FOUNDATIONS / PROJECT BRIEF

Workshop Badge Printer

THE ASSIGNMENT

A community workshop needs badges that look consistent even when the signup text contains extra spaces. Build a terminal program that turns a name, a role, and a seat number into a readable badge. A variable gives a value a name so the rest of the program can refer to its purpose. String methods return a transformed string. Save the result when you need to reuse it. An f-string inserts values between braces, and a format specification such as 03d gives an integer a minimum width of three digits with leading zeroes. Start with sample values so your first run works without typing anything.

ACCEPTANCE TARGET

The sample prints Name: Ada Rivera without leading or trailing spaces.

START HERE

Install Python 3.12 with its standard library. Extract the study bundle to a folder you can edit. In a terminal, open that folder and run python --version (or py -3.12 --version on Windows). Use a plain-text code editor and UTF-8.

```
python projects/day_001.py
# Windows alternative: py -3.12 projects/day_001.py
```

Each day spans five pages: frame the task, build it, inspect the two solution sections, and test an extension. The PDF outline links to every day. The code bundle includes a searchable project index.

JUMP TO A TRACK

Day 001 Start writing Python
Day 026 Automate real tasks
Day 051 Work with data
Day 076 Build interactive projects

FOUNDATIONS / GUIDED BUILD

Workshop Badge Printer

Create a working copy of day_001.py. Use the project contract and implement each step before comparing the reference.

STEP 01

Create badge.py and put three raw signup values near the top. Keep the seat as an integer and the other values as quoted text.

STEP 02

Trim the name with strip. Split the role into words and join those words with one space, then uppercase the result.

STEP 03

Format the seat separately. The 03d specification pads short numbers and preserves all digits in a longer number.

STEP 04

Join the badge lines with a newline so one print call produces the entire badge.

STEP 05

Add checks for the cleaned name, role, and formatted number. Run python badge.py from the folder where you saved it.

FIRST EXPERIMENT

Run the smallest valid example. Record the exact result you observed.

FOUNDATIONS / REFERENCE SOLUTION

Workshop Badge Printer

day_001.py | Part 1 of 2

```python
if __name__ == "__main__":
    raw_name = "  Ada Rivera "
    raw_role = "  tool   mentor "
    seat_number = 7

    name = raw_name.strip()
    words = raw_role.split()
    role = " ".join(words).upper()
    seat = f"{seat_number:03d}"

```

READ THE DESIGN

Follow the assignments in order. Write down a value after each transformation and compare it to the acceptance target.

FOUNDATIONS / RUN AND INSPECT

Workshop Badge Printer

day_001.py | Part 2 of 2

```
    heading = "COMMUNITY WORKSHOP"
    name_line = f"Name: {name}"
    role_line = f"Role: {role}"
    seat_line = f"Seat: {seat}"
    lines = [heading, "=" * len(heading), name_line, role_line, seat_line]
    badge = "\n".join(lines)

    assert name == "Ada Rivera"
    assert role == "TOOL MENTOR"
    assert seat == "007"
    print(badge)
```

RUN FROM THE EXTRACTED BUNDLE

```
python projects/day_001.py
```

OBSERVED CONSOLE OUTPUT

```
COMMUNITY WORKSHOP
==================
Name: Ada Rivera
Role: TOOL MENTOR
Seat: 007
```

FOUNDATIONS / CHECK AND EXTEND

Workshop Badge Printer

VERIFY THE BEHAVIOR

01 The sample prints Name: Ada Rivera without leading or trailing spaces.

02 The sample prints Role: TOOL MENTOR with exactly one internal space.

03 Seat 7 prints Seat: 007. Change it to 1234 and update the assertion to check Seat: 1234.

DEBUG A LIKELY FAILURE

A ValueError about format code d usually means the seat is quoted. Inspect type(seat_number) and store the seat as an integer.

TAKE IT FURTHER

Add a table number on a sixth line. Verify that the table and seat are displayed separately.

LAB NOTES

What did the checks prove? What remains untested?

FOUNDATIONS / PROJECT BRIEF

Recipe Batch Card

THE ASSIGNMENT

A volunteer kitchen scales a four-serving recipe to serve seven people. The quantities should all change by the same factor. Derive that factor once from the desired and original serving counts, then multiply each starting quantity by it. In Python, division with / gives a floating point result. Keep the values numeric through the calculation and format them to one decimal place when printing the card. This preserves more information for later calculations. A quick check with the original number of servings should recover the original quantities. Label each measure clearly so a cook can use the card without guessing the units.

ACCEPTANCE TARGET

At seven servings, flour is 420.0 g and milk is 350.0 mL.

PLAN THE BOUNDARY

Write down the base servings and the three base quantities. Store the desired servings in its own variable.

BEFORE YOU CODE

Describe the input and result you will use for your first run.

Predict one case the program should reject or handle separately.

FOUNDATIONS / GUIDED BUILD

Recipe Batch Card

Create a working copy of day_002.py. Use the project contract and implement each step before comparing the reference.

STEP 01

Write down the base servings and the three base quantities. Store the desired servings in its own variable.

STEP 02

Calculate desired divided by base. For seven over four, the factor should be 1.75.

STEP 03

Multiply each quantity by the common factor. Use names that keep grams, milliliters, and teaspoons distinguishable.

STEP 04

Print a card that states both serving counts. Apply one decimal place to each displayed quantity.

STEP 05

Try the base serving count before trying a fractional scale. The first check exposes a reversed ratio quickly.

FIRST EXPERIMENT

Run the smallest valid example. Record the exact result you observed.

