Lab 10 — The Winter Count

Dictionaries: ask, change, loop, count; counting the looks, and the slow line

DURATION ~3 hours **COVERS** Lectures 17 & 18 **TASKS** 8 files **CHECKER** `check.py`

How this lab works

- Every section opens with a worked example. It is a file in your folder, `example1.py` to `example5.py`: open it in Thonny, run it, and see the run printed on the sheet.
- Work top to bottom. Read a task, then its five keys. Four are practical: type a line in Thonny's Shell, or change one thing in the example file and run it again, and look at what it shows. The fifth is worked out by hand, Shell closed, with the small numbers it gives.
- Type the five answers into `check.py` when it asks. All five right, and the task's file opens. A wrong answer costs nothing: type the line again and look.
- Tasks are the deliverable: `task01.py` … `task08.py` in your `lab10` folder. Aim to pass them all. A failed check costs nothing: read its reason and try again.
- Run `check.py` as often as you like.
- Ask. Talk to your neighbour, the TA or the instructor about the idea. Type your own code: no copy-paste.
- The minutes on a task are a guide for pacing, not a limit.
- This lab assumes Lab 09 and the labs before it: a loop that reads n lines, `a, b = line.split()`, `try` and `except`, a bisection search with `low` and `high`, and a function that returns its answer.

Before you start: get the files

Download the `lab10` folder (from the course LMS page) onto your machine. It contains `task01.py` … `task08.py`, the worked examples `example1.py` … `example5.py`, and `check.py`. Keep everything in that **one folder**, and do not rename any file — the checker finds your work by its file name. The example files are yours to change; nothing in them is handed in.

The story of this lab

Every winter, thousands of water birds fly south from Siberia to the lakes of Pakistan. Their path along the Indus River is called the Indus Flyway. Each January, counters at lakes such as Haleji and Keenjhar count the birds, species by species. Ringers catch some birds, put a small numbered ring on one leg, and let them go. A ring read again years later tells how long that bird has lived. All the code in this lab is about that work.

1 The Field Guide (~23 min)

Ref: Lecture 17 — Dictionaries

A dictionary stores pairs: a key, and the value it looks up. Here the key is a bird's name. `d[key]` gives its value, and `key in d` asks if the key is there. A key that is not there stops the program with a `KeyError`.

Worked example: wingspans

A field guide gives the wingspan of each species: how wide its open wings are, tip to tip. This program adds one species to the guide, then looks up the bird you type. It is `example1.py` in your folder: open it, run it, type `flamingo`, and see this run.

```
wingspan = {"teal": 58, "coot": 75, "pelican": 300}
wingspan["flamingo"] = 150
bird = input()
if bird in wingspan:
    print("Wingspan:", wingspan[bird], "cm")
else:
    print("Unknown")
print("Species:", len(wingspan))
```

Input

```
flamingo
```

Output

```
Wingspan: 150 cm
Species: 4
```

Notice: `bird in wingspan` is asked first, so `wingspan[bird]` only runs for a bird that is there. A bird that is not there takes the `else` part. Key 1.2 tries a bird with a capital letter.

Four things check.py is strict about

- Read each line with `input()` and nothing inside the brackets, as in the CCC problems. A prompt such as `input("n: ")` is printed too, so the task fails.
- A key's answer is what the Shell showed, typed as it showed it: a number, `True`, `False`, an error's name such as `KeyError`, or the one word a `print` showed. No units, no quotes. A by-hand key's answer has the same shape.
- In Tasks 6 and 7, `check.py` also calls your functions by themselves, as you can in the Shell after F5. So keep the exact names and parameters the task gives, hand the answer back with `return`, and print nothing inside a function.
- Every task has 10 seconds. In Task 8 that is the test: a slow program shows as `TIMEOUT`.

TASK 1 Weigh the Net

EASY `task01.py` ~11 min

⌨ Ringers catch birds in a soft net, and let them go after. The starter holds a second guide, `weight`: the usual weight of five species, in grams. The team wants the total weight of the birds in the net, from the guide. A bird that is not in the guide is counted as unknown.

Your task: Read the birds. Add up the guide's weight of every bird in the guide, and count the birds that are not in it. Print the total, then the count.

Input: Line 1: n, the number of birds in the net, 1 or more. Then n lines: one species name per line, in small letters.

