Python Programming

The Big, Comprehensive Coding Workbook: 250+ Lessons on Python Basics, Data Analysis, Automation, Web Scraping and Machine Learning – Hands-On Labs with Full Solutions

Code24Lab · code24lab.com

Python Programming: The Big, Comprehensive Coding Workbook: 250+ Lessons on Python Basics, Data Analysis, Automation, Web Scraping and Machine Learning – Hands-On Labs with Full Solutions

First edition 2026

© 2026 Babala Özdemir, Füllinsdorf, Switzerland. All rights reserved. No part of this book may be reproduced or distributed without the written permission of the publisher.

Publisher: Babala Özdemir (Code24Lab), Füllinsdorf, Switzerland · code24lab.com

The lessons, examples and exercises were written for the Code24Lab courses with the help of AI tools and reviewed editorially. All companies, people, data and numbers in the examples are fictional. This book is for education only and is not professional advice. Independent training, not affiliated with or endorsed by the owners of any product, language or trademark mentioned. Despite careful checking, no guarantee of error-free content is given. Feedback is welcome at code24lab.com.

How to use this book

12 courses with 254 hands-on lessons on Python: clear explanations, code examples, labs, tasks, final exams and a full solutions appendix.

Every lesson has a goal, a short explanation with code examples, and tasks. Most labs show a small table (a *worksheet*) with the data, code or output of the lesson: you read it like a debugger or a spreadsheet and work out the answer yourself. Some lessons ask you to write code in a file; the starter code and the tasks are printed in the lesson.

How to work with a lab. 1) Read the explanation. 2) Look at the worksheet: the column letters and row numbers are printed around the table, so a cell like *B3* is easy to find. The **shaded cells** are your work area. 3) Work out the result of each task with pencil and paper, in a spreadsheet, or by running the code in your own editor, and write down your answer. 4) Turn to the **Appendix: Solutions** at the end of the book, compare, and correct. Use the hints when you are stuck.

A note on menus. Some hints describe the interactive Code24Lab editor in your browser. In a book you simply use your own tools; everything can be done with a free code editor, a spreadsheet and a database such as SQLite.

Interactive version. Every course in this book is also available as an interactive course on **code24lab.com**, where every answer is checked instantly.

This volume has 254 lessons.

Contents

Python Foundations: Variables, Loops, Functions & Files ... 10

Your First Script: Variables, Arithmetic and print() ... 11
Numbers: int, float, //, % and ** ... 13
Types and Conversion: str, int, float and type() ... 15
Strings I: Indexing, Slicing and len() ... 17
Strings II: Methods, split() and f-strings ... 19
Comparisons and Booleans: and, or, not ... 21
Decisions: if, elif and else ... 23
Lists: Index, Slice, append(), pop() and insert() ... 25
for Loops and range() ... 27
while Loops: Repeat Until a Condition Fails ... 29
Loop Patterns: Counters, Accumulators, break and continue ... 31
Dictionaries: Counting and Summing by Key ... 33
Sets, Tuples and Membership ... 35
Functions: def, Parameters, Defaults and return ... 37
Functions II: Composition, Return Values and Scope ... 39
Built-in Helpers: sum, max, sorted, zip and enumerate ... 41
Reading Files: open(), with and Parsing Lines ... 43
Writing Files: write(), Modes "w" and "a", and Newlines ... 45
Handling Errors: try, except and Clean Input ... 47
Final Project: The Daily Rental Report ... 49
Final exam ... 51

Python in Practice: OOP, Modules, Comprehensions & Testing ... 54

Classes & Objects: Model Products with __init__ and Methods ... 55
Class vs Instance Attributes: Shared Defaults and Shadowing ... 57
Properties: Validated Attributes with @property and Setters ... 60
Inheritance & super(): Extend Products without Copying Code ... 63
Polymorphism & MRO: One Call, Many Shipping Classes ... 66
Dunder Methods I: __repr__, __eq__, __lt__ and Sorting Objects ... 69
Dunder Methods II: Make a Cart Behave Like a List ... 71
Operator Overloading: A Money Class with +, * and Exact Cents ... 74
Dataclasses: Less Boilerplate, Ordering and __post_init__ ... 76
Modules: Import Once, Names, Bindings and reload() ... 78
Packages: __init__.py, Relative Imports and Import Cost ... 80
List Comprehensions: Map, Filter and Conditional Expressions ... 82
Dict & Set Comprehensions: Group, Deduplicate, Last-Wins ... 84
Built-ins with key=, Generators and next(): Idiomatic Searching ... 86
Exceptions: try, except, else, finally and Control Flow ... 88
Custom Exceptions: Hierarchies, raise and Handler Order ... 90
Regular Expressions I: Anchors, Classes and Quantifiers ... 92
Regular Expressions II: Groups, findall and Tuples ... 94
Testing with pytest: assert, parametrize and Boundary Cases ... 95
pytest Fixtures, Scopes and Coverage ... 97
Capstone: A Tested Checkout with Classes, Exceptions and Exact Cents ... 99
Final exam ... 102

Python Mastery: Decorators, Generators, Async & Performance ... 105

Functions as Objects: Closures & nonlocal State ... 106

Late Binding: The Loop-Variable Closure Trap ... 108
Writing Decorators: Wrappers, State & functools.wraps ... 110
Decorator Factories & Stacking Order ... 112
Memoization with lru_cache: Hits, Misses & Cache Growth ... 114
The Iterator Protocol: __iter__, __next__ & Exhaustion ... 116
Generators & Laziness: Short-Circuiting and Partial Consumption ... 118
send(), Priming & Generator-Based Coroutines ... 120
itertools in Practice: groupby, accumulate & Shared Iterators ... 122
Lazy Pipelines vs Lists: Memory Peaks & batched() ... 124
Context Managers: __enter__, __exit__ & Rollback ... 126
@contextmanager, try/finally & Nested Timers ... 128
Type Hints: Static Checks, Any & Run-Time Reality ... 130
asyncio Scheduling: await, gather & Semaphores ... 132
Blocking the Event Loop: time.sleep vs await & to_thread ... 134
Big-O in Practice: Counting Comparisons, list vs set ... 136
Profiling with cProfile: tottime, Hotspots & Amdahl's Law ... 138
__slots__ & Dataclasses: Memory per Object ... 140
Pythonic Design Patterns: Strategy & Registration Decorators ... 142
Packaging: pyproject.toml & Version Specifiers ... 144
Capstone: A Retrying, Batched, Rate-Limited Async Uploader ... 146
Final exam ... 148

Python Data Foundations: NumPy & Pandas Basics ... 151

Lists vs NumPy Arrays: Element-wise Maths ... 152
Creating Arrays: arange, linspace, zeros and shape ... 154
Indexing and Slicing 1-D Arrays ... 156
2-D Arrays: Rows, Columns and the axis Argument ... 158
Vectorised Maths and Broadcasting ... 160
Boolean Masks and np.where ... 162
Aggregations: median, std, argmax and cumsum ... 164
dtypes, Floor Division and Modulo ... 166
pandas Series: Values with Labels ... 168
DataFrame Anatomy: shape, head, tail and NaN ... 170
Selecting Data with loc and iloc ... 172
Filtering Rows with Conditions ... 174
Sorting, Top N and rank() ... 176
New Columns with Vectorised Arithmetic ... 178
describe(): std, Quartiles and the IQR ... 180
value_counts, unique and nunique ... 182
groupby: Split, Apply, Combine ... 184
Missing Data: isna, dropna and fillna ... 186
Reading and Writing CSV Files ... 188
Final Project: The Weekly Store Report ... 190
Final exam ... 192

Python Data Analysis: GroupBy, Merge, Time Series & Plots ... 194

First Look: Sum, Mean & value_counts on a Real Orders Table ... 195
Boolean Masks: Filtering Rows with &, | and ~ ... 197
groupby Basics: size, sum and mean per Region ... 200
size vs count: Groups with Missing Values ... 202
Named Aggregation: min, max and nunique in One Report ... 204
Two Keys and unstack: From MultiIndex to Cross-Table ... 206
transform: Group Totals and Shares on Every Row ... 208