FOUNDATIONS / REFERENCE SOLUTION

Recipe Batch Card

day_002.py | Part 1 of 2

```
if __name__ == "__main__":
    base_servings = 4
    target_servings = 7

    base_flour_g = 240
    base_milk_ml = 200
    base_spice_tsp = 2

    scale = target_servings / base_servings
    flour_g = base_flour_g * scale
    milk_ml = base_milk_ml * scale
    spice_tsp = base_spice_tsp * scale

```

READ THE DESIGN

Follow the assignments in order. Write down a value after each transformation and compare it to the acceptance target.

FOUNDATIONS / RUN AND INSPECT

Recipe Batch Card

day_002.py | Part 2 of 2

```
    print("RECIPE BATCH CARD")
    print(f"Scale from {base_servings} to {target_servings} servings")
    print(f"Flour: {flour_g:.1f} g")
    print(f"Milk:  {milk_ml:.1f} mL")
    print(f"Spice: {spice_tsp:.1f} tsp")

    assert scale == 1.75
    assert flour_g == 420
    assert milk_ml == 350
    assert spice_tsp == 3.5
```

RUN FROM THE EXTRACTED BUNDLE

```
python projects/day_002.py
```

OBSERVED CONSOLE OUTPUT

```
RECIPE BATCH CARD
Scale from 4 to 7 servings
Flour: 420.0 g
Milk:  350.0 mL
Spice: 3.5 tsp
```

FOUNDATIONS / CHECK AND EXTEND

Recipe Batch Card

VERIFY THE BEHAVIOR

01 At seven servings, flour is 420.0 g and milk is 350.0 mL.

02 At seven servings, spice prints as 3.5 tsp.

03 Set the target to four and update the assertions to expect scale 1, 240.0 g, 200.0 mL, and 2.0 tsp.

DEBUG A LIKELY FAILURE

If amounts decrease when you ask for more servings, print scale by itself. The target belongs in the numerator.

TAKE IT FURTHER

Show each original quantity beside its scaled quantity. Label both columns and retain each unit.

LAB NOTES

What did the checks prove? What remains untested?

FOUNDATIONS / PROJECT BRIEF

Fair Cents Splitter

THE ASSIGNMENT

A club divides a shared receipt among members and needs to collect the exact amount. Store the receipt as an integer number of cents. The quotient gives every member the same base share, and the remainder tells you how many members should pay one additional cent. Python's divmod returns both numbers together. This arithmetic guarantees that each share differs from the next by at most one cent and that no cent disappears. The example gives the extra cent to the earliest listed members. Split the displayed share into dollars and cents after you have computed each member's amount.

ACCEPTANCE TARGET

1001 cents among three prints USD 3.34, USD 3.34, and USD 3.33.

PLAN THE BOUNDARY

Set the receipt to 1001 cents and the group size to three. Restrict group size to positive numbers.

BEFORE YOU CODE

Describe the input and result you will use for your first run.

Predict one case the program should reject or handle separately.

FOUNDATIONS / GUIDED BUILD

Fair Cents Splitter

Create a working copy of day_003.py. Use the project contract and implement each step before comparing the reference.

STEP 01

Set the receipt to 1001 cents and the group size to three. Restrict group size to positive numbers.

STEP 02

Use divmod to obtain the base share and extra count. Recombine them to check the original receipt total.

STEP 03

Visit each person with range. The first extra_count people receive the higher share, using a zero-based position in the comparison.

STEP 04

Convert each share to dollars and cents with another divmod call. Pad the cents field to two digits.

STEP 05

Accumulate paid_cents as each row prints. Check that the amount collected equals the receipt exactly.

FIRST EXPERIMENT

Run the smallest valid example. Record the exact result you observed.

FOUNDATIONS / REFERENCE SOLUTION

Fair Cents Splitter

day_003.py | Part 1 of 2

```
if __name__ == "__main__":
    receipt_cents = 1001
    people = 3
    if receipt_cents < 0 or people < 1:
        raise ValueError("Use a nonnegative receipt and a positive group")

    base_share, extra_count = divmod(receipt_cents, people)
    assert base_share * people + extra_count == receipt_cents

```

READ THE DESIGN

Follow the assignments in order. Write down a value after each transformation and compare it to the acceptance target.

FOUNDATIONS / RUN AND INSPECT

Fair Cents Splitter

day_003.py | Part 2 of 2

```
    paid_cents = 0
    print(f"Split {receipt_cents} cents among {people} people")
    for position in range(people):
        share = base_share
        if position < extra_count:
            share += 1

        dollars, cents = divmod(share, 100)
        print(f"Person {position + 1}: USD {dollars}.{cents:02d}")
        paid_cents += share

    assert paid_cents == receipt_cents
    print(f"Collected: {paid_cents} cents")
```

RUN FROM THE EXTRACTED BUNDLE

```
python projects/day_003.py
```

OBSERVED CONSOLE OUTPUT

```
Split 1001 cents among 3 people
Person 1: USD 3.34
Person 2: USD 3.34
Person 3: USD 3.33
Collected: 1001 cents
```

FOUNDATIONS / CHECK AND EXTEND

Fair Cents Splitter

VERIFY THE BEHAVIOR

01 1001 cents among three prints USD 3.34, USD 3.34, and USD 3.33.

02 Change the receipt to 900 cents. All three shares print as USD 3.00.

03 Change the group to zero. A ValueError explains the positive group requirement.

DEBUG A LIKELY FAILURE

If the printed total exceeds the receipt, check the comparison with extra_count. Using <= assigns one additional cent too many.

TAKE IT FURTHER

Rotate the extra cents. With three members and a start at position two, give the first extra cent to member three and the next to member one.

LAB NOTES

What did the checks prove? What remains untested?