Lab 11 — The Codebreakers

Selection sort, the sieve, counters; Caesar's shift, mixed alphabets, counting letters

DURATION ~3 hours **COVERS** Lectures 19 & 20 **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 `lab11` 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.
- Tasks 1 to 4 meet every idea of the two lectures, once. Tasks 5 to 8 are a second round: the same ideas, one step further. Do 1 to 4 first.
- This lab assumes Lab 10 and the labs before it: reading a line of numbers with `[int(x) for x in input().split()]`, counting in a dictionary (a new key is set, an old one is added to), a function that returns its answer, and an `assert` line as a probe.

Before you start: get the files

Download the `lab11` 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

In World War II, armies sent their orders by radio, in secret code. Anyone could hear the radio, but nobody could read it without the key. At Bletchley Park in England, thousands of people worked day and night to break these codes. Alan Turing, one of the first computer scientists, worked there. All the code in this lab is about their work.

1 Sort and Sieve (~23 min)

Ref: Lecture 19 — Sorting

Selection sort finds the smallest item and swaps it to the front. Then it does the same for the rest of the list, one place further on. A list of n items takes $n-1$ passes. `a, b = b, a` swaps two values in one line. The sieve finds every prime up to `top` at once, and a list of counters counts how often each value comes.

Worked example: sort the times

Listeners write down the time of each message they hear, as a number: 905 means 9:05, and 1340 means 13:40. This program sorts five times, earliest first, and prints the list after each pass. It is `example1.py` in your folder: open it and run it; nothing is typed.

```
1 times = [1520, 905, 1340, 1105, 830]
2 n = len(times)
3 for i in range(n - 1):
4     smallest = i
5     for j in range(i + 1, n):
6         if times[j] < times[smallest]:
7             smallest = j
8     times[i], times[smallest] = times[smallest], times[i]
9     print(times)
```

Output

```
[830, 905, 1340, 1105, 1520]
[830, 905, 1340, 1105, 1520]
[830, 905, 1105, 1340, 1520]
[830, 905, 1105, 1340, 1520]
```

Notice: each pass starts with `smallest = i`: the first item of what is left. The inner loop finds the smallest of the rest, and one line swaps it to place `i`. A pass can change nothing, as passes 2 and 4 do. Key 1.2 changes the `<`.

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`, `None`, an error's name such as `IndexError`, or the one word a `print` showed. No quotes. A by-hand key's answer has the same shape.
- In Task 1, print a list with `print(L)`: the checker wants it exactly as Python shows it, such as `[90, 60, 40]`.
- In Tasks 3 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.

TASK 1 Strongest First

EASY `task01.py` ~11 min

A listener gives each radio signal a strength, from 1 to 99. The hut reads the strongest signals first. So the list must be sorted the other way: largest first.

Your task: Read the strengths. Sort them strongest first with selection sort: find the largest of what is left, and swap it to the front of what is left. Print the list after each pass. Do not use `sorted()` or `.sort()`.

Input: Line 1: the strengths, with a space between them, 2 or more.

Output: the list after each pass, one line per pass, as `print` shows a list. A list of n strengths takes $n - 1$ passes.

Sample 1 (the checker uses this one)

Input

```
20 60 40 90 75
```

Output

```
[90, 60, 40, 20, 75]
[90, 75, 40, 20, 60]
[90, 75, 60, 20, 40]
[90, 75, 60, 40, 20]
```

Why: Pass 1 swaps 90 with 20, the first item. Pass 2 finds 75 in the rest and swaps it with 60, and so on.

Hint: start from the worked example. Read the list as in Lab 08: `strengths = [int(x) for x in input().split()]`. Then change one test, as Key 1.2 does. The name `largest` reads better than `smallest` here.

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`. How many lines does it print?

Answer: .....

Key 1.2 Change `times[j] < times[smallest]` to `times[j] > times[smallest]` and run it again. What is the last number on the last line now?

Answer: .....

Key 1.3 Type `L = [5, 9]`, then `L[0], L[1] = L[1], L[0]`, then `L[0]`. What came back?

