Lab 05 — Doing It Again

Iteration: `while`, `for`, `range`

DURATION ~3 hours **COVERS** Lectures 7 & 8 **TASKS** 12 files **CHECKER** `check.py`

How this lab works

- Work top to bottom. Read a task, answer its keys, then write its code.
- ✎ means pen on this sheet, Shell closed. Only calculators are allowed. ⌨ means at the machine, in the task's file.
- Tasks are the deliverable: `task01.py` … `task12.py` in your `lab05` folder. Every task must PASS.
- Run `check.py` as often as you like.
- Type the code yourself. No copy-paste.
- This lab assumes Lab 04: `if/elif/else`, comparisons joined with `and` and `or`, `int()` on what `input()` gives, and f-strings.

Before you start: get the files

Download the `lab05` folder (from the course LMS page) onto your machine. It contains `task01.py` … `task12.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.

1 The `while` Loop: Set, Test, Change (~33 min)

Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

You will write `while` loops that set, test and change a loop variable. You will count their rounds, even when there are none. Task 2 is a loop that never stops.

Two 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. If a loop never stops, press **Ctrl + c** in the Shell. `check.py` stops it after 10 seconds and marks it TIMEOUT.

TASK 1 Prime or Not

EASY `task01.py` ~7 min

A **divisor** of a number divides it with no remainder. For example, 3 is a divisor of 15, because `15 % 3` is 0. Every number has 1 and itself as divisors. A **prime** number has no other divisor: 7 is prime, but 15 is not, because 3 and 5 also divide it. The **smallest divisor** of n (not counting 1) is the first of 2, 3, 4, ... that divides n. If that is n itself, then n is prime.

Your task: Read a number n. Print its smallest divisor, then say whether n is prime.

Input: one line: a whole number n, 2 or more.

Output: two lines.
Line 1: `Smallest divisor: D`, where `D` is the smallest divisor of n.
Line 2: `Prime` if `D` equals n, otherwise `Not prime`.

Sample 1 (the checker uses this one)

Input
```
15
```

Output
```
Smallest divisor: 3
Not prime
```

Why: 2 does not divide 15, but 3 does. 3 is not 15, so 15 is not prime.

Sample 2

Input
```
7
```

Output
```
Smallest divisor: 7
Prime
```

Why: None of 2, 3, 4, 5, 6 divides 7. The search reaches 7 itself, so 7 is prime.

Hint: use a `while` loop and a variable `d` for the divisor. Give `d` its three jobs: set it to 2 before the loop, test `n % d` in the `while` line, and add 1 to it in the body. When the loop stops, `d` is the smallest divisor. Compare it with `n` to choose the second line.

Pen on this sheet, Shell closed, before you open `task01.py`. Only calculators are allowed.

Key 1.1 The only even prime number is:
(a) 0
(b) 2
(c) 3
(d) 4

Answer: .....

Key 1.2 The user inputs 33. Predict the number the program should print after `Smallest divisor:`.

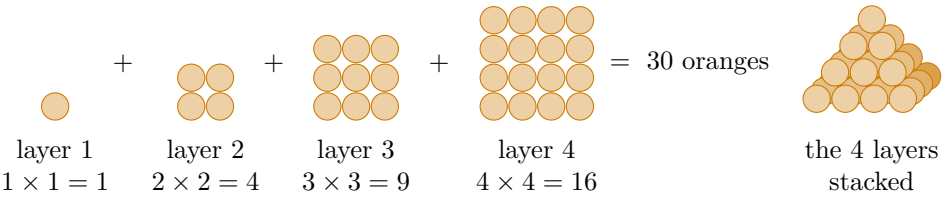
Answer: .....

Key 1.3 The user inputs 25. Predict the two lines the program should print.
(a) `Smallest divisor: 25`, then `Prime`
(b) `Smallest divisor: 1`, then `Not prime`
(c) `Smallest divisor: 5`, then `Not prime`

TASK 2 Kepler's Oranges

EASY `task02.py` ~8 min

A fruit seller stacks oranges in a pyramid. The top layer holds 1 orange, the next 4, then 9, then 16: layer k holds $k \times k$ oranges. The astronomer Kepler studied this stacking in year 1611. `task02.py` should add up the layers and print the total for 4 layers: $1+4+9+16=30$. Right now it never stops.



layer 1 $1 \times 1 = 1$ | layer 2 $2 \times 2 = 4$ | layer 3 $3 \times 3 = 9$ | layer 4 $4 \times 4 = 16$ | the 4 layers stacked

Your task: It has exactly **two** bugs. Find and fix them, so it prints the total. Do not rewrite the program.

