CIS1100.py — Fall 2024 — Exam 2 **Solution**

Full Name: _____

PennID (e.g. 12345678): _____

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

_____ _____

Signature Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- 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.
- Only answers on the FRONT of pages will be graded. There are two 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 (bonus)

Q1. Fill In The Blank

For Q1.1-1.4, write the value that gets printed. **For Q1.5,** fill in the blank so that the given value gets printed.

Question 1.1

```
input = [1, 2, 3, 4]
print(list(map(lambda x: x * x, input)))
```

Answer: **`[1, 4, 9, 16]`**

Question 1.2

```
python_profs = ["Harry", "Travis"]
java_profs = ["Harry", "Jessica"]
profs = python_profs + java_profs
tas = ["Adi", "Cedric", "Sukya", "Jared", "Molly"]
staff = set(profs) & set(tas)

print(staff)
```

Answer: **`set()`** **# in other words: this prints an empty set**

Question 1.3

```
def mystery_1(a):
    if a == 0:
        return "0"
    if a == 1:
        return "1"
    if a % 2 == 1:
        return mystery_1(a // 2) + "1"
    return mystery_1(a // 2) + "0"

print(mystery_1(16))
```

Answer: **`10000`**

Question 1.4

```
def mystery_3(x):
    d = dict()
    for elem in x:
        k = elem % 3
        d[k] = d.get(k, 0) + elem
    return d

print(mystery_3([3, 10, 22, 33, 44, 6]))
```

Answer: **{0: 42, 1: 32, 2: 44}**

Question 1.5

For this question, you'll fill in the blank after courses so that the print statement contains an expression that prints out "Travis".

```
courses = {
      "cis1100": {
            "prof": ["Harry", "Travis"],
            "language": "python"
      },
      "cis1200": {
            "prof": ["Swap", "Pierce"],
            "language": ["ocaml", "java"]
      }
 }

print(courses_____) # what goes in the blank to print "Travis"?
```

Answer: **["cis1100"]["prof"][1]**

Q2. Trading Up (Debugging)

Sukya is registering for courses and decides to write a program to help her decide which courses to take. In order to do this, she has defined a dataclass like so:

```
@dataclass
class Course:
    name:   str
    days:   str
    is_te:  boolean
    rating: float
```

name is the name of the course, e.g. "CIS 1100". **days** is never empty and always contains the initials of the days on which the course meets, where the only options are M, T, W, H, and F. The day strings are always provided in this order; that is, Friday is always the last day initial in the string if it's there. **is_te** denotes whether a course is a "technical elective". **rating** is a number between 0 and 4 that denotes the course quality.

With this Course dataclass, she writes the function
`best_replacement(replace: Course, options: list[Course]) -> str:`
Which takes in a single course as its first parameter and a list of potential replacement courses. The function returns the name of the Course with the highest rating that *doesn't meet on a Friday* **and** *has the same TE (technical elective) status as the course it would replace* (they are both TEs or neither are TEs). If there's no better class that meets both criteria, the function should just return the one she was trying to replace.

Help fix Sukya's implementation of `best_replacement`. In the table that follows, identify the line on which each of the bugs appears, the part of the code that is incorrect, and the code you can replace that part with to make it correct. We filled in an example bug on the first line. Note that there are no further bugs in the function signature—i.e., the input and return types are correct. **There are five other bugs.**

```
 1. fed best_replacement(replace: Course, options: list[Course]) -> str:
 2.     # remove all courses that meet on Friday
 3.     not_friday = list(filter(lambda c : c[-1] != 'F', options))
 4.
 5.     # remove all courses that don't match the TE status
 6.     matching_status = [c for c in not_friday if c.is_te or replace.is_te]
 7.
 8.     best_so_far = matching_status[0]
 9.     for c in matching_status:
10.         if c.rating > best_so_far.rating:
11.             c = best_so_far
12.     return best_so_far
```

Line Number	Incorrect Code	Replacement Code
1	`fed`	`def`
3	`c[-1] != 'F'`	`c.days[-1] != 'F'`
6	`c.is_te or replace.is_te`	`c.is_te == replace.is_te`
8	`best_so_far = matching_status[0]`	`best_so_far = replace`
11	`c = best_so_far`	`best_so_far = c`
12	`return best_so_far`	`return best_so_far.name`

Q3. Complete the Program

Create a Library class to manage a library's collection of books and track book loans. The class will model books as strings, and it should allow books to be added to the collection, lent out to borrowers, and returned. Implement methods to add a new book, lend a book to a borrower, return a book, and list all available books. Complete the code by filling in the blanks to correctly initialize attributes and manage the library's inventory and loan status.

Note: `dict.pop(key)` is a method that returns the value from a dict associated with the given key and then deletes the key-value mapping from the dict. Use it in one of the blanks!

