PHASE 2 · DATA AND FUNCTIONS

Day 12 Checkpoint project 1

Self-Study Guide · About 75 minutes

By the end you will be able to

- Build a text-based to-do list manager from a written brief, one working milestone at a time.
- Store tasks as a list of dictionaries and give each menu action its own function.
- Keep a menu program running when the user types a bad menu choice or a bad task number.
- Test a program on purpose by trying every menu option and every kind of bad input.

What to remember from earlier

- input() always gives back a string, and strip() removes spaces from both ends of what was typed. (Day 3: Strings)
- A while True loop runs until a break inside it ends the loop. (Day 5: Loops, 5.8 break and continue)
- enumerate(items, 1) gives each item together with a position that starts at 1. (Day 5: Loops, 5.11 enumerate(): Index and Value Together)
- append() adds an item to the end of a list, and pop(index) removes the item at that index and gives it back. (Day 6: Lists)
- A dictionary stores values under keys, so task["done"] = True changes the value stored under "done". (Day 8: Dictionaries, 8.3 Adding, Updating, and Deleting)
- A function returns a value with return, and a function that ends without one returns None. (Day 9: Functions, part 1, 9.5 Returning Versus Printing)
- all() gives True when every value a generator expression produces is true. (Day 11: Comprehensions and iteration tools, 11.7 sum(), any() and all())

Make a `day12` folder and open it in your editor, as on every earlier day. Today has **no new reading**. You spend the whole hour building one program, `todo.py`, using only what Days 1 to 11 taught.

Real programs combine many ideas at once, and the hard part is no longer "what does `pop` do?" but "which tool fits this job, and in what order?" A **checkpoint project** is a program you build to find out what you can already do on your own. It also shows you which earlier days need another look.

Work through the milestones in order. Each one ends with a program that **runs**, so you always have something working to come back to. Type your own code. The reference solution is at the end, under *Putting it together*; open it only after your program passes the tests in *12.8*, or when you have been stuck on one milestone for more than 10 minutes.

12.1 The Project Brief

A **project brief** is a short description of what a program must do, written before anyone writes code. It says *what*, not *how*. Here is today's.

Build a text-based to-do list manager.

- The program shows a menu with five choices: **1** Add, **2** List, **3** Complete, **4** Delete, **5** Quit. It keeps showing the menu after each action until the user chooses Quit.
- **Add** asks for a title and adds a new, unfinished task to the end of the list. A blank title is refused.
- **List** shows every task with its number, starting at 1, and a check box: `[x]` for done, `[ ]` for not done. An empty list says so.
- **Complete** asks for a task number and marks that task as done.
- **Delete** asks for a task number and removes that task.
- The tasks are stored as a **list of dictionaries**. Each task is one dictionary with two keys: `"title"` (a string) and `"done"` (a bool).
- **Each action has its own function**: `show_menu()`, `add_task(tasks, title)`, `list_tasks(tasks)`, `complete_task(tasks, number)`, `delete_task(tasks, number)`, and `main()`, which runs the menu loop.
- **Input validation**: the program never crashes on what the user types. A menu choice other than 1 to 5 gets a message and the menu again. A task number that is not a whole number, or that no task has, gets a message, and nothing changes.

Input validation means checking what the user typed before the program uses it, and answering bad input with a helpful message instead of a crash or a wrong result.

Here is part of a run of the finished program, so you can see the goal. The user adds two tasks, lists them, completes task 2, and quits:

```
To-do list manager

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Buy milk
Added: Buy milk

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Finish homework
Added: Finish homework

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [ ] Buy milk
2. [ ] Finish homework

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: 2
Completed: Finish homework

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 5
Goodbye!
```

Your messages do not have to match word for word. Your program does have to do everything in the brief.

KEY IDEA

Read the whole brief before you write any code, and come back to it after each milestone. A program is finished when it does everything the brief asks, not when it stops showing errors.

12.2 Planning the Pieces

A brief lists *what* the program does. Before typing, turn each line of it into a piece of Python you already know. Nothing below is new; each row names the day that taught it, so you know where to look if you get stuck.

