Lab 06 — Black Boxes

Functions: `def`, `return`, frames, `lambda`

DURATION ~3 hours **COVERS** Lectures 9 & 10 **TASKS** 7 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` ... `task07.py` in your `lab06` folder. Every task must PASS.
- Tasks 8 to 10 are a bonus: optional, and not counted.
- Run `check.py`, check or submit as often as you like.
- Type the code yourself. No copy-paste.
- This lab assumes Lab 05: `while` and `for` loops, `range(start, stop)`, a running total, and walking through a string by its indexes.

Before you start: get the files

Download the `lab06` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task10.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 A Function Is a Black Box (~23 min)

Ref: Lecture 9 — Abstraction, Decomposition, Functions

You will define a function, call it, and use the value it returns. A function hands its answer back with `return`. The lines that call it decide what to print. So every task has two parts: the function, and the program that calls it. Task 2 is a function with two bugs.

Three 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 with options (a), (b), (c) and (d) takes its letter and nothing else. Every other key takes one number or one word, with no units: type `25`, not `25 m`.
- `check.py` also calls your function by itself, as you can in the Shell after F5. So keep the exact name and parameters the task gives, hand the answer back with `return`, and print nothing inside the function. If you see the word `None`, a function printed its answer, or has no `return`.

TASK 1 Eight Minutes Old

EASY `task01.py` ~6 min

Light travels 18 million km in one minute. The Sun is 150 million km away, so the sunlight you see left the Sun about 8 minutes ago.

Part 1, the function: `light_minutes(million_km)` returns the whole minutes light takes to travel that far. It prints nothing. In the Shell, `light_minutes(4500)` gives `250`.

Part 2, the program: Read a distance, call the function, and print the minutes, then the same time in hours and minutes.

Input: one line: a distance in millions of km, a whole number, 0 or more.

Output: two lines.
Line 1: `Minutes: T`, where `T` is the distance `//` 18.
Line 2: `H h M min`: the same time as whole hours and the minutes left over.

Sample 1 (the checker uses this one)

Input
```
4500
```

Output
```
Minutes: 250
4 h 10 min
```

Why: Neptune is 4500 million km away. 4500 // 18 is 250 minutes: 4 hours and 10 minutes.

Sample 2

Input
```
150
```

Output
```
Minutes: 8
0 h 8 min
```

Why: The Earth. 150 // 18 is 8: `//` drops the part after the whole number.

Hint: write the `def` above the lines that call it. Its body is one `return` line. Call it once, and keep what it returns in a variable.

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

Key 1.1 The user inputs 108, the distance of Venus. Predict the number the program should print after `Minutes:`.

Answer: .....

Key 1.2 Predict what this code prints.

```
1 def tiles(side):
2     return side * side
3
4 print(tiles(3) + tiles(2))
```

Answer: .....

Key 1.3 Predict what this code prints.

```
def half(n):
    n // 2

x = half(10)
print(x)
```

Answer: .....

Key 1.4 The program printed `Minutes: 43` for Jupiter. Work out the smallest distance the user could have input.

Answer: .....

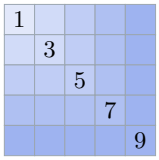
Key 1.5 The user inputs 35. Light needs almost two minutes for that distance. Predict the number the program should print after `Minutes:`.

Answer: .....

TASK 2 Squares by Adding

EASY `task02.py` ~8 min

In the 1790s the French engineer de Prony hired about 80 people to work out huge tables by hand. They were called computers, and many could only add and subtract. A square needs only adding: row 1 of the table holds 1, and each new row adds the next odd number.



row 1: 1 row 2: $1+3=4$ row 3: $4+5=9$

row 4: $9+7=16$ row 5: $16+9=25$

In `task02.py`, the function `square_of(n)` should build row n this way and return it. Right now the program prints two lines, and both are wrong.

Your task: It has exactly **two** bugs. Find and fix them, so it prints the square. Do not rewrite the program. When both are fixed, in the Shell `square_of(6)` gives `36`.