Input: none: the program reads nothing.

Output: one line: `4 layers: 30 oranges`.

Sample 1 (the checker uses this one)

Input	Output
nothing is typed	`4 layers: 30 oranges`

Why: The fixed program adds the four layers, $1+4+9+16$, and prints 30.

Hint: find the bugs with a **print probe**. Add `print(layer, total)` as the last line of the loop body. Run the file and read what it prints. Fix one bug at a time. Delete the probe before you run `check.py`.

Pen on this sheet, Shell closed, before you open `task02.py`. Only calculators are allowed.

Key 2.1 Predict how many times this code prints `x`.

```
1 while False:
2     print('x')
```

Answer: .....

Key 2.2 Predict how many times this code prints.

```
1 x = 1
2 while x < 5:
3     print(x)
```

(a) 4 times
(b) 5 times
(c) For ever: it never stops.

Answer: .....

Key 2.3 When does a `while` loop stop?
(a) As soon as its test becomes `False`, even in the middle of the body.
(b) When its test is `False` at the top of a round.
(c) When Python reaches the last line of its body.

Key 2.4 A loop adds numbers into `total`. Where does `total = 0` go?
(a) Before the loop.
(b) As the first line of the loop body.
(c) After the loop.

TASK 3 Hailstone Numbers

MEDIUM `task03.py` ~11 min

Start with a whole number n and apply one rule to it:

- if n is even *(divides by 2 with no remainder)*, halve it;
- if n is odd, multiply it by 3 and add 1.

Each use of a rule is one **step**. Repeat on the new number until you reach 1. In year 1937, Lothar Collatz guessed that every start reaches 1. Nobody has proved it yet.

Your task: Read the start n. Print each new number the rules give, one per line, down to 1, then the number of steps.

Input: one line: the start n, a whole number, $n \geq 1$

Output: one line for each step: the new number, as an integer (`3`, not `3.0`).
the last of these is `1`. The start itself is not printed.
Every last line: `Steps: S`, where `S` is the number of steps.

Sample 1 (the checker uses this one)

Input
```
6
```

Output
```
3
10
5
16
8
4
2
1
Steps: 8
```

Why: 6 is even, so 3. 3 is odd, so 10. The rules go on until 1: eight new numbers, so eight steps.

Sample 2

Input
```
1
```

Output
```
Steps: 0
```

Why: The start is already 1, so no rule is used. Nothing prints but the steps line.

Pen on this sheet, Shell closed, before you open `task03.py`. Only calculators are allowed.

Key 3.1 The user inputs 7. Predict the third line the program should print.
Answer: .....

Key 3.2 The `while` line should test `n != 1`, where `n` is the number. Why not `n > 0`?
(a) `n > 0` would stop the loop one number too early.
(b) A `while` line cannot compare a number with 0.
(c) After 1 come 4, 2 and 1 again, so `n` never reaches 0.

Key 3.3 A student halves with `n = n / 2`. The user inputs 10. Predict the first line their program prints.
(a) `5`
(b) `5.0`
(c) `10`

Key 3.4 The user inputs 8. Predict how many lines the program should print in all.

Answer: .....

Key 3.5 A student prints `n` at the start of each round, before changing it. The user inputs 3. Predict the first line their program prints.

Answer: .....

2 Reading Inside the Loop (~23 min)

Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

Now `input()` sits inside the loop, so each round reads one new line. In Task 4, a stop value ends the reading. In Task 5, a total ends it.

TASK 4 Geiger Counter

EASY `task04.py` ~7 min

⌨ A Geiger counter clicks each time radiation passes through it. A student writes down how many clicks it made in each minute. A minute is **high** when it has more than 30 clicks.

Your task: Read the clicks of each minute, up to the `-1`, and count the high minutes.

Input: one line per minute: its clicks, a whole number, 0 or more.
Last line: `-1`. It marks the end; it is not a minute. It can be the only line.

Output: one line: `High minutes: H`, where `H` is the number of high minutes.

Sample 1 (the checker uses this one)

Input
```
12
45
18
60
33
-1
```

Output
```
High minutes: 3
```

Why: 45, 60 and 33 are more than 30; 12 and 18 are not. The `-1` ends the reading and is not a minute.

Sample 2

Input
```
-1
```

Output
```
High minutes: 0
```

Why: The first line is already `-1`: no minutes at all, so the count stays 0.

