh trong mã Python được biên dịch thông thường.

To give an idea of performance difference, consider a NumPy array of 1 million integers, & equivalent Python list:

```
In [7]: import numpy as np
In [8]: my_arr = np.arange(1_000_000)
In [9]: my_list = list(range(1_000_000))
```

Multiply each sequence by 2:

```
In [10]: %timeit my_arr2 = my_arr * 2
715 us +- 13.2 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
In [11]: %timeit my_list2 = [x * 2 for x in my_list]
48.8 ms +- 298 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

NumPy-based algorithms are generally 10–100 times faster (or more) than their pure Python counterparts & use significantly less memory.

- 4.1. NumPy ndarray: A Multidimensional Array Object. 1 of key features of NumPy is its N-dimensional array object, or `ndarray`, which is a fast, flexible container for large datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to equivalent operations between scalar elements.

  To give a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, 1st import NumPy & create a small array:

```
In [12]: import numpy as np
In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])
In [14]: data
Out[14]:
array([[ 1.5, -0.1, 3.],
[  0., -3. , 6.5]])
```

  Then write mathematical operations with `data`:

```
In [15]: data * 10
Out[15]:
array([[ 15., -1., 30.],
[ 0., -30., 65.]])
In [16]: data + data
Out[16]:
array([[ 3. , -0.2, 6. ],
[ 0. , -6. , 13. ]])
```

  In 1st example, all of elements have been multiplied by 10. In 2nd, corresponding values in each "cell" in array have been added to each other.

  Remark 16. *In this chap & throughout book, use standard NumPy convention of always using* `import numpy as np`*. Possible to put* `from numpy import *` *in your code to avoid having to write* `np.`*, but advise against making a habit of this.* `numpy` *namespace is large & contains a number of functions whose names conflict with built-in Python functions (like* `min`, `max`*). Following standard conventions like these is almost always a good idea.*

  An `ndarray` is a generic multidimensional container for homogeneous data, i.e., all of elements must be same type. Every array has a `shape`, a tuple indicating size of each dimension, & a `dtype`, an object describing *data type* of array:

```
In [17]: data.shape
Out[17]: (2, 3)
In [18]: data.dtype
Out[18]: dtype('float64')
```

  This chap will introduce to basics of using NumPy arrays, & should be sufficient for following along with rest of book. While unnecessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming & thinking is a key step along way to becoming a scientific Python guru.

  Remark 17. *Whenever see "array," "NumPy array," or "ndarray" in book text, in most cases they all refer to* `ndarray` *object.*

  - Creating `ndarrays`. Easiest way to create an array is to use `array` function. This accepts any sequence-like object (including other arrays) & produces a new NumPy array containing passed data. E.g., a list is a good candidate for conversion:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]
In [20]: arr1 = np.array(data1)
In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

    Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
In [23]: arr2 = np.array(data2)
In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
[5, 6, 7, 8]])
```

Since `data2` was a list of lists, NumPy array `arr2` has 2D, with shape inferred from data. Can confirm this by inspecting `ndim` & `shape` attributes:

```
In [25]: arr2.ndim
Out[25]: 2
In [26]: arr2.shape
Out[26]: (2, 4)
```

Unless explicitly specified, `numpy.array` tries to infer a good data type for array that it creates. Data type is stored in a special `dtype` metadata object; e.g., in previous 2 examples have:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')
In [28]: arr2.dtype
Out[28]: dtype('int64')
```

In addition to `numpy.array`, there are a number of other functions for creating new arrays. As examples, `numpy.zeros` & `numpy.ones` create arrays of 0s or 1s, resp., with a given length or shape. `numpy.empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for shape:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0.]])
In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
[0., 0.],
[0., 0.]],
[[0., 0.],
[0., 0.],
[0., 0.]]])
```

Remark 18. *Not safe to assume* `numpy.empty` *will return an array of all 0s. This function returns uninitialized memory & thus may contain nonzero "garbage" values. Should use this function only if intend to populate new array with data.*

`numpy.arrange` is an array-valued version of built-in Python `range` function:

```
In [32]: np.arange(15)
Out[32]: array([ 0, 1,
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

See Table 4.1: Some important NumPy array creation functions for a short list of standard array creation functions. Since NumPy is focused on numerical computing, data type, if not specified, will in many cases be `float64` (floating point).

`Function: Description`

- `array`: Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies input data by default
- `asarray`: Convert input to ndarray, but do not copy if input is already an ndarray
- `arrange`: Like built-in `range` but returns an ndarray instead of a list
- `ones, ones_like`: Produce an array of all 1s with given shape & data type; `ones_like` takes another array & produces a `ones` array of same shape & data type
- `zeros, zeros_like`: Like `ones, ones_like` but producing arrays of 0s instead
- `empty, empty_like`: Create new arrays by allocating new memory, but do not populate with any values like `ones, zeros`
- `full, full_like`: Produce an array of given shape & data type with all values set to indicated "fill value"; `full_like` takes another array & produces a filled array of same shape & data type
- `eye, identity`: Create a square $N \times N$ identity matrix (1s on diagonal & 0s elsewhere)

∗ Data Types for `ndarrays`. *Data type* or `dtype` is a special object containing information (or *metadata*, data about data) ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)
In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)
In [35]: arr1.dtype
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Data types are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it possible to read & write binary streams of data to disk & to connect to code written in a low-level language like C or FORTRAN. Numerical data types are named same way: a type name, like `float` or `int`, followed by a number indicating number of bits per element. A standard double-precision floating-point value (what's used under hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See Table 4.2: NumPy data types for a full listing of NumPy's supported data types.

Remark 19. *Don't worry about memorizing NumPy data types, especially if you're a new user. Often only necessary to care about general* kind *of data you're dealing with, whether floating point, complex, integer, Boolean, string, or general Python object. When need more control over how data is stored in memory & on disk, especially large datasets, good to know what you have control over storage type.*

Remark 20. *There are both* signed, unsigned *integer types, & many readers will not be familiar with this terminology. A* signed *integer can represent both positive & negative integers, while an* unsigned *integer can only represent nonzero integers. E.g.,* `int8` *(signed 8-bit integer) can represent integers from* -128 *to* 127 *(inclusive), while* `uint8` *(unsigned 8-bit integer) can represent 0 through 255.*

Can explicitly convert or *cast* an array from 1 data type to another using ndarray's `astype` method:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])
In [38]: arr.dtype
Out[38]: dtype('int64')
In [39]: float_arr = arr.astype(np.float64)
In [40]: float_arr
Out[40]: array([1., 2., 3., 4., 5.])
In [41]: float_arr.dtype
Out[41]: dtype('float64')
```

In this example, integers were cast to floating point. If cast some floating-point numbers to be of integer data type, decimal part will be truncated:

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
In [43]: arr
Out[43]: array([ 3.7, -1.2, -2.6,
0.5, 12.9, 10.1])
In [44]: arr.astype(np.int32)
Out[44]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

If have an array of strings representing numbers, can use `astype` to convert them to numeric form:

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)
In [46]: numeric_strings.astype(float)
Out[46]: array([ 1.25, -9.6 , 42. ])
```

Remark 21. *Be cautious when using* `numpy.string_` *type, as string data in NumPy is fixed size & may truncate input without warning. pandas has more intuitive out-of-box behavior or non-numeric data.*

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `ValueError` will be raised. Before, was a bit lazy & wrote `float` instead of `np.float64`; NumPy aliases Python types to its own equivalent data types.

Can also use another array's `dtype` attribute:

```
In [47]: int_array = np.arange(10)
In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
In [49]: int_array.astype(calibers.dtype)
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

There are shorthand type code strings you can also use to refer to a `dtype`:

```
In [50]: zeros_uint32 = np.zeros(8, dtype="u4")
In [51]: zeros_uint32
Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```

Remark 22. *Call* `astype` always *creates a new array (a copy of data), even if new data type is same as old data type.*

* Arithmetic with NumPy Arrays. Arrays are important because they enable you to express batch operations on data without writing any `for` loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays apply operation element-wise:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
In [53]: arr
Out[53]:
array([[1., 2., 3.],
[4., 5., 6.]])
In [54]: arr * arr
Out[54]:
array([[ 1., 4., 9.],
[16., 25., 36.]])
In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
[0., 0., 0.]])
```

Arithmetic operations with scalars propagate scalar argument to each element in array:

```
In [56]: 1 / arr
Out[56]:
array([[1.    , 0.5, 0.3333],
[0.25 , 0.2, 0.1667]])


In [57]: arr ** 2
Out[57]:
array([[ 1., 4., 9.],
[16., 25., 36.]])
```

Comparisons between arrays of same size yield Boolean arrays:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
In [59]: arr2
Out[59]:
array([[ 0., 4., 1.],
[ 7., 2., 12.]])
In [60]: arr2 > arr
Out[60]:
array([[False, True, False],
[ True, False, True]])
```

Evaluating operations between differently sized arrays is called *broadcasting* (see Appendix A). Having a deep understanding of broadcasting is not necessary for most of this book.

∗ Basic Indexing & Slicing. NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements. 1D arrays are simple; on surface they act similarly to Python lists:

```
In [61]: arr = np.arange(10)
In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [63]: arr[5]
Out[63]: 5
In [64]: arr[5:8]
Out[64]: array([5, 6, 7])
In [65]: arr[5:8] = 12
In [66]: arr
Out[66]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

If assign a scalar value to a slice, as in `arr[5:8] = 12`, value is propagated (or *broadcast* henceforth) to entire selection.

Remark 23. *An important 1st distinction from Python's built-in lists: array slices are views on original array. I.e., data is not copied, & any modifications to view will be reflected in source array.*

E.g., 1st create a slice of `arr`:

```
In [67]: arr_slice = arr[5:8]
In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

Now, when change values in `arr_slice`, mutations are reflected in original array `arr`:

```
In [69]: arr_slice[1] = 12345
```

```
In [70]: arr
Out[70]:
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

"Bare" slice `[:]` will assign to all values in an array:

```
In [71]: arr_slice[:] = 64
In [72]: arr
Out[72]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

If new to NumPy, might be surprised by this, especially if have used other array programming languages that copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, could imagine performance & memory problems if NumPy insisted on always copying data.

Remark 24. *If want to copy of a slice of an ndarray instead of a view, will need to explicitly copy array – e.g.,* `arr[5:8].copy()`*. pandas works this way, too.*

With higher dimensional arrays, have many more options. In a 2D array, elements at each index are no longer scalars but rather 1D arrays:

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [74]: arr2d[2]
Out[74]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [75]: arr2d[0][2]
Out[75]: 3
In [76]: arr2d[0, 2]
Out[76]: 3
```

See Fig. 4.1: Indexing elements in a NumPy array for an illustration of indexing on a 2D array. Find it helpful to think of axis 0 as "rows" of array & axis 1 as "columns."

In multidimensional arrays, if omit later indices, returned object will be a lower dimensional ndarray consisting of all data along higher dimensions. So in $2 \times 2 \times 3$ array `arr3d`:

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
In [78]: arr3d
Out[78]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

`arr3d[0]` is a 2×3 array:

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
[4, 5, 6]])
```

Both scalar values & arrays can be assigned to `arr3d[0]`:

```
In [80]: old_values = arr3d[0].copy()
In [81]: arr3d[0] = 42
In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
[42, 42, 42]],
[[ 7, 8, 9],
[10, 11, 12]]])
In [83]: arr3d[0] = old_values
In [84]: arr3d
Out[84]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],
[[ 7, 8, 9],
[10, 11, 12]]])
```

Similarly, `arr3d[1, 0]` gives all of values whose indices start with `(1, 0)`, forming a 1D array:

```
In [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

This expression is same as though had indexed in 2 steps:

```
In [86]: x = arr3d[1]
In [87]: x
Out[87]:
array([[ 7, 8, 9],
[10, 11, 12]])
In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Note: in all of these cases where subsections of array have been selected, returned arrays are views.

Remark 25. *This multidimensional indexing syntax for NumPy arrays will not work with regular Python objects, e.g. lists of lists.*

· Indexing with slices. Like 1D objects e.g. Python lists, ndarrays can be sliced with familiar syntax:

```
In [89]: arr
Out[89]: array([ 0, 1, 2, 3, 4, 64, 64, 8, 9])
In [90]: arr[1:6]
Out[90]: array([ 1, 2, 3, 4, 64])
```

Consider 2D array from before, `arr2d`. Slicing this array is a bit different:

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
[4, 5, 6]])
```

It has sliced along axis 0, 1st axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read expression `arr2d[:2]` as "select 1st 2 rows of `arr2d`."
Can pass multiple slices just like you can pass multiple indexes:

```
In [93]: arr2d[:2, 1:]
Out[93]:
array([[2, 3],
[5, 6]])
```

When slicing like this, always obtain array reviews of same number of dimensions. By mixing integer indexes & slices, get lower dimensional slices.
E.g., can select 2nd row but only 1st 2 columns, like so:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Here, while `arr2d` is 2D, `lower_dim_slice` is 1D, & its shape is a tuple with 1 axis size:

```
In [95]: lower_dim_slice.shape
Out[95]: (2,)
```

Similarly, can select 3rd column but only 1st 2 rows, like so:

```
In [96]: arr2d[:2, 2]
Out[96]: array([3, 6])
```

