

Designing and Verifying Classes

Keep the six tasks: rectangles, stacks, word counts, temperatures, a gradebook refactor, and design decisions. Specify the interface first, then verify independent state, failed updates, results, and the A/B/C learning records.

Due: Homework 3 · Tue 17 Nov, 23:59

Goals and submission entries

Apply [L09](#) to initialization, instance methods, shared references, and interface checks. The six original tasks are organized as A: four basic classes, B: a gradebook refactor, and C: design choices and debugging records.

Use the five code entries and three text cards on the [HW3 submission page](#). Save and test locally, then submit each item to its own entry. Saving a local file is not an online submission. HW3 covers weeks 7–9; its deadline comes from the shared course calendar rather than a separate “this Sunday” deadline.

Part	Code	Written record	Raw points
A	Rectangle, Stack, WordCounter, Temperature: 4 each	A design/verification card: 8 staff-graded	24
B	Gradebook: 8 automatic	B refactor/regression card: 4 staff-graded	12
C	No code entry	Choices, debugging, and operations: 0	0

A/B total 36 raw points. Passing code tests does not award staff points automatically. C is required learning work without extra points. These raw points feed into HW3 without changing the course assessment weights.

Week 9 uses Level 1

AI assistance is allowed after you establish interfaces, domains, failures, and verification. Put a brief usage declaration in every code submission's notes, and retain the complete account on the A/B text cards. If you did not use AI, say so. HW3's overall badge follows its earliest week. Apply each week's own lab policy: week 7 and solo assignment 1 remain independent implementation tasks.

Every class must work as an independent program interface. The evaluator does not run a slide first or supply global state from another problem. Do not read tests with `input()` or add an interactive main program.

Annotate public method signatures. Use the conventional receiver name `self`; no separate annotation for `self` is required. Signatures document expectations, not runtime enforcement. Evaluation primarily checks the stated behavior; written records explain decisions and evidence.

Part A · Basic classes and independent state

9-1 Rectangle

Store dimensions in public attributes `width` and `height`, with this interface:

```
class Rectangle:
    def __init__(self, width: float, height: float) -> None: ...
    def area(self) -> float: ...
    def perimeter(self) -> float: ...
    def is_square(self) -> bool: ...
    def scale(self, factor: float) -> None: ...
```

Contract:

- Dimensions and scale factors must be finite and positive. Zero, negative values, NaN, and either infinity are invalid; construction or scaling raises `ValueError`.

- Area, perimeter, and square queries preserve dimensions. `is_square()` compares the two stored dimensions; approximate geometric equality is not required here.
- `scale(factor)` updates **both dimensions in place** and returns **None**, not a new rectangle.
- The resulting dimensions must also remain finite and positive. If floating-point scaling produces infinity or zero, raise `ValueError` and retain both old dimensions.
- Ordinary tests use representable areas and perimeters. Real parameters arrive as integers or floats, not other object types.

```
r = Rectangle(3, 4)
r.area()           → 12
r.perimeter()      → 14
r.is_square()      → False
r.scale(2)         → None
r.width, r.height  → 6, 8
r.area()           → 48
```

Test separate instances, squares, invalid starting dimensions, successful scaling, and full state preservation after failure. Compare floating results with an appropriate tolerance.

The A card explains this in-place contract and contrasts an alternative returning a new object. The explanation cannot choose that alternative while the code violates this entry's required behavior.

Code entry: `w09-rectangle` .

9-2 Stack

Implement last-in, first-out storage for each instance:

```

class Stack:
    def __init__(self) -> None: ...
    def push(self, item: int) -> None: ...
    def pop(self) -> int: ...
    def peek(self) -> int: ...
    def is_empty(self) -> bool: ...
    def size(self) -> int: ...

```

- Push appends an integer to the top and returns None.
- Pop removes and returns the top; peek returns it without removal.
- Empty pop and peek **both raise IndexError** and leave the stack empty. Do not return None or rely on the evaluator to check emptiness first.
- A new stack is empty with size zero. Operations on two instances must not affect each other.

```

s.push(7); s.push(9)
s.peek()      → 9
s.size()      → 2
s.pop()       → 9
s.pop()       → 7
s.is_empty()  → True
s.pop()       → IndexError

```

On the A card, compare returning None, raising an exception, and requiring a caller precondition. Those are real design alternatives; this code entry uses one fixed exception contract.