Filtering Whole Groups: groupby().filter() as HAVING ... 210
Ranking Inside Groups: rank, cumcount and Top-N ... 212
merge on a Key: What an Inner Join Keeps and Drops ... 214
Left Joins, indicator=True and the Duplicate-Key Trap ... 216
concat and drop_duplicates: Combining Overlapping Exports ... 218
pivot_table: Default mean, aggfunc and margins ... 220
melt: From Wide Spreadsheets to Tidy Long Data ... 222
Missing Values: isna, fillna and dropna Without Fooling Yourself ... 224
Gaps in a Series: ffill, bfill and interpolate ... 226
Dates with the .dt Accessor: Weekdays, Weekends and Names ... 227
resample: Weekly Totals, Counts and Means from Raw Orders ... 229
Rolling Windows: Moving Averages and min_periods ... 231
shift, diff and pct_change: Month-over-Month Growth ... 232
Plotting with pandas: Line for Time, Bars for Categories ... 233
Final Project: Merge, Filter, Group and Rank a Margin Report ... 235
Final exam ... 237

Python Data Mastery: Performance, Pipelines & Advanced Pandas ... 239

Vectorization vs Loops: Measure What a Python Loop Costs ... 240
dtypes and Memory: Downcast Integers Safely ... 242
Categoricals: When Strings Get Cheap (and When They Do Not) ... 244
Vectorized Conditions: np.where and np.select ... 246
apply, iterrows and the Hidden Python Loop ... 248
Copy-on-Write and Safe Assignment with .loc ... 250
MultiIndex: Select by Tuple, Cross-Section and Level ... 252
stack, unstack and pivot: Reshape Without Loops ... 254
groupby transform: Group Statistics on Every Row ... 256
Rolling Windows: window, min_periods and center ... 257
expanding and ewm: Running Records and Smoothing ... 259
Time-Based Windows: rolling("3D") on Irregular Dates ... 261
shift, diff and pct_change per Group ... 263
Ranking Within Groups: rank Methods and Top-N per Group ... 265
merge with validate: Catch Row Explosions Before They Count Twice ... 267
Data Quality Checks: Duplicates, Ranges and the NaN Trap ... 269
Method Chaining and .pipe: Pipelines You Can Read ... 271
Chunked Processing: Combine Partial Results Correctly ... 273
Reproducible Results: Seeds, Key-Based Splits and Stable Sorts ... 275
Capstone: The Larkspur Weekly Revenue Pipeline ... 277
Final exam ... 279

Python Automation Foundations: Files, Excel, CSV & Email Tasks ... 282

Automation Thinking: Measure the Manual Work First ... 283
Paths with pathlib: name, stem, suffix and File Sizes ... 285
Finding Files: glob(), rglob() and Wildcards ... 287
Batch Renaming: sorted(), enumerate() and Zero Padding ... 289
Tidying a Folder: shutil.move(), dict.get() and Safe Skips ... 291
Reading CSV Files: csv.reader, Header Rows and float() ... 293
DictReader and Messy Data: strip(), try and except ... 295
Writing CSV Files: csv.writer, Modes "w" and "a" ... 297
JSON Config Files: json.load(), Nested Keys and json.dump() ... 299
JSON Records: Lists of Dicts, dict.get() and max(key=…) ... 301
Excel Workbooks I: Reading Cells with openpyxl ... 303
Excel Workbooks II: Writing Values, append() and Formulas ... 305

Text Processing: strip(), split() and Parsing Log Lines ... 307
Regular Expressions: Extract Invoice Numbers and Amounts ... 309
Dates and Times: date, timedelta and weekday() ... 311
Timestamps in File Names: strptime(), strftime() and Retention ... 313
Command-Line Arguments: sys.argv and argparse ... 315
Unattended Scripts: logging Levels and Retry Loops ... 317
Building Emails: EmailMessage, Attachments and Size Limits ... 319
Mail Merge: Templates, Filters and a Dry Run ... 321
Final Project: The Automated Daily Sales Summary ... 323
Final exam ... 325

Python Web Scraping & APIs: Requests, HTML Parsing & JSON ... 327

First Crawl: Status Codes, Bytes and Timing of requests.get ... 328
raise_for_status: Permanent vs Retryable Errors ... 330
Query Parameters and Pagination Math ... 332
Headers, Compression and Text vs Bytes ... 333
Timeouts and Sessions: What Waiting Really Costs ... 335
Beautiful Soup: find, find_all and Reading Fields per Card ... 337
CSS Selectors with select and select_one ... 339
Attributes, data-* Values and urljoin ... 342
Cleaning Scraped Text: Values, Units and Prices per 100 g ... 344
Scraping an HTML Table into a Rate Lookup ... 346
Pagination Loops: next Links, Short Pages and break ... 348
JSON APIs: r.json(), Nested Keys and Bearer Tokens ... 350
Flattening Nested JSON: Orders into Lines ... 352
Offset vs Cursor Pagination in JSON APIs ... 354
Rate Limits: Headers, Pacing and 429 ... 356
Retries with Exponential Backoff and a Cap ... 357
robots.txt, Crawl-delay and Polite Scraping ... 359
Conditional Requests: ETag, 304 and Caching ... 361
Deduplicating Scraped Rows and Handling Missing Values ... 363
Saving Results: csv.DictWriter, Quoting and Line Endings ... 365
Final Project: A Polite Weekly Price Monitor ... 367
Final exam ... 369

Python Mastery: FastAPI Services, Scheduling & Testing ... 371

FastAPI Routes: Query Parameters, 422 & Pagination ... 372
Pydantic Models: Coercion, Constraints & the 422 Error List ... 374
Dependency Injection: Depends, yield & the Per-Request Cache ... 376
Safe Retries: Idempotency Keys for POST Endpoints ... 378
async def or def? The Event Loop, the Thread Pool & Blocking Calls ... 379
BackgroundTasks: Work After the Response – and What Gets Lost ... 381
Job Queues & Workers: Utilization, Sizing & Little's Law ... 382
Retries Done Right: Exponential Backoff, Caps & Jitter ... 383
Rate Limiting: Fixed Windows, Sliding Estimates & 429 ... 385
Scheduling with Cron: Fields, Steps & Run Counts ... 386
Interval Jobs: Overlaps, max_instances & Skipped Runs ... 387
pytest Fixtures: Scopes, autouse & Set-up Counts ... 388
Parametrized Tests: Cross Products, Fixture Params & Skips ... 390
Mocks in pytest: side_effect, call_count & Retry Tests ... 392
Test Coverage: Lines, Branches & fail_under ... 394
Logging: Logger Levels, Handler Levels & Propagation ... 396
Structured Logs: Error Rate, p95 Latency & Filtered Means ... 398

Configuration: pydantic-settings, Env Vars & Precedence ... 400
Packaging the Service: Wheel Tags, abi3 & manylinux ... 402
Deployment: Worker Processes, Memory & CPU Capacity ... 404
Rolling Deploys: maxSurge, maxUnavailable & Health Probes ... 406
Capstone: A Scheduled, Rate-Limited, Retrying Import Job ... 408
Final exam ... 410

Python Machine Learning Foundations: Data Prep & Regression ... 413

Features & Target: Your First Look at an ML Dataset ... 414
Baseline First: DummyRegressor and MAE ... 416
Train/Test Split: Holding Data Back Honestly ... 418
Missing Values: SimpleImputer Without Leakage ... 420
StandardScaler: z-Scores with ddof=0 ... 422
MinMaxScaler: 0 to 1, and Test Data Outside the Range ... 424
One-Hot Encoding: Text Categories as Model Inputs ... 426
Correlation: Which Feature Moves the Target? ... 428
Simple Linear Regression: Slope, Intercept, Prediction ... 430
Residuals & Least Squares: Why This Line Wins ... 432
Error Metrics: MAE, MSE, RMSE and max_error ... 434
R^2: How Much Does the Model Explain? ... 435
Multiple Regression: Predicting with coef_ and intercept_ ... 437
Gradient Descent: One Step by Hand ... 438
Polynomial Features & Overfitting: The U-Curve ... 440
Bias vs Variance: Reading a Learning Curve ... 442
Ridge Regularisation: Paying for Large Weights ... 443
k-Fold Cross-Validation: Fold Sizes and Negative Scores ... 445
Logistic Regression: From Score to Probability ... 447
Log Loss: Grading Probabilities, Not Just Labels ... 449
Decision Threshold & Accuracy ... 451
Final Project: Train, Predict and Beat the Baseline ... 453
Final exam ... 455