See Fig. 4.2: 2D array slicing for an illustration. note: a colon by itself means to take entire axis, so can slice only higher dimensional axes by doing:

```
In [97]: arr2d[:, :1]
Out[97]:
array([[1],
[4],
[7]])
```

Of course, assigning to a slice expression assigns to whole selection:

```
In [98]: arr2d[:2, 1:] = 0
In [99]: arr2d
Out[99]:
array([[1, 0, 0],
[4, 0, 0],
[7, 8, 9]])
```

* Boolean Indexing. Consider an example where have some data in an array & an array of names with duplicates:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2],
.....:                    [-12, -4], [3, 4]])
In [102]: names
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
In [103]: data
Out[103]:
array([[ 4, 7],
[ 0, 2],
[-5, 6],
[ 0, 0],
[ 1, 2],
[-12, -4],
[ 3, 4]]),
```

Suppose each name corresponds to a row in `data` array & wanted to select all rows with corresponding name `"Bob"`. Like arithmetic operations, comparisons (e.g. `==`) with arrays are also vectorized. Thus, comparing `names` with string `"Bob"` yields a Boolean array:

```
In [104]: names == "Bob"
Out[104]: array([ True, False, False, True, False, False, False])
```

This Boolean array can be passed when indexing array:

```
In [105]: data[names == "Bob"]
Out[105]:
array([[4, 7],
[0, 0]])
```

Boolean array must be of same length as array axis it's indexing. Can even mix & match Boolean arrays with slices or integers (or sequences of integers; more on this later).
In these examples, select from rows where `names == "Bob"` & index columns, too:

```
In [106]: data[names == "Bob", 1:]
Out[106]:
array([[7],
[0]])
In [107]: data[names == "Bob", 1]
Out[107]: array([7, 0])
```

To select everything but `"Bob"` can either use `!=` or negate condition using `~`:

```
In [108]: names != "Bob"
Out[108]: array([False, True, True, False, True, True, True])
In [109]: ~(names == "Bob")
Out[109]: array([False, True,
True, False, True, True, True])
In [110]: data[~(names == "Bob")]
Out[110]:
array([[ 0, 2],
[-5, 6],
[ 1, 2],
[-12,-4],
[ 3, 4]])
```

`~` operator can be useful when want to invert a Boolean array referenced by a variable:

```
In [111]: cond = names == "Bob"
```

```
In [112]: data[~cond]
Out[112]:
array([[ 0, 2],
[-5, 6],
[ 1, 2],
[-12,-4],
[ 3, 4]])
```

To select 2 of 3 names to combine multiple Boolean conditions, use Boolean arithmetic operators like `&` (and) & `|` (or):

```
In [113]: mask = (names == "Bob") | (names == "Will")
In [114]: mask
Out[114]: array([ True, False,
True, True, True, False, False])
In [115]: data[mask]
Out[115]:
array([[ 4, 7],
[-5, 6],
[ 0, 0],
[ 1, 2]])
```

Selecting data from an array by Boolean indexing & assigning result to a new variable *always* create a copy of data, even if returned array is unchanged.

Remark 26. *Python keywords* `and`, `or` *do not work with Boolean arrays. Use* `&` *(and) &* `|` *(or) instead.*

Setting values with Boolean arrays works by substituting value or values on RHS into locations where Boolean array's values are `True`. To set all of negative value in `data` to 0, need only do:

```
In [116]: data[data < 0] = 0
In [117]: data
Out[117]:
array([[4, 7],
[0, 2],
[0, 6],
[0, 0],
[1, 2],
[0, 0],
[3, 4]])
```

Can also set whole rows or columns using a 1D Boolean array:

```
In [118]: data[names != "Joe"] = 7
In [119]: data
Out[119]:
array([[7, 7],
[0, 2],
[7, 7],
[7, 7],
[7, 7],
[0, 0],
[3, 4]])
```

These types of operations on 2D data are convenient to do with pandas.

* Fancy Indexing. *Fancy indexing* is a term adopted by NumPy to describe indexing using integer arrays. Suppose had an 8×4 array:

```
In [120]: arr = np.zeros((8, 4))
In [121]: for i in range(8):
.....:
arr[i] = i
In [122]: arr
Out[122]:
array([[0., 0., 0., 0.],
[1., 1., 1., 1.],
[2., 2., 2., 2.],
[3., 3., 3., 3.],
[4., 4., 4., 4.],
```

```
[5., 5., 5., 5.],
[6., 6., 6., 6.],
[7., 7., 7., 7.]])
```

To select a subset of rows in a particular order, can simply pass a list or ndarray of integers specifying desired order:

```
In [123]: arr[[4, 3, 0, 6]]
Out[123]:
array([[4., 4., 4., 4.],
[3., 3., 3., 3.],
[0., 0., 0., 0.],
[6., 6., 6., 6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from end:

```
In [124]: arr[[-3, -5, -7]]
Out[124]:
array([[5., 5., 5., 5.],
[3., 3., 3., 3.],
[1., 1., 1., 1.]])
```

Passing multiple index arrays does something slightly different; it selects a 1D array of elements corresponding to each tuple of indices:

```
In [125]: arr = np.arange(32).reshape((8, 4))
In [126]: arr
Out[126]:
array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11],
[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23],
[24, 25, 26, 27],
[28, 29, 30, 31]])
In [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[127]: array([ 4, 23, 29, 10])
```

To learn more about `reshape` method, have a look at Appendix A.
Here elements `(1, 0)`, `(5, 3)`, `(7, 1)`, `(2, 2)` were selected. Result of fancy indexing with as many integer arrays as there are axes is always 1D.
Behavior of fancy indexing in this case is a bit different from what some users might have expected, which is rectangular region formed by selecting a subset of matrix's rows & columns. 1 way to get that:

```
In [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[128]:
array([[ 4, 7, 5, 6],
[20, 23, 21, 22],
[28, 31, 29, 30],
[ 8, 11, 9, 10]])
```

Keep in mind: fancy indexing, unlike slicing, always copies data into a new array when assigning result to a new variable. If assign values with fancy indexing, indexed values will be modified:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[129]: array([ 4, 23, 29, 10])
In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0
In [131]: arr
Out[131]:
array([[ 0, 1, 2, 3],
[ 0, 5, 6, 7],
[ 8, 9, 0, 11],
[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 0],
[24, 25, 26, 27],
[28, 0, 30, 31]])
```

* Transposing Arrays & Swapping Axes. Transposing is a special form of reshaping that similarly returns a view on underlying data without copying anything. Arrays have `transpose` method & special `T` attribute:

```
In [132]: arr = np.arange(15).reshape((3, 5))
In [133]: arr
Out[133]:
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])
In [134]: arr.T
Out[134]:
array([[ 0, 5, 10],
[ 1, 6, 11],
[ 2, 7, 12],
[ 3, 8, 13],
[ 4, 9, 14]])
```

When doing matrix computations, may do this very often – e.g., when computing inner matrix product using `numpy.dot`:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])

In [16]: arr
Out[16]:
array([[ 0,  1,  0],
[ 1,  2, -2],
[ 6,  3,  2],
[-1,  0, -1],
[ 1,  0,  1]])

In [17]: np.dot(arr.T, arr)
Out[17]:
array([[39, 20, 12],
[20, 14,  2],
[12,  2, 10]])
```

`@infix` operator is another way to do matrix multiplication:

```
In [138]: arr.T @ arr
Out[138]:
array([[39, 20, 12],
[20, 14, 2],
[12, 2, 10]])
```

Simple transposing with `.T` is a special case of swapping axes. ndarray has method `swapaxes`, which takes a pair of axis numbers & switches indicated axes to rearrange data:

```
In [139]: arr
Out[139]:
array([[ 0, 1, 0],
[ 1, 2, -2],
[ 6, 3, 2],
[-1, 0, -1],
[ 1, 0, 1]])
In [140]: arr.swapaxes(0, 1)
Out[140]:
array([[ 0, 1, 6, -1, 1],
[ 1, 2, 3, 0, 0],
[ 0, -2, 2, -1, 1]])
```

`swapaxes` similarly returns a view on data without making a copy.

- 4.2. Pseudorandom Number Generation. `numpy.random` module supplements built-in Python `random` module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. e.g., can get a 4×4 array of samples from standard normal distribution using `numpy.random.standard_normal`:

```
In [141]: samples = np.random.standard_normal(size=(4, 4))
In [142]: samples
Out[142]:
```

```
array([[-0.2047, 0.4789, -0.5194, -0.5557],
[ 1.9658, 1.3934, 0.0929, 0.2817],
[ 0.769 , 1.2464, 1.0072, -1.2962],
[ 0.275 , 0.2289, 1.3529, 0.8864]])
```

Python's built-in `random` module, by contrast, samples only 1 value at a time. As can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [143]: from random import normalvariate
In [144]: N = 1_000_000
In [145]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.04 s +- 11.4 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
In [146]: %timeit np.random.standard_normal(N)
21.9 ms +- 155 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

These random numbers are not truly random (rather, *pseudorandom*) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like `numpy.random.standard_normal` use `numpy.random` module's default random number generator, but your code can be configured to use an explicit generator:

```
In [147]: rng = np.random.default_rng(seed=12345)
In [148]: data = rng.standard_normal((2, 3))
```

`seed` argument is what determines initial state of generator, & state changes each time `rng` object is used to generate data. Generator object `rng` is also isolated from other code which might use `numpy.random` module:

```
In [149]: type(rng)
Out[149]: numpy.random._generator.Generator
```

See Table 4.3: NumPy random number generator methods for a partial list of methods available on random generator objects like `rng`. Will use `rng` object created above to generate random data throughout rest of chap. Method: Description

* `permutation`: Return a random permutation of a sequence, or return a permuted range
* `shuffle`: Randomly permute a sequence in place
* `uniform`: Draw samples from a uniform distribution
* `integers`: Draw random integers from a given low-to-high range
* `standard_normal`: Draw samples from a normal distribution with mean 0 & standard deviation 1
* `binomial`: Draw samples from a binomial distribution
* `normal`: Draw samples from a normal (Gaussian) distribution
* `beta`: Draw samples from a beta distribution
* `chisquare`: Draw samples from a chi-square distribution
* `gamma`: Draw samples from a gamma distribution
* `uniform`: Draw samples from a uniform $[0, 1)$ distribution

- 4.3. Universal Functions: Fast Element-Wise Array Functions. A A universal function, or `ufunc`, is a function that performs element-wise operations on data in ndarrays. Can think of them as fast vectorized wrappers for simple functions that take 1 or more scalar values & produce 1 or more scalar results.

  Many ufuncs are simple element-wise transformations, like `numpy.sqrt` or `numpy.exp`:

```
In [150]: arr = np.arange(10)
In [151]: arr
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [152]: np.sqrt(arr)
In [152]: np.sqrt(arr)
Out[152]:
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

In [153]: np.exp(arr)
Out[153]:
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
2.98095799e+03, 8.10308393e+03])
```

These are referred to as *unary* ufuncs. Others, e.g. `numpy.add` or `numpy.maximum`, take 2 arrays (thus, *binary* ufuncs) & return a single array as result:

```
In [154]: x = rng.standard_normal(8)

In [155]: y = rng.standard_normal(8)

In [156]: x
Out[156]:
array([-1.3677927 ,  0.6488928 ,  0.36105811, -1.95286306,  2.34740965,
0.96849691, -0.75938718,  0.90219827])

In [157]: y
Out[157]:
array([-0.46695317, -0.06068952,  0.78884434, -1.25666813,  0.57585751,
1.39897899,  1.32229806, -0.29969852])

In [158]: np.maximum(x, y)
Out[158]:
array([-0.46695317,  0.6488928 ,  0.78884434, -1.25666813,  2.34740965,
1.39897899,  1.32229806,  0.90219827])
```

In this example, `numpy.maximum` computed element-wise maximum of elements in `x, y`.

While not common, a ufunc can return multiple arrays. `numpy.modf` is 1 example: a vectorized version of built-in Python `math.modf`, it returns fractional & integral parts of a floating-point array:

```
In [35]: arr = rng.standard_normal(7) * 5

In [36]: arr
Out[36]:
array([ 4.51459671, -8.10791367, -0.7909463 ,  2.24741966, -6.71800536,
-0.40843795,  8.62369966])

In [37]: remainder, whole_part = np.modf(arr)

In [38]: remainder
Out[38]:
array([ 0.51459671, -0.10791367, -0.7909463 ,  0.24741966, -0.71800536,
-0.40843795,  0.62369966])

In [39]: whole_part
Out[39]: array([ 4., -8., -0.,  2., -6., -0.,  8.])
```

Ufuncs accept an optional `out` argument that allows them to assign their results into an existing array rather than create a new one:

```
In [40]: arr
Out[40]:
array([ 4.51459671, -8.10791367, -0.7909463 ,  2.24741966, -6.71800536,
-0.40843795,  8.62369966])

In [41]: out = np.zeros_like(arr)

In [42]: np.add(arr, 1)
Out[42]:
array([ 5.51459671, -7.10791367,  0.2090537 ,  3.24741966, -5.71800536,
0.59156205,  9.62369966])

In [43]: np.add(arr, 1, out=out)
Out[43]:
array([ 5.51459671, -7.10791367,  0.2090537 ,  3.24741966, -5.71800536,
0.59156205,  9.62369966])

In [44]: out
```

```
Out[44]:
array([ 5.51459671, -7.10791367,  0.2090537 ,  3.24741966, -5.71800536,
0.59156205,  9.62369966])
```

See Table 4.4: Some unary universal functions & Table 4.5: Some binary universal functions for a listing of some of NumPy's ufuncs. New ufuncs continue to be added to NumPy, so consulting online NumPy documentation is best way to get a comprehensive listing & stay up to date.