Output: `Total grams: t`, the weights added up (0 if no bird is known); then `Unknown: u`, how many birds are not in the guide.

Sample 1 (the checker uses this one)

Input

```
4
teal
pelican
heron
teal
```

Output

```
Total grams: 9700
Unknown: 1
```

Why: The guide gives teal 350 and pelican 9000: 350 + 9000 + 350 is 9700. A heron is not in the guide.

Hint: start from the worked example's `if bird in ...` test. Set the total and the count to 0 above the loop. Without the test, the heron stops the program (Key 1.3).

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task01.py` opens.

Key 1.1 Run `example1.py` and type `pelican`. What number is on the `Wingspan:` line?
Answer: .....

Key 1.2 Run it again and type `Pelican`, with a capital P. What word is printed first?
Answer: .....

Key 1.3 Type `{"coot": 800}["heron"]`. The last line starts with an error's name. Which?
Answer: .....

Key 1.4 Type `800 in {"coot": 800}`. What came back?
Answer: .....

Key 1.5 **By hand, Shell closed.** The guide gives teal 350 g and coot 800 g. The net holds coot, heron, teal, coot. What number is on the `Total grams:` line?
Answer: .....

TASK 2 Morning and Evening

EASY `task02.py` ~12 min

In the morning the counters saw three species on the lake. Through the day more birds land, one at a time. `task02.py` starts the evening count from the morning count and adds 1 for every bird that lands. A species seen for the first time is added too. It prints the evening count, then the morning total, which must not change. It runs, but it prints the wrong numbers.

Your task: It has exactly **two** bugs. Find and fix them. Do not rewrite the program.

Input: Line 1: n, the number of birds that land, 1 or more. Then n lines: one species name per line.

Output: one line per species, its name and evening count: the morning's species first, then new ones in the order they first landed; then `Morning total: m`.

Sample 1 (the checker uses this one)

Input

```
5
teal
flamingo
teal
coot
flamingo
```

Output

```
teal 42
coot 26
mallard 12
flamingo 2
Morning total: 77
```

Why: Two teal land (40 to 42) and one coot (25 to 26). Two flamingos land: a new species. The morning had 40 + 25 + 12 = 77.

Hint: find the bugs by **asserting invariants**. An invariant is a fact that must stay true all the way through. Here the morning total is always 77, and the evening total grows by exactly 1 with each bird. An `assert` line for each (Keys 2.2 and 2.4) shows which one breaks, and where. Key 2.3 is the fact behind the first bug.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task02.py` opens.

Key 2.1 ⌨ Open `task02.py`, run it as it is and type the sample's six lines. What number is on the `Morning total:` line?

Answer: .....

Key 2.2 ⌨ Put `assert sum(morning.values()) == 77, i` as the last line of the loop's body: four spaces in, under the `else` part. Run it on the sample. The last line says `AssertionError:` and a number. Which number?

Answer: .....

Key 2.3 ⌨ Type `a = {"teal": 40}`, then `b = a`, then `b["teal"] = 99`, then `a["teal"]`. What came back?

Answer: .....

Key 2.4 ⌨ Change that `assert` line to `assert sum(evening.values()) == 78 + i, bird` and run it again. Which word comes after `AssertionError:`?

Answer: .....

Key 2.5 ✎ **By hand, Shell closed.** The morning holds `{"teal": 3}`. Then a heron, a teal and a heron land. In a correct program, what number follows `heron`?

Answer: .....

2 The Ring Book (~11 min)

Ref: Lecture 17 — Dictionaries

`d[key] = value` adds a pair, or changes the value of a key that is already there: a key appears only once. `del d[key]` takes a pair out. Both `del` and `d[key]` stop with a `KeyError` for a missing key, so ask with `in` first, or use `try`.

Worked example: three changes and a lookup

A ring book maps each ring's code to the species wearing it. This program adds a ring, changes one, deletes one, and then looks up the code you type. It is `example2.py`: run it and type `K20`.

```
book = {"K12": "teal", "K15": "coot"}
book["K20"] = "crane"
book["K15"] = "mallard"
del book["K12"]
ring = input()
try:
    print(ring, book[ring])
except KeyError:
    print(ring, "unknown")
print("Rings:", len(book))
```

Input

```
K20
```

Output

```
K20 crane
Rings: 2
```

Notice: `book["K15"] = "mallard"` changes K15; it does not add a second K15. `book[ring]` sits inside `try`, so a missing ring prints `unknown`. Key 3.3 deletes a ring that is not there.