Input: one line: a whole number n, 1 or more.

Output: one line: `n squared is S`, where `S` is $n \times n$.

Sample 1 (the checker uses this one)

Input	Output
`6`	`6 squared is 36`

Why: Rows 1 to 6 hold 1, 4, 9, 16, 25 and 36. The fixed function adds 3, 5, 7, 9 and 11.

Sample 2

Input	Output
`1`	`1 squared is 1`

Why: Row 1 is already 1, so nothing is added.

Hint: find the bugs by **shrinking the failing input**. When the answer for 6 is wrong, try 3, then 2, then 1. Stop at the smallest input that is still wrong, and follow the code by hand for that input. Fix one bug at a time.

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

Key 2.1 Row 2 is made by adding 3 to row 1, and row 3 by adding 5 to row 2. Predict the number that is added to row 7 to make row 8.

Answer: .....

Key 2.2 Predict the second line this code prints.

```
1 def show(n):
2     print(n)
3
4 a = show(3)
5 print(a)
```

Answer: .....

Key 2.3 Predict how many times this code prints `x`.

```
1 for row in range(2, 2):
2     print("x")
```

Answer: .....

Key 2.4 A loop must run once for each of the rows 2, 3, 4, 5, 6 and 7. It is written `for row in range(2, ____)`. Work out the number that fills the blank.

Answer: .....

Key 2.5 A program gives a wrong answer for the inputs 6, 3 and 2. It gives the right answer for 1. You will follow its code by hand for one input. Work out the best input to choose.

Answer: .....

2 Decisions and Loops Inside a Function (~28 min)

Ref: Lecture 9 — Abstraction, Decomposition, Functions

An `if` and a `for` work inside a function as they do outside. Only the way out is new: a `return` ends the function at once, even in the middle of a loop.

TASK 3 Star Colours — EASY — `task03.py` — ~7 min

⌨ A star's colour shows how hot its surface is. Around 1900, women hired as computers at Harvard Observatory sorted about 350,000 stars this way, by hand.

Surface temperature	Colour	A star like this
10000 K or more	`blue`	Rigel, 12100 K
7500 K to 9999 K	`white`	Sirius, 9900 K
5200 K to 7499 K	`yellow`	the Sun, 5800 K
3700 K to 5199 K	`orange`	Arcturus, 4300 K
below 3700 K	`red`	Betelgeuse, 3500 K

Part 1, the function: `colour(kelvin)` returns the star's colour as a string. It prints nothing. In the Shell, `colour(5800)` gives `'yellow'`.

Part 2, the program: Read two temperatures. Print the colour of each star, then whether the two colours are the same.

Input: two lines: the surface temperature of each star in kelvin, a whole number.
Output: three lines.
Line 1: the colour of the first star.
Line 2: the colour of the second star.
Line 3: `Same colour: True` or `Same colour: False`.

Sample 1 (the checker uses this one)

Input
```
10000
5800
```

Output
```
blue
yellow
Same colour: False
```

Why: 10000 is on the edge: 10000 or more is blue. 5800 is the Sun, a yellow star.

Sample 2

Input
```
7500
9999
```

Output
```
white
white
Same colour: True
```

Why: 7500 is the lowest white temperature and 9999 is the highest.

Hint: one `return` in each branch of an `if/elif` chain. Line 3 compares what two calls return.

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

Key 3.1 The user inputs 8200 and 4900. Predict the second line the program should print.

Answer: .....

Key 3.2 Predict what this code prints.

```
1 def fee(age):
2     if age < 18:
3         return 50
4     return 80
5
6 print(fee(3) + fee(18))
```

Answer: .....

Key 3.3 The user inputs 5200 and 5199. Predict the word the program should print after `Same colour:`.

Answer: .....

Key 3.4 Four stars have the temperatures 3699, 3700, 9999 and 10000. Work out how many different colours the program should print for them.

Answer: .....

Key 3.5 Work out how many different whole-number temperatures the program should call `orange`.

Answer: .....