Answer: .....

Key 1.4 Type `print([52, 17, 88].sort())`. What came back?

Answer: .....

Key 1.5 By hand, Shell closed. The strengths are `20 50 30`. After the first pass of strongest-first selection sort, what is the second number in the list?

Answer: .....

TASK 2 Prime Endings

EASY `task02.py` ~12 min

The key maker needs prime numbers. A prime has exactly two divisors: 1 and itself. `task02.py` makes a sieve up to `top`. Then it counts the primes by their last digit, in a list of 10 counters, one for each digit 0 to 9. It prints each digit that ends a prime, with its count, and then how many primes there are. It runs, but it prints the wrong lines.

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

Input: Line 1: `top`, a whole number, 2 or more.

Output: one line for each last digit that ends a prime, in digit order: the digit and its count; then `Primes: p`, how many primes there are up to `top`.

Sample 1 (the checker uses this one)

Input

```
50
```

Output

```
1 3
2 1
3 4
5 1
7 4
9 2
Primes: 15
```

Why: The 15 primes up to 50 end in 1 (11, 31, 41), 2 (2), 3 (3, 13, 23, 43), 5 (5), 7 (7, 17, 37, 47) or 9 (19, 29).

Hint: find the bugs by **asserting invariants**. An invariant is a fact that must stay true all the way through. Here a prime is never crossed out by its own loop: after it, `flags[p]` is still `True`. And every last digit has a counter: `k % 10` is less than `len(counts)`. An `assert` line for each (Keys 2.2 and 2.4) shows which one breaks, and where.

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 50. What number is on the `Primes:` line?

Answer: .....

Key 2.2 Put `assert flags[p], p` as the last line of the `if flags[p]:` part: eight spaces in, under the inner `for` line, not inside its loop. Run it on 50. The last line says `AssertionError:` and a number. Which number?

Answer: .....

Key 2.3 Type `list(range(3, 13, 3))[0]`. What came back?

Answer: .....

Key 2.4 Fix the line Key 2.3 is about. Then put `assert k % 10 < len(counts), k` just above the `counts[k % 10] = ...` line, at the same indent, and run it again. Which number comes after `AssertionError:`?

Answer: .....

Key 2.5 **By hand, Shell closed.** In a correct program, `top` is 20. What number follows `3` in the output?

Answer: .....

2 Caesar's Shift (~15 min)

Ref: Lecture 20 — Codes & Codebreakers

A letter's place is `ABC.index(ch)`: 0 for A, up to 25 for Z. Caesar's code moves every letter k places along. `% 26` goes round the wheel: after Z comes A again. To unlock, move each letter k places back. With only 26 keys, a codebreaker can try them all.

Worked example: lock one word

This program locks a word: it moves every letter k places along. It is `example2.py`: run it, type `TURING`, then 3.

```
ABC = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
word = input()
k = int(input())
out = ""
for ch in word:
    i = ABC.index(ch)
    out = out + ABC[(i + k) % 26]
print(out)
```

Input

```
TURING
3
```

Output

```
WXULQJ
```

Notice: `out` starts empty, and each pass adds one letter to it. `(i + k) % 26` keeps the place between 0 and 25. Key 3.2 types the key 26.

TASK 3 Try Every Key

MEDIUM `task03.py` ~15 min

⌨ Every morning the enemy sent a weather report, and it always held the word `WEATHER`. So the hut can try all 26 keys on a secret. The right key is the one whose message holds `WEATHER`. To unlock with a key, every capital letter moves k places back; before A comes Z. A space stays as it is.

Part 1, the function: `unlock(secret, k)` returns the secret with every capital letter moved k places back. It prints nothing. In the Shell, `unlock("ZHDWKHU", 3)` gives `'WEATHER'`, and `unlock("A", 1)` gives `'Z'`.

Part 2, the program: Read the secret. Try the keys 0, 1, 2, ... 25 in turn. At the first key whose message holds `WEATHER`, print the key and the message, and stop. If no key does, print `No key`.