TASK 3 The Ring Book

EASY `task03.py` ~11 min

⌨ The team sends one order per line. `add K27 teal` puts ring K27 in the book, on a teal. `find K27` prints the ring and its species. `remove K27` takes the ring out: it came off the bird. `task03.py` does all of it but one gap, marked `# TODO`: the remove order. A remove for a ring that is not in the book does nothing.

Your task: Replace the `pass` under `# TODO` with two lines: if the ring is in the book, delete it. Change nothing else.

Input: Line 1: n, the number of orders, 1 or more. Then n lines, one order each: `add`, a code and a species; or `find` or `remove`, and a code.

Output: one line per find order: the code and its species, or the code and `unknown`; then `Rings: r`, the rings in the book at the end.

Sample 1 (the checker uses this one)

Input

```
6
add K27 teal
add K31 crane
find K27
remove K27
remove K99
find K27
```

Output

```
K27 teal
K27 unknown
Rings: 1
```

Why: K27 is found, then removed, so the second find prints unknown. K99 was never added: removing it does nothing. Only K31 is left.

Hint: `if ring in book:` and, under it, `del`. Without the `if`, `remove K99` stops the program (Key 3.3).

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task03.py` opens.

Key 3.1 ⌨ Run `example2.py` and type `K15`. Which species is printed?

Answer: .....

Key 3.2 ⌨ Run it again and type `K12`. What word follows `K12`?

Answer: .....

Key 3.3 ⌨ Change `del book["K12"]` to `del book["K99"]` and run it again. The last line starts with an error's name. Which?

Answer: .....

Key 3.4 ⌨ Type `d = {"K1": "teal"}`, then `d["K1"] = "coot"`, then `len(d)`. What came back?

Answer: .....

Key 3.5 ✎ **By hand, Shell closed.** The book starts empty. The orders are `add K1 teal`, `add K2 coot`, `remove K1`, `remove K7`. What number is on the `Rings:` line?

Answer: .....

3 The Census (~27 min)

Ref: Lecture 17 — Dictionaries

`for key in d` goes through the keys, in the order they were added. `for key, value in d.items()` gives each key with its value. To count with a dictionary, a key seen for the first time is set, and a key already there is added to.

Worked example: the big flocks

A census dictionary holds each species and how many were counted. This program prints every species with 100 birds or more, and the total. It is `example3.py`: run it; nothing is typed.

```
1 census = {"teal": 320, "coot": 95, "flamingo": 140, "pelican": 12}
2 total = 0
3 for bird, count in census.items():
4     total = total + count
5     if count >= 100:
6         print(bird, count)
7 print("Total:", total)
```

Output

```
teal 320
flamingo 140
Total: 567
```

Notice: each pass of the loop gets one species in `bird` and its number in `count`, in the order the census lists them. Key 4.2 changes the 100.

TASK 4 Census Tally

EASY `task04.py` ~12 min

⌨ On census day, counters stand at different points around the lake. Each one sends lines such as `teal 40`: a species, and how many birds of it they saw. The same species can come in many lines. The lake's total for a species is all its numbers added up.

Your task: Read the lines and add up each species' total in a dictionary. Print each species and its total, in the order the species first came in. Then print how many species there are, and the largest total.

Input: Line 1: n, the number of lines that follow, 1 or more. Then n lines: a species name, a space, and a whole number.

Output: one line per species, its name and total; then `Species: s`, and `Largest: g`, the largest total.

Sample 1 (the checker uses this one)

Input

```
5
teal 40
coot 25
teal 15
flamingo 120
coot 5
```

Output

```
teal 55
coot 30
flamingo 120
Species: 3
Largest: 120
```

Why: Teal is 40 + 15 = 55 and coot 25 + 5 = 30. Flamingo comes once: 120, which is also the largest.

Hint: read a line with `bird, count = input().split()`, then `int(count)`. If `bird` is already in the dictionary, add to its total; if not, its total is this count (Key 4.3). Then loop over `items()`, as in the worked example.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task04.py` opens.

Key 4.1 ⌨ Run `example3.py`. What number is on the `Total:` line?

Answer: .....