Python Machine Learning in Practice: Classification & Pipelines ... 457

First Look: Predictions, Accuracy & Class Balance ... 458
The Confusion Matrix: TP, FP, FN and TN ... 460
Precision, Recall, F1 and Specificity ... 462
Multi-class Reports: Per-class, Macro and Weighted Averages ... 463
Moving the Threshold: the Precision–Recall Trade-off ... 465
ROC Curve: TPR and FPR at Every Threshold ... 467
AUC Two Ways: Trapezoids and Ranked Pairs ... 469
Gini Impurity: How a Tree Scores a Split ... 471
Entropy and Information Gain ... 472
Choosing the Best Split Among Candidates ... 473
Tracing a Decision Tree's Predictions ... 474
max_depth and Overfitting: Train vs Validation ... 476
k-NN: Euclidean and Manhattan Distances ... 477
k-NN Voting: k, predict_proba and Distance Weights ... 479
Feature Scaling for k-NN: StandardScaler and MinMaxScaler ... 481
k-Fold Cross-Validation: Folds, Mean and Std ... 483
GridSearchCV: Candidates, Fits and best_score_ ... 484
Pipelines: Fit on Train, Transform the Rest ... 486
ColumnTransformer and One-Hot Encoding ... 488
Imbalanced Classes: Baselines and class_weight ... 490

Choosing a Threshold by Business Cost ... 491
Capstone: Final Evaluation of the Churn Pipeline ... 493
Final exam ... 495

Python Machine Learning Mastery: Ensembles, Tuning & Model Evaluation ... 497

Why Ensembles Work: Averaging Errors Away ... 498
Bootstrap Samples and Out-of-Bag Rows ... 500
Random Forests: Soft Votes, Hard Votes and Decorrelation ... 502
Impurity-Based Feature Importance (MDI) by Hand ... 504
Permutation Importance and the Correlated-Feature Trap ... 506
AdaBoost: Weighted Error, Amount of Say and Reweighting ... 508
Gradient Boosting by Hand: Residuals, Stumps and Learning Rate ... 510
Boosting for Classification: Log-Odds, Newton Leaves, Probabilities ... 512
Learning Curves and Early Stopping ... 514
Ridge (L2) Regularization: Shrinking a Coefficient ... 516
Lasso (L1) and Soft Thresholding: Coefficients That Hit Zero ... 517
Grid Search: What It Costs and How It Chooses ... 519
Random Search and Successive Halving: Spending the Budget Well ... 521
Nested Cross-Validation and the Optimism of best_score_ ... 523
Data Leakage in Preprocessing: Fit on Train Only ... 525
Leakage Through Splits: Time Series and Groups ... 527
Calibration: Reliability Bins, ECE and the Brier Score ... 529
Platt Scaling and Log Loss ... 531
Choosing the Decision Threshold by Cost ... 533
Class Imbalance: Weights, Baselines and Honest Metrics ... 535
Regression Metrics: MAE, RMSE, MAPE and Pinball Loss ... 536
Capstone: Select, Justify and Price the Final Model ... 538
Final exam ... 540

Appendix: Solutions ... 542

Python Foundations: Variables, Loops, Functions & Files

Learn Python by thinking like the interpreter: trace variables, strings, lists, loops, dictionaries, functions, files and error handling for a fictional bike rental, with every answer checked as you go. · Beginner · 20 lessons

- Your First Script: Variables, Arithmetic and print()
- Numbers: int, float, //, % and **
- Types and Conversion: str, int, float and type()
- Strings I: Indexing, Slicing and len()
- Strings II: Methods, split() and f-strings
- Comparisons and Booleans: and, or, not
- Decisions: if, elif and else
- Lists: Index, Slice, append(), pop() and insert()
- for Loops and range()
- while Loops: Repeat Until a Condition Fails
- Loop Patterns: Counters, Accumulators, break and continue
- Dictionaries: Counting and Summing by Key
- Sets, Tuples and Membership
- Functions: def, Parameters, Defaults and return
- Functions II: Composition, Return Values and Scope
- Built-in Helpers: sum, max, sorted, zip and enumerate
- Reading Files: open(), with and Parsing Lines
- Writing Files: write(), Modes "w" and "a", and Newlines
- Handling Errors: try, except and Clean Input
- Final Project: The Daily Rental Report

Your First Script: Variables, Arithmetic and print()

Lesson 1 · about 15 min

> **Goal.** You write a real Python script that stores data in variables, calculates, decides with if/else, uses a function and a loop, and prints a clean receipt for Bluefin Bikes.

Your mission today

You just started helping **Bluefin Bikes**, a small (fictional) bike rental by the lake. Every evening the owner adds up the day by hand. In this lab you write the Python script that does it for her: from the first variable to a printed receipt. Real Python runs right here in your browser, and your file grows task by task.

How Python reads your file

A **script** is a list of instructions. Python reads it **from top to bottom, one line at a time**, like a recipe. That is why a name must be created before you use it: if line 5 uses `rate` but `rate` only appears on line 9, Python stops with a `NameError`.

Variables are labelled boxes

`rate = 4.5` puts the value 4.5 in a box with the label `rate`. The = means "store", not "is equal". Later lines can read the box, and if the price changes you edit one line instead of hunting for every 4.5 in your file.

```
bikes = 12           # int: whole number
rate = 4.5           # float: decimal number, always with a dot
shop_name = "Bluefin Bikes"   # str: text, always in quotes
print(type(rate))    # <class 'float'>
```

Text and numbers are different things

`12` is a number you can calculate with; `"12"` is text that only looks like a number. Python refuses to guess what `"Bikes: " + 7` should mean, so you convert explicitly with `str(7)`, or you use an **f-string**, which does the conversion for you:

```
revenue = 189.0
print(f"Revenue: ${revenue:.2f}")   # Revenue: $189.00
```

> **Tip.** If you get stuck, open the hint first: it gives you the key idea without spoiling the whole answer.

All companies, names and numbers are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **Store the shop's facts.** Bluefin Bikes rents out bikes by the hour. `shop_name` is already there. Below it, create `bikes = 12` and `rate = 4.5` (the price per hour in dollars).

Starter code:

```
# Bluefin Bikes: end-of-day report
# Python runs this file from top to bottom.

# Task 1: store the shop's facts in variables
shop_name = "Bluefin Bikes"
```

2. **Let Python do the maths.** Today the bikes were rented for `hours_today` hours in total. Create `revenue` that multiplies `hours_today` by `rate`.
3. **Print with an f-string.** Print exactly `Bluefin Bikes earned $189.00 today`, built from `shop_name` and `revenue` with an f-string. Show the money with two decimals.
4. **Mix text and numbers.** Create `available = bikes - out_now`. Then build `label` as `"Bikes available: "` joined with the number, and print it. Watch out: `"text" + 7` crashes in Python.
5. **Decide with if/else.** If `revenue` is 150 or more, set `mood = "Great day"`, otherwise `mood = "Quiet day"`. Then `print(mood)`.
6. **Write your first function.** Define `rental_price(hours)` that **returns** the price of one ride: `hours * rate`. Example: `rental_price(2)` gives `9.0`.
7. **Loop over the rides.** `rentals` lists this morning's rides in hours. Write a `for` loop that adds `rental_price(hours)` of each ride to `total`.
8. **Print the receipt.** Finish the day: loop over `rentals` again and print one line per ride like `2 h: $9.00`. After the loop, print `Total: $47.25 from 4 rides` using `total` and `len(rentals)`.