TASK 4 The Wow! Signal

MEDIUM `task04.py` ~11 min

In 1977 a radio telescope in Ohio printed the strength of the radio waves it heard, one character for each reading. `0` to `9` mean strengths 0 to 9. Then `A` means 10, `B` means 11, and so on up to `Z`, which means 35. One night it printed `6EQUJ5`. An astronomer circled it and wrote "Wow!". Nobody knows what sent it. The scale is one string, already in your file:

```
SCALE = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

A character's strength is its index in `SCALE`. Any character that is not in `SCALE`, such as a space or a small letter, has strength 0.

Part 1, the function: `strength(ch)` returns the strength of one character. It prints nothing. In the Shell, `strength('E')` gives `14`.

Part 2, the program: Read the printed code. Print the strength of each character, then the total.

Input: one line: the code, made of digits, capital letters and spaces.

Output: one line for each character of the code, in order: its strength.
Last line: `Total: T`, where `T` is the strengths added together.

Sample 1 (the checker uses this one)

Input

```
6EQUJ5
```

Output

```
6
14
26
30
19
5
Total: 100
```

Why: E is 14, Q is 26, U is 30 and J is 19. The six strengths add up to 100.

Sample 2

Input

```
1 Z
```

Output

```
1
0
35
Total: 36
```

Why: The space is not on the scale, so it counts 0. Z is the top of the scale, 35.

Hint: in `strength`, walk the indexes of `SCALE` with a `for` loop, and return as soon as the character matches. Decide what the function returns when the loop ends with no match.

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

Key 4.1 The user inputs `7C`. Predict the number the program should print after `Total:`.
Answer: .....

Key 4.2 The user inputs `HELLO`. Predict how many lines the program should print.
Answer: .....

Key 4.3 The program printed `11`, then `10`, then `13`, then `Total: 34`. Work out the code the user input.

Answer: .....

Key 4.4 Predict what this code prints.

```
def find(limit):
    for n in range(1, 10):
        if n * n > limit:
            return n
    return 0

print(find(10) + find(200))
```

Answer: .....

Key 4.5 Predict what this code prints.

```
def first_big(a, b):
    for n in range(a, b):
        if n * n > 50:
            return n

print(first_big(1, 5))
```

Answer: .....

Key 4.6 The user inputs `a1`, with a small `a`. Predict the first line the program should print.

Answer: .....

Key 4.7 The code has 4 characters. Work out the largest number the program can ever print after `Total:`.

Answer: .....

3 Functions That Call Functions (~17 min)

Ref: Lecture 9 — Abstraction, Decomposition, Functions

A big job is easier as small functions, each with one job of its own. One function can call another and use what it returns. You will build a date calculator from two functions.

TASK 5 **Day of the Year** MEDIUM `task05.py` ~11 min

Astronomers number the days of a year: 1 January is day 1, 1 February is day 32, and 31 December is day 365. April, June, September and November have 30 days. February has 28 (this task has no leap years). The other seven months have 31.

Part 1, the functions: `month_days(month)` returns the number of days in that month. `day_number(day, month)` adds the days of all the months before this one, by calling `month_days`, then adds `day`, and returns the sum. Neither prints. In the Shell, `month_days(2)` gives 28 and `day_number(1, 2)` gives 32.

Part 2, the program: Read a date. Print its day number, then the days left in the year.

Input: Line 1: the day of the month, 1 to 31.
Line 2: the month, 1 to 12.
Output: two lines.
Line 1: `Day number: D`.
Line 2: `Days left: L`, where `L` is 365 minus `D`.

Sample 1 (the checker uses this one)

Input
```
14
8
```

Output
```
Day number: 226
Days left: 139
```

Why: January to July have $31 + 28 + 31 + 30 + 31 + 30 + 31 = 212$ days. $212 + 14$ is 226.

Sample 2

Input
```
1
1
```

Output
```
Day number: 1
Days left: 364
```

Why: No month comes before January, so nothing is added but the day.

Hint: `day_number` needs a running total and a `for` loop over the months before this one. Call `month_days` inside the loop.

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

Key 5.1 The user inputs 10 and 3, for 10 March. Predict the number the program should print after `Day number:`.

Answer: .....

Key 5.2 The user wants the day number of 5 November. Work out the number the user must input on line 2.

Answer: .....

Key 5.3 Predict what this code prints.