- 4.4. Array-Oriented Programming with Arrays. Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is referred to by some people as *vectorization*. In general, vectorized array operations will usually be significantly faster than their pure Python equivalents, with biggest impact in any kind of numerical computations. Later, in Appendix A, explain *broadcasting*, a powerful method for vectorizing computations.

  E.g., suppose wished to evaluate function `sqrt(x^2 + y^2)` across a regular grid of values. `numpy.meshgrid` function takes 2 1D arrays & produces 2 2D matrices corresponding to all pairs of `(x, y)` in 2 arrays:

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 equally spaced points
In [170]: xs, ys = np.meshgrid(points, points)
In [171]: ys
Out[171]:
array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],
[-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
[-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
...,
[ 4.97, 4.97, 4.97, ..., 4.97, 4.97, 4.97],
[ 4.98, 4.98, 4.98, ..., 4.98, 4.98, 4.98],
[ 4.99, 4.99, 4.99, ..., 4.99, 4.99, 4.99]])
```

Now, evaluating function is a matter of writing same expression you would write with 2 points:

```
In [172]: z = np.sqrt(xs ** 2 + ys ** 2)
In [173]: z
Out[173]:
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
[7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
[7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
...,
[7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
[7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
[7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

As a preview of Chap. 9, use matplotlib to create visualizations of this 2D array:

```
In [174]: import matplotlib.pyplot as plt
In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
Out[175]: <matplotlib.image.AxesImage at 0x7f624ae73b20>
In [176]: plt.colorbar()
Out[176]: <matplotlib.colorbar.Colorbar at 0x7f6253e43ee0>
In [177]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[177]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values')
```

In Fig. 4.3: Plot of function evaluated on a grid, used matplotlib function `imshow` to create an image plot from a 2D array of function values.

If working in IPython, can close all open plot windows by executing `plt.close("all")`:

```
In [179]: plt.close("all")
```

Remark 27. *Term* vectorization *is used to describe some other computer science concepts, but in this book use it to describe operations on whole arrays of data at once rather than going value by value using a Python* `for` *loop.*

- Expressing Conditional Logic as Array Operations. `numpy.where` function is a vectorized version of ternary expression `x if condition else y`. Suppose had a Boolean array & 2 arrays of values:

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
In [182]: cond = np.array([True, False, True, True, False])
```

Suppose wanted to take a value from `xarr` whenever corresponding value in `cond` is `True`, & otherwise take value from `yarr`. A list comprehension doing this might look like:

```
In [183]: result = [(x if c else y)
.....:            for x, y, c in zip(xarr, yarr, cond)]
In [184]: result
Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

This has multiple problems. 1st, it will not be very fast for large arrays (because all work is being done in interpreted Python code). 2nd, it will not work with multidimensional arrays. With `numpy.where` you can do this with a single function call:

```
In [185]: result = np.where(cond, xarr, yarr)
In [186]: result
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

2nd & 3rd arguments to `numpy.where` don't need to be arrays; 1 or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose had a matrix of randomly generated data & wanted to replace all positive values with 2 & all negative values with -2. This is possible to do with `numpy.where`:

```
In [187]: arr = rng.standard_normal((4, 4))
In [188]: arr
Out[188]:
array([[ 2.6182, 0.7774, 0.8286, -0.959 ],
[-1.2094, -1.4123, 0.5415, 0.7519],
[-0.6588, -1.2287, 0.2576, 0.3129],
[-0.1308, 1.27 , -0.093 , -0.0662]])
In [189]: arr > 0
Out[189]:
array([[ True, True, True, False],
[False, False, True, True],
[False, False, True, True],
[False, True, False, False]])
In [190]: np.where(arr > 0, 2, -2)
Out[190]:
array([[ 2, 2, 2, -2],
[-2, -2, 2, 2],
[-2, -2, 2, 2],
[-2, 2, -2, -2]])
```

Can combine scalars & arrays when using `numpy.where`. E.g., can replace all positive values in `arr` with constant 2, like so:

```
In [191]: np.where(arr > 0, 2, arr) # set only positive values to 2
```

* Mathematical & Statistical Methods. A set of mathematical functions that compute statistics about an entire array or about data along an axis are accessible as methods of array class. Can use aggregations (sometimes called *reductions*) like `sum, mean, std` (standard deviation) either by calling array instance method or using top-level NumPy function. When use NumPy function, like `numpy.sum`, have to pass array you want to aggregate as 1st argument – phải truyền mảng bạn muốn tổng hợp làm đối số thứ nhất.

  Here generate some normally distributed random data & compute some aggregate statistics:

```
In [192]: arr = rng.standard_normal((5, 4))
In [193]: arr
Out[193]:
array([[-1.1082, 0.136 , 1.3471, 0.0611],
[ 0.0709, 0.4337, 0.2775, 0.5303],
[ 0.5367, 0.6184, -0.795 , 0.3
],
[-1.6027, 0.2668, -1.2616, -0.0713],
[ 0.474 , -0.4149, 0.0977, -1.6404]])
In [194]: arr.mean()
Out[194]: -0.08719744457434529
In [195]: np.mean(arr)
Out[195]: -0.08719744457434529
In [196]: arr.sum()
Out[196]: -1.743948891486906
```

Functions like `mean, sum` take an optional `axis` argument that computes statistic over given axis, resulting in an array with 1 less dimension:

```
In [197]: arr.mean(axis=1)
Out[197]: array([ 0.109 , 0.3281, 0.165 , -0.6672, -0.3709])
In [198]: arr.sum(axis=0)
Out[198]: array([-1.6292, 1.0399, -0.3344, -0.8203])
```

Here, `arr.mean(axis=1)` means "compute mean across columns," where `arr.sum(axis=0)` means "compute sum down rows."
Other methods like `cumsum` (tổng xuất tinh!) & `cumprod` (sản phẩm/tích tụ xuất tinh!) do not aggregate, instead producing an array of intermediate results:

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
In [200]: arr.cumsum()
Out[200]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like `cumsum` return an array of same size but with partial aggregates computed along indicated axis according to each lower dimensional slice:

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
In [202]: arr
Out[202]:
array([[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
```