Numbers: int, float, //, % and **

Lesson 2 · about 16 min

> **Goal.** You pack Bluefin's helmet delivery into boxes with // and %, turn minutes into hours, grow savings with ** and see exactly what // does with a negative balance.

Today at Bluefin Bikes

A delivery of **47 new helmets** just arrived at Bluefin Bikes, and they have to be packed into boxes of 6 for the branch shops. How many boxes? How many helmets are left? How long will packing take? These are questions about **whole numbers and leftovers**, and Python has exactly the right tools for them. Your script starts fresh but keeps the shop's numbers from last time.

Two kinds of numbers: int and float

An **int** is a whole number (`47`, `-17`), a **float** has a decimal point (`4.5`, `3.0`). The plain slash `/` is **true division** and **always** gives a float, even when the result is whole: `12 / 4` is `3.0`. That is perfect for averages, but wrong for counting boxes.

// and %: the box-packing pair

`//` (floor division) answers "how many **full** groups?", `%` (remainder, or modulo) answers "what is **left over**?". Think of a muffin tray: 47 muffins in trays of 6 fill 7 trays, and 5 muffins wait for the next tray.

```
print(47 // 6)        # 7  full boxes
print(47 % 6)         # 5  helmets left over
print(7 * 6 + 5)      # 47 (a // b) * b + a % b == a, always
print(135 // 60, 135 % 60)   # 2 15  -> 2 h 15 min
```

A comparison gives `True` or `False`, and in arithmetic `True` counts as `1` and `False` as `0`. So `full + (left > 0)` adds one extra box only when something is left over.

Powers with **

`2 ** 3` is `8`. Powers are perfect for growth: money at 2 % interest for 3 years is `500 * 1.02 ** 3`, because each year multiplies by 1.02 again. `**` is calculated before `*` and `+`. Careful: `^` is **not** power in Python (it is a bit operation for ints and an error for floats).

Negative numbers: // rounds down

`//` always rounds **down**, towards minus infinity, not towards zero. `-17 / 5` is -3.4, so `-17 // 5` is **-4**, and `-17 % 5` is **3** (because -4 × 5 + 3 = -17). If you want to cut towards zero instead, use `int(-17 / 5)`, which gives -3. Many other languages cut towards zero, so this surprises a lot of people.

> **Tip.** Floats are stored in binary, so `0.1 + 0.2` prints `0.30000000000000004`. That is normal. For money, round when you show or store a result: `round(x, 2)` or `f"{x:.2f}"`.

All companies, names and numbers are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **Average ride with `/`.** Bluefin's bikes were out for `hours_today` hours across `rides_today` rides. Create `avg_ride` with true division `/`, then print `Average ride: 3.5 h` with an f-string.

   Starter code:

   ```
   # Bluefin Bikes: helmet delivery and the numbers behind it
   # int = whole number (47), float = decimal number (4.5)

   shop_name = "Bluefin Bikes"
   bikes = 12
   rate = 4.5
   hours_today = 42
   rides_today = 12

   # Task 1: average ride length with true division /
   ```

2. **Full boxes with `//`.** 47 helmets arrive and 6 fit in one box. Create `full_boxes`: how many boxes are **completely** full? Use floor division `//`.
3. **The rest with `%`.** Create `left_over`: the helmets that do not fill a whole box, using the remainder operator `%`. Then print `7 full boxes, 5 helmets left over` from your two variables.
4. **Count every box.** Define `boxes_needed(items, size)` that returns all boxes needed, including one for the leftovers. `boxes_needed(47, 6)` gives 8, but `boxes_needed(48, 6)` gives 8 too: no empty extra box!
5. **Minutes into hours.** The longest ride today was 135 minutes. Define `format_duration(minutes)` that **returns** text like `2 h 15 min` (hours and leftover minutes).
6. **Interest with `**`.** The deposit money earns 2 % interest per year, and the interest earns interest too. Define `savings_after(years)` that returns `deposit * (1 + interest) ** years`, rounded to 2 decimals. `savings_after(3)` gives `530.6`.
7. **The negative trap.** A customer owes $17, so `balance = -17`. Create `fives = balance // 5`, `rest = balance % 5` and `cut = int(balance / 5)`, then print all three. Before you run it: guess the values!
8. **The packing report.** Loop over `shipments`: for each `n`, print `47 helmets: 8 boxes` using `boxes_needed(n, per_box)` and add the boxes to `total_boxes`. After the loop, print `Total: 19 boxes, packing time 1 h 16 min` (4 minutes per box, via `format_duration`).

Types and Conversion: str, int, float and type()

Lesson 3 · about 16 min

> **Goal.** You turn text from Bluefin's booking form into int and float, glue numbers back into text with str(), avoid the int() and bool() traps and total the evening's web orders.

Today at Bluefin Bikes

Bluefin Bikes now takes bookings on its website. The first evening batch looks fine at first sight, but your script prints `33` bikes where Mia ordered 3 twice. The reason: **everything a web form sends is text**, even when it looks like a number. In this lab you learn to see the type of a value and to convert it on purpose.

Every value has a type

Python always knows what kind of value it is holding, and `type()` tells you. The type decides what operators do with the value, so it is the first thing to check when a result looks strange.

```
print(type(3))       # <class 'int'>    whole number
print(type(3.0))     # <class 'float'>  decimal number
print(type("3"))     # <class 'str'>    text
print(type(True))    # <class 'bool'>   True or False
```

Same operator, different meaning

For numbers, `+` adds and `*` multiplies. For strings, `+` **joins** and `*` **repeats**: `"3" + "3"` is `"33"`, and so is `"3" * 2`. Mixing them, like `"Bikes: " + 3`, is a `TypeError`: Python refuses to guess whether you meant text or maths.

Converting on purpose

- `int("42")` → `42`, but `int("12.50")` raises a `ValueError`: int() only reads whole-number text.
- `float("12.50")` → `12.5`, the safe choice for decimal text.
- `int(9.99)` → `9`: int() **cuts** the decimals off (towards zero). It never rounds; that is `round()`'s job.
- `str(37)` → `"37"`, so you can glue a number into a sentence.

Think of a currency exchange desk: you cannot pay in a shop with a foreign note, you exchange it first. Converting a value is that exchange, and you do it once, at the border, right where the data comes in.

The bool() trap

`bool()` of a string only checks whether it is empty. `bool("")` is `False`, but `bool("no")` and even `bool("False")` are `True`. To read a checkbox value, compare the text: `text == "yes"`.

> **Tip.** Error messages name the problem: `ValueError` means the text cannot be read as that type, `TypeError` means two types do not fit together. Read the last line of the message first.

All companies, names and numbers are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **What type is it?.** The form sent Mia's booking: `qty_text = "3"`. Store `type(qty_text)` in `qty_type` and print it. Is it a number?
   Starter code:

   ```
   # Bluefin Bikes: the new online booking form
   # Everything a web form sends arrives as text (str), even numbers.

   shop_name = "Bluefin Bikes"
   rate = 4.5

   # What the form sent for Mia's booking:
   qty_text = "3"
   price_text = "12.50"
   hours_text = "2"

   # Task 1: ask Python for the type of qty_text
   ```

2. **Text * 2 is not 6.** Mia wants twice as many bikes. Turn the text into a whole number with `qty = int(qty_text)`, create `doubled = qty * 2` and print `Doubled: 6`. Curious? Try `print(qty_text * 2)` first.
3. **A price with cents.** Convert `price_text` into `price`, calculate `cost = qty * price` and print `Cost: $37.50`.
4. **Back to text with str().** Build `summary = "3 bikes for 2 h"` from `qty` and `hours_text` by joining strings with `+`, then print it. Do not type the numbers yourself.
5. **int() cuts, it does not round.** Bluefin gives 1 loyalty point per **full** dollar. Define `points_for(amount)` that returns the points as an int (`points_for(9.99)` is 9). Then print `Mia earns 37 points` with `points_for(cost)`.
6. **The conversion that crashes.** Hours arrive as `"2"`, `"2.0"` or `"3.75"`. Define `hours_from_form(text)` that returns the full hours as an int, for all of them, without crashing.
7. **The bool() trap.** Create `surprise = bool("no")` and print it (guess first!). Then define `wants_helmet(text)` that returns `True` only for the text `"yes"`.
8. **Process the web orders.** Define `order_total(qty_text, price_text, helmet_text)`: bikes × price, plus `helmet_fee` when `wants_helmet` says so. Then loop with `for q, p, h in web_orders:`, print `3 x $12.50, helmet: True, total $40.50` for each order, add up `evening_total` and finish with `Evening total: $63.00`.