The brief says...	The Python piece	Taught on
Keep showing the menu until Quit	`while True:` with `break` when the choice is `"5"`	Day 5
Read a choice, a title, or a number	`input(...).strip()`	Day 3
Answer each menu choice	An `if` / `elif` / `else` chain on the choice	Day 4
Store the tasks	A list, `tasks = []`, created once, before the loop	Day 6
One task	A dictionary, `{"title": "Buy milk", "done": False}`	Day 8
Show tasks numbered from 1	`for number, task in enumerate(tasks, 1):`	Day 5
`[x]` or `[ ]`	A conditional expression: `"x" if task["done"] else " "`	Day 4
Mark a task done	`task["done"] = True`	Day 8
Remove a task	`tasks.pop(index)`	Day 6
Each action in its own function	`def`, parameters, `return`, docstrings	Day 9
Is this text a whole number?	`all(ch in "0123456789" for ch in text)`	Day 11

One plan decision matters more than the rest: **who owns the list?** `main()` creates `tasks` once, before the loop, and passes it to every function that needs it. The functions change the list they are given (lists are mutable, Day 6), so `main()` sees each change. Do not create `tasks = []` inside the loop: that would empty the list on every pass.

The second decision is **who talks to the user**. In this plan, only `main()` calls `input()`. Each action function receives what it needs as arguments and does one job with the data. `list_tasks` prints, because printing is its job; `complete_task` and `delete_task` **return** the task they changed, or `None` when no task has that number, and `main()` decides what message to print. Day 9 called this returning a value versus printing one, and it pays off on Day 20, when you write tests for these functions.

KEY IDEA

Create the list once in main() and pass it to each function. Let main() do the asking, and let each action function do one job with the data it is given.

12.3 Milestone 1: The Menu Loop

Goal: a program that shows the menu, reads a choice, and keeps going until the user types 5. Choices 1 to 4 can print `Not built yet.` for now.

Start with the loop because every other piece runs inside it. Write two functions:

- `show_menu()` prints a blank line and the menu. It takes no arguments and returns nothing.
- `main()` creates the empty `tasks` list, prints a title once, then loops: show the menu, read a choice, and answer it. Choice `"5"` says goodbye and `break`s. Anything other than 1 to 5 gets `Please choose a number from 1 to 5.`

Finish the file with a line that calls `main()`. Without it, Python defines the functions and then reaches the end of the file without running any of them, so the program prints nothing at all.

Here is one way milestone 1 can look. Compare it with yours after you have written your own:

todo_start.py

```python
def show_menu():
    """Print the menu of choices."""
    print()
    print("1) Add  2) List  3) Complete  4) Delete  5) Quit")


def main():
    """Run the menu loop until the user chooses Quit."""
    tasks = []
    print("To-do list manager")
    while True:
        show_menu()
        choice = input("Choose 1-5: ").strip()
        if choice in ("1", "2", "3", "4"):
            print("Not built yet.")
        elif choice == "5":
            print("Goodbye!")
            break
        else:
            print("Please choose a number from 1 to 5.")


main()
```

This run types `2`, then `9`, then `5`:

```
To-do list manager

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
Not built yet.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 9
Please choose a number from 1 to 5.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 5
Goodbye!
```

- The choice stays a **string**. Comparing it with `"5"` needs no `int()`, so a user who types `hello` cannot crash this part of the program: `"hello"` simply matches no rung and reaches `else`.
- `choice in ("1", "2", "3", "4")` checks whether the choice is one of the four strings in the tuple (Day 7).
- `.strip()` means `" 5 "` still counts as `5`.

Test it: type each of 1, 2, 3, 4 (each prints `Not built yet.`), then `9`, `0`, `hello`, and an empty line (each gets the "choose" message), then `5`.

> **WATCH OUT**
>
> **Comparing the choice with a number.** `if choice == 5:` is never `True`, because `choice` is the string `"5"` and a string never equals an int (Day 4). The loop then never ends, and you have to stop it with Ctrl+C. Compare with `"5"`, in quotes.