Key 4.2 ⌨ Change `count >= 100` to `count >= 95` and run it again. How many lines are printed above `Total:`?

Answer: .....

Key 4.3 ⌨ Type `d = {}`, then `d["coot"] = d["coot"] + 5`. The last line starts with an error's name. Which?

Answer: .....

Key 4.4 ⌨ Type `max({"teal": 55, "coot": 30}.values())`. What came back?

Answer: .....

Key 4.5 ✎ **By hand, Shell closed.** The lines are `coot 10`, `teal 4`, `coot 6`. What number follows `coot` in the output?

Answer: .....

TASK 5 Since Last Year

MEDIUM task05.py ~15 min

⌨ The same census lines as Task 4. The starter holds last year's totals in a dictionary, `last_year`. The team wants to know how this year compares, species by species: how much each one changed, which species are new, and which are gone.

Your task: Compare the two years:

Step 1: Add up this year's totals as in Task 4 (you may copy those lines). Print the dictionary and run it on the sample: you should see `{'teal': 55, 'coot': 30, 'flamingo': 120}`.

Step 2: Take that print out. Loop over this year's totals. For each species, print its name and this year's total minus last year's. If last year has no such species, print its name and `new`. Run it: you should see the first three lines of the sample.

Step 3: Loop over `last_year`. Print each species that this year does not have, and `gone`. Run it: you should see the sample.

Input: Line 1: n, the number of lines that follow, 1 or more. Then n lines: a species name, a space, and a whole number.

Output: one line per species this year, in the order it first came in: its name and the change, or its name and `new`; then one line per species gone, in last year's order: its name and `gone`.

Sample 1 (the checker uses this one)

Input

```
5
teal 40
coot 25
teal 15
flamingo 120
coot 5
```

Output

```
teal -15
coot 0
flamingo new
pelican gone
mallard gone
```

Why: Last year had teal 70, coot 30, pelican 8 and mallard 22. Teal fell from 70 to 55, and coot stayed at 30. Pelican and mallard were not seen.

Hint: two loops after the reading, one over each dictionary. In each, ask `in` or `not in` about the *other* one (Keys 5.1 and 5.3).

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task05.py` opens.

Key 5.1 ⌨ Type `"flamingo" in {"teal": 70, "coot": 30}`. What came back?

Answer: .....

Key 5.2 ⌨ Type `for b in {"teal": 70, "coot": 30}: print(b)` and press Enter twice. What is the last word printed?

Answer: .....

Key 5.3 ⌨ Type `"pelican" not in {"teal": 55}`. What came back?

Answer: .....

Key 5.4 ⌨ Type `last = {"teal": 70}`, then `55 - last["teal"]`. What came back?

Answer: .....

Key 5.5 ✎ **By hand, Shell closed.** Last year had crane 9 and coot 4. This year's lines are `coot 6` and `heron 2`. How many lines does the program print?

Answer: .....

4 Count the Looks (~27 min)

Ref: Lecture 18 — Counting Steps, Big-O

Count the steps, not the seconds. In a search the step that counts is a look: one ring checked. A search from the front is $O(n)$: twice the rings, twice the looks. Halving a sorted list is $O(\log n)$: twice the rings, one look more. In this section the ring book is a sorted list of ring numbers.

Worked example: looks from the front

This program makes a book of 8 ring numbers, 10 to 80, and searches it from the front for the ring you type. It counts every ring it checks. It is `example4.py`: run it and type 50.

```
book = list(range(10, 90, 10))
ring = int(input())
looks = 0
for r in book:
    looks = looks + 1
    if r == ring:
        break
print("Rings in the book:", len(book))
print("Looks:", looks)
```

Input

```
50
```

Output

```
Rings in the book: 8
Looks: 5
```

Notice: 50 is the fifth ring, so the loop stops after 5 looks. A ring that is not in the book is the worst case: every ring is looked at. Key 6.2 doubles the book.

TASK 6 Count the Looks

EASY `task06.py` ~12 min

`ring in book` hides this same loop: Python checks the rings one by one from the front, until it finds the one it wants. The team wants to see how many looks each search takes.

Part 1, the function: `looks(book, ring)` returns how many looks a search from the front takes. It stops at the first ring equal to `ring`. If no ring is, it has looked at all of them. It prints nothing. In the Shell, `looks([12, 25, 31, 40], 31)` gives 3, and `looks([12, 25, 31, 40], 50)` gives 4.