Strings I: Indexing, Slicing and len()

Lesson 4 · about 16 min

> **Goal.** You take Bluefin's booking codes apart with indexes, negative indexes and slices, counting from 0, mask the booking number and check broken codes without an IndexError.

Today at Bluefin Bikes

Every web booking at Bluefin Bikes now gets a code like `BLF-2026-ZRH-0417`: shop, year, branch and booking number. The staff at the counter need the parts separately, the printed receipt must hide the number, and one partner site sends broken codes. Today you take strings apart, character by character.

A string is a row of numbered boxes

Picture the code as a row of lockers, one character per locker. The numbers on the doors start at **0**, not 1, because an index says how many steps you move from the start. The first locker is zero steps away.

```
code = "BLF-2026-ZRH-0417"
#       B  L  F  -  2  0  2  6  -  Z  R  H  -  0  4  1  7
#       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
print(len(code))   # 17
print(code[0])     # B
print(code[4])     # 2
print(code[-1])    # 7  (negative indexes count from the end)
```

`len(code)` is 17, so the last index is **16**. `code[17]` raises an `IndexError`: there is no locker 17. That is why `code[-1]` is so handy: it is always the last character, whatever the length.

Slices: start included, stop excluded

`code[start:stop]` gives a new string from `start` up to, **but not including**, `stop`. The nice side effect: `stop - start` is the number of characters, so `code[4:8]` is exactly 4 characters, `"2026"`.

```
print(code[4:8])    # 2026
print(code[:3])     # BLF   (from the start)
print(code[13:])    # 0417  (to the end)
print(code[-4:])    # 0417  (last four)
print(code[:-4])    # BLF-2026-ZRH-  (all but the last four)
```

Strings cannot be changed

`code[0] = "X"` fails with a `TypeError`: strings are **immutable**. Every slice and every `+` builds a **new** string, so you write `new = code[:-4] + "****"` instead.

> **Tip.** Slices are forgiving, single indexes are not: `code[10:99]` simply stops at the end and `"BLF"[5:6]` is `""`, but `"BLF"[5]` crashes. Use that when data might be too short.

All companies, names and numbers are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **Count the characters.** Create `code_length` with `len()` and print `Code length: 17`. The dashes count too!
   Starter code:

```
# Bluefin Bikes: reading booking codes
# Every web booking gets a code: shop - year - branch - number
#
#   B  L  F  -  2  0  2  6  -  Z  R  H  -  0  4  1  7
#   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16

shop_name = "Bluefin Bikes"
code = "BLF-2026-ZRH-0417"

# Task 1: how many characters does the code have?
```

2. **Index 0 is the first.** Store the first character in `first` and the last one in `last` (use a negative index), then `print(first, last)`. It should show `B 7`.
3. **Your first slices.** Cut out `prefix` (`"BLF"`) and `year` (`"2026"`) with slices `code[start:stop]`. Use the ruler in the comment.
4. **Branch and number.** Create `branch = code[9:12]` and `number` with the last four characters (slice from `-4`). Turn it into `booking_no` with `int()` and print `Booking 417 at ZRH`.
5. **Slices inside functions.** Define `branch_of(c)` that returns the branch letters and `year_of(c)` that returns the year as an int. They must work for every code, e.g. `branch_of("BLF-2025-BSL-0003")` gives `"BSL"`.
6. **Hide the number.** Define `masked(c)` that returns the code with the last four characters replaced by `****`: `"BLF-2026-ZRH-****"`.
7. **Slices never crash.** Define `dashes(c)` that returns the characters at index 3, 8 and 12 joined together (`"---"` for a good code). For the short code `"BLF-26"` it must not crash.
8. **The booking code report.** Loop over `todays_codes`. If `dashes(c) == "---"`, count it in `valid` and print `BLF-2026-ZRH-****: booking 417 at ZRH, 2026`; otherwise print `BLF-26-GVA: broken code`. After the loop print `3 of 4 codes are valid`.

Strings II: Methods, split() and f-strings

Lesson 5 · about 17 min

> **Goal.** You clean customer input from Bluefin's booking form with strip(), title(), replace(), find() and lower(), split a booking row into parts and print finished receipts with zfill() and formatted f-strings.

Today at Bluefin Bikes

The booking form is live, and customers type whatever they like: `" theo BAUMANN "`, phone numbers with dashes, e-mails in capitals. Before anything is printed on a receipt it has to be cleaned. Last time you cut strings with indexes and slices; today you use the ready-made tools every string carries: its **methods**.

Methods: tools that come with the string

A method is a function that belongs to a value. You call it with a dot and brackets: `name.strip()`. Think of a Swiss army knife: the string is the knife, and `strip`, `title`, `replace` are the blades you fold out.

```
raw = "  theo BAUMANN  "
print(raw.strip())            # 'theo BAUMANN'  spaces at both ends gone
print(raw.strip().title())    # 'Theo Baumann'  every word capitalised
print("zrh".upper())          # 'ZRH'
print("079 123".replace(" ", ""))   # '079123'
print("a@b.example".find("@"))      # 1  (index, or -1 if not found)
```

The original never changes

Strings are immutable, so a method cannot change the string it is called on. It **returns a new one**. `raw.strip()` on its own line does nothing useful; you have to store the result: `name = raw.strip()`. Because every method returns a string, you can **chain** them: `raw.strip().title()`.

split(): from one text to a list

`"helmet,lock".split(",")` cuts at every comma and returns the list `["helmet", "lock"]`. You pick a part by its position, starting at 0. Without an argument, `split()` cuts at whitespace instead. Careful: `"".split(",")` is `[""]`, one empty part, not an empty list.

f-strings with formats

In `f"{name}: ${cost:.2f}"` any expression can go inside the braces, and after a colon you add a **format**: `.2f` means two decimals, so 13.5 is shown as `13.50`. For leading zeros, the string method `str(417).zfill(6)` gives `"000417"`.

> **Tip.** Not sure what a method returns? Print it with square brackets around it, like `print(f"[{name}]")`. Hidden spaces become visible at once.

All companies, names, phone numbers and e-mail addresses are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **strip() the spaces.** Customers type extra spaces. Create `name = name_raw.strip()` and print it in square brackets: `[theo BAUMANN]`.

   Starter code:

   ```
   # Bluefin Bikes: cleaning up what customers type into the booking form
   # String methods return a NEW string. The original stays as it is.

   shop_name = "Bluefin Bikes"
   rate = 4.5
   name_raw = "  theo BAUMANN  "

   # Task 1: remove the spaces at both ends
   ```

2. **Chain strip() and title().** Define `clean_name(raw)` that removes the outer spaces **and** starts every word with a capital: `clean_name(" theo BAUMANN ")` gives `"Theo Baumann"`. Print `clean_name(name_raw)`.
3. **replace() spaces and dashes.** Define `clean_phone(p)` that removes all spaces and dashes: `"079 123-45 67"` becomes `"0791234567"`. Print `clean_phone(phone_raw)`.
4. **split() into a list.** Cut `extras_text` at the commas into the list `extras`, store the second extra in `second` and `print(len(extras), second)`: `3 lock`.
5. **Pad with zfill().** Define `order_ref(n)` that returns `"BLF-"` plus the number padded to 6 digits: `order_ref(417)` gives `"BLF-000417"`. Print it.
6. **Format with an f-string.** Define `receipt_line(name, hours)` that returns `Theo Baumann: 2.5 h, $11.25` (price = hours × `rate`, always 2 decimals). Print `receipt_line(clean_name(name_raw), 2.5)`.
7. **find() and slice.** Define `email_domain(e)` that returns everything after the `@`, in small letters: `"Theo.Baumann@LakeMail.example"` gives `"lakemail.example"`. Print `email_domain(email)`.
8. **Tonight's receipts.** Define `receipt(row)`: split the row at `;`, clean the name, build the order reference, count the extras and return `BLF-000417 Theo Baumann | 2.5 h | extras: 2 | $15.25` (hours × `rate` + extras × `extra_fee`). Then print a receipt for every row in `bookings`.