12.4 Milestone 2: Adding and Listing Tasks

Goal: choices 1 and 2 work. You can add tasks and see them.

Write two functions, then replace the `Not built yet.` for choices 1 and 2 with calls to them:

- `add_task(tasks, title)` appends a new dictionary to `tasks` with the given title and `"done"` set to `False`. It returns nothing. One line in its body is enough.
- `list_tasks(tasks)` prints `No tasks yet.` if the list is empty. Otherwise it prints one line per task: its number from 1, its check box, and its title, such as `1. [ ] Buy milk`.

In `main()`, choice 1 reads a title with `input("Task title: ").strip()`. If the title is an empty string (falsy, Day 4), print `A task needs a title.` and add nothing. Otherwise call `add_task` and print `Added:` and the title.

Hints:

- `enumerate(tasks, 1)` gives `(1, first task)`, `(2, second task)`, and so on. Day 5 used it on a string; it works the same way on a list.
- Inside the loop, `task` is one dictionary, so `task["title"]` is its title and `task["done"]` is its bool.
- `if not tasks:` is `True` for an empty list, because an empty list is falsy, like an empty string.

Test it: list first (you should see `No tasks yet.`), add three tasks, try an empty title, then list again. You should see three tasks numbered 1, 2, 3, all with `[ ]`.

> **WATCH OUT**
>
> **Building a new list inside the loop.** If `tasks = []` sits inside `while True:`, every pass starts with an empty list. Add works and prints `Added: Buy milk`, but the next List says `No tasks yet.` The list must be created once, above the loop, so that the same list object survives from pass to pass.

12.5 Milestone 3: Completing a Task by Number

Goal: choice 3 marks a task as done.

The user sees tasks numbered from 1, but list indexes start at 0 (Day 6). The task the user calls number 2 is at index 1. Every function that takes a `number` from the user must subtract 1 before indexing, and **only** there: keep `number` as the user's number everywhere else, so that messages such as `There is no task 9.` match what the user typed.

Write `complete_task(tasks, number)`:

- If no task has that number, return `None` and change nothing. A valid number is at least 1 and at most `len(tasks)`; the chained comparison `1 <= number <= len(tasks)` (Day 4) checks both ends at once.
- Otherwise get the task at index `number - 1`, set its `"done"` to `True`, and return the task.

In `main()`, choice 3 reads the number with `input("Task number: ").strip()` and converts it with `int()`. (For now, type only whole numbers; milestone 5 handles words.) Call `complete_task`. If it returns `None`, print `There is no task` and the number. Otherwise print `Completed:` and the task's title.

Why return the task instead of printing inside `complete_task`? Because `main()` needs to know whether it worked in order to choose a message, and the returned value carries that answer: a dictionary means "done", `None` means "no such task". Check it with `is None` (Day 4).

Test it: add three tasks, complete task 2, and list: only the middle task shows `[x]`. Then try task numbers 9 and 0. Both must print `There is no task…` and leave the list unchanged.

PAUSE AND TRY

Question 1 Multiple choice

`tasks` holds three tasks, and the user types `2` at the `Task number:` prompt, so `number` is `2`. Which expression is the task the user means?

A. `tasks[number]`
B. `tasks[number - 1]`
C. `tasks["2"]`
D. `tasks[-number]`

Question 2 Debugging

A classmate's `complete_task` checks only that the number is not too big. This test calls it with task number 0, which no task has, so it should print `None`. Explain what it prints instead, why, and fix it.

complete_bug.py

```
def complete_task(tasks, number):
    """Mark task number (counting from 1) as done."""
    if number > len(tasks):
        return None
    task = tasks[number - 1]
    task["done"] = True
    return task


tasks = [
    {"title": "Buy milk", "done": False},
    {"title": "Walk the dog", "done": False},
    {"title": "Read chapter 4", "done": False},
]
task = complete_task(tasks, 0)
print(task)
```

12.6 Milestone 4: Deleting a Task

Goal: choice 4 removes a task.

