

CIS1100.py – Practice Exam Spring 2026 – 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.

- 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 blue or black 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	Q2	Q3	Q4	Q5	Q6
Matching Snippets	OOP Details	Executing Code	Structured Data	Nested Data	Bonus

Q1. Understanding Code Snippets (20 pts)

Below, we have four snippets that each operate on `groups`, a list of sets. For each **original snippet**, your job is twofold:

- Select the single **alternative snippet** that produces exactly the same value of `result`.
- Compute what the value of `result` would be given:

```
groups = [{"a", "b", "c"}, {"b", "c", "d"}, {"c", "d", "e"}]
```

Original Snippets:

1.

```
def mystery_one(groups):
    seen = set()
    result = set()
    for s in groups:
        result = result | (seen & s)
        seen = seen | s
    return result
```

3.

```
def mystery_three(groups):
    result = []
    running = set()
    for s in groups:
        running = running | s
        result.append(len(running))
    return result
```

2.

```
def mystery_two(groups):
    all_elems = set()
    for s in groups:
        all_elems = all_elems | s
    in_multiple = set()
    seen = set()
    for s in groups:
        in_multiple = in_multiple | (
            seen & s)
        seen = seen | s
    return all_elems - in_multiple
```

4.

```
def mystery_four(groups):
    result = groups[0]
    for s in groups[1:]:
        result = result ^ s
    return result
```

Alternative Snippets:

A.

```
def mystery_a(groups):
    total = set()
    for s in groups:
        total = total | s
    result = set()
    for elem in total:
        count = 0
        for s in groups:
            if elem in s:
                count += 1
        if count % 2 == 1:
            result.add(elem)
    return result
```

B.

```
def mystery_b(groups):
    result = set()
    for i in range(len(groups)):
        for j in range(i + 1, len(groups)):
            result = result | (groups[i] & groups[j])
    return result
```

C.

```
def mystery_c(groups):
    result = set()
    for i in range(len(groups)):
        unique_to_i = groups[i].copy()
        for j in range(len(groups)):
            if i != j:
                unique_to_i = unique_to_i - groups[j]
        result = result | unique_to_i
    return result
```

D.

```
def mystery_d(groups):
    result = []
    for i in range(len(groups)):
        combined = set()
        for j in range(i + 1):
            combined = combined | groups[j]
        result.append(len(combined))
    return result
```

Answer Here:

Original Snippet	Matching Alternate (Select A-D)	Result for groups = [{"a", "b", "c"}, {"b", "c", "d"}, {"c", "d", "e"}] (Write the Value)
1	B	{"b", "c", "d"}
2	C	{"a", "e"}
3	D	[3, 4, 5]
4	A	{"a", "c", "e"}

Q2. Banking Banking Banking (17 pts)

This question is about your understanding of how classes are written and used. Here's an example of a class that describes how a `BankAccount` object behaves.

You don't need to know how the methods here work, so don't spend too much time reading the method bodies here.

```
class BankAccount:
    def __init__(self, owner, bank, balance, active):
        self.owner = owner
        self.bank = bank
        self.balance = balance
        self.active = active

    def is_active(self):
        return self.active

    def deposit(self, amount):
        if self.is_active():
            self.balance += amount

    def withdraw(self, amount):
        if not self.is_active():
            print("This account is inactive.")
        elif amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient funds")
```

Question 2.1

What are the names of all of the **attributes** of the `BankAccount` class? (Sometimes we have referred to these as *fields* or *instance variables*, too.)

Solution: owner, bank, balance, status (with or without self)

Question 2.2

What are the names of all of the **methods** of the `BankAccount` class? (Magic methods are methods, too.)