Hint: structure is similar to the 'lost forest' we did in the class.

Pen on this sheet, Shell closed, before you open `task04.py`. Only calculators are allowed.

Key 4.1 The user inputs 40, 20, 35, then `-1`. Predict the number the program should print after `High minutes:`.

Answer: .....

Key 4.2 The user inputs 12, 45, 18, then `-1`. How many minutes did the student write down as input?

(a) 4
(b) 3
(c) 1

Key 4.3 The user inputs 31, 30, then `-1`. Predict the number the program should print after `High minutes:`.

Answer: .....

TASK 5 Edge of Space

MEDIUM `task05.py` ~11 min

Space begins 100 km above the ground, at the Kármán line. A test rocket starts on the ground, at height 0. Each minute it reports how many km it climbed in that minute. Its height is the sum of all the climbs so far.

Your task: Read the reports one at a time, and stop as soon as the height is 100 km or more. Print how many minutes that took, and the height then.

Input: one line per minute: the km climbed, a whole number. Nothing marks the end: read no more lines once the height reaches 100.

Output: two lines.
Line 1: `Minutes: M`, where `M` is the number of lines you read.
Line 2: `Height: H km`, where `H` is the height after the last one.

Sample 1 (the checker uses this one)

Input

```
30
25
35
20
```

Output

```
Minutes: 4
Height: 110 km
```

Why: The height goes 30, 55, 90, 110. It first reaches 100 after the 4th minute, so reading stops there.

Sample 2

Input

```
150
```

Output

```
Minutes: 1
Height: 150 km
```

Why: One climb of 150 already passes 100, so only one line is read.

Hint: the `while` line tests the height, not a stop value. Put `input()` inside the loop, so each round reads one minute.

Pen on this sheet, Shell closed, before you open `task05.py`. Only calculators are allowed.

Key 5.1 The user inputs 40, 45, 30 and 10. Predict the number the program should print after `Height:`

Answer: .....

Key 5.2 This code should add numbers until the total reaches 10. The user inputs 6, 2, 9 and 1. Predict what it prints.

```
1 total = 0
2 n = int(input())
3 while total < 10:
4     total = total + n
5 print(total)
```

Answer: .....

Key 5.3 Why does Task 5 need a `while` loop, and not an `if`?
(a) An `if` cannot have `input()` inside it.
(b) An `if` runs its body once at most, so it reads only one climb.
(c) An `if` runs for ever while its test is true.

Key 5.4 Which line can the program never print?
(a) `Height: 99 km`
(b) `Height: 100 km`
(c) `Height: 130 km`

Key 5.5 The user inputs 70, 30 and 20. Predict the number the program should print after `Minutes:`.

Answer: .....

3 Counting with `for` and `range` (~22 min)

Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

You will choose a `range`'s start, stop and step. Then you will keep a running product and a running total in one `for` loop. Before you run a loop, find the last value its `range` gives.

TASK 6 Cicada Years

EASY `task06.py` ~7 min

Some cicadas *(an exotic insect species)* live underground for exactly 17 years. Then millions of them come out together in one spring. So if a group came out in 2004, it comes out again in 2021, 2038, and so on.

Your task: Read the year a group came out and a last year. List every year the group comes out, from the first year up to the last.

Input: two lines.
Line 1: a year the cicadas came out.
Line 2: the last year to list. It is the same year or later.

Output: one line for each year the cicadas come out, from the first year up to the last year, oldest first: `Out in Y`. The first year is always on the list. The last year is on the list only if the cicadas come out that year.

Sample 1 (the checker uses this one)

Input
```
2004
2060
```

Output
```
Out in 2004
Out in 2021
Out in 2038
Out in 2055
```

Why: From 2004, add 17 again and again: 2021, 2038, 2055. The next, 2072, is past 2060.

Sample 2

Input
```
1970
1980
```

Output
```
Out in 1970
```

Why: The first year is always listed. The next, 1987, is past 1980.

Hint: you can use `for` loops here. Think about the step size?

Pen on this sheet, Shell closed, before you open `task06.py`. Only calculators are allowed.

Key 6.1 The user inputs 1990 and 2030. Predict the last year the program should print.

Answer: .....

Key 6.2 Sana says `range(5, 125, 5)` gives 120 last. Bilal says 125. Asma says 130. Who is right?

(a) Sana
(b) Bilal
(c) Asma

Key 6.3 The user inputs 1987 and 2021. Predict how many lines the program should print.