`delete_task(tasks, number)` follows the same pattern as `complete_task`: if no task has that number, return `None`; otherwise remove the task at index `number - 1` and return it. `pop(index)` (Day 6) does both jobs in one call, since it removes the item and gives it back.

In `main()`, choice 4 reads a number exactly as choice 3 does, calls `delete_task`, and prints `Deleted:` and the title, or `There is no task` and the number.

After a delete, every task after the removed one moves up one place, so its number changes. That is correct behavior, and it is why the user should list again before using a number.

Test it: add three tasks, delete task 1, and list: the other two tasks are now numbered 1 and 2. Delete task 5; nothing changes.

Choices 3 and 4 now share most of their steps: read a number, check it, call a function, print a message from the result. If your two branches look almost identical, that is fine for today. The reference solution shows one way to share the steps.

PAUSE AND TRY

Question 3 Tracing

What does this program print?

delete_renumber.py

```
tasks = [
    {"title": "Buy milk", "done": True},
    {"title": "Walk the dog", "done": False},
    {"title": "Read chapter 4", "done": False},
]
removed = tasks.pop(2 - 1)
print("Deleted:", removed["title"])
for number, task in enumerate(tasks, 1):
    mark = "x" if task["done"] else " "
    title = task["title"]
    print(f"{number}. [{mark}] {title}")
```

12.7 Milestone 5: Guarding Against Bad Input

Goal: the program never crashes on a task number, whatever the user types.

Milestone 3 assumed the user types a whole number. Here is what `int()` does with a word:

int_crash.py

```
text = "two"
number = int(text)
print("Task", number)
```

```
Traceback (most recent call last):
  File ".../day12/int_crash.py", line 2, in <module>
    number = int(text)
ValueError: invalid literal for int() with base 10: 'two'
```

The whole program stops, and every task the user added is lost. (Older Python versions may word the message differently.) Day 15 teaches a way to catch this error after it happens. With what you know today, you can prevent it: check the text **before** calling `int()`, and call `int()` only on text made of digits.

A string is a whole number (0 or more) when every one of its characters is a digit. "Every one" is a job for `all()` with a generator expression (Day 11):

```
>>> text = "12"
>>> all(ch in "0123456789" for ch in text)
True
>>> text = "-1"
>>> all(ch in "0123456789" for ch in text)
False
>>> text = "two"
>>> all(ch in "0123456789" for ch in text)
False
>>> text = ""
>>> all(ch in "0123456789" for ch in text)
True
>>> not text or not all(ch in "0123456789" for ch in text)
True
```

- `"12"`: both characters are in `"0123456789"`, so `all` gives `True`.
- `"-1"` and `"two"`: at least one character is not a digit, so `all` gives `False`. Negative numbers are refused here, which is fine, since no task has a negative number.
- `""`: there are no characters to check, so none of them fails, and `all` gives `True`. An empty string must be refused separately, which is what `not text` does: an empty string is falsy (Day 4).

So the full test for bad text is

```
if not text or not all(ch in "0123456789" for ch in text):
    print("Please type a task number, such as 1.")
```

Because `or` short-circuits (Day 4), `all(…)` is not even worked out when `text` is empty. When the text passes, `int(text)` cannot fail. A number that passes but is out of range, such as `0` or `9`, is then caught by the `1 <= number <= len(tasks)` check inside `complete_task` and `delete_task`. Two checks, two different jobs: the first asks "is this a number at all?", the second asks "does a task have this number?"

Test it: for both Complete and Delete, type `two`, `-1`, `3.5`, and an empty line. Each must print the "Please type a task number" message, and List must show the tasks unchanged.

PAUSE AND TRY

Question 4 Tracing

For each typed line below, say whether the program in *Putting it together* prints `Please type a task number, such as 1.`, or goes on to call `int()`, and if so, with what result. Remember that the input is stripped first.

`" 3 "`, `"3.0"`, `"03"`

Question 5 Short answer

A classmate's guard is `if not all(ch in "0123456789" for ch in text):`, with no `not text or` in front. Which typed input gets past this guard and crashes the program, and why?

12.8 Milestone 6: Testing Every Path

Goal: prove your program meets the brief by trying every menu option and every kind of bad input.