Step 1: Write `unlock`: the worked example's loop inside a function, with `if ch in ABC` first and `(i - k) % 26`. Run the file, then type `unlock("ZHDWKHU", 3)` in the Shell: you should see `'WEATHER'`.

Step 2: Read the secret. Loop `for k in range(26):`. When `"WEATHER" in` the message, print the two lines and `break`. Run it on the sample: you should see it.

Step 3: Set `found = False` above the loop and `True` where you print. After the loop, print `No key` if it is still `False`. Run it on `HELLO`: you should see `No key`.

Input: Line 1: the secret: capital letters and spaces.

Output: `Key: k`, then the message; or one line, `No key`, when no key gives `WEATHER`.

Sample 1 (the checker uses this one)

Input

```
DLHAOLY JSLHY HA KHDU
```

Output

```
Key: 7
WEATHER CLEAR AT DAWN
```

Why: Keys 0 to 6 give messages without WEATHER. Key 7 moves D back to W, L to E, H to A: WEATHER.

Hint: without `if ch in ABC`, the space stops the program (Key 3.3). Moving back past A goes round to Z (Key 3.5).

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 `ZOO`, then 1. What word is printed?

Answer: .....

Key 3.2 ⌨ Run it again and type `HUT`, then 26. What word is printed?

Answer: .....

Key 3.3 ⌨ Run it again, type `HI THERE`, then 1. The last line starts with an error's name. Which?

Answer: .....

Key 3.4 ⌨ Type `"WEATHER" in "weather"`. What came back?

Answer: .....

Key 3.5 ✎ **By hand, Shell closed.** `unlock("B", 3)` returns one letter. Which?

Answer: .....

3 Al-Kindi Counts (~16 min)

Ref: Lecture 20 — Codes & Codebreakers

A mixed alphabet is a key of the 26 letters in a new order: each letter always becomes the same other letter. There are far too many such keys to try. But a long English message has more E's than any other letter, then T's, then A's. So count the letters of a secret in a dictionary, and sort them most common first. `sorted(counts, key=lambda ch: counts[ch], reverse=True)` gives the letters, most common first.

Worked example: count the letters

This program counts the letters of a short text, then sorts them by their counts. It prints the most common letter and the number of different letters. It is `example3.py`: run it; nothing is typed.

```
text = "KEEP THE SECRET"
counts = {}
for ch in text:
    if ch != " ":
        if ch in counts:
            counts[ch] = counts[ch] + 1
        else:
            counts[ch] = 1
ranked = sorted(counts, key=lambda ch: counts[ch], reverse=True)
print(ranked[0], counts[ranked[0]])
print("Different letters:", len(counts))
```

Output

```
E 5
Different letters: 8
```

Notice: `sorted` on a dictionary sorts its keys, the letters. `key=lambda ch: counts[ch]` sorts them by their counts, and `reverse=True` puts the largest first. Key 4.2 takes the `reverse` away.

TASK 4 Count the Letters

MEDIUM `task04.py` ~16 min

The hut has a secret locked with a mixed alphabet. Its most common letter most likely stands for E, the second for T, and the third for A. The starter holds `GUESS = "ETA"`. So the letter at place `i` of the top three becomes the letter at place `i` of `GUESS`: a tiny mixed alphabet, run backwards.

Your task: Break the secret, as far as counting goes:

Step 1: Read the secret. Count each capital letter in a dictionary, `counts`, as the worked example does. Print `len(counts)` and run it on the sample: you should see 13.

Step 2: Take that print out. Sort the letters most common first, and print the top three, each with its count. Run it: you should see the first three lines of the sample.

Step 3: Print the secret again: each of the top three letters becomes its guess, every other capital letter becomes `-`, and a space stays a space. Run it: you should see the sample.

Input: Line 1: the secret: capital letters and spaces, with at least three different letters.

Output: three lines, the three most common letters, each with its count (two letters with the same count keep the order they first came in: `sorted` keeps it); then the secret with the top three letters guessed.