```
1 def half(n):
2     return n // 2
3 def quarter(n):
4     return half(half(n))
5
6 print(quarter(22) + half(7))
```

Answer: .....

Key 5.4 The user inputs 25 and 12, for 25 December. Predict the number the program should print after `Days left:`.

Answer: .....

Key 5.5 The program printed `Day number: 100`. Work out the month number the user input.

Answer: .....

Key 5.6 In this code, the fare rises: line 2 becomes `return km * 3`. Predict what the code prints then.

```
1 def fare(km):
2     return km * 2
3 def return_trip(km):
4     return fare(km) + fare(km)
5
6 print(return_trip(10))
```

Answer: .....

Key 5.7 The user inputs 31 and 1, for 31 January. Predict the number the program should print after `Day number:`.

Answer: .....

4 Default Parameters (~11 min)

Ref: Lecture 9 — Abstraction, Decomposition, Functions

A parameter can have a usual value, called its default. A call may leave that argument out, give it by position, or give it by name. You will write one function and call it in all three ways.

TASK 6 **World Clock** EASY `task06.py` ~7 min

The world sets its clocks from UTC, the time kept at Greenwich in England. Lahore is 5 hours ahead of UTC, Makkah is 3 and Tokyo is 9. A clock shows the hours 0 to 23: after 23 comes 0.

Part 1, the function: `local_hour(utc, offset=5)` returns the hour on a clock `offset` hours ahead of UTC. It prints nothing. In the Shell, `local_hour(20)` gives `1` and `local_hour(20, 3)` gives 23.

Part 2, the program: Read the UTC hour and print the hour in three cities. Call the function three ways: with no offset for Lahore, with `3` for Makkah, with `offset=9` for Tokyo.

Input: one line: the hour in UTC, a whole number from 0 to 23.

Output: Line 1: `Lahore: H`, where `H` is the hour in Lahore, from 0 to 23.
Line 2: `Makkah: H`, the same for Makkah.
Line 3: `Tokyo: H`, the same for Tokyo.

Sample 1 (the checker uses this one)

Input
```
20
```

Output
```
Lahore: 1
Makkah: 23
Tokyo: 5
```

Why: $20 + 5$ is 25, one hour past midnight: Lahore shows 1. $20 + 9$ is 29: Tokyo shows 5.

Sample 2

Input
```
19
```

Output
```
Lahore: 0
Makkah: 22
Tokyo: 4
```

Why: $19 + 5$ is 24: exactly midnight, which a clock shows as 0.

Hint: the default goes in the `def` line. `%` can wrap an hour past 23 round to 0.

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

Key 6.1 The user inputs 9. Predict the number the program should print after `Lahore:`.

Answer: .....

Key 6.2 Predict what this code prints.