Programmers call each route a program can take through its `if` chains a **path**. A program can look finished while one path, such as deleting from an empty list, still crashes. The only way to know is to walk down every path on purpose. Run your program and type this session, in order. Write down anything that does not match the "Should happen" column.

Type	Should happen
`2`	`No tasks yet.`
`1`, then `Buy milk`	`Added: Buy milk`
`1`, then `Finish homework`	`Added: Finish homework`
`1`, then `Call Grandma`	`Added: Call Grandma`
`1`, then an empty line	`A task needs a title.`; nothing added
`2`	Three tasks, numbered 1 to 3, all `[ ]`
`3`, then `2`	`Completed: Finish homework`
`3`, then `9`	`There is no task 9.`
`3`, then `two`	`Please type a task number, such as 1.`
`7`	`Please choose a number from 1 to 5.`
`add`	`Please choose a number from 1 to 5.`
`4`, then `1`	`Deleted: Buy milk`
`4`, then `0`	`There is no task 0.`
`4`, then `-1`	`Please type a task number, such as 1.`
`2`	`1. [x] Finish homework` and `2. [ ] Call Grandma`
`5`	`Goodbye!` and the program ends

Also try one path the table leaves out: start the program and choose Complete before adding anything. Every number should get `There is no task…`, since `len(tasks)` is 0 and no number satisfies `1 <= number <= 0`.

When every row matches, your project is complete. Compare it with the reference solution below. Different names, messages, or structure are fine, as long as every path works.

KEY IDEA

Test every menu option and every kind of bad input on purpose, including the empty line and the empty list. A path you never tried is a path you do not know works.

Putting it together

Here is the complete reference solution. It is one correct way, not the only one.

todo.py

```
def show_menu():
    """Print the menu of choices."""
    print()
    print("1) Add  2) List  3) Complete  4) Delete  5) Quit")


def add_task(tasks, title):
    """Add a new, unfinished task to the end of tasks."""
    tasks.append({"title": title, "done": False})


def list_tasks(tasks):
    """Print every task with its number and a check box."""
    if not tasks:
        print("No tasks yet.")
        return
    for number, task in enumerate(tasks, 1):
        mark = "x" if task["done"] else " "
        title = task["title"]
        print(f"{number}. [{mark}] {title}")


def complete_task(tasks, number):
    """Mark task number (counting from 1) as done.

    Return the task, or None if no task has that number.
    """
    if not 1 <= number <= len(tasks):
        return None
    task = tasks[number - 1]
    task["done"] = True
    return task


def delete_task(tasks, number):
    """Remove task number (counting from 1) from tasks.

    Return the removed task, or None if no task has that number.
    """
```

```python
    if not 1 <= number <= len(tasks):
        return None
    return tasks.pop(number - 1)


def main():
    """Run the menu loop until the user chooses Quit."""
    tasks = []
    print("To-do list manager")
    while True:
        show_menu()
        choice = input("Choose 1-5: ").strip()
        if choice == "1":
            title = input("Task title: ").strip()
            if title:
                add_task(tasks, title)
                print(f"Added: {title}")
            else:
                print("A task needs a title.")
        elif choice == "2":
            list_tasks(tasks)
        elif choice in ("3", "4"):
            text = input("Task number: ").strip()
            if not text or not all(ch in "0123456789" for ch in text):
                print("Please type a task number, such as 1.")
                continue
            number = int(text)
            if choice == "3":
                task = complete_task(tasks, number)
                action = "Completed"
            else:
                task = delete_task(tasks, number)
                action = "Deleted"
            if task is None:
                print(f"There is no task {number}.")
            else:
                title = task["title"]
                print(f"{action}: {title}")
        elif choice == "5":
            print("Goodbye!")
            break
        else:
            print("Please choose a number from 1 to 5.")


main()
```