Sample 1 (the checker uses this one)

Input

```
JDS SGSTL XVSSJ KQCJU QJ JDS EQJS
```

Output

```
S 7
J 6
Q 3
T-E E-E-- --EET -A-T- AT T-E -ATE
```

Why: S comes 7 times, J 6 and Q 3. So S becomes E, J becomes T and Q becomes A. The first word, JDS, is T-E: most likely THE.

Hint: `top = ranked[:3]`. For a letter in `top`, `GUESS[top.index(ch)]` is its guess (Key 4.3). Ask `if ch in top` first, then `elif ch in ABC`.

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 follows `E`?

Answer: .....

Key 4.2 ⌨ Change `reverse=True` to `reverse=False` and run it again. Which letter is printed first now?

Answer: .....

Key 4.3 ⌨ Type `"ETA"["SJQ".index("Q")]`. What came back?

Answer: .....

Key 4.4 ⌨ Type `"X" in ["S", "J", "Q"]`. What came back?

Answer: .....

Key 4.5 ✎ **By hand, Shell closed.** A secret's top three letters are W, R and B, in that order. Which letter does R stand for?

Answer: .....

4 Second Round: Sieve and Counters (~21 min)

Ref: Lecture 19 — Sorting

Round two: the same ideas, one step further. Here you write the sieve yourself, and then the counting. A list with a place for every number from 0 to `top` needs `top + 1` places: `flags[k]` says if k is prime, and `counts[h]` says how many times h came.

Worked example: the sieve up to 30

A prime has exactly two divisors: 1 and itself. The sieve starts with every number marked `True`. For each number still `True`, it crosses out its multiples. Then it answers the number you type. It is `example4.py`: run it and type 29.

```
top = 30
flags = [True] * (top + 1)
flags[0] = False
flags[1] = False
for p in range(2, top + 1):
    if flags[p]:
        for m in range(2 * p, top + 1, p):
            flags[m] = False
k = int(input())
print(k, flags[k])
print([x for x in range(top + 1) if flags[x]])
```

Input

```
29
```

Output

```
29 True
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Notice: the list has `top + 1` places, 0 to 30, so `flags[30]` exists. Crossing out starts at `2 * p`, never at `p` itself. Key 5.4 makes `top` 1000.

TASK 5 Prime or Not

EASY `task05.py` ~11 min

A key maker checks many numbers, up to 1000: is each one prime? 7 is prime, but 91 is not (7×13), and 1 is not. Testing every number by dividing is slow. So the program makes the sieve once, and then each answer is one look.

Your task: Make a sieve up to 1000, once. Then read the numbers, and print for each one if it is prime. Then print how many were prime.

Input: Line 1: m, how many numbers follow, 1 or more. Then m lines: one whole number per line, from 0 to 1000.

Output: one line per number: the number and `prime`, or the number and `not prime`; then `Primes: c`, how many of them were prime.

Sample 1 (the checker uses this one)

Input

```
6
97
91
1
2
997
1000
```

Output

```
97 prime
91 not prime
1 not prime
2 prime
997 prime
1000 not prime
Primes: 3
```

Why: 97, 2 and 997 have no divisor but 1 and themselves. 91 is 7 x 13, 1000 is 2 x 500, and 1 is never prime.

Hint: copy the sieve from the worked example, with `top = 1000`, above the reading. Without the `+ 1`, the number 1000 has no place (Key 5.3). Task 2's sieve is the same.

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 ⌨ Run `example4.py` and type 27. Which word follows 27?

Answer: .....

Key 5.2 ⌨ Change `top = 30` to `top = 100` and run it again on 29. What is the last number on the second line?

Answer: .....

Key 5.3 ⌨ Type `flags = [True] * 1000`, then `flags[1000]`. The last line starts with an error's name. Which?

Answer: .....

Key 5.4 ⌨ Change `top = 100` to `top = 1000`, run it again and type `997`. Which word follows `997`?

Answer: .....

Key 5.5 ✎ **By hand, Shell closed.** A sieve goes up to 10. Which number does `p = 3` cross out that `p = 2` did not?

Answer: .....

TASK 6 Messages by the Hour

EASY `task06.py` ~10 min

⌨ Every message heard has the hour it was heard: a whole number from 0 (midnight) to 23 (11 at night). The hut wants a chart: how many messages came in each hour, in hour order, and the busiest hour. `task06.py` does all of it but one gap, marked `# TODO`: the counting.

Your task: Replace the `# TODO` line with three lines. Make a list of 24 counters, one for each hour from 0 to 23, all 0, and name it `counts`. Then add 1 to the counter of every hour in the list `hours`. Change nothing else.

Input: Line 1: the hours, with a space between them, 1 or more.

Output: one line for each hour that had a message, in hour order: the hour and its count; then `Busiest: b`, the hour with the most (the earliest, if two tie).

Sample 1 (the checker uses this one)

Input