```
1 def seats(rows, per_row=4):
2     return rows * per_row
3
4 print(seats(3) + seats(2, 10) + seats(per_row=1, rows=5))
```

Answer: .....

Key 6.3 A function is defined with `def seats(rows, per_row=4)`. Which call makes Python stop with an error?
(a) `seats(6)`
(b) `seats(per_row=2, 6)`
(c) `seats(6, per_row=2)`
(d) `seats(per_row=2, rows=6)`

Key 6.4 The program printed `Tokyo: 2`. Work out the UTC hour the user input.

Answer: .....

Key 6.5 The user inputs 21. Predict the number the program should print after `Makkah:`.

Answer: .....

5 Frames: Where a Name Lives (~17 min)

Ref: Lecture 10 — Environment Diagrams, Lambda Functions

Every call gets its own frame, and the frame is thrown away when the call returns. So a function cannot change a variable outside it. It returns the new value, and the caller stores it.

TASK 7 Rover Battery — MEDIUM — `task07.py` — ~11 min

The Mars rover Opportunity ran on sunlight for 5,111 Mars days, called sols. Each sol its solar panels added charge to the battery, and driving used some. In this task a full battery holds 100 units. The level never goes above 100 and never below 0.

Part 1, the function: `change(level, amount)` returns the new level: `level + amount`, but 100 if that is more than 100, and 0 if that is less than 0. It prints nothing. In the Shell, `change(50, 60)` gives `100`.

Part 2, the program: Read the start level and the number of sols. For each sol, read its change, update the level with the function, and print the level.

Input: Line 1: the start level, 0 to 100.
Line 2: the number of sols n, 0 or more.
Then n lines: the change in each sol, a whole number. It is positive when the panels add more than the rover uses, and negative otherwise.

Output: one line for each sol: `Sol K: L`. `K` is 1, 2, 3, ... and `L` is the level after that sol.
Last line: `Final level: L`. With no sols, it is the only line.

Sample 1 (the checker uses this one)

Input

```
50
3
60
-70
-90
```

Output

```
Sol 1: 100
Sol 2: 30
Sol 3: 0
Final level: 0
```

Why: $50 + 60$ is 110, so sol 1 stops at 100. $30 - 90$ is -60, so sol 3 stops at 0.

Sample 2

Input

```
80
0
```

Output

```
Final level: 80
```

Why: No sols, so the level never changes. Only the last line prints.

Hint: write `FULL = 100` above the function, and let the function read it. A function cannot change the level in your main program: store the value it returns.

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

Key 7.1 The user inputs 30, 2, 50 and `-20`: a start of 30, then two sols. Predict the number the program should print after `Final level:`.

Answer: .....

Key 7.2 Predict what this code prints.

```
1 def boost(speed):
2     speed = speed + 5
3
4 speed = 10
5 boost(speed)
6 print(speed)
```

Answer: .....

Key 7.3 Predict what this code prints.

```
1 def burn(fuel):
2     return fuel - 3
3
4 fuel = 8
5 fuel = burn(fuel)
6 burn(fuel)
7 print(fuel)
```

Answer: .....

Key 7.4 Count the frames Python makes while this code runs. Count the Global Frame too.

```
def tick(n):
    return n + 10

a = tick(1)
b = tick(a)
c = tick(b)
```

Answer: .....

Key 7.5 The user inputs 90, 2, 30 and `-50`. Predict the number the program should print after `Sol 2:`.

Answer: .....

Key 7.6 The start level is 60 and there is one sol. The program printed `Sol 1: 100`. Work out the smallest change the user could have input.

Answer: .....

Key 7.7 The user inputs 10, 3, `-40`, 30 and `-5`. Predict the number the program should print after `Final level:`.

Answer: .....

6 Bonus: Functions as Data (~41 min)

Optional. Not counted. From Lecture 10 — Environment Diagrams, Lambda Functions

A function is a value. You can store it in a variable, pass it to another function, and write a small one in place with `lambda`. A name with no brackets is the function itself. Brackets call it.

TASK 8 What Would You Weigh? EASY `task08.py` ~8 min

⌨ Gravity is weaker on the Moon and on Mars than on Earth, and stronger on Jupiter. So a bathroom scale shows a different number on each world.

World	The scale shows	In Python
Moon	one sixth of the Earth reading	`kg // 6`
Mars	38 percent of it	`kg * 38 // 100`
Jupiter	two and a half times it	`kg * 5 // 2`

Part 1, the functions: `on_moon(kg)`, `on_mars(kg)` and `on_jupiter(kg)`. Each returns what the scale shows on that world, and prints nothing. In the Shell, `on_mars(60)` gives 22.

Part 2, the program: Read a world and two Earth readings. Store the right function in a variable, then call it through that variable for each reading.

Input: three lines.
Line 1: the world: `moon`, `mars` or `jupiter`.
Lines 2 and 3: the two Earth readings in kg, each a whole number.

Output: two lines.
Line 1: `First: A kg`, where `A` is what the scale shows there for the first reading.
Line 2: `Second: B kg`, the same for the second reading.

Sample 1 (the checker uses this one)

Input