Solution: `__init__`, `is_active`, `deposit`, `withdraw`

Question 2.3

"Maya" wants to create a new account with the bank "Citi Bank". She has a starting balance of 2000 dollars. Upon creation, this account should also be active (i.e. active should be set to True). How would I create a new `BankAccount` object representing her new account and save it in a variable `acct`? **Write one statement (i.e. line of code)**, but you can write across multiple lines if you need the space.

```
Solution: acct = BankAccount("Maya", "Citi Bank", 2000, True)
```

Question 2.4

Maya wants to go out to dinner, but needs to withdraw some cash for the bill. How can I modify the `BankAccount` stored in variable `acct` **using a method** to reflect the \$50 Maya removed from her account? **Write one statement**, but you can write across multiple lines if you need the space.

```
Solution: acct.withdraw(50)
```

Question 2.5

I want to add a feature to the `BankAccount` object allowing account owners to transfer funds. Using the existing `is_active`, `deposit` and `withdraw` methods, write a method, `transfer`, that transfers funds from the current account to a `recipient_account`.

If one or both of the accounts is not active, or if the current account does not have enough funds for the transfer, the method should print "Transfer failed." and nothing else. Otherwise the method should remove the funds from the current account and transfer them into the `recipient_account`. Write a method below, using the provided signature. The function should not return anything.

```
1
2 def transfer(self, amount: int, recipient_account: BankAccount) -> None:
3     if not (self.is_active() and recipient_account.is_active()):
4         print("Transfer Failed.")
5     elif amount <= self.balance:
6         self.withdraw(amount)
7         recipient_account.deposit(amount)
8     else:
9         print("Transfer failed.")
```

Q3. Subsequences (20 pts)

A string s is called a **subsequence** of string t if all the characters in s appear in t in the same order, but not necessarily consecutively. For example:

s	t	Is s a subsequence of t?
"ace"	"abcde"	Yes
"aec"	"abcde"	No
"db"	"abcde"	No
"cat"	"carat"	Yes
" "	"hello"	Yes

Question 3.1

Finish this implementation of `is_subsequence`, which returns `True` if s is a subsequence of t and `False` otherwise.

```
def is_subsequence(s: str, t: str) -> bool:
    i = 0
    for ch in _____BLANK_ONE_____:
        if i < len(s) and _____BLANK_TWO_____:
            _____BLANK_THREE_____
    return _____BLANK_FOUR_____
```

Blank	Code
ONE	<code>t</code>
TWO	<code>ch == s[i]</code>
THREE	<code>i += 1</code>
FOUR	<code>i == len(s)</code>

Question 3.2

Now we want to write the function `find_pairs`, which takes in a list of words and returns a list of tuples `(s, t)` containing each pair of **distinct** words where `s` is a subsequence of `t`. For example, `find_pairs(["ab", "ace", "abcde", "race"])` should return (the order doesn't matter):

```
[("ab", "abcde"), ("ace", "abcde"), ("ace", "race")]
```

Fill in the blanks.

```
def find_pairs(words: list) -> list:
    result = []
    for i in range(_____BLANK_ONE_____):
        for j in range(_____BLANK_TWO_____):
            if _____BLANK_THREE_____:
                continue
            if is_subsequence(_____BLANK_FOUR_____):
                result.append(_____BLANK_FIVE_____)
    return result
```

Blank	Code
ONE	<code>len(words)</code>
TWO	<code>len(words)</code>
THREE	<code>i == j</code>
FOUR	<code>words[i], words[j]</code>
FIVE	<code>(words[i], words[j])</code>

Question 3.3

The output of `find_pairs` above was a list of tuples `(s, t)`. Lyla suggests storing the output instead as a list of lists (e.g. `[["ace", "abcde"], ...]`).

Select the best reason to prefer a list of tuples.

- (A) **[CORRECT]** The pairs are fixed relationships between strings, meaning they shouldn't be changed, and tuples are immutable.
- (B) tuples can be indexed but list cannot, so using a list of lists wouldn't allow us to get the first element of each pair
- (C) tuples use less memory than lists only when it comes to storing strings.
- (D) You can't iterate through a list of lists using a for loop.

Q4. Course Scheduling (38 pts)

Rami is a sophomore student planning his semester and wants to write a program to help him decide on his course schedule. The program will check if a potential new course has a time conflict with any courses in his existing schedule, which he has stored in a JSON.