```
14 3 23 14 0 3 14 23
```

Output

```
0 1
3 2
14 3
23 2
Busiest: 14
```

Why: Hour 14 came three times, hours 3 and 23 twice, and hour 0 once. No other hour came.

Hint: `counts = [0] * 24`, then a loop over `hours` that adds 1 to `counts[h]`. With 23 counters, hour 23 has no place (Key 6.2).

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 Type `c = [0] * 24`, then `c[23]`. What came back?

Answer: .....

Key 6.2 Type `c = [0] * 23`, then `c[23] = c[23] + 1`. The last line starts with an error's name. Which?

Answer: .....

Key 6.3 Type `c = [0, 0, 0]`, then `c[2] = c[2] + 1` twice, then `c[2]`. What came back?

Answer: .....

Key 6.4 Type `3 > 3`. What came back?

Answer: .....

Key 6.5 **By hand, Shell closed.** The hours are `5 9 5 9 2`. What number follows `Busiest:`?

Answer: .....

5 Second Round: Codes (~27 min)

Ref: Lecture 20 — Codes & Codebreakers

Round two for the codes. A full mixed alphabet has all 26 letters in a new order: a letter at place `i` of `ABC` becomes the letter at place `i` of `KEY`. And counting breaks Caesar's shift in one step: every letter moved by the same k, so the top letter alone gives the key.

Worked example: lock with a mixed key

This key is the letters of a keyboard, read down its columns. The program locks the message you type. It is `example5.py`: run it and type `HUT SIX`.

```
ABC = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
KEY = "QAZWSXEDCRFVTGBYHNUJMIKOLP"
message = input()
out = ""
for ch in message:
    if ch in ABC:
        out = out + KEY[ABC.index(ch)]
    else:
        out = out + ch
print(out)
```

Input

```
HUT SIX
```

Output

```
DMJ UCO
```

Notice: `ABC.index(ch)` finds the letter's place, and `KEY[...]` reads the letter at the same place in the key. Key 7.2 turns the line round, to unlock.

TASK 7 The Keyword Alphabet

MEDIUM task07.py ~15 min

⌨ A mixed alphabet of 26 letters is hard to remember. So code makers built it from a keyword. The key is the keyword's letters, each one only once, then the rest of the alphabet in order. For `BLETCHLEY` the key starts `BLETCHY`: the second L and the second E are skipped. Then come A, D, F, G and the rest, up to Z.

Part 1, the function: `make_key(word)` returns the 26-letter key for that keyword. It prints nothing. In the Shell, `make_key("ZEBRA")` gives `'ZEBRACDFGHIJKLMNOPQSTUVWXY'`.

Part 2, the program: Read the keyword and the message. Print the key, then the message locked with it.

Step 1: Write `make_key`. Start with `key = ""`. Loop over every letter of `word + ABC`. Add a letter to the key only if it is `not in` the key yet. Return the key. Run the file, then type `make_key("ZEBRA")` in the Shell: you should see the key above.

Step 2: Read the keyword and print `Key:` and its key. Run it on the sample: you should see the first line.

Step 3: Read the message and lock it with that key, as the worked example does with `KEY`. Run it: you should see the sample.

Input: Line 1: the keyword: capital letters, no spaces. Line 2: the message: capital letters and spaces.

Output: `Key:` and the 26 letters; then the locked message, a space staying a space.

Sample 1 (the checker uses this one)

Input