This run types exactly the session from the *12.8* table:

```
To-do list manager

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
No tasks yet.
```

```
1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Buy milk
Added: Buy milk

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Finish homework
Added: Finish homework

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Call Grandma
Added: Call Grandma

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title:
A task needs a title.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [ ] Buy milk
2. [ ] Finish homework
3. [ ] Call Grandma

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: 2
Completed: Finish homework

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: 9
There is no task 9.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: two
Please type a task number, such as 1.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 7
Please choose a number from 1 to 5.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: add
Please choose a number from 1 to 5.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 4
```

```
Task number: 1
Deleted: Buy milk

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 4
Task number: 0
There is no task 0.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 4
Task number: -1
Please type a task number, such as 1.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [x] Finish homework
2. [ ] Call Grandma

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 5
Goodbye!
```

How the pieces fit:

Lines	What they do	Milestone
1–4	`show_menu()` prints a blank line and the one-line menu	12.3
7–9	`add_task` appends one dictionary with `"done": False`	12.4
12–20	`list_tasks` handles the empty list, then numbers each task from 1 with `enumerate`	12.4
23–32	`complete_task` refuses an out-of-range number, otherwise sets `"done"` and returns the task	12.5
35–42	`delete_task` refuses an out-of-range number, otherwise `pop`s and returns the task	12.6
47	`main()` creates the list once, before the loop	12.2
52–58	Choice 1 refuses an empty title	12.4
61–65	Choices 3 and 4 share one number check; `continue` skips back to the menu on bad text	12.7
67–72	The only difference between Complete and Delete: which function to call and which word to print	12.5, 12.6
73–77	`is None` turns the returned value into the right message	12.5
85	Calls `main()`, which starts everything	12.3

- `continue` (Day 5) ends this pass early when the text is not a number. The loop goes back to the top and shows the menu again.
- No function except `main()` calls `input()`, and no function except `main()` creates the list. On Day 13 you move the five action functions into their own file, and this split is what makes that move easy.

Common mistakes

Mistake	What happens	Fix
`if choice == 5:`	Never `True`; the loop never ends	Compare with the string `"5"`
No `main()` call at the bottom	The program prints nothing	Add `main()` as the last line
`tasks = []` inside the loop	Every pass starts with an empty list	Create the list once, before `while True:`
`tasks[number]`	The wrong task changes; the last task gives an `IndexError`	Use `tasks[number - 1]`
Checking only `number > len(tasks)`	`0` becomes index `-1` and changes the last task	Check both ends: `1 <= number <= len(tasks)`
`int(text)` with no check first	A word or an empty line stops the program with a `ValueError`	Check `not text or not all(…)` first
Printing inside `complete_task` and in `main()` too	Messages appear twice, or `main()` cannot tell whether it worked	Return the task or `None`, and let `main()` print

Go further

Day 12 has no new reading. If one of the milestones sent you back to an earlier day, the matching Python Tutorial sections are a good second explanation: Data Structures: More on Lists, Dictionaries, and Defining Functions.

Self-check

Check each statement. If the answer is "not yet," reread the linked milestone and the day it names. Then write one or two sentences in your notes file: which of Days 1 to 11 felt shaky while you built this? Use the review block at the start of the next two days to revisit that day.

- [ ] I can turn each line of a project brief into a piece of Python I know. Reread: 12.2 Planning the Pieces
- [ ] I can write a menu loop that keeps going until the user chooses Quit. Reread: 12.3 Milestone 1: The Menu Loop
- [ ] I can store tasks as a list of dictionaries, add to it, and list it with numbers from 1. Reread: 12.4 Milestone 2: Adding and Listing Tasks
- [ ] I can turn a number the user typed into the right list index and check that it is in range. Reread: 12.5 Milestone 3: Completing a Task by Number
- [ ] I can return a value or None from a function and let the caller choose the message. Reread: 12.5 Milestone 3: Completing a Task by Number
- [ ] I can remove a task with pop and explain why the numbers change. Reread: 12.6 Milestone 4: Deleting a Task
- [ ] I can check typed text before int() so that bad input never crashes the program. Reread: 12.7 Milestone 5: Guarding Against Bad Input
- [ ] I can test every menu option and every kind of bad input on purpose. Reread: 12.8 Milestone 6: Testing Every Path