CIS 1100

Exam Tips

Python

Spring 2025

University of Pennsylvania

Different Sequences, Different Behaviors

Type	Ordered	Indexable	Common Methods	Notes
String	Yes	Yes	`.find()`, `.replace()`, `.split()`	Immutable
List	Yes	Yes	`.append()`, `.extend()`	Mutable
Set	No	n	`.add()`, `.remove()`, `&`, `|`	Unordered, no duplicates

Common mistakes:

- Using the same methods across all sequences (DO NOT DO THIS)
- Forgetting you can't index into a set
- Misreading `{}` vs `[]`. You can always use `set()`, `dict()`, `list()` if you want to be super clear when writing code.

Formatting and Comprehensions

```python
list_one = list("CIS4480ISTHEBEST")
list_two = list("CIS1100ISTHEBEST")

result_one = {char for char in list_one if char not in list_two}
result_two = [char for char in list_one if char not in list_two]

print(result_one) # {'4', '8'}
print(result_two) # ['4', '4', '8']
```

- `[x for x in seq]` → list comprehension
- `{x for x in seq}` → set comprehension

Misidentifying the structure leads to wrong answers. Make sure you know exactly what you're working with.

Map, Filter, Reduce: Quick Reminders

Tool	Behavior	Good Lambda
`map`	Applies function to each item	`lambda x: x * 2`
`filter`	Keeps items passing a test	`lambda x: x > 0`
`reduce`	Combines all items to a single value	`lambda x, y: x + y`

- **map** & **filter**:
  - Forgetting to wrap `map` or `filter` in `list()` if you want a list of items
- **reduce**:
  - DO NOT WRITE lambdas that modify the accumulator in-place.
  - Forgetting to provide an initial value to catch edge cases (if necessary).

Lambdas Must Be Exact

- Must have the **correct number of arguments**. If you know it needs two, make sure to give it two arguments.
- In `reduce`, **don't modify** the accumulator directly — always **return a new object** (this is a bit on cusp of being out of scope of the class, but this is possible to happen with lists! To be safe, do not perform `.method` calls on the accumulator.)

Bad example ❌:

```
reduce(lambda acc, elem: acc.append(elem), nums, [])
```

Good example ✅:

```
reduce(lambda acc, elem: acc + [elem], nums, [])
```

How Recursion Accumulates Results

Return Type	Accumulation Behavior
String	Concatenation (`+`)
List	Concatenation (`+ [item]`)
Number	Addition (`+`) or Multiplication (`*`)

```python
def build_str(s):
    if not s:
        return ""
    return s[0] + build_str(s[1:])
```

```python
def sum_nums(nums):
    if not nums:
        return 0
    return nums[0] + sum_nums(nums[1:])
```

You should be able to answer questions like: Suppose you call `build_str("1234")` and `sum_nums(["1", "2", "3", "4"])`.

Do they both succeed? If so, what does each function return?

0 Modulo Anything Is Always 0

No exceptions:

```
0 % 5 == 0
0 % 99 == 0
0 % (-8) == 0
```

Misapplying `%` often causes wrong conditionals. Please do not miss questions becuase you don't know this. From 0 things, there is nothing "remaining" possible.

note: 0 % 0 gives an error.

Handling Missing Keys

When updating dictionaries:

- You must check if a key exists first.
- Otherwise you get a `KeyError`.

Manual Defaulting

Without any help:

```
d = {}
if key not in d:
    d[key] = 1 # if we don't do this; bugs!
else:
    d[key] += 1
```

Tedious, easy to mess up.

`defaultdict`: Automatic Defaults

A better way:

```python
from collections import defaultdict

d = defaultdict(int)
d[key] += 1
```

No need to check if the key exists, will default to `0`!

Example: Start New Keys at 1

You can control the default:

```
from collections import defaultdict

d = defaultdict(lambda: 1)
print(d["apples"])  # 1
d["apples"] += 1
print(d["apples"])  # 2
```

`defaultdict(list)`: Default Empty Lists

```python
from collections import defaultdict

d = defaultdict(list)
d["fruits"].append("apple")
d["fruits"].append("banana")

print(d)
# {'fruits': ['apple', 'banana']}
```

- New keys automatically start with an empty list `[]`.