```
BLETCHLEY
MEET AT HUT SIX
```

Output

```
Key: BLETCHYADFGIJKMNOPQRSUVWXZ
JCCR BR ASR QDW
```

Why: M is place 12 of ABC, and place 12 of the key is J. E (place 4) becomes C, and T (place 19) becomes R.

Hint: `for ch in word + ABC:` goes through the keyword and then the whole alphabet. Without the `not in` test the key is too long (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 ⌨ Run `example5.py` and type `ABC`. What word is printed?

Answer: .....

Key 7.2 ⌨ Change `KEY[ABC.index(ch)]` to `ABC[KEY.index(ch)]`, run it again and type `QAZ`. What word is printed?

Answer: .....

Key 7.3 ⌨ Type `"L" not in "BLE"`. What came back?

Answer: .....

Key 7.4 ⌨ Type `len("ZEBRA" + "ABCDEFGHIJKLMNOPQRSTUVWXYZ")`. What came back?

Answer: .....

Key 7.5 ✎ **By hand, Shell closed.** `make_key("CAB")` starts with C, A and B. Which letter comes next?

Answer: .....

TASK 8 Break the Shift

EASY `task08.py` ~12 min

⌨ This long secret was locked with Caesar's code, and its key is lost. The most common letter of the secret stands for E. E is place 4, so the key is the top letter's place minus 4, `% 26`.

Your task: Start from your `task04.py`: copy it into this file, then change it. Keep the counting and the three top lines, and take out the guess line. Print `Key:` and the key the top letter gives. Then copy `unlock` from your `task03.py`, and print the secret unlocked with that key.

Input: Line 1: the secret: capital letters and spaces, with at least three different letters.

Output: three lines, the three most common letters, each with its count; then `Key: k`; then the message, unlocked with key k.

Sample 1 (the checker uses this one)

Input

```
COFOX CRSZC COOX XOKB DRO MYKCD
```

Output

```
O 6
C 5
X 3
Key: 10
SEVEN SHIPS SEEN NEAR THE COAST
```

Why: O comes 6 times, the most. O is place 14, and 14 - 4 is 10. Each letter 10 places back gives the message.

Hint: the key is
`(ABC.index(ranked[0]) - ABC.index("E")) % 26`.
It goes round when the top letter comes before E (Key 8.3).

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 `task08.py` opens.

Key 8.1 ⌨ Type `"ABCDEFGHIJKLMNOPQRSTUVWXYZ".index("E")`. What came back?
Answer: .....

Key 8.2 ⌨ Type `("ABCDEFGHIJKLMNOPQRSTUVWXYZ".index("H") - 4) % 26`. What came back?
Answer: .....

Key 8.3 ⌨ Type `("ABCDEFGHIJKLMNOPQRSTUVWXYZ".index("B") - 4) % 26`. What came back?
Answer: .....

Key 8.4 ⌨ Run `example2.py` and type `SEVEN`, then 10. What word is printed?
Answer: .....

Key 8.5 ✎ **By hand, Shell closed.** The most common letter of a secret is `G`. What is the key?
Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

☐ I can sort a list with selection sort, smallest first or largest first, and say what the list looks like after each pass.

- ☐ I can swap two places of a list in one line, and I know why two lines lose a value.
- ☐ I know that `.sort()` gives back `None`, and that `sorted()` gives a new list.
- ☐ I can find a bug by asserting an invariant: a fact that must stay true all the way through.
- ☐ I can make a sieve, and then say at once if a number is prime.
- ☐ I know that a list for the numbers 0 to `top` needs `top + 1` places.
- ☐ I can count into a list of counters, one for each value.
- ☐ I can lock and unlock a message with Caesar's shift, and I know how `% 26` goes round the wheel.
- ☐ I can try every key, and stop at the first one that works.
- ☐ I can lock a message with a mixed alphabet, and build the key from a keyword.
- ☐ I can count the letters of a secret, sort them most common first with `key=`, and turn the top letter into a key.