Expression `arr.cumsum(axis=0)` computes cumulative sum along rows, while `arr.cumsum(axis=1)` computes sums along columns:

```
In [203]: arr.cumsum(axis=0)
Out[203]:
array([[ 0, 1, 2],
[ 3, 5, 7],
[ 9, 12, 15]])
In [204]: arr.cumsum(axis=1)
Out[204]:
array([[ 0, 1, 3],
[ 3, 7, 12],
[ 6, 13, 21]])
```

See Table 4.6: Basic array statistical methods for a full listing. See many examples of these methods in action in later chaps.
Method: Description

- `sum`: Sum of all elements in array or along an axis; zero-length arrays have sum 0
- `mean`: Arithmetic mean; invalid (returns `NaN`) on zero-length arrays
- `std, var`: Standard deviation & variance, resp.
- `min, max`: Minimum & maximum
- `argmin, argmax`: Indices of minimum & maximum elements, resp.
- `cumsum`: Cumulative sum of elements starting from 0
- `cumprod`: Cumulative product of elements starting from 1

* Methods for Boolean Arrays. Boolean values are coerced to 1 (`True`) & 0 (`False`) in preceding methods. Thus, `sum` is often used as a means of counting `True` values in a Boolean array:

```
In [205]: arr = rng.standard_normal(100)
In [206]: (arr > 0).sum() # Number of positive values
Out[206]: 48
In [207]: (arr <= 0).sum() # Number of non-positive values
Out[207]: 52
```

Parentheses here in expression `(arr > 0).sum()` are necessary to be able to call `sum()` on temporary result of `arr > 0`.
2 additional methods, $\boxed{\forall, \exists}$ `any, all` are useful especially for Boolean arrays. `any` tests whether 1 or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [208]: bools = np.array([False, False, True, False])
In [209]: bools.any()
Out[209]: True
In [210]: bools.all()
```

```
Out[210]: False
```

These methods also work with non-Boolean arrays, where nonzero elements are treated as `True`.

* Sorting. Like Python's built-in list type, NumPy arrays can be sorted in place with `sort` method:

```
In [45]: arr = rng.standard_normal(6)

In [46]: arr
Out[46]:
array([ 2.61815943,  0.77736134,  0.8286332 , -0.95898831, -1.20938829,
-1.41229201])

In [47]: arr.sort()

In [48]: arr
Out[48]:
array([-1.41229201, -1.20938829, -0.95898831,  0.77736134,  0.8286332 ,
2.61815943])
```

Can sort each 1D section of values in a multidimensional array in place along an axis by passing axis number to `sort`. In this example data:

```
In [49]: arr = rng.standard_normal((5, 3))

In [50]: arr
Out[50]:
array([[ 0.54154683,  0.7519394 , -0.65876032],
[-1.22867499,  0.25755777,  0.31290292],
[-0.13081169,  1.26998312, -0.09296246],
[-0.06615089, -1.10821447,  0.13595685],
[ 1.34707776,  0.06114402,  0.0709146 ]])
```

`arr.sort(axis=0)` sorts values within each column, while `arr.sort(axis=1)` sorts across each row:

```
In [51]: arr.sort(axis=0)

In [52]: arr
Out[52]:
array([[-1.22867499, -1.10821447, -0.65876032],
[-0.13081169,  0.06114402, -0.09296246],
[-0.06615089,  0.25755777,  0.0709146 ],
[ 0.54154683,  0.7519394 ,  0.13595685],
[ 1.34707776,  1.26998312,  0.31290292]])

In [53]: arr.sort(axis=1)

In [54]: arr
Out[54]:
array([[-1.22867499, -1.10821447, -0.65876032],
[-0.13081169, -0.09296246,  0.06114402],
[-0.06615089,  0.0709146 ,  0.25755777],
[ 0.13595685,  0.54154683,  0.7519394 ],
[ 0.31290292,  1.26998312,  1.34707776]])
```

Top-level method `numpy.sort` returns a sorted copy of an array (like Python built-in function `sorted`) instead of modifying array in place. E.g.:

```
In [221]: arr2 = np.array([5, -10, 7, 1, 0, -3])
In [222]: sorted_arr2 = np.sort(arr2)
In [223]: sorted_arr2
Out[57]: array([-10,  -3,   0,   1,   5,   7])
```

For more details on using NumPy's sorting methods, & more advanced techniques like indirect sorts, see Appendix A. Several other kinds of data manipulations related to sorting (e.g., sorting a table of data by 1 or more columns) can also be found in pandas.

* Unique & Other Set Logic. NumPy has some basic set operations for 1D ndarrays. A commonly used one is `numpy.unique`, which returns sorted unique values in an array:

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])
In [225]: np.unique(names)
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
In [227]: np.unique(ints)
Out[227]: array([1, 2, 3, 4])
```

Contrast `numpy.unique` with pure Python alternative:

```
In [228]: sorted(set(names))
Out[228]: ['Bob', 'Joe', 'Will']
```

In many cases, NumPy version is faster & returns a NumPy array rather than a Python list.
Another function, `numpy.in1d`, tests membership of values in 1 array in another, returning a Boolean array"

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])
In [230]: np.in1d(values, [2, 3, 6])
Out[230]: array([ True, False, False, True, True, False, True])
```

See Table 4.7: Array set operations for a listing of array set operations in NumPy.

- 4.5. File Input & Output with Arrays. NumPy is able to save & load data to & from disk in some text or binary formats. In this sect, discuss only NumPy's built-in binary format, since most users will prefer pandas & other tools for loading text or tabular data (see Chap. 6 for much more).

  `numpy.save, numpy.load` are 2 workhorse functions for efficiently saving & loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`:

  ```
  In [231]: arr = np.arange(10)
  In [232]: np.save("some_array", arr)
  ```

  If file path does not already end in `.npy`, extension will be appended. Array on disk can then be loaded with `numpy.load`:

  ```
  In [233]: np.load("some_array.npy")
  Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```

  Can save multiple arrays in an uncompressed archive using `numpy.savez` & passing arrays as keyword arguments:

  ```
  In [234]: np.savez("array_archive.npz", a=arr, b=arr)
  ```

  When loading an `.npz` file, get back a dictionary-like object that loads individual arrays lazily:

  ```
  In [235]: arch = np.load("array_archive.npz")
  In [236]: arch["b"]
  Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```

  If your data compresses well, may wish to use `numpy.savez_compressed` instead:

  ```
  In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
  ```

- 4.6. Linear Algebra. Linear algebra operations, like matrix multiplication, decompositions, determinants, & other square matrix math, are an important part of many array libraries. Multiplying 2 2D arrays with `*` is an element-wise product, while matrix multiplications require using a function. Thus, there is a function `dot`, both an array method & a function in `numpy` namespace, for matrix multiplication:

  ```
  In [241]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
  In [242]: y = np.array([[6., 23.], [-1, 7], [8, 9]])
  In [243]: x
  Out[243]:
  array([[1., 2., 3.],
  [4., 5., 6.]])
  In [244]: y
  Out[244]:
  array([[ 6., 23.],
  [-1., 7.],
  [ 8., 9.]])
  In [245]: x.dot(y)
  ```

```
Out[245]:
array([[ 28., 64.],
[ 67., 181.]])
```

x.dot(y) ⇔ np.dot(x, y):

```
In [246]: np.dot(x, y)
Out[246]:
array([[ 28., 64.],
[ 67., 181.]])
```

A matrix product between a 2D array & a suitably sized 1D array results in a 1D array:

```
In [247]: x @ np.ones(3)
Out[247]: array([ 6., 15.])
```

`numpy.linalg` has a standard set of matrix decompositions & things like inverse & determinant:

```
In [58]: from numpy.linalg import inv, qr

In [59]: X = rng.standard_normal((5, 5))

In [60]: mat = X.T @ X

In [61]: inv(mat)
Out[61]:
array([[10.58672281, -5.77153265, -5.93417247,  4.21878576, -0.98785662],
[-5.77153265,  3.80844049,  3.598377  , -2.30090931,  0.38710317],
[-5.93417247,  3.598377  ,  4.15734351, -2.40805923,  0.08345156],
[ 4.21878576, -2.30090931, -2.40805923,  3.05312068, -0.07713403],
[-0.98785662,  0.38710317,  0.08345156, -0.07713403,  0.55496624]])

In [62]: mat @ inv(mat)
Out[62]:
array([[ 1.00000000e+00, -7.58919943e-16, -1.86401885e-15,
-2.42458295e-15, -5.77969161e-17],
[ 5.29870145e-16,  1.00000000e+00, -3.45485441e-16,
3.05312695e-16,  1.43017029e-16],
[-6.35265213e-15, -2.20265557e-16,  1.00000000e+00,
-2.42543158e-15,  4.46456528e-16],
[ 4.74051610e-16, -8.60778149e-16,  5.01637067e-16,
1.00000000e+00, -1.77867729e-16],
[-1.78072592e-15,  1.16156604e-15,  4.76405781e-16,
-3.51857823e-16,  1.00000000e+00]])
```

Expression `X.T.dot(X)` computes dot product of `X` with its transpose `X.T`.
See Table 4.8: Commonly used `numpy.linalg` functions for a list of some of most commonly used linear algebra functions.

* `diag`: Return diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with 0s on off-diagonal
* `dot`: Matrix multiplication
* `trace`: Compute sum of diagonal elements
* `det`: Compute matrix determinant
* `eig`: Compute eigenvalues & eigenvectors of a square matrix
* `inv`: Compute inverse of a square matrix
* `pinv`: Compute Moore–Penrose pseudoinverse of a matrix
* `qr`: Compute QR decomposition
* `svd`: Compute singular value decomposition (SVD)
* `solve`: Solve linear system $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, where A: a square matrix
* lstsq: Compute least-squares solution to $A\mathbf{x} = \mathbf{b}$

◦ 4.7. Example: Random Walks. Simulation of random walks provides an illustrative application of utilizing array operations. 1st consider a simple random walk starting at 0 with steps of 1 & -1 occurring with equal probability.
A pure Python way to implement a single random walk with 1000 steps using built-in `random` module:

```
#! blockstart
import random
position = 0
walk = [position]
nsteps = 1000
for _ in range(nsteps):
step = 1 if random.randint(0, 1) else -1
position += step
walk.append(position)
#! blockend
```

See Fig. 4.4: A simple random walk for an example plot of 1st 100 values on 1 of these random walks:

```
In [255]: plt.plot(walk[:100])
```

Might make observation: `walk` is cumulative sum of random steps & could be evaluated as an array expression. Thus, use `numpy.random` module to draw 1000 coin flips at once, set these to ± 1, & compute cumulative sum:

```
In [256]: nsteps = 1000
In [257]: rng = np.random.default_rng(seed=12345) # fresh random generator
In [258]: draws = rng.integers(0, 2, size=nsteps)
In [259]: steps = np.where(draws == 0, 1, -1)
In [260]: walk = steps.cumsum()
```

From this, can begin to extract statistics like minimum & maximum value along walk's trajectory:

```
In [261]: walk.min()
Out[261]: -8
In [262]: walk.max()
Out[262]: 50
```

A more complicated statistic: *1st crossing time*, step at which random walk reaches a particular value. Here might want to know how long it took random walk to get at least 10 steps away from origin 0 in either direction. `np.abs(walk) >= 10` gives us a Boolean array indicating where walk has reached or exceeded 10, but want index of *1st* 10 or -10. Turns out, can compute this using `argmax`, which returns 1st index of maximum value in Boolean array (`True` is maximum value):

```
In [263]: (np.abs(walk) >= 10).argmax()
Out[263]: 155
```

Note: using `argmax` here is not always efficient because it always makes a full scan of array. In this special case, once a `True` is observed we know it to be maximum value.

* Simulating Many Random Walks at Once. If goal was to simulate many random walks, say 5000 of them, can generate all of random walks with minor modifications to preceding code. If passed a 2-tuple, `numpy.random` functions will generate a 2D array of draws, & can compute cumulative sum for each row to compute all 5000 random walks in 1 shot:

  ```
  In [65]: nwalks = 5000

  In [66]: nsteps = 1000

  In [67]: draws = rng.integers(0, 2, size=(nwalks, nsteps)) # 0 or 1

  In [68]: steps = np.where(draws > 0, 1, -1)

  In [69]: walks = steps.cumsum(axis=1)

  In [70]: walks
  Out[70]:
  array([[  1,   2,   1, ..., -34, -33, -34],
  [ -1,  -2,  -1, ...,   2,   3,   2],
  [  1,   0,   1, ..., -26, -27, -26],
  ...,
  [  1,   0,   1, ...,  24,  23,  24],
  [  1,   0,   1, ...,   4,   3,   2],
  [  1,   2,   3, ...,  12,  11,  10]])
  ```

Now, can compute maximum & minimum values obtained over all of walks:

```
In [270]: walks.max()
Out[270]: 114
In [271]: walks.min()
Out[271]: -120
```

Out of these walks, compute minimum crossing time to ±30. This is strictly tricky because not all 5000 of them reach 30. Can check this using `any` method:

```
In [272]: hits30 = (np.abs(walks) >= 30).any(axis=1)
In [273]: hits30
Out[273]: array([False, True,
True, ..., True, False, True])
In [274]: hits30.sum() # Number that hit 30 or -30
Out[274]: 3395
```

Can use this Boolean array to select rows of `walks` that actually cross absolute 30 level, & call `argmax` across axis 1 to get crossing times:

```
In [275]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(axis=1)
In [276]: crossing_times
Out[276]: array([201, 491, 283, ..., 219, 259, 541])
```

Lastly, compute average minimum crossing time:

```
In [277]: crossing_times.mean()
Out[277]: 500.5699558173785
```

Feel free to experiment with other distributions for steps other than equalized coin flips. Need only use a different random generator method, like `standard_normal` to generate normally distributed steps with some mean & standard deviation:

```
In [278]: draws = 0.25 * rng.standard_normal((nwalks, nsteps))
```

Remark 28. *Keep in mind: this vectorized approach requires creating an array with* `nwalks * nsteps` *elements, which may use a large amount of memory for large simulations. If memory is more constrained, then a different approach will be required.*

- 4.8. Conclusion. While much of rest of book will focus on building data wrangling skills with pandas – xây dựng kỹ năng xử lý dữ liệu với pandas, continue to work in a similar array-based style. In Appendix A, dig deeper into NumPy features to help you further develop your array computing skills.

- 5. Getting Started with pandas. pandas will be a major tool of interest throughout much of rest of book. It contains data structures & data manipulation tools designed to make data cleaning & analysis fast & convenient in Python. pandas is often used in tandem with numerical computing tools like NumPy & SciPy, analytical libraries like statsmodels & scikit-learn, & data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions & a preference for data processing without `for` loops.

While `pandas` adopts many coding idioms from NumPy, biggest difference: pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneously typed numerical array data.

Since becoming an open source project in 2010, pandas has matured into a quite large library that's applicable in a broad set of of real-world use cases. Developer community has grown to > 2500 distinct contributors, who've been helping build project as they used it to solve their day-to-day data problems. Vibrant pandas developer & user communities have been a key part of its success.

Remark 29. *Many people don't know that I haven't been actively involved in day-to-day pandas development since 2013; it has been an entirely community-managed project since then. Be sure to pass on your thanks to core development & all contributors for their hard work!*

Throughout rest of book, use following import conventions for NumPy & pandas:

```
In [1]: import numpy as np
In [2]: import pandas as pd
```

Thus, whenever you see `pd.` in code, it's referring to pandas. May also find it easier to import Series & DataFrame into local namespace since they are so frequently used:

```
In [3]: from pandas import Series, DataFrame
```

- **5.1. Introduction to pandas Data Structures.** To get started with pandas, will need to get comfortable with its 2 workhorse data structures: *Series & DataFrame*. while they are not a universal solution for every problem, they provided a solid foundation for a wide variety of data tasks.
  - **Series.** A Series is a 1D array-like object containing a sequence of values (of similar types to NumPy types) of same type & an associated array of data labels, called its *index*. Simplest Series is formed from only an array of data:

```
In [14]: obj = pd.Series([4, 7, -5, 3])
In [15]: obj
Out[15]:
0  4
1  7
2 -5
3  3
dtype: int64
```

String representation of a Series displayed interactively shows index on left & values on right. Since did not specify an index for data, a default one consisting of integers `0` through `N - 1` (where `N`: length of data) is created. Can get array representation & index object of Series via its `array` & `index` attributes, resp.:

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64
In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1)
```

Result of `.array` attribute is a `PandasArray` which usually wraps a NumPy array but can also contain special extension array types discussed more in Sect. 7.3: Extension Data Types.
Often, want to create a Series with an index identifying each data point with a label:

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])
In [19]: obj2
Out[19]:
d  4
b  7
a -5
c  3
dtype: int64
In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, can use labels in index when selecting single values or a set of values:

```
In [21]: obj2["a"]
Out[21]: -5
In [22]: obj2["d"] = 6
In [23]: obj2[["c", "a", "d"]]
Out[23]:
c  3
a -5
d  6
dtype: int64
```

Here `["c", "a", "d"]` is interpreted as a list of indices, even though it contains strings instead of integers.
Using NumPy functions or NumPy-like operations, e.g. filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve index-value link:

```
In [24]: obj2[obj2 > 0]
Out[24]:
d  6
b  7
c  3
dtype: int64
In [25]: obj2 * 2
Out[25]:
d  12
```

```
b  14
a -10
c  6
dtype: int64

In [26]: import numpy as np
In [27]: np.exp(obj2)
Out[27]:
d 403.428793
b 1096.633158
a 0.006738
c 20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dictionary, as it is a mapping of index values to data values. It can be used in many contexts where might use a dictionary:

```
In [28]: "b" in obj2
Out[28]: True
In [29]: "e" in obj2
Out[29]: False
```

Should have data contained in a Python dictionary, can create a Series from it by passing dictionary:

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}
In [31]: obj3 = pd.Series(sdata)
In [32]: obj3
Out[32]:
Ohio 35000
Texas 71000
Oregon 16000
Utah 5000
dtype: int64
```

A Series can be converted back to a dictionary with its `to_dict` method:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

When only passing a dictionary, index in resulting Series will respect order of keys according to dictionary's `keys` method, which depends on key insertion order. Can override this by passing an index with dictionary keys in order you want them to appear in resulting Series:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]
In [35]: obj4 = pd.Series(sdata, index=states)
```

- 5.2. Essential Functionality.
- 5.3. Summarizing & Computing Descriptive Statistics.
- 5.4. Conclusion.

- 6. Data Loading, Storage, & File Formats.
  - 6.1. Reading & Writing Data in Text Format.
  - 6.2. Binary Data Formats.
  - 6.3. Interacting with Web APIs.
  - 6.4. Interacting with Databases.
  - 6.5. Conclusion.
- 7. Data Cleaning & Preparation.
  - 7.1. Handling Missing Data.
  - 7.2. Data Transformation.
  - 7.3. Extension Data Types.
  - 7.4. String Manipulation.

- A.4. Advanced `ufunc` Usage.
- A.5. Structured & Record Arrays.
- A.6. More About Sorting.
- A.7. Writing Fast NumPy Functions with Numba.
- A.8. Advanced Array Input & Output.
- A.9. Performance Tips.

- B. More on IPython System.
  - B.1. Terminal Keyboard Shortcuts.
  - B.2. About Magic Commands.
  - B.3. Using Command History.
  - B.4. Interacting with OS.
  - B.5. Software Development Tools.
  - B.6. Tips for Productive Code Development Using IPython.
  - B.7. Advanced IPython Features.
  - B.8. Conclusion.

2 Miscellaneous

Tài liệu

[Gru19] Joel Grus. *Data Science from Scratch: First Principles with Python*. 2nd edition. O'Reilly Media Publisher, 2019, p. 403.

[McK22] Wes McKinney. *Python for Data Analysis: Data Wrangling with pandas, NumPy, & Jupyter*. 3rd edition. O'Reilly Media Publisher, 2022, p. 579.