Part 2, the program: Read the book and the rings to find. For each ring, print the ring and its looks. Then print the total.

Input: Line 1: the book, ring numbers from small to large, with a space between them. Line 2: m, how many rings to find, 1 or more. Then m lines, a ring number each.
Output: one line per ring, the ring and its looks; then `Total looks: t`.

Sample 1 (the checker uses this one)

Input

```
12 25 31 40 47 58 63 71
3
40
71
50
```

Output

```
40 4
71 8
50 8
Total looks: 20
```

Why: 40 is the fourth ring. 71 is the last: 8 looks. 50 is not in the book, so all 8 rings are looked at.

Hint: read the book as in Lab 08: `book = [int(x) for x in input().split()]`. Inside the function, `return` the count as soon as a ring matches, and return it again after the loop. An empty book gives 0 (Key 6.4).

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task06.py` opens.

Key 6.1 Run `example4.py` and type `85`. What number is on the `Looks:` line?

Answer: .....

Key 6.2 Change `range(10, 90, 10)` to `range(10, 170, 10)`, twice as many rings, and run it on `85` again. What number is on the `Looks:` line now?

Answer: .....

Key 6.3 Type `[12, 25, 31, 40].index(31)`. What came back?

Answer: .....

Key 6.4 Type `len([])`. What came back?

Answer: .....

Key 6.5 **By hand, Shell closed.** The book is `12 25 31`. The rings to find are 25, then 99. What number is on the `Total looks:` line?

Answer: .....

TASK 7 Halve the Book

MEDIUM `task07.py` ~15 min

The book is sorted, small to large. So a search can halve it, as a bisection search does: look at the middle ring, then keep only the half that can hold the ring. Each middle ring checked is one look.

Part 1, the new function: Start from your `task06.py`: copy it into this file, then change it. Keep `looks` as it is. `halving_looks(book, ring)` returns how many looks a halving search takes. It prints nothing. In the Shell, `halving_looks([10, 20, 30, 40, 50, 60, 70], 40)` gives `1`, and with 60 in place of 40 it gives `2`.

Part 2, the program: The same input as Task 6. For each ring, print the ring, its looks and its halving looks. Then print both totals.

Step 1: Write `halving_looks`. Set `low = 0` and `high = len(book) - 1`. While `low <= high`: set `mid = (low + high) // 2` and count one look. If `book[mid] == ring`, return the looks. If `book[mid] < ring`, set `low = mid + 1`; otherwise `high = mid - 1`. After the loop, return the looks. Run the file, then type the two calls above in the Shell: you should see 1, then 2.

Step 2: Add the third number to each ring's line. Run it on the sample: you should see `40 4 1`, `71`

`8 4` and `50 8 3`.

Step 3: Replace the `Total looks:` line with the two totals. Run it: you should see the sample.

Input: the same as Task 6.

Output: one line per ring: the ring, its looks and its halving looks; then `Linear: a` and `Halving: b`, the two totals.

Sample 1 (the checker uses this one)

Input

```
12 25 31 40 47 58 63 71
3
40
71
50
```

Output

```
40 4 1
71 8 4
50 8 3
Linear: 20
Halving: 8
```

Why: For 71, halving looks at 40, 58, 63 and 71: 4 looks. For 50 it looks at 40, 58 and 47; then low passes high, so 50 is not there.

Hint: the loop runs while `low <= high`: with one ring left, `low` and `high` are equal, and that ring still needs a look (Key 7.4).

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task07.py` opens.

Key 7.1 ⌨ Type `(0 + 7) // 2`. What came back?

Answer: .....

Key 7.2 ⌨ Type `[12, 25, 31, 40, 47, 58, 63, 71][3] < 71`. What came back?

Answer: .....

Key 7.3 ⌨ Type `(4 + 7) // 2`. What came back?

Answer: .....

Key 7.4 ⌨ Type `low = 4`, then `high = 4`, then `low <= high`. What came back?

Answer: .....

Key 7.5 ✎ **By hand, Shell closed.** The book is `10 20 30 40 50 60 70`. How many looks does `halving_looks` take to find 30?

Answer: .....

