# CIS 1100 Fall 2026 Practice Exam II

Name: _____ PennID (e.g. 12345): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_____ _____

Signature Date

- Do not open this exam until told by the proctor.
- You will have exactly 60 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts.
- Food and gum are not permitted—don't be noisy or messy.
- You may not use your phone or open your bag for any reason, including to retrieve or put away pens or pencils, until you have left the exam room.
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Python format.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- There are blank pages at the end of the exam if you need extra space for any graded answers.
- Use a pencil or a dark pen to complete the exam.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

# Q1. Code Snippets [10 points]

Below are five **original snippets**. Each operates on some list `words`. For each original snippet, your job is twofold:

- Select the single **alternative snippet** (A–E, on the next page) that produces exactly the same value.
- Write the value it produces for `words = ["yes", "no", "let's", "go"]`.

**Original Snippets:**

**1.**

```
def mystery_one(words):
    if words == []:
        return []
    first = words[0]
    rest = words[1:]
    if len(first) > 2:
        return [first + first] + mystery_one(rest)
    else:
        return mystery_one(rest)
```

**2.**

```
def mystery_two(words):
    if words == []:
        return []
    first = words[0]
    rest = words[1:]
    if len(first) > 4:
        return [first + first] + mystery_two(rest)
    else:
        return mystery_two(rest)
```

**3.**

```
def mystery_three(words):
    if len(words) == 1:
        return len(words[0])
    first = words[0]
    rest = words[1:]
    result = mystery_three(rest)
    if len(first) >= result:
        return len(first)
    return result
```

**4.**

```
def mystery_four(words):
    result = []
    for elem in words:
        result = [elem] + result
    return result
```

**5.**

```
def mystery_five(words):
    transforms = []
    for word in words:
        transforms.append(len(word))
    result = 0
    for x in transforms:
        result += x
    return result
```

**Alternative Snippets:**

**A.**

```
def mystery_a(words):
    if words == []:
        return words
    return mystery_a(words[1:]) + [words[0]]
```

**B.**

```
def mystery_b(words):
    doubled = []
    for word in words:
        doubled.append(transform(word))
    res = []
    for word in doubled:
        if predicate(word):
            res.append(word)
    return res

def transform(word):
    return word + word

def predicate(word):
    return len(word) > 4
```

**C.**

```
def mystery_c(words):
    lengths = []
    for word in words:
        lengths.append(transform(word))
    return max(lengths)

def transform(word):
    return len(word)
```

**D.**

```
def mystery_d(words):
    filtered = []
    for word in words:
        if predicate(word):
            filtered.append(word)
    res = []
    for word in filtered:
        res.append(transform(word))
    return res

def predicate(word):
    return len(word) > 4

def transform(word):
    return word + word
```

**E.**

```
def mystery_e(words):
    acc = 0
    for word in words:
        acc = combine(acc, word)
    return acc

def combine(acc, word):
    return acc + len(word)
```

**Answer Here:** Each row is worth 2 points: 1 for the matching alternative, 1 for the value.

| **Original** | **Matching Alternative (A–E)** | **Value for** `words = ["yes", "no", "let's", "go"]` |
|---|---|---|
| 1 | B | `['yesyes', "let'slet's"]` |
| 2 | D | `["let'slet's"]` |
| 3 | C | `5` |
| 4 | A | `['go', "let's", 'no', 'yes']` |
| 5 | E | `12` |

1. Doubling a word then keeping those longer than 4 is the same as keeping words longer than 2 then doubling them, since doubling exactly doubles the length.
2. Only `"let's"` has more than 4 characters. Here the filter happens *before* the doubling.
3. This is a recursive maximum over the word lengths; `"let's"` is longest at 5.
4. Each element is placed in front of the accumulated result, which reverses the list.
5. 3 + 2 + 5 + 2 = 12. Accumulating a running total is exactly the reduce pattern.

Either quote style is accepted for string values.