The course data Rami uses is stored in a particular JSON format. An example of which is shown below.

Note that start and end times are stored in a 24-hour time format, so "1115" represents 11:15 AM, and "1445" represents 2:45 PM.

```
[
  {
    "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"
  }
]
```

For the questions that follow, make sure to write functions and code that can handle any size JSON file for any number of courses! You may assume that all JSON files are well-formed and that every course contains all of the keys shown below.

Rami determines that he needs to check two things about a new course to see if it overlaps with any existing courses. Two courses overlap if they are at both the same **time** AND they are on any of the same **days**.

Rami wrote the following function to check if a potential course is at the same **time** as an existing course. Note that two courses can be at the same time but run on different days and therefore not overlap.

We provide the function signature here. The full implementation of the function is available in the appendix if you would like to reference it.

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

Question 4.1

Write a function `day_overlap` that takes a new course and a list of existing courses and returns a list of all existing courses that share at least one day with the new course. This function should not take time into account and should **only** check overlap among days.

```
def day_overlap(new_course: dict, existing_courses: list[dict]) -> list[dict]:
    # TODO: your function here
    new_days = new_course["days"]

    overlap = []

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

    return overlap

# NOTE: Technically you should cast overlap to a set before returning it.
# Don't take points off for that-- whether they do it or not.
```

Question 4.2

Using the `time_overlap` and `day_overlap` functions, write a function, `overlapping_courses`, that returns a list of all courses that overlap in time and on at least one day with `new_course`.

```
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
```

OR

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

    overlap_days = day_overlap(new_course, existing_courses)
    overlap_times = time_overlap(new_course, existing_courses)

    overlaps = []

    for day in overlap_days:
        for time in overlap_times:
            if day == time:
                overlaps.append(day)

    return overlaps
```

Question 4.3

After finalizing his schedule, Rami realizes that he has selected some courses that are only available to senior students. Write a function called `remove_senior_courses` that takes in a list of courses and returns a new list that excludes courses with a code greater than or equal to 4000.

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

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

    return final_list
```

FOR USE IN QUESTION 4.4:

```
CIS_1100 = {  
  "name": "Intro Computing",  
  "dept": "CIS",  
  "code": "1100",  
  "days": ["Wed", "Fri"],  
  "start_time": "1200",  
  "end_time": "1300"  
}
```

```
HIST_1460 = {  
  "name": "American History",  
  "dept": "HIST",  
  "code": "1460",  
  "days": ["Tues", "Thurs"],  
  "start_time": "1115",  
  "end_time": "1230"  
}
```

```
PSYC_1210 = {  
  "name": "Research Methods",  
  "dept": "PSYC",  
  "code": "1210",  
  "days": ["Mon", "Wed"],  
  "start_time": "1000",  
  "end_time": "1100"  
}
```

```
ART_2250 = {  
  "name": "Mesopotamian Art",  
  "dept": "ART",  
  "code": "2250",  
  "days": ["Tues", "Thurs"],  
  "start_time": "1330",  
  "end_time": "1445"  
}
```

```
BIO_3210 = {  
  "name": "Cell Biology",  
  "dept": "BIO",  
  "code": "3210",  
  "days": ["Tues"],  
  "start_time": "1200",  
  "end_time": "1400"  
}
```

```
PHYS_4540 = {  
  "name": "Quantum Physics",  
  "dept": "PHYS",  
  "code": "4540",  
  "days": ["Mon", "Thurs"],  
  "start_time": "1045",  
  "end_time": "1200"  
}
```