Answer: .....

TASK 7 Rice on the Chessboard

MEDIUM `task07.py` ~11 min

An old story says the inventor of chess asked a king for rice. He wanted 1 grain on the first square of the board, 2 on the second, 4 on the third: each square gets double the grains of the one before it.

Your task: Read n. Add up the grains on squares 1 to n, and print the total.

Input: one line: the number of squares n, from 1 to 64.

Output: one line: `Total grains: T`, where `T` is the grains on squares 1 to n added together.

Sample 1 (the checker uses this one)

Input	Output
6	`Total grains: 63`

Why: Squares 1 to 6 hold 1, 2, 4, 8, 16 and 32 grains. Their sum is 63.

Sample 2

Input	Output
1	`Total grains: 1`

Why: Only the first square: 1 grain.

Hint: keep two values. `grains` is the grains on the current square: a running product that starts at 1 and doubles in every round. `total` is a running total that starts at 0. In each round, add `grains` to `total`, then double `grains`.

Pen on this sheet, Shell closed, before you open `task07.py`. Only calculators are allowed.

Key 7.1 How many grains are on square 4 alone?

Answer: .....

Key 7.2 The user inputs 3. Predict the line the program should print.

(a) `Total grains: 4`
(b) `Total grains: 7`
(c) `Total grains: 15`

Key 7.3 The user inputs 4. Predict the number the program should print after `Total grains:`.

Answer: .....

Key 7.4 The program printed `Total grains: 31`. What number did the user input?

Answer: .....

Key 7.5 The totals for 1, 2, 3 and 4 squares are 1, 3, 7 and 15: each is one less than the grains on the next square. The user inputs 10. Predict the number the program should print after `Total grains:`.

Answer: .....

4 Walking Through Strings (~32 min)

Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

A `for` loop can visit a string one character at a time. With `range`, it visits the indexes instead, and can walk two strings at once. The first character is at index 0.

TASK 8 Clean the Read

EASY `task08.py` ~7 min

DNA is written with four letters: A, C, G and T. A DNA machine reads a strand and prints its letters in groups, with spaces or dashes between them. When it cannot read a letter, it prints N instead.

Your task: Clean the read: keep only the letters A, C, G and T, and drop every N, space and dash. Print the clean read, then how many letters it has.

Input: one line: the machine's read, made of the capital letters A, C, G, T and N, spaces and dashes.

Output: two lines.
Line 1: the A, C, G and T letters of the read, in the same order, with nothing between them. If there are none, this line is empty.
Line 2: `Letters:` `L`, where `L` is the number of letters on line 1.

Sample 1 (the checker uses this one)

Input
```
ACG TNA-GGT
```

Output
```
ACGTAGGT
Letters: 8
```

Why: The space, the N and the dash are dropped. The eight letters left, in order, are ACGTAGGT.

Sample 2

Input
```
GATTACA
```

Output
```
GATTACA
Letters: 7
```

Why: Nothing to drop: every character is one of A, C, G and T.

Hint: There are many ways to do this. What if you have an empty string by your side, you read original string character by character and keep on "adding" the valid letters in order in the new string!

Pen on this sheet, Shell closed, before you open `task08.py`. Only calculators are allowed.

Key 8.1 The user inputs `TA-NC`. Predict the first line the program should print.

Answer: .....

Key 8.2 `ch` is one character of the read. Which test is true only when `ch` is A, C, G or T?
(a) `if ch == "ACGT":`
(b) `if "ACGT" in ch:`
(c) `if ch in "ACGT":`

Key 8.3 The user inputs `NN-N`. Predict the number the program should print after `Letters:`.

Answer: .....

TASK 9 Point Mutations

EASY `task09.py` ~7 min

A point mutation changes one letter of a DNA strand. To find mutations, compare a strand before and after, place by place: the 1st letter with the 1st, the 2nd with the 2nd, and so on. In Sample 1 the letters differ at the 3rd, 6th and 7th places.

Your task: Read the strand before and the strand after. Count the places where their letters differ.

Input: two lines.
Line 1: the strand before, in capital letters.
Line 2: the strand after, with the same number of letters.

Output: one line: `Mutations: M`, where `M` is the number of places where the letters differ.

Sample 1 (the checker uses this one)

Input
```
GATTACA
GACTATT
```

Output
```
Mutations: 3
```

Why: Place by place: G=G, A=A, T≠C, T=T, A=A, C≠T, A≠T. Three places differ.