Comparisons and Booleans: and, or, not

Lesson 6 · about 16 min

> **Goal.** You turn Bluefin's e-bike rules into comparisons and boolean functions with and, or and not, test the exact age limits, avoid the truthy-text traps and check in eight customers.

Today at Bluefin Bikes

Bluefin now rents e-bikes, and the insurance comes with rules: minimum age, helmet or guided tour, standard or reduced price. Mia used to check every customer by hand at the counter. Today you turn those rules into Python expressions that answer with a clear **yes** or **no**, so the evening check-in runs by itself.

Comparisons give a bool

A comparison is a question, and its answer is a value of type `bool`: `True` or `False` (capital letter, no quotes). You can store it, print it and return it like any other value. The six operators are `==`, `!=`, `<`, `<=`, `>` and `>=`.

```
age = 17
print(age >= 18)        # False
print(age == 17)        # True   (== compares, a single = assigns!)
print(age != 17)        # False
print(16 <= age < 65)   # True   a chain: 16 <= age and age < 65
print("ZRH" == "zrh")   # False  text is compared exactly
```

and, or, not

- `a and b` is True only if **both** are True.
- `a or b` is True if **at least one** is True.
- `not a` flips the value.
- `and` is evaluated before `or`, like `*` before `+`. Use brackets to make the meaning obvious: `age >= 16 and (helmet or guided)`.

Truthy and falsy

Every value can stand in a condition. A few values count as **falsy**: `False`, `None`, `0`, `0.0`, the empty text `""` and empty collections like `[]` and `{}`. **Everything else is truthy**, including `"0"`, `"False"`, `" "` and `[0]`. Think of it as asking: is there anything in it? That is also why `bool("no")` is True: the word "no" is not empty.

> **Tip.** Boundaries cause most bugs. Whenever you write a comparison, test the exact limit: does 18 count? Does 65? One = more or less changes the answer.

All companies, names and rules are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **A comparison is a question.** Ask Python whether the customer is an adult: `adult = age >= 18`. Then `print(adult, type(adult))`: `False <class 'bool'>`.

   Starter code:

   ```
   # Bluefin Bikes: the e-bike rules
   # A comparison asks a yes/no question. The answer is a bool: True or False.

   shop_name = "Bluefin Bikes"
   rate = 4.5
   age = 17

   # Task 1: is this customer an adult (18 or older)?
   ```

2. **Return a bool.** Define `is_adult(age)` that returns `True` for 18 and older. Then `print(is_adult(17), is_adult(18))`.
3. **Chained comparison.** Standard price is for ages 16 up to, but not including, 65. Define `standard_price(age)` with one chained comparison and `print(standard_price(17), standard_price(65))`.
4. **and, or and brackets.** Define `can_ebike(age, helmet, guided)`: the rider must be at least 16, **and** wear a helmet **or** ride on a guided tour. Print `can_ebike(17, True, False), can_ebike(12, False, True)`: `True False`.
5. **Flip it with not.** Everybody who does not pay the standard price pays the reduced one. Define `reduced_price(age)` using your `standard_price` and `not`. Print it for 12, 40 and 70.
6. **Truthy and falsy text.** Define `has_note(note)` that returns `True` only if the note contains real characters. A note of only spaces counts as no note. Print `has_note(note), has_note("late return")`.
7. **Count the falsy values.** Define `count_falsy(values)` that counts how many values in a list are falsy. Then `print(count_falsy(lineup))`.
8. **Tonight's e-bike check.** Define `can_ride(row)`: split the row at `;`, turn the age into a number and `"yes"/"no"` into True/False, then use `can_ebike`. Print every name with its result (`Lea False`) and finish with `Allowed: 4`.

Decisions: if, elif and else

Lesson 7 · about 17 min

> **Goal.** You price Bluefin's rentals with if, elif and else, prove why the order of the branches matters, avoid the two-ifs and or-text traps and print the day's price list.

Today at Bluefin Bikes

Bluefin's new price list has tiers: short rides cost more per hour, long rides less. There are discounts by age, a late fee and a surcharge for partner stations. Last time your code answered yes/no questions; today it **acts** on the answers and picks exactly one price for every rental.

if, elif, else

```
if hours <= 2:
    price = hours * 6
elif hours <= 5:
    price = 12 + (hours - 2) * 4
else:
    price = 24 + (hours - 5) * 2
```

- Every condition line ends with a **colon**.
- The block that belongs to it is **indented by 4 spaces**. Indentation is not decoration in Python: it decides which lines are inside the branch.
- `elif` ("else if") adds another condition; you can have as many as you need.
- `else` has no condition. It catches everything that is left.

Only the first true branch runs

Python checks the conditions from top to bottom and runs **only the first branch whose condition is True**. All the others are skipped, even if they would also be True. With `hours = 4`, `hours <= 2` is False, `hours <= 5` is True, so the price is 20 and the else is never looked at. That is also why the elif does not need `2 < hours`: if we get there, the first condition already failed.

Why the order matters

Think of a sorting machine with several holes: a parcel drops into the **first** hole it fits. If the large hole comes first, the small parcels fall through it too and the small hole never gets anything. So put the **narrowest** condition first. Python never warns you about a branch that can never run; it simply gives the wrong answer.

> **Tip.** Test every rule at its edges: exactly 2 hours, exactly 15 minutes, exactly 65 years. Most branch bugs live on the boundary.

All companies, names, prices and rules are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **Your first if.** Rentals of more than 3 hours need a deposit. Below `deposit = 0`, write an `if` that sets `deposit = 20` when `hours > 3`. Then `print("Deposit:", deposit)`.

   Starter code:

   ```
   # Bluefin Bikes: the new price list
   # if / elif / else: Python checks the branches from top to bottom
   # and runs ONLY the first one whose condition is True.

   shop_name = "Bluefin Bikes"
   hours = 4

   # Task 1: rentals of more than 3 hours need a deposit of $20
   deposit = 0
   ```

2. **if and else.** Define `deposit_for(hours)` that returns 20 for more than 3 hours and 0 otherwise, using `if` and `else`. Print `deposit_for(2), deposit_for(5)`.
3. **Three tiers with elif.** Define `price(hours)` with `if`, `elif` and `else` following the three tiers in the comment. Print `price(1), price(4), price(7)`: `6 20 28`.
4. **Prove the order matters.** Theo's `price_theo` has the same branches in a different order. Define `differs(h)` that returns True when the two prices differ, then loop over `test_hours` and print each hour where they disagree, like `1 h: Mia 6, Theo 8`.
5. **Narrowest condition first.** Define `discount(age)`: under 12 gets 50, under 18 gets 30, 65 or older gets 20, everyone else 0. Print `discount(10), discount(16), discount(70)`: `50 30 20`.
6. **One answer, not two.** Define `late_fee(minutes, member)`: up to 15 minutes free, more than 60 minutes 25 for everyone, otherwise members pay 5 and everyone else 10. Print the four example calls: `0 5 10 25`.
7. **Compare text in branches.** Define `zone(code)`: ZRH is `"city"`, BSL or BRN is `"regional"`, every other code `"partner"`. Codes may arrive in small letters. Print `zone("zrh"), zone("BRN"), zone("GVA")`.
8. **Today's price list.** Define `rental_total(row)`: split the row, take `price(hours)`, subtract the discount in percent and add $3 for partner stations. Print one line per rental (`R-301 city $6.00`) and finish with the day's total (`Total: $...`).

Lists: Index, Slice, append(), pop() and insert()

Lesson 8 · about 17 min

> **Goal.** You follow Bluefin's e-bike battery list through indexes, slices, append, pop and insert, keep a real copy instead of a second name, build new lists in a loop and simulate a full day at the station.