## Q2. Dictionaries [5 points]

In HW04: Food Recommender we wrote code answering questions about data stored in dictionaries. Each drink maps to a tuple of `(strength, ingredients, price)`. Write what the function below prints when called on this dictionary d. (None of them raise errors.)

```
d = {
    "Espresso": (2, ["espresso"], 1.85),
    "Red Eye": (8, ["espresso", "drip"], 4.85),
    "Americano": (6, ["espresso", "water"], 3.00),
    "Macchiato": (3, ["espresso", "milk"], 3.45),
    "Cortado": (5, ["espresso", "milk", "foam"], 4.45),
    "Cappuccino": (5, ["espresso", "milk", "foam", "cocoa"], 5.95),
    "Mocha": (5, ["espresso", "milk", "foam", "cocoa", "whipped
       cream"], 5.95),
    "Cold Brew": (5, ["drip", "ice"], 4.95),
}
```

### Question 2.1 *Dictionaries: drinks under a price* [1 points]

```
def one(drinks):
    result = set()
    for drink, info in drinks.items():
        if info[2] <= 3:
            result.add(drink)
    return result
print(one(d))
```

**Solution:** `{"Espresso", "Americano"}`
**Note:** Only Espresso ($1.85) and Americano ($3.00) are at or below $3. Sets print in an arbitrary order, so either order is accepted.

### Question 2.2 *Dictionaries: counting distinct ingredients* [1 points]

```
def two(drinks):
    result = set()
    for drink in drinks.keys():
        for x in drinks[drink][1]:
            result.add(x)
    return len(result)
print(two(d))
```

**Solution:** 8
**Note:** The distinct ingredients are espresso, drip, water, milk, foam, cocoa, whipped cream and ice.

### Question 2.3 *Dictionaries: drinks with many ingredients* [1 points]

```
def three(drinks):
    result = set()
    for drink, info in drinks.items():
        if len(info[1]) > 3:
            result.add(drink)
    return result
print(three(d))
```

**Solution:** `{"Cappuccino", "Mocha"}`
**Note:** Cappuccino has 4 ingredients and Mocha has 5; every other drink has 3 or fewer.

### Question 2.4 *Dictionaries: finding a minimum* [1 points]

```
def four(drinks):
    result = 9999
    for info in drinks.values():
        if info[0] < result:
            result = info[0]
    return result
print(four(d))
```

**Solution:** 2
**Note:** This is the running-minimum pattern over the first element of each tuple.

### Question 2.5 *Dictionaries: grouping into a dictionary of sets* [1 points]

```
def five(drinks):
    result = {"espresso drinks": set(), "drip drinks": set()}
    for drink, info in drinks.items():
        if "espresso" in info[1] and info[0] > 5:
            result["espresso drinks"].add(drink)
        if "drip" in info[1] and info[0] > 5:
            result["drip drinks"].add(drink)
    return result
print(five(d))
```

**Solution:** `{"espresso drinks": {"Red Eye", "Americano"}, "drip drinks": {"Red Eye"}}`

**Note:** The test is strictly greater than 5, so Cold Brew (strength 5) is excluded from the drip drinks.

# Q3. Coding: Exam Registration [2 points]

## Question 3.1 *Coding: students who missed every exam* [1 points]

You run a class preparing students for a licensing exam. A student must take at least one attempt in a given year to be eligible the next year. At each session, the names of everyone in attendance are collected as a set.

Write `missed_all_exams()`, taking a roster (a `set[str]` of names) and a `list[set[str]]` of attendance records from the past year, returning the `set[str]` of students who did not take any exam.

The examples below use the `roster = {"anna", "becca", "cedric", "deniz"}`.

| Call | Returns |
|---|---|
| `missed_all_exams(roster, [{"anna", "cedric"}, {"deniz", "anna"}, set(), {"becca"}])` | `set()` |
| `missed_all_exams(roster, [{"anna", "becca"}, {"deniz", "anna"}, set(), {"becca"}])` | `{'cedric'}` |

Write your implementation below.

```
def missed_all_exams(full_roster: set[str], sessions: list[set[str
    ]]) -> set[str]:
```

**Reference solution:**

One correct solution:

`attended_any |= session` is set union; `attended_any = attended_any.union(sess` is the same thing.

```
def missed_all_exams(full_roster: set[str], sessions: list[set[str
    ]]) -> set[str]:
    attended_any = set()
    for session in sessions:
        attended_any |= session
    return full_roster - attended_any
```

## Question 3.2 *Coding: students registered for too many sessions* [1 points]

At certain times of year, three sessions in a row give exactly the same exam, and students may not register for more than one of them.