Code entry: `w09-stack`.

9-3 WordCounter

```
class WordCounter:
    def __init__(self) -> None: ...
    def add(self, word: str) -> None: ...
    def get(self, word: str) -> int: ...
    def total(self) -> int: ...
    def most_common(self, n: int) -> list[tuple[str, int]]: ...
```

- Treat the supplied string as one exact key. Do not tokenize, strip punctuation, or change case. The empty string is also a valid key.
- Unseen keys have count zero. Total is the sum of occurrences, not the number of distinct keys.
- Return at most `n` `(word, count)` pairs, count descending and ties ascending in Python string order. For `n=0` return an empty list; a larger `n` returns all available pairs.
- `n` is a nonnegative integer. Negative `n` raises `ValueError`; noninteger `n` is outside these tests.
- Queries and sorting preserve counts. Every instance needs its own dictionary.
- A named sort-key function is sufficient; `lambda` is not needed before its later dedicated treatment.

Add "pear", "apple", "pear", "banana" in that order.

```
get("pear")      -> 2
get("missing")   -> 0
total()          -> 4
most_common(3)   -> [("pear", 2), ("apple", 1), ("banana", 1)]
```

The A card records an actual two-instance experiment: operations, expected observations, actual observations, and a conclusion. "Tests passed" is not a substitute for the observations.

Code entry: `w09-word-counter` .

9-4 Temperature

Store Celsius in the nonpublic field `self._celsius`:

```
class Temperature:
    def __init__(self, celsius: float) -> None: ...
    def celsius(self) -> float: ...
    def fahrenheit(self) -> float: ...
    def kelvin(self) -> float: ...
    def set_celsius(self, value: float) -> bool: ...
```

- Values must be finite real numbers at least -273.15°C . Inputs arrive as integers or floats, not other object types.
- Invalid initialization raises `ValueError`.
- An invalid update returns **False** without replacing the old value. An accepted update stores the new value and returns **True**.
- Queries preserve state. Fahrenheit is `c * 9 / 5 + 32`; Kelvin is `c + 273.15`.
- Do not store an invalid candidate first and repair it later.

```
t = Temperature(25)
t.celsius()          → 25
t.fahrenheit()       → 77
t.kelvin()           → 298.15
t.set_celsius(-300) → False
t.celsius()          → 25
t.set_celsius(100)  → True
```

Exercise the exact lower boundary, values below it, NaN/infinity, all three queries after rejection, and separate instances. Explain why construction and updates need validation and why an underscore does not enforce it.

Code entry: `w09-temperature`.

A design and verification card

Text entry: `w09-a-design-notes`. Four responses carry 2 staff points each:

1. **Mutation or new result:** explain Rectangle's contract and the return values and alias effects of the alternative.
2. **Empty-stack policy:** compare None, an exception, and a caller precondition; justify consistency with this IndexError interface.
3. **Instance isolation:** give the operations and expected/actual results for two WordCounters.
4. **Valid state:** provide invalid-constructor and failed-setter evidence for Temperature, including state before/after and the meaning of the underscore.

A separate zero-point AI declaration records tool/version, requested task, adopted or changed content, and verification evidence. State nonuse explicitly. A generated answer or an unsupported personal conclusion is not test evidence.