Today at Bluefin Bikes

The ZRH station has a rack of e-bikes, and every bike reports its battery level. Bikes leave, come back, go to the workshop and return. One value per bike would mean eight variables today and twelve tomorrow. A **list** holds all of them in one name, in order, and can grow and shrink during the day.

Index and slice, just like strings

```
levels = [82, 45, 97, 13]
print(levels[0])     # 82   the first
print(levels[-1])    # 13   the last
print(levels[:2])    # [82, 45]   a slice is a new list
print(len(levels))   # 4
```

Lists can change

Strings are immutable; lists are **mutable**. Their methods change the list in place and mostly return `None`, so you call them on their own line:

- `levels.append(55)` adds 55 at the end.
- `levels.insert(0, 13)` puts 13 at index 0 and shifts all others one place right.
- `x = levels.pop(3)` removes the element at index 3 and returns it; `levels.pop()` takes the last one.
- `levels[0] = 100` replaces one element.
- `levels.index(97)` tells you where a value is; `max(levels)`, `min(levels)` and `sum(levels)` work on any list of numbers.

After every change the indexes move. After `pop(3)`, the element that was at index 4 is now at index 3. When you trace list code, write the list down after **each** line.

Two names, one list

`backup = levels` does not copy anything. It sticks a second label on the **same** box, so a change through one name shows up under the other. The same happens when you pass a list to a function: the function gets the original. If you need an independent list, make one with `levels.copy()` or `levels[:]`.

> **Tip.** Not sure what a line did? Add `print(levels)` right after it. Tracing the list step by step beats guessing every time.

All companies, names and figures are fictional. Independent training. Not affiliated with or endorsed by the Python Software Foundation.

Your coding tasks

These tasks are written for a Python file called `main.py`. Create the file in your editor (or the online editor of your choice), type the starter code, and complete the tasks one by one. When you are done, compare your work with the solution in **Appendix: Solutions**.

1. **Count and pick.** Print the number of bikes, the first battery level and the last one in one line: `8 82 90`.
   Starter code:

   ```
   # Bluefin Bikes: e-bike batteries at the ZRH station
   # A list keeps values in order. Index 0 is the first, -1 the last.
   #
   #   index:    0   1   2   3   4   5   6   7
   levels = [82, 45, 97, 13, 60, 71, 28, 90]

   # Task 1: how many bikes, and what do the first and the last show?
   ```

2. **Slices give lists.** Store the first three levels in `first_three` and the last two in `last_two`, using slices. Print both.
3. **append() adds at the end.** A bike came back with 55 %. Add it to the end of `levels` and `print(levels)`.
4. **pop() removes and returns.** Take the bike at index 3 out of the list and keep its value in `low`. Then `print(low, levels)`.
5. **insert() at a position.** Put the repaired bike back at the very front. Then `print(levels[:3], len(levels))`: `[13, 82, 45] 9`.
6. **Copy, not a second name.** Save a copy of `levels` in `backup`, then set `levels[0] = 100`. `print(backup[0], levels[0])` must show `13 100`.
7. **Build a new list in a loop.** Define `low_bikes(levels, limit)` that returns a NEW list with all levels below `limit`, in their order. The list passed in must not change. Print `low_bikes(levels, 50)`.
8. **Simulate the day.** Define `run_day(start, events)` that works on a copy of `start`: `"rent"` removes the bike with the highest level (if there is any bike), `"return N"` appends N. Return the list, then print `today` and the summary line.

for Loops and range()

Lesson 9 · about 13 min

> **Goal.** You trace a for loop week by week in a trace table and work out how many numbers range(start, stop, step) produces and which one comes last.

A **for loop** repeats its indented block once for every element of a sequence: `for week in range(1, 7):` runs the block with week = 1, 2, 3, 4, 5, 6. The variable keeps its last value after the loop.

`range(stop)` counts from 0 up to stop - 1, `range(start, stop)` from start up to stop - 1, `range(start, stop, step)` in steps. **stop is never included**: `range(1, 7)` has six numbers, `range(3, 40, 4)` gives 3, 7, 11, …, 39.

The pattern `total = 0` before the loop and `total = total + …` inside it is an **accumulator**. To trace it, make a table with one row per iteration and write the value of total *after* the body ran. That is exactly what the trace table in A17:B23 is for.

Counting a range without listing it: the number of values is `ceil((stop - start) / step)` (0 if stop <= start), and the last value is `start + (count - 1) * step`.

	A	B
1	**Code · training_plan.py**	
2	*base_km = 15*	
3	`total = 0`	
4	`for week in range(1, 7):`	
5	`    total = total + week * base_km`	
6	`print(total)`	
7		
8	*start, stop, step = 3, 40, 4*	
9	`service_km = list(range(start, stop, step))`	
10	`print(len(service_km), service_km[-1])`	
11		
12	`base_km`	15
13	`start`	3
14	`stop`	40
15	`step`	4
16		
17	**week**	**total after the loop body**
18	1	
19	2	
20	3	
21	4	
22	5	
23	6	
24		
25	`len(service_km)`	
26	`service_km[-1]`	

Shaded cells are your work area – fill them in, then compare with the Appendix: Solutions at the end of the book.

Your tasks

Try it on your computer: open a blank spreadsheet, type the numbers from the worksheet above into the same cells (A1, B1 …), then work on the shaded cells. When you are done, compare with the solutions in **Appendix: Solutions** at the end of this book and correct your sheet.

1. Fill the trace table B18:B23 with the value of `total` after each week.
2. In B25, compute `len(service_km)` from start, stop and step.
3. In B26, compute `service_km[-1]`, the last number of the range.

while Loops: Repeat Until a Condition Fails

Lesson 10 · about 12 min

> **Goal.** You work out how many trips an e-bike battery allows before it hits the reserve, and how a one-character change in the condition breaks the rule.

A **while loop** repeats its block **as long as** its condition is True. The condition is checked *before* every round, so the body may run zero times. Something in the body must change the condition, or the loop runs forever (stop a runaway script with Ctrl+C).

Use `while` when you do not know the number of rounds in advance: *until the battery reaches the reserve*, *until the user types quit*. Use `for` when you loop over a known sequence.

Trace a while loop by asking, before each round: *is the condition still True?* With battery 96, reserve 20 and drain 7, Mia's condition `battery - drain >= reserve` asks *will the next trip still leave the reserve?* Theo's `battery > reserve` only asks *are we above the reserve now?*, so his last trip dips below it. Off-by-one errors in loop conditions are among the most common bugs.

Two keywords control loops: `break` leaves the loop at once, `continue` skips the rest of this round and jumps back to the condition. You will use both in the next lesson.

	A	B
1	**Code · battery_trips.py (Mia)**	
2	*battery = 96*	
3	*reserve = 20*	
4	*drain = 7 # % per trip*	
5	`trips = 0`	
6	`while battery - drain >= reserve:`	
7	`    battery = battery - drain`	
8	`    trips += 1`	
9	`print(trips, battery)`	
10		
11	**Code · Theo's version (only the loop line differs)**	
12	`while battery > reserve:`	
13	`    battery = battery - drain`	
14	`    trips += 1`	
15		
16	`battery (start, %)`	96
17	`reserve (%)`	20
18	`drain per trip (%)`	7
19		
20	`Mia: trips`	
21	`Mia: battery at the end`	
22	`Theo: trips`	
23	`Theo: battery at the end`	
24		
25	`while repeats as long as its condition is True. It is checked BEFORE every round.`	

Shaded cells are your work area – fill them in, then compare with the Appendix: Solutions at the end of the book.

Your tasks

Try it on your computer: open a blank spreadsheet, type the numbers from the worksheet above into the same cells (A1, B1 …), then work on the shaded cells. When you are done, compare with the solutions in **Appendix: Solutions** at the end of this book and correct your sheet.

1. In B20, compute the number of `trips` Mia's loop makes, with a formula on B16:B18.
2. In B21, compute the `battery` value that line 9 prints for Mia.
3. In B22, compute the number of trips with Theo's condition `battery > reserve`.
4. In B23, compute Theo's final battery level.