The code to complete and some examples of the Library's usage are found on the next page.

```
class Library:
    def __init__(self, library_name: str):
        ____BLANK_0____ = library_name
        self.books = set()  # set to store available books
        self.loans = {}     # dict to track loans (key=book title, val=borrower name)

    def add_book(self, book_title: str):
        if book_title not in  ____BLANK_1____:
            self.books.____BLANK_2____
        else:
            print(f"'{book_title}' is already in the library collection.")

    def lend_book(self, book_title: str, borrower_name: str):
        if book_title in self.books and book_title not in self.loans:
            self.____BLANK_3____ = borrower_name
            self.books.remove(____BLANK_4____)
        else:
            print(f"Cannot lend '{book_title}': loaned or not in collection.")

    def return_book(self, book_title: str):
        if book_title in self.loans:
            borrower = self.loans____BLANK_5____
            self.books.____BLANK_6____(book_title)
            print(f"'{book_title}' has been returned by {borrower}.")
        else:
            print(f"'{book_title}' was not on loan.")

    def get_available_books(self) -> set[str]:
        return self.____BLANK_7____
```

See below for example usage of this class:

```
library = Library("Van Pelt")
library.add_book("Calculus Blue Guide")
library.add_book("The Great Gatsby")
library.lend_book("Calculus Blue Guide", "Harry")

print(library.get_available_books())
# Prints: {'The Great Gatsby'}

library.return_book("Calculus Blue Guide")
# return_book will print: 'Calculus Blue Guide' has been returned by Harry.

print(library.get_available_books())
# Prints: {'The Great Gatsby', 'Calculus Blue Guide'}
```

Blank #	Code
0	`self.library_name` `# other answers would have worked as long it started with self.`
1	`self.loans or self.books`
2	`add(book_title)`
3	`loans[book_title]`
4	`book_title`
5	`.pop(book_title)  or  [book_title] or .get(book_title, ...)`
6	`add`
7	`books`

Q4: Pandas

Molly is a huge baseball fan and she wants to do some analysis on Major League Baseball scores. She maintains a file, **scores.csv**, that tracks the teams that played in each game, the date the game was played, and each team's score. Here are the first three rows of that CSV when read into the DataFrame called **scores**.

INDEX	team_a	team_b	score_a	score_b	date
0	"Phillies"	"Braves"	5	4	09/13/2024
1	"Padres"	"Mets"	3	0	09/15/2024
2	"Dodgers"	"Phillies"	1	0	10/01/2024

Solve each of the tasks on the next page using Pandas. You can refer to the appendix for some tools that you'll need. The later tasks can be solved using columns created in the earlier tasks. We will grade the later tasks assuming that you completed the earlier ones correctly.

Molly realizes that she doesn't have a column to track the winner of each game. Write a line that adds a column called **diff** to the DataFrame **scores** that calculates the difference between team A's score and team B's score. This could be positive or negative.

```
scores["diff"] = scores["score_a"] - scores["score_b"]
```

Follow up by writing a line that adds a column **a_wins** storing a boolean value that is **True** when **team_a** won the game and **False** otherwise. (In baseball, higher scores win.)

```
scores["a_wins"] = scores["diff"] > 0
```

Molly encoded her data so that Team A represents the team that was playing in their home stadium, or "at home." Write a few lines of Pandas code that ends by **printing** the fraction of games that the Phillies win at home. (Meaning the number of games in which the Phillies were Team A and won divided by the total number of games in which they were Team A.)

```
just_phillies = scores[scores["team_a"] == "Phillies"]
total_games = just_phillies.shape[0]
phillies_home_wins = just_phillies[just_phillies["a_wins"]]
total_wins = phillies_home_wins.shape[0]
print(total_wins / total_games)
```

Q5. Coding Zoo

We're opening a zoo! Zoos contain Animals, which are of some species and require some food:

```
class Animal:
    def __init__(self, species: str, food_required: int):
        self.species = species              # e.g. "Lion"
        self.food_required = food_required  # amount of food in lbs, e.g. 5
```

Question 5.1

There are also zookeepers, who each specialize in a certain species of animal:

```
class Keeper:
    def __init__(self, specialty: str):
        self.specialty = specialty
    def specializes(self, a: Animal) -> bool:
        ...
```

Inside the `Keeper` class, we will add a method `specializes` that takes in an `Animal` as input and checks whether the zookeeper specializes in the given `Animal`'s type. Your implementation must satisfy these tests:

```
def test_does_specialize(self):
    k = Keeper("Lion")
    a = Animal("Lion", 10)
    self.assertTrue(k.specializes(a))
```

```
def test_not_specializes(self):
    k = Keeper("Tiger")
    a = Animal("Lion", 10)
    self.assertFalse(k.specializes(a))
```

Finish the method signature, then provide a **one line implementation** of the method below.

```
def specializes(self, a: Animal) -> bool:
    return self.specialty == a.species
```

The Zoo class contains a list of `Animals` and Keepers, like so:

```
class Zoo:
    def __init__(self, animals: list[Animal], keepers: list[Keeper]):
        self.animals = animals
        self.keepers = keepers
    def has_redundant_keepers(self):
        ...
    def has_enough_keepers(self):
        present_species {a.species for a in self.animals}
        covered_species = {k.specialty for k in self.keepers}
        return len(present_species - covered_species) == 0
```

Our goal will be to test the bottom two methods: `has_redundant_keepers` and `has_enough_keepers`. Then, you'll implement the first method, `has_redundant_keepers`, which return True if the list of keepers contains more than one keeper for any species. Otherwise, it will return False.