Part B · Refactoring and regression

9-5 Gradebook

Retain the original dictionary task: store students' subject scores, query an average, and identify the highest average. Inspect this candidate before writing a class:

```
def add_score(book, name, subject, score):
    if name not in book:
        book[name] = {}
    book[name][subject] = score

def average(book, name):
    scores = book[name].values()
    return sum(scores) / len(scores)

def top_student(book):
    names = list(book)
    best = names[0]
    for name in names[1:]:
        if average(book, name) > average(book, best):
            best = name
    return best
```

This is a review excerpt, not a complete runnable entry. The refactor needs neither lambda nor a generator expression.

```
class Gradebook:
    def __init__(self) -> None: ...
    def add(self, name: str, subject: str, score: float) -> None: ...
    def average(self, name: str) -> float: ...
    def top_student(self) -> str | None: ...
```

Contract:

- Student and subject names are nonempty strings, retained exactly without trimming or case changes. Other name types are not tested.
- Scores are finite real numbers in 0–100 inclusive. An invalid score or empty name/subject raises `ValueError` and leaves the **entire book unchanged**, without a newly created student or empty subject.
- Adding the same student/subject replaces the old score. Different subjects accumulate. `Add` returns `None`.
- Average is the arithmetic mean of the current subjects. A missing student raises `KeyError` without creating a record.
- An empty book has `top_student() == None`. Otherwise return the highest-average name; ties choose the smallest name in Python string order, not the first insertion.
- Books are independent; queries preserve their data.

```
add("Alex", "math", 87)
add("Alex", "physics", 92)
add("Blake", "math", 65)
average("Alex")      -> 89.5
top_student()        -> "Alex"
empty top_student() -> None
average("missing")   -> KeyError
```

The candidate lacks a declared empty-book behavior and an explained missing-student failure; tied winners also depend on insertion order. The new contract intentionally allows

KeyError. Do not describe the mere persistence of an undocumented error as a completed fix.

Regression includes ordinary examples, empty and missing data, ties, replacement, and state after rejection.

Code entry: `w09-gradebook`.

B refactor and regression card

Text entry: `w09-b-refactor-notes`.

- **Contract and corrections, 2 staff points:** compare original behavior with the chosen contract and explain changes, including missing students and an empty book.
- **Regression evidence, 2 staff points:** record inputs, expected/actual outcomes, and conclusions for normal data, ties, replacement, and failed additions.
- **AI declaration, 0 points:** make it consistent with the code submission; state nonuse if applicable.

Do not put these answers in another task's code file or paste complete implementations into the text card.

Part C · Design decisions and debugging records

9-6 Class, function, or record?

Keep the four original requirements. Choose and justify an organization for each rather than choosing classes because of the week's title:

1. Convert Celsius to Fahrenheit.
2. Maintain a deck that can be shuffled, dealt from, and counted.
3. Test whether a string is a palindrome.
4. Retain conversation history and use it to organize later replies.

Discuss retained state, cohesive operations, lifetime, and interface boundaries. No model service or chat-data upload is required. A single instance, one method, or read-only data is not an automatic veto against classes.

Observe two objects in a debugger

Use your local Point or another already-taught class:

1. Construct two instances and set a breakpoint in an instance method.
2. Call it from each instance and inspect self and the relevant fields.
3. Update one instance and observe whether the other remains unchanged.
4. If an alias exists, explain how it differs from new construction.

Record the breakpoint location, call sequence, receiver selection, and actual values. If the debugger cannot start, record why and use print for the same before/after evidence. Do not invent a screenshot. This reuses L08's debugging actions without requiring new tool installation.

Text entry: `w09-c-design-tools`. Its three fields cover design decisions, debugging, and declarations/operations. All are zero-point learning records, not executable code. Zero points do not mean the task is optional.

Before submitting

- ☐ Each code file uses its own entry, specified method names, and this lab's return contracts.
- ☐ Required annotations are present; self need not have an extra annotation.
- ☐ Two-instance, alias, and shared-input observations have actual evidence.
- ☐ Failed Rectangle scaling preserves both dimensions; empty Stack pop/peek both raise IndexError.
- ☐ WordCounter ties, n=0, empty state, and nonmutating queries work.
- ☐ Temperature checks initialization, updates, finiteness, and absolute zero.
- ☐ Gradebook handles empty/missing/tied/replaced data and preserves all state after rejected additions.
- ☐ Per-code notes and A/B declarations match actual assistance; C decisions and debugging records are complete.

The statement, starter, tests, and staff criteria use one contract. Automatic scores do not replace design evaluation or change HW3's course weight.