Write `took_too_many()`, taking the registration records for those three sessions and returning the names of students registered for more than one.

| Call | Returns |
|---|---|
| `took_too_many({"anna", "becca"}, {"cedric"}, {"deniz"})` | `set()` |
| `took_too_many({"anna", "becca"}, {"cedric", "becca"}, {"deniz"})` | `{'becca'}` |
| `took_too_many({"anna", "becca", "deniz"}, {"cedric", "anna"}, {"anna"})` | `{'anna'}` |

Write your implementation below.

```
def took_too_many(s1: set[str], s2: set[str], s3: set[str]) -> set[
    str]:
```

**Reference solution:**

One correct solution:

A name in more than one session must appear in at least one of the three pairwise intersections.

```
def took_too_many(s1: set[str], s2: set[str], s3: set[str]) -> set[
    str]:
     return (s1 & s2) | (s1 & s3) | (s2 & s3)
```

# Q4. Recursion [3 points]

We **intertwine** two strings by building a new string from the first characters of both, then the second characters, and so on. If one string is longer, the intertwining stops when the shorter string runs out and the remaining characters appear at the end. The output length always equals the sum of the input lengths.

## Question 4.1 *Recursion: why intertwine needs two base cases* [1 points]

| Call | Expected Output |
|---|---|
| `intertwine("abcd", "1234")` | `"a1b2c3d4"` |
| `intertwine("1234", "abcd")` | `"1a2b3c4d"` |
| `intertwine("cis", "1100")` | `"c1i1s00"` |
| `intertwine("", "banana")` | `"banana"` |

Which is the best explanation for why a recursive solution to this problem might require two base cases?

(A) There are two inputs to the function, and we need one base case per input.

(B) There is no guarantee as to which of the two inputs will be shorter. ← **correct**

(C) The sizes of the inputs must change at different rates with each recursive call to reach a base case.

(D) The cases where an input has a length of 1 and a length of 0 need to be handled differently.

**Note:** Either string can run out first, so recursion has to stop on whichever one empties — hence one base case per string.

## Question 4.2 *Recursion: intertwining with the empty string* [1 points]

When calling `intertwine(x, "")` for any string `x`, the output will always be equal to what?

**Solution:** `x` (also accepted: `intertwine("", x)`, `intertwine('', x)`)
**Note:** With nothing to interleave, every character of `x` is what remains.

## Question 4.3 *Recursion: implementing intertwine* [1 points]

Fill in the blanks to implement `intertwine`. Use all the blanks available and only those blanks — although other implementations are possible, you are responsible for one matching this structure.

*Don't forget about string indexing and slicing.*

```
def intertwine(a: str, b: str) -> str:
    # base cases
    if len(a) == ________BLANK_ONE________:

        ________________BLANK_TWO_____________

    if ___________BLANK_THREE___________:

        ___________BLANK_FOUR___________

    # recursive call

    return ___________BLANK_FIVE_____________________________
```

Fill in each blank below. Each is worth an equal share of the credit.

| **Blank** | **Code** |
|---|---|
| BLANK_ONE | `0` |
| BLANK_TWO | `return b` |
| BLANK_THREE | `len(b) == 0` |
| BLANK_FOUR | `return a` |
| BLANK_FIVE | `a[0] + b[0] + intertwine(a[1:], b[1:])` |

BLANK_THREE also accepts `b == ''`, `b == ""`, `not b`, `len(b) < 1`.

## Q5. Coding: Course Scheduling [4 points]

Rami is planning his semester and wants a program that checks whether a potential new course conflicts with the courses already in his schedule, stored as a list of dictionaries. Start and end times use a 24-hour format, so `"1115"` is 11:15 AM and `"1445"` is 2:45 PM.