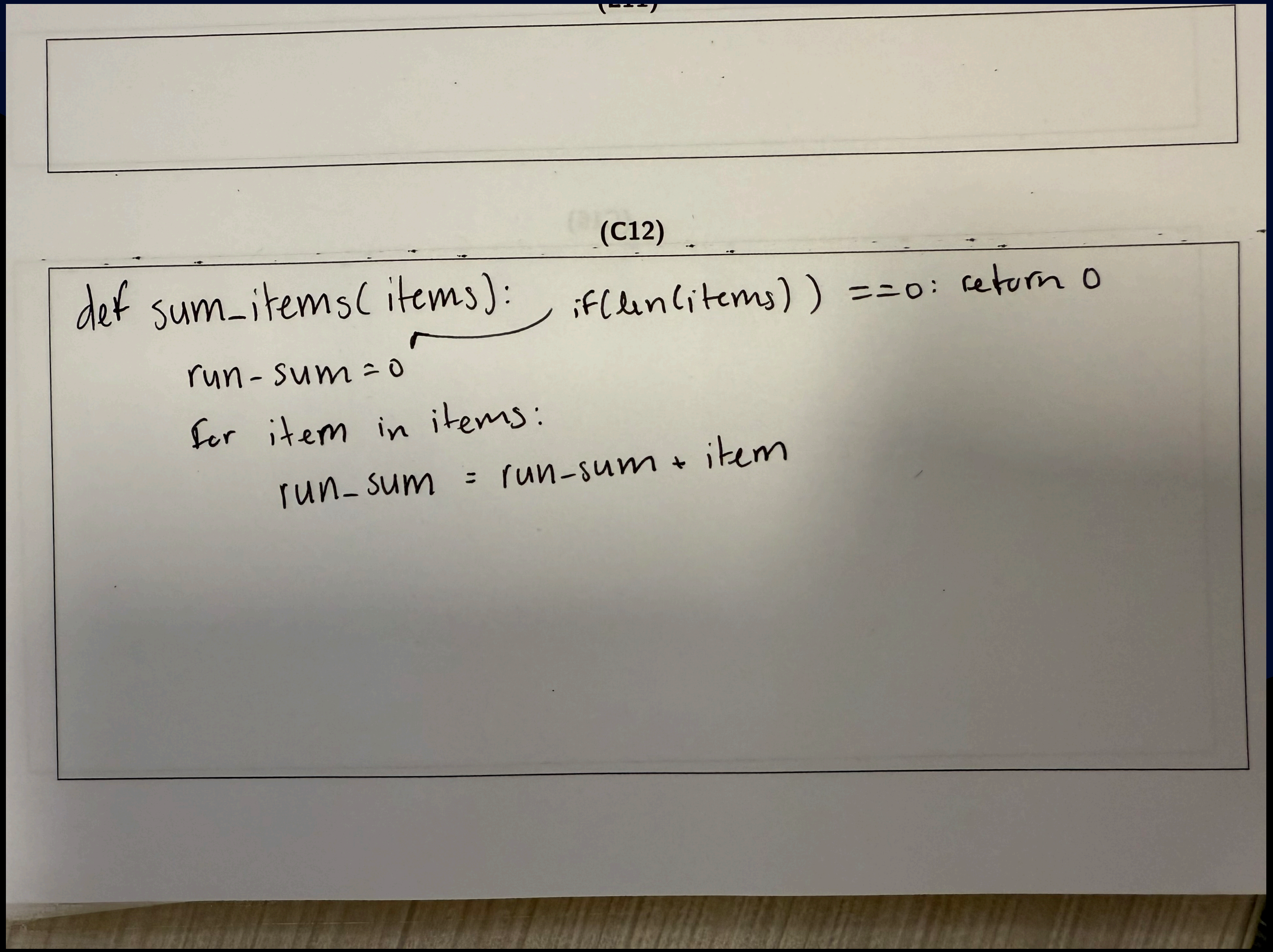
When to Use `defaultdict`

Use it when:

- You want **missing keys** to safely have a starting value.
- You want **less code** and **fewer mistakes**.

Otherwise, manual checks are required.

What Is Missing from This Image?



(C12)

```
def sum_items( items):   if(len(items)) ) ==0: return 0
    run-sum = 0
    for item in items:
        run_sum = run-sum + item
```

- You are tasked with **returning the sum of elements in a list**.
- Look carefully: **What key part is missing?**

`return` !!!!

DO NOT FORGET TO RETURN!

DO NOT FORGET TO RETURN WHAT YOU NEED TO RETURN.

DO NOT LOSE POINTSSSS!!!

No return = less points.

Always double-check: What is the function supposed to give back? Common mistake made by people who leave early. `PRINTING IS NOT THE SAME AS RETURNING!`

Fill in the Blank Tips

Complete a function that sums the odd elements in a list and then prints the result.

```python
import sys

def sum_odd(args):
    total = 0
    for arg in args:
        num = int(arg)
        if ___BLANK_1___:
            ___BLANK_2___
    return total

def main():
    args = ___BLANK_3___
    result = ___BLANK_4___
    print(result)
```

- **Anchor yourself on what cannot be wrong.**
- **Start from the lines where behavior is most predictable and necessary.**

Example Walkthrough:

- Work upward from certainties.
  - `print(result)` means `result` must exist.
  - `result = ...` must call `sum_odd` — there is no other option.
  - `args = ...` must strip out the filename: slice `sys.argv[1:]`.
- Then move inside the function.
  - `for arg in args:` means we are looping through numbers as strings.
  - We need to check for odd numbers, so use `num % 2 != 0`.
  - If odd, add to total: `total += num`.
- Prioritize what must happen, check if variable names give hints, and infer!

Start with immovable facts, then fill in conditions and logic around them.

That's it!