Sample 2

Input
```
ACGT
ACGT
```

Output
```
Mutations: 0
```

Why: The two strands are the same, so no place differs.

Hint: you probably know where this one is going...

Pen on this sheet, Shell closed, before you open `task09.py`. Only calculators are allowed.

Key 9.1 The user inputs `AGT`, then `ACA`. Predict the number the program should print after `Mutations:`.

Answer: .....

Key 9.2 Which line compares the letters at the same place?

(a) `if before != after:`
(b) `if before[i] != after[i + 1]:`
(c) `if before[i] != after[i]:`

Key 9.3 The user inputs `GTCA`, then `GTCC`: only the last letter changed. Predict the number the program should print after `Mutations:`.

Answer: .....

TASK 10 ISBN Check

MEDIUM `task10.py` ~11 min

⌨ Every book has a 13-digit number, its ISBN, printed above the barcode on the back. Its last digit is there to catch typing mistakes. To check an ISBN:

1. Multiply the 1st, 3rd, 5th, ..., 13th digits by 1.
2. Multiply the 2nd, 4th, 6th, ..., 12th digits by 3.
3. Add all 13 results. This is the total.

The ISBN is valid when the total ends in 0 (divides by 10 with no remainder). For Sample 1 the total starts $9 + 7 \times 3 + 8 + 0 \times 3 + \cdots$ and comes to 100.

Your task: Read the 13 digits. Work out the total and print it, then say whether the ISBN is valid.

Input: one line: the 13 digits of an ISBN, with no dashes or spaces.

Output: two lines.
Line 1: `Total: T`, where `T` is the total.
Line 2: `Valid` if the total ends in 0, otherwise `Not valid`.

Sample 1 (the checker uses this one)

Input
```
9780306406157
```

Output
```
Total: 100
Valid
```

Why: $9 + 7{\times}3 + 8 + 0{\times}3 + 3 + 0{\times}3 + 6 + 4{\times}3 + 0 + 6{\times}3 + 1 + 5{\times}3 + 7 = 100$, which ends in 0.

Sample 2

Input
```
9780306406158
```

Output
```
Total: 101
Not valid
```

Why: Only the last digit changed, from 7 to 8, so the total is 101. It does not end in 0.

Hint: walk the indexes with a `range` and a step of 2. `int()` turns one digit character into a number.

Pen on this sheet, Shell closed, before you open `task10.py`. Only calculators are allowed.

Key 10.1 An ISBN starts 979. Work out how much these three digits add to the total.
Answer: .....

Key 10.2 Which `range` gives the indexes of the digits multiplied by 3?
(a) `range(2, 13, 2)`
(b) `range(0, 13, 3)`
(c) `range(1, 13, 2)`

Key 10.3 List the numbers `range(0, 13, 2)` gives. How many are there?
Answer: .....

Key 10.4 A student writes `total = total + isbn[i]`, with no `int()`. Predict what happens.
(a) The total is still right.
(b) Python stops with a `TypeError`.
(c) The total stays 0.

Key 10.5 The user inputs `0000000000000`. Predict the lines the program should print.
(a) `Total: 0`, then `Valid`
(b) `Total: 0`, then `Not valid`
(c) Nothing: 0 is not an ISBN.

5 A Loop Inside a Loop (~15 min)

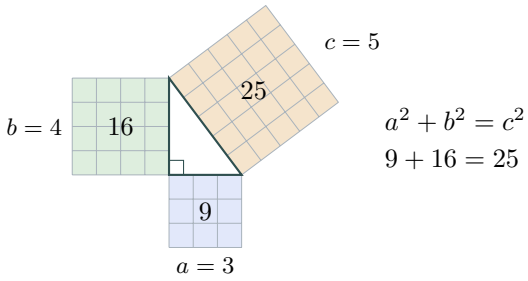
Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

Some problems need every pair of values. You will put one `for` loop inside another, as in Roll the Dice.

TASK 11 Babylonian Triangles

MEDIUM `task11.py` ~12 min

About 3,800 years ago, a Babylonian clay tablet listed right triangles with whole-number sides. A right triangle has two short sides a and b and a long side c, and $a^2 + b^2 = c^2$. For example, $3^2 + 4^2 = 5^2$, since $9 + 16 = 25$.



Your task: Read the long side c. Find every pair of whole numbers a and b, both 1 or more, with $a \le b$ and $a^2 + b^2 = c^2$. Print each pair, then how many there are.

Input: one line: the long side c, a whole number, 1 or more.