This is an example of a list of dictionaries that represents a course schedule.

```
[
    {
        "name": "Intro to Psychology",
        "dept": "PSYC",
        "code": "1000",
        "days": ["Mon", "Wed", "Fri"],
        "start_time": "1000",
        "end_time": "1115"
    },
    {
        "name": "History of Art",
        "dept": "ART",
        "code": "2130",
        "days": ["Tue", "Thu"],
        "start_time": "0900",
        "end_time": "1015"
    }
]
```

## Question 5.1 *Coding: dropping senior-only courses* **[1 points]**

Two courses overlap if they are at the same **time** AND on any of the same **days**. Rami already wrote `time_overlap`, which checks only the time:

```
def time_overlap(
    new_course: dict,
    existing_courses: list[dict]
) -> list[dict]:

    overlap = []
    new_start = new_course["start_time"]
    new_end = new_course["end_time"]

    for course in existing_courses:
        # if new course starts before and ends during or after
           existing course
        if (new_start < course["start_time"]) and (new_end > course[
           "start_time"]):
            overlap.append(course)

        # if new course starts during an existing course
        elif course["start_time"] <= new_start < course["end_time"]:
            overlap.append(course)
    return overlap
```

Write code that handles any number of courses. You may assume every file is well formed and every course has all the keys shown.

Rami realises he picked some courses only open to seniors. Write `remove_senior_courses`, taking a list of courses and returning a new list that **excludes** courses with a code greater than or equal to `4000`.

| Call | Returns |
|---|---|
| `remove_senior_courses( [{"code": "1100"}, {"code": "4540"}, {"code": "2250"}])` | `[{'code': '1100'}, {'code': '2250'}]` |

Write your implementation below.

```
def remove_senior_courses(courses: list[dict]) -> list[dict]:
```

**Reference solution:**

One correct solution:

Two problems with the version in the exam's answer key. Its condition is inverted — it *keeps* the senior courses instead of excluding them. And `code` is stored as a string in this JSON, so comparing it against the integer `4000` raises `TypeError`; it needs `int(...)`.

```
def remove_senior_courses(courses: list[dict]) -> list[dict]:
```

```
    final_list = []

    for course in courses:
        if int(course["code"]) < 4000:
            final_list.append(course)

    return final_list
```

## Question 5.2 *Coding: courses sharing a day* [1 points]

Two courses overlap if they are at the same **time** AND on any of the same **days**. Rami already wrote `time_overlap`, which checks only the time:

```
def time_overlap(
    new_course: dict,
    existing_courses: list[dict]
) -> list[dict]:

    overlap = []
    new_start = new_course["start_time"]
    new_end = new_course["end_time"]

    for course in existing_courses:
        # if new course starts before and ends during or after
           existing course
        if (new_start < course["start_time"]) and (new_end > course[
           "start_time"]):
            overlap.append(course)

        # if new course starts during an existing course
        elif course["start_time"] <= new_start < course["end_time"]:
            overlap.append(course)
    return overlap
```

Write code that handles any number of courses. You may assume every file is well formed and every course has all the keys shown.

Write `day_overlap`, taking a new course and a list of existing courses, and returning a list of every existing course sharing at least one day with the new course. Ignore time entirely — check only days.

| Call | Returns |
|---|---|
| `day_overlap({"days": ["Mon"]}, [{"days": ["Mon", "Wed"]}, {"days": ["Tue"]}])` | `[{'days': ['Mon', 'Wed']}]` |
| `day_overlap({"days": ["Mon", "Wed"]}, [{"days": ["Mon", "Wed"]}])` | `[{'days': ['Mon', 'Wed']}]` |

Write your implementation below.

```
def day_overlap(new_course: dict, existing_courses: list[dict]) ->
    list[dict]:
```

**Reference solution:**

One correct solution:

The exam's answer key wrote `overlap.add(course)`, but `overlap` is a list and lists have no `.add()` — that raises `AttributeError`. The `break` also matters: without it, a course sharing two days is added twice.

```
def day_overlap(new_course: dict, existing_courses: list[dict]) ->
    list[dict]:
    new_days = new_course["days"]

    overlap = []

    for course in existing_courses:
        for day in new_days:
            if day in course["days"]:
                overlap.append(course)
                break

    return overlap
```