5 The Slow Line (~15 min)

Ref: Lecture 18 — Counting Steps, Big-O

`x in L` and `L.index(x)` each hide a loop over the list: n steps. Put one inside a loop over n items, and the program is $O(n^2)$: twice the data, four times the steps. A dictionary answers `x in d` and `d[x]` in one step, $O(1)$. So build it once, then ask it.

Worked example: two lists become a dictionary

The ringers keep two lists in the same order: ring numbers, and the year each ring was put on. This program builds a dictionary from ring to year with one loop, then asks it. It is `example5.py`: run it and type 717.

```python
rings = [703, 710, 717, 724]
years = [2019, 2021, 2021, 2024]
year_of = {}
for i in range(len(rings)):
    year_of[rings[i]] = years[i]
ring = int(input())
if ring in year_of:
    print("Ringed in", year_of[ring])
else:
    print("Not ours")
```

Input
```
717
```

Output
```
Ringed in 2021
```

Notice: `rings[i]` and `years[i]` belong together, so one loop over `i` pairs them up. After that, no list is searched: `ring in year_of` is one step. Key 8.2 types a year instead of a ring.

TASK 8 The Slow Line

MEDIUM `task08.py` ~15 min

Each winter the team reads the rings on the birds it catches. A caught bird whose ring is in the lists was caught again. Its age is 2026 minus the year it was ringed. The real data is not ready, so `task08.py` makes up n rings and n catches with simple number rules, to test its speed. It gives the right answers. But with $n = 300000$ it does not finish in 10 seconds: two of its lines hide a loop over all the rings.

Your task: Make it fast. Do not change what it prints.

Step 1: Run it and type 1000: you should see `Caught again: 333` and `Oldest: 26 years`. Run it again and type 300000. It does not finish: press Stop, or Ctrl + c.

Step 2: Below the `# TODO`, build a dictionary `year_of` from ring to year, with one loop over the two lists, as in the worked example. Run it on 1000: nothing changes yet.

Step 3: Change `if c in rings:` and `year = years[rings.index(c)]` to ask `year_of`. Run it on 300000: it finishes in under a second, with the sample.

Input: one line: n, how many rings and catches to make up, 0 or more.

Output: `Caught again: k`, how many caught birds were in the lists; then `Oldest: a years`, the oldest of them (0 if none).

Sample 1 (the checker uses this one)

Input
```
300000
```

Output
```
Caught again: 100000
Oldest: 26 years
```

Why: The rings are the odd numbers 1, 3, 5, ... and the catches 0, 3, 6, 9, ...: one catch in three is an odd number in the lists. The first rings were put on in 2000.

Hint: fix both lines. With only `c in rings` fixed, `rings.index(c)` still searches the list for each of 100,000 caught birds, and the program is still too slow.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task08.py` opens.

Key 8.1 Run `example5.py` and type `724`. What number is printed?

Answer: .....

Key 8.2 Run it again and type `2021`. What is the first word printed?

Answer: .....

Key 8.3 Type `r = [703, 710]`, then `y = [2019, 2021]`, then `y[r.index(710)]`. What came back?

Answer: .....

Key 8.4 Type `300000 * 300000`. What came back? It is about how many looks the slow program makes.

Answer: .....

Key 8.5 **By hand, Shell closed.** With $n = 1000$, the slow program makes up to 1000×1000 looks. With $n = 2000$, how many times as many looks is that?

Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

- ☐ I can make a dictionary, look a value up with `d[key]`, and ask `key in d` first.
- ☐ I know that `in` asks about the keys, never the values, and that keys are exact.
- ☐ I can add, change and delete a pair, and I know which of them stop with a `KeyError`.
- ☐ I can catch a `KeyError` with `try` and `except`.
- ☐ I know that `b = a` gives one dictionary two names, and that `a.copy()` makes a new one.
- ☐ I can find a bug by asserting an invariant: a fact that must stay true all the way through.
- ☐ I can count with a dictionary: a new key is set, an old one is added to.
- ☐ I can loop over a dictionary's keys, or over its `items()`, and compare two dictionaries.
- ☐ I can count the looks of a search from the front, and of a halving search.
- ☐ I can say what happens to the steps when the data doubles: twice for $O(n)$, one more for $O(\log n)$, four times for $O(n^2)$.
- ☐ I can spot a hidden loop (`x in L`, `L.index(x)`) inside a loop, and replace it with a dictionary built once.