Output: one line for each pair: `a b`, the two numbers with one space between, smallest `a` first.
Last line: `Triangles: T`, where `T` is the number of pairs.
If there are no pairs, print only the last line.

Sample 1 (the checker uses this one)

Input
```
25
```

Output
```
7 24
15 20
Triangles: 2
```

Why: $7^2 + 24^2 = 49 + 576 = 625 = 25^2$, and $15^2 + 20^2 = 225 + 400 = 625$. No other pair works.

Sample 2

Input
```
7
```

Output
```
Triangles: 0
```

Why: No two squares of whole numbers add up to $49 = 7^2$, so only the last line prints.

Hint: put one `for` loop inside another. The outer loop picks `a`, and the inner loop picks `b`. Test each pair with an `if`.

Pen on this sheet, Shell closed, before you open `task11.py`. Only calculators are allowed.

Key 11.1 The user inputs 10. Predict the number the program should print after `Triangles:`.

Answer: .....

Key 11.2 The program printed `12 16`, then `Triangles: 1`. Work out the number the user input.

Answer: .....

Key 11.3 Where does `print(a, b)` go?
(a) After both loops.
(b) Inside the outer loop, after the inner loop.
(c) Inside the `if`, inside the inner loop.

Key 11.4 A student puts `count = 0` inside the outer loop, above the inner loop. The user inputs 13. Predict the number their program prints after `Triangles:`.

Answer: .....

Key 11.5 A student's inner loop starts `b` at 1, not at `a`. The user inputs 5. Predict how many lines their program prints before `Triangles:`.

Answer: .....

6 Three Letters at a Time (~14 min)

Ref: Lectures 7 & 8 — Iteration, Problem Set (CCC Pset 2)

A slice can hold three letters at once. You will slide it along a strand, one letter at a time, as in Happy or Sad.

TASK 12 Start Codons

MEDIUM task12.py ~11 min

A cell reads DNA three letters at a time. The three letters ATG, side by side, mark where a gene starts: a **start codon**. The letters of a strand are numbered from 1: the first letter is position 1. In Sample 1, the first ATG starts at position 3.

Your task: Read a strand. Print the position of every ATG in it, left to right, then how many there are.

Input: one line: a DNA strand, made of the capital letters A, C, G and T.

Output: one line for each ATG, from left to right: `ATG at P`, where `P` is the position of its A.
Last line: `Start codons: S`, where `S` is the number of ATGs.
If there are none, print only the last line.

Sample 1 (the checker uses this one)

Input

```
CCATGCATGTATGC
```

Output

```
ATG at 3
ATG at 7
ATG at 11
Start codons: 3
```

Why: Counting letters from 1, an A followed by TG starts at positions 3, 7 and 11.

Sample 2

Input

```
GGCCTA
```

Output

```
Start codons: 0
```

Why: There is no ATG in GGCCTA, so only the last line prints.

Hint: look at three letters at a time with a slice, as in Happy or Sad.

Pen on this sheet, Shell closed, before you open `task12.py`. Only calculators are allowed.

Key 12.1 The user inputs `CATGG`. Predict the number the program should print after `ATG at`.

Answer: .....

Key 12.2 Which slice holds the three letters that start at index `i`?

(a) `dna[i:i + 3]`
(b) `dna[i:i + 2]`
(c) `dna[i + 1:i + 4]`

Key 12.3 The program printed `ATG at 1`, `ATG at 4`, then `Start codons: 2`. Which strand did the user input?

(a) `ATGCATG`
(b) `CATGATG`
(c) `ATGATG`

Key 12.4 The strand has 8 letters. Predict the biggest number the program can ever print after `ATG at`.

Answer: .....

Key 12.5 The user inputs `TTATG`. Predict the number the program should print after `Start codons:`.

Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

- ☐ I can give a loop variable its three jobs: set, test and change.
- ☐ I can say how many rounds a `while` loop runs, even when it is zero.
- ☐ I can find the missing change in a loop that never stops.
- ☐ I can use a print probe to watch values change round by round.
- ☐ I can read one line per round, and stop at a stop value or at a total.
- ☐ I can keep a count, a running total and a running product.
- ☐ I can choose start, stop and step, and I know `range` never gives the stop.
- ☐ I can walk through a string by its characters and by its indexes.
- ☐ I can use `in` to test whether a character is one of a group.
- ☐ I can compare two strings place by place.
- ☐ I can put one loop inside another to try every pair.
- ☐ I can look at three letters at a time with a slice.