## Question 5.3 *Unit Testing: tests for overlapping_courses* [1 points]

Two courses overlap if they are at the same **time** AND on any of the same **days**. Rami already wrote `time_overlap`, which checks only the time:

```
def time_overlap(
    new_course: dict,
    existing_courses: list[dict]
) -> list[dict]:

    overlap = []
    new_start = new_course["start_time"]
    new_end = new_course["end_time"]
```

```
    for course in existing_courses:
        # if new course starts before and ends during or after
           existing course
        if (new_start < course["start_time"]) and (new_end > course[
           "start_time"]):
            overlap.append(course)

        # if new course starts during an existing course
        elif course["start_time"] <= new_start < course["end_time"]:
            overlap.append(course)
    return overlap
```

Write code that handles any number of courses. You may assume every file is well formed and every course has all the keys shown.

Write two unit tests for `overlapping_courses`: one where the proposed new course does overlap an existing course, and one where there are no overlaps. You may use these course dictionaries:

```
CIS_1100 = {"name": "Intro Comp", "dept": "CIS", "code": "1100",
        "days": ["Wed", "Fri"], "start_time": "1200", "end_time": "
           1300"}
PSYC_1210 = {"name": "Methods", "dept": "PSYC", "code": "1210",
         "days": ["Mon", "Wed"], "start_time": "1000", "end_time": "
            1100"}
BIO_3210 = {"name": "Cell Biology", "dept": "BIO", "code": "3210",
        "days": ["Tues"], "start_time": "1200", "end_time": "1400"}
HIST_1460 = {"name": "Amer. Hist.", "dept": "HIST", "code": "1460",
         "days": ["Tues", "Thurs"], "start_time": "1115", "end_time"
            : "1230"}
ART_2250 = {"name": "Mesopot. Art", "dept": "ART", "code": "2250",
        "days": ["Tues", "Thurs"], "start_time": "1330", "end_time":
            "1445"}
PHYS_4540 = {"name": "Quantum Phs.", "dept": "PHYS", "code": "4540",
         "days": ["Mon", "Thurs"], "start_time": "1045", "end_time":
             "1200"}
```

Write your implementation below.

```
import pytest


def test_overlapping_case():
    # TODO: finish this unit test


def test_nonoverlapping_case():
    # TODO: put your second test here
```

**Reference solution:**

One correct solution:

Any correct pair of tests earns credit, as long as one exercises an overlapping case and one a non-overlapping case.

```
import pytest


def test_overlapping_case():
    # BIO_3210 is Tues 1200-1400; HIST_1460 is Tues/Thurs 1115-1230.
    # They share Tuesday and their times overlap.
    actual = overlapping_courses(BIO_3210, [HIST_1460, PSYC_1210])
    assert actual == [HIST_1460]


def test_nonoverlapping_case():
    # CIS_1100 is Wed/Fri 1200-1300; ART_2250 is Tues/Thurs
       1330-1445.
    # No shared day and no shared time.
    actual = overlapping_courses(CIS_1100, [ART_2250])
    assert actual == []
```

Write code that handles any number of courses. You may assume every file is well formed and every course has all the keys shown.

Using `time_overlap` and `day_overlap`, write `overlapping_courses`, returning a list of all courses that overlap the new course in time *and* on at least one day.

Write your implementation below.

```
def overlapping_courses(new_course: dict, existing_courses: list[
   dict]) -> list[dict]:
```

**Reference solution:**

One correct solution:

Feeding the day-overlapping courses into `time_overlap` applies both conditions in one pass. The exam's answer key also showed a longer variant that builds a list named `overlaps` but then writes `return overlap` — a `NameError`.

```
def overlapping_courses(new_course: dict, existing_courses: list[
   dict]) -> list[dict]:
    overlap_days = day_overlap(new_course, existing_courses)
    overlap = time_overlap(new_course, overlap_days)

    return overlap
```

## Extra Work Space