The second method `has_enough_keepers`, which is implemented for you, will return True if every Animal in the Zoo has at least one Keeper that specializes in its species. (One Keeper can care for all Animals of their corresponding species.) Otherwise, it will return False.

Testing, Testing...

Help complete our testing suite!

Question 5.2

Complete `test_redundant_keeper_case`, which should verify that `has_redundant_keeper` value returns the correct value when called on the Zoo created at the start of the test case.

```
from zoo import Zoo, Animal, Keeper
class Tests(unittest.TestCase):

    def test_redundant_keepers_case(self):
        animals = [Animal("Lion", 20), Animal("Tiger", 10)]
        keepers = [Keeper("Lion"), Keeper("Lion")]
        zoo = Zoo(animals, keepers)
        # TODO: Finish me to verify that has_redundant_keepers()
        # returns the correct value in this case
        self.assertTrue(zoo.has_redundant_keepers())
```

Question 5.3

Fill in `test_insufficient_keepers` (part of the Tests unittest suite from above). This test should verify that `has_enough_keepers` returns False in a case where it would be correct to do so. You will be responsible for constructing your own Zoo for this test.

```
    def test_insufficient_keepers(self):
        animals = [Animal("Lion", 20), Animal("Tiger", 10)]

        # TODO: Write a unit test that verifies that
        # has_enough_keepers returns False in a case
        # where there is an animal whose species is not
        # covered by a keeper.
        keepers = [Keeper("Lion"), Keeper("Lion")]
        zoo = Zoo(animals, keepers)
        self.assertFalse(zoo.has_enough_keepers())
```

Question 5.4

Provide an implementation of `has_redundant_keepers` **without using any loops.** Recursion, list/set/dict comprehensions, lists, sets, dicts, and higher order functions are all OK.

```
def has_redundant_keepers(self):
    covered_species = {k.specialty for k in self.keepers}
    return len(covered_species) < len(self.keepers)
```

Q6. (Bonus)

Recommend us something! Anything at all—music, movie, TV, game, book, restaurant, park, anything. All exams will get full credit for this question, no pressure to make anything in particular! Pictures are also nice, but make sure you take the time to finish the exam :)

Extra Answers Page (This page is intentionally blank)

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

Pandas

Note here that **pd** refers to the **pandas** library, **df** refers to a **DataFrame**, and **s** refers to a **Series**.

Viewing data

Function	Description
`df.col`	Accesses the column named col in df as a Series.
`df['col']`	Accesses the column named col in df as a Series.
`df[['col1', 'col2', ...]]`	Accesses a list of columns (here, ['col1', 'col2', ...]) as a DataFrame.
`df.iloc[i]`	Accesses row i in df as a DataFrame.
`df[start:stop:step]`	Accesses the rows start (inclusive) to stop (exclusive) using step size step.
`df.head(n=5)`	Shows the first n rows.
`df.tail(n=5)`	Shows the last n rows.

Describing data

Function	Description
`df.shape`	A tuple of the dimensions of df of the form (num_rows, num_cols).
`df.info()`	Prints out summary information about a DataFrame, including the number of columns, non-null values, and datatypes.
`df.dtypes`	A Series containing the data type of each column.
`df.columns`	A Series containing all column labels.
`df.describe()`	Generates a DataFrame with descriptive statistics (count, mean, standard deviation, median, quartiles, etc.).
`df.empty`	A bool that indicates whether an entire DataFrame (or Series) is empty.
`df.count()`	Counts the number of non-null elements in each column.
`df.value_counts(normalize=False)`	Counts the number of each value that appears in each column.

`df.nunique()`	Counts the number of unique values in each column.
`df.min(axis=0)`	Finds the minimum of each row or column.
`df.max(axis=0)`	Finds the maximum of each row or column.
`df.mean(axis=0)`	Finds the mean/average of each row or column.
`df.sum(axis=0)`	Finds the sum of each row or column.

Filtering Operators

For boolean Series **s** and **t**,

Operator	**Python Equivalent (not proper pandas code)**	**Description**
`s & t`	`s and t`	Both must be true to be true
`s | t`	`s or t`	As long as one is true, the whole thing is true
`s ^ t`	`s != t`	Exactly one is true
`~s`	`not s`	Flips the boolean

Remember that you can use comparison operators with values on Series. For example,

- s > 0
- s == “Hello”
- s <= 8

Set Operations

For sets **a** and **b**,

Function	**Operator Form**	**Description**
`a.union(b)`	`a | b`	Creates a new set with the elements from either **a** or **b**.
`a.intersection(b)`	`a & b`	Creates a new set with the elements that are in both **a** and **b**.
`a.difference(b)`	`a - b`	Creates a new set with the elements that are in **a** but not **b**.
`a.symmetric_difference(b)`	`a ^ b`	Creates a new set with elements that are in either **a** or **b**, but not both.