Loop Patterns: Counters, Accumulators, break and continue

Lesson 11 · about 13 min

> **Goal.** You trace a loop that skips closed days with continue, counts busy days and stops with break as soon as Bluefin reaches its rental target.

Most loops follow a few **patterns**: an *accumulator* (`total += n`), a *counter* (`busy_days += 1` inside an `if`), a *search* that stops early. Initialise the variables **before** the loop, update them inside, use them after.

`continue` ends the current round at once and jumps to the next element; the lines below it are skipped for this round only. Here closed days (0 rentals) are skipped. `break` leaves the loop completely: no more elements are processed, and the code continues after the loop.

The order of the lines inside the body matters. In this script a day is first added and counted, and **then** the target check runs. So the day that reaches the target is still counted, and every later day is ignored.

To trace it, build a **running total** column (C17:C26). The loop stops on the first day whose running total is at least the target. Busy days only count up to and including that day.

	A	B	C
1	**Code · stop_when_target.py**		
2	*rentals = [18, 0, 31, 22, 40, 0, 27, 35, 12, 29]*		
3	*busy, target = 25, 150*		
4	*total = 0*		
5	`busy_days = 0`		
6	*for n in rentals:*		
7	`    if n == 0:`		
8	`        continue          # shop closed`		
9	`    total += n`		
10	`    if n >= busy:`		
11	`        busy_days += 1`		
12	`    if total >= target:`		
13	`        break`		
14	`print(total, busy_days)`		
15			
16	**day (index)**	**rentals**	**running total**
17	0	18	
18	1	0	
19	2	31	
20	3	22	
21	4	40	
22	5	0	
23	6	27	
24	7	35	
25	8	12	
26	9	29	

	A	B	C
27			
28	busy (rentals)	25	
29	target	150	
30			
31	printed total		
32	printed busy_days		

Shaded cells are your work area – fill them in, then compare with the Appendix: Solutions at the end of the book.

Your tasks

Try it on your computer: open a blank spreadsheet, type the numbers from the worksheet above into the same cells (A1, B1 …), then work on the shaded cells. When you are done, compare with the solutions in **Appendix: Solutions** at the end of this book and correct your sheet.

1. Fill the running total in C17:C26 (sum of rentals up to that day).
2. In B31, give the `total` that line 14 prints.
3. In B32, give the printed `busy_days`: days with at least `busy` rentals, up to and including the break day.

Dictionaries: Counting and Summing by Key

Lesson 12 · about 13 min

> **Goal.** You build Bluefin's per-station statistics with a dictionary, the counting pattern with get(), and learn which lookup crashes.

A **dictionary** maps keys to values: `prices = {"city": 4.5, "ebike": 9.0}`. You read a value with `prices["city"]`, add or change one with `prices["kids"] = 3.0`, and check a key with `"kids" in prices`. Keys are unique; assigning to an existing key overwrites its value.

`d[key]` raises a **KeyError** when the key is missing. `d.get(key, default)` returns the default instead (and `d.get(key)` returns `None`). That makes the classic **counting pattern** possible: `counts[s] = counts.get(s, 0) + 1`, the first time it starts from 0.

Summing per key works the same way: `minutes_by[s] = minutes_by.get(s, 0) + minutes`. A loop over pairs can **unpack** them directly: `for station, minutes in log:`.

In the sheet, a dictionary built like this is a COUNTIFS / SUMIFS per key. The `continue` for test rides becomes an extra criterion on the minutes.

	A	B	C	D	E
1	**Code · station_stats.py**				
2	*min_minutes = 10*				
3	`counts = {}`				
4	`minutes_by = {}`				
5	`for station, minutes in log:`				
6	`    if minutes < min_minutes:`				
7	`        continue          # test ride, ignore`				
8	`    counts[station] = counts.get(station, 0) + 1`				
9	`    minutes_by[station] = minutes_by.get(station, 0) + minutes`				
10	`avg_harbour = minutes_by["Harbour"] / counts["Harbour"]`				
11					
12					
13	**log: station**	**minutes**		**Expression**	**Value**
14	`Harbour`	34		`counts["Harbour"]`	
15	`Old Town`	12		`minutes_by["Old Town"]`	
16	`Harbour`	6		avg_harbour	
17	`Station Sq`	45			
18	`Old Town`	27		**Line**	**Code (after the loop)**
19	`Harbour`	18		1	`counts.get("Airport", 0)`
20	`Old Town`	8		2	`counts["Airport"]`
21	`Harbour`	52		3	*"Airport" in counts*
22	`Station Sq`	15		4	`counts.get("Airport")`
23	`Old Town`	40		5	*len(counts)*
24	`Harbour`	9			
25	`min_minutes`	10		Which line raises a KeyError?	

Shaded cells are your work area – fill them in, then compare with the Appendix: Solutions at the end of the book.

Your tasks

Try it on your computer: open a blank spreadsheet, type the numbers from the worksheet above into the same cells (A1, B1 …), then work on the shaded cells. When you are done, compare with the solutions in **Appendix: Solutions** at the end of this book and correct your sheet.

1. In E14, compute `counts["Harbour"]` after the loop.
2. In E15, compute `minutes_by["Old Town"]`.
3. In E16, compute `avg_harbour` (line 10).
4. Nobody rented at the Airport, so it is not a key in `counts`. Which of the five lines in D19:E23 raises a KeyError? Type its number in E25.

Sets, Tuples and Membership

Lesson 13 · about 12 min

> **Goal.** You count unique and returning customers with sets and trace a tuple swap that builds a number sequence.

A **set** is an unordered collection **without duplicates**: `set([104, 117, 104])` is `{104, 117}`. That makes `len(set(items))` the fastest way to count distinct values. Membership tests (`104 in s`) are very fast on sets.

Sets have operators from maths: `a | b` (union: in a or b), `a & b` (intersection: in both), `a - b` (in a but not in b). A useful check: `len(a & b) == len(a) + len(b) - len(a | b)`.

A **tuple** is like a list that cannot change: `point = (47.56, 7.59)`. Python packs and unpacks tuples automatically: `a, b = b, a + b` first builds the tuple `(b, a + b)` from the **old** values, then assigns both at once. No temporary variable is needed.

Spreadsheet helpers for checking: UNIQUE() removes duplicates, TOCOL() stacks two columns into one, COUNTA() counts the result.

	A	B	C	D	E	F	G
1	**Code · regulars.py**						
2	*mon = [104, 117, 104, 122, 131, 117, 140, 108]*						
3	*tue = [117, 150, 131, 104, 162, 150, 175, 104]*						
4	`unique_mon = len(set(mon))`						
5	`all_customers = len(set(mon) | set(tue))`						
6	`both_days = len(set(mon) & set(tue))`						
7							
8	*a, b = 2, 3*						
9	`for _ in range(3):`						
10	`    a, b = b, a + b`						
11	`print(a, b)`						
12							
13	**rental no.**	**mon (customer id)**	**tue (customer id)**		**Expression**	**Value**	
14	1	104	117		unique_mon		
15	2	117	150		all_customers		
16	3	104	131		both_days		
17	4	122	104		a (printed)		
18	5	131	162				
19	6	117	150		**round**	**a**	**b**
20	7	140	175		1		
21	8	108	104		2		
22					3		
23							
24	`a (start)`	2					
25	`b (start)`	3					

Shaded cells are your work area – fill them in, then compare with the Appendix: Solutions at the end of the book.

Your tasks

Try it on your computer: open a blank spreadsheet, type the numbers from the worksheet above into the same cells (A1, B1 …), then work on the shaded cells. When you are done, compare with the solutions in **Appendix: Solutions** at the end of this book and correct your sheet.

1. In F14, compute `unique_mon`.
2. In F15, compute `all_customers` (union of both days).
3. In F16, compute `both_days` (intersection). Tip: use the identity from the theory.
4. Trace the tuple swap in F20:G22 (a and b after each round), then put the printed `a` in F17.