Question 4.4

Write two unit tests for `overlapping_courses`. One test should check a case where the proposed new course does overlap with an existing course, and one test should check the case where there are no overlapping courses. We have provided some courses defined as dictionaries on the previous page, feel free to use these in your tests.

```
import unittest

class TestOverlappingCourses(unittest.TestCase):
    def test_overlapping_case(self):
        # TODO: Finish this unit test!

        courses = [PSYC_1210, HIST_1460, ART_2250]

        expected = [HIST_1460, ART_2250]
        actual = overlapping_courses(BIO_3210, courses)

        self.assertEqual(expected, actual)

    def test_nonoverlapping_case(self):
        # TODO: Put your second test here!

        courses = [PSYC_1210, HIST_1460, ART_2250]

        expected = []
        actual = overlapping_courses(CIS_1100, courses)

        self.assertEqual(expected, actual)
```

Q5. Scraping Encyclopedia

[12 points]

You're writing a script to extract content from an online wiki that is stored in xml format. Each page has the same structure. Here the first page of the wiki:

```
<head><title>CIS1100 Encyclopedia</title></head>
<body>
  <h1>Page 1: Dogs</h1>
  <div class="nav"><a link="next_page.xml">Next Entry</a></div>
  <ul class="main-content">
    <li class="fact">The first dog was discovered at least 10 years ago.</li>
    <li class="fact">You've probably met a dog before.</li>
    <li class="ad">Dogs love Harry's Pet Chow.</li>
    <li class="fact">Dogs can have as many as four legs usually.</li>
  </ul>
</body>
```

Each page has an unordered list (indicated by a `ul` element) that contains bullet points. Some represent **facts**, others represent paid **advertisements**.

Question 5.1

Write the name of the tag that represents a single bullet point.

Solution: `li`

Question 5.2

Describe how you can use the XML to determine which bullet points are facts and which are advertisements. (You can be brief—full sentences are not required.)

Solution: By looking at the `class` attribute

Question 5.3

Write a short snippet that prints the fraction of bullet points on the page that are ads as a number. This should work for any page that has this structure. You can assume you have already parsed the xml file as follows: `soup = BeautifulSoup(file, "xml")`. Remember *find* and *find_all*!

Solution:

One option:

```
ads = soup.find_all("li", class_="ad")
total = soup.find_all("li")
print(len(ads) / len(total))
```

Another way (manual counting):

```
count = 0
total = soup.find_all("li")
for item in total:
    if item["class"] == "ad":
        count += 1
print(count / len(total))
```

Question 5.4

Most pages contain a link to the next page in the wiki, contained in the tag `<a>`. The last page does not contain any links at all, and it is displayed below.

```
<head><title>CIS1100 Encyclopedia</title></head>
<body>
  <h1>Page 123812398123: Geese (Band)</h1>
  <div class="nav">That's it!</div>
  <ul class="main-content">
    <li class="fact">Despite the name, this musical group has no birds as
      members.</li>
    <li class="fact">They write some good tunes.</li>
    <li class="ad">Stream Geese on Suhani's premiere streaming service:
      Suhanify</li>
  </ul>
</body>
```

Finish the function below that returns the URL of the linked page, or `None` if this is the last page. New pages are being added all the time, so you can't assume that the specific page above will always be the last one; instead, determine this based on the structure of the page itself.

```
def next_link(soup: BeautifulSoup) -> str | None:
```

.

Solution:

```
a = soup.find("a")
if not a:
    return None
return a["link"]
```

Q6. Bonus Point

Recommend something for the course staff: a book, a movie, a song, whatever. Or, if not, then you could create a piece of art! All exams will get full credit for this question even if you leave it blank.

Extra Work Space

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.

Appendix

Q4 time_overlap function

```
1 def time_overlap(  
2     new_course: dict,  
3     existing_courses: list[dict]  
4 ) -> list[dict]:  
5  
6     overlap = []  
7     new_start = new_course["start_time"]  
8     new_end = new_course["end_time"]  
9  
10    for course in existing_courses:  
11        # if new course starts before and ends during or after existing course  
12        if (new_start < course["start_time"]) and (new_end > course["  
13            start_time"]):  
14            overlap.append(course)  
15  
16        # if new course starts during an existing course  
17        elif course["start_time"] <= new_start < course["end_time"]:  
18            overlap.append(course)  
19    return overlap
```