```
mars
60
85
```

Output

```
First: 22 kg
Second: 32 kg
```

Why: 60 × 38 is 2280, and 2280 // 100 is 22. 85 × 38 is 3230, and 3230 // 100 is 32.

Sample 2

Input

```
jupiter
1
60
```

Output

```
First: 2 kg
Second: 150 kg
```

Why: 1 × 5 is 5, and 5 // 2 is 2, because `//` drops the half. 60 × 5 is 300, and 300 // 2 is 150.

Hint: a function's name with no brackets is the function itself. Choose it with `if`/`elif`/`else`.

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

Key 8.1 The user inputs `jupiter`, 40 and 7. Predict the number the program should print after `Second:`.

Answer: .....

Key 8.2 Predict what this code prints.

```
1 def twice(n):
2     return n * 2
3
4 job = twice
5 print(job(9) + twice(1))
```

Answer: .....

Key 8.3 Predict how many lines this code prints.

```
1 def beep():
2     print("beep")
3
4 beep
5 beep()
6 print("end")
```

Answer: .....

Key 8.4 The readings were 12 and 30. The program printed `First: 2 kg`, then `Second: 5 kg`. Work out the world the user input.

Answer: .....

Key 8.5 The world is `moon`. The program printed `First: 9 kg`. Work out the smallest Earth reading the user could have input.

Answer: .....

TASK 9 Bouncing Balls

MEDIUM `task09.py` ~11 min

⌨ Drop a ball and it bounces back, but not as high. From a height of `cm`, a basketball comes back to three quarters of it: `cm * 3 // 4`. A tennis ball comes back to half: `cm // 2`. Each bounce starts from the height of the bounce before.

Part 1, the functions: `basketball(cm)` and `tennis_ball(cm)` each return the height after one bounce. `height_after(rule, cm, bounces)` applies the function `rule` again and again, `bounces` times, each time to the height before, and returns the last height. None of them prints. In the Shell, `height_after(tennis_ball, 200, 3)` gives 25.

Part 2, the program: Read the drop height and the number of bounces. Print the height of each ball after its last bounce.

Input: Line 1: the drop height in cm, a whole number.
Line 2: the number of bounces, 0 or more.

Output: two lines.
Line 1: `Basketball: X cm`, where `X` is its height after the last bounce.
Line 2: `Tennis ball: Y cm`, the same for the tennis ball.

Sample 1 (the checker uses this one)

Input
```
200
3
```

Output
```
Basketball: 84 cm
Tennis ball: 25 cm
```

Why: The basketball goes 200, 150, 112, 84. The tennis ball goes 200, 100, 50, 25.

Sample 2

Input
```
200
0
```

Output
```
Basketball: 200 cm
Tennis ball: 200 cm
```

Why: No bounce yet: both balls are still at the drop height.

Hint: in `height_after`, each round of a loop gives the old height to `rule`. Pass a rule by its name alone, with no brackets.

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

Key 9.1 The user inputs 64 and 2: a drop of 64 cm, then 2 bounces. Predict the number the program should print after `Basketball:`.

Answer: .....

Key 9.2 Predict what this code prints.

```
1 def add_two(n):
2     return n + 2
3 def run(job, n):
4     return job(n) * job(1)
5
6 print(run(add_two, 4))
```

Answer: .....

Key 9.3 The drop height is 9 cm and there are 2 bounces. Predict the number the program should print after `Basketball:`.

Answer: .....

Key 9.4 The user inputs 96 for the height. The program printed `Tennis ball: 12 cm`. Work out the number of bounces the user input.

Answer: .....

Key 9.5 Predict what this code does.

```
def add_two(n):
    return n + 2
def run(job, n):
    return job(n)

print(run(add_two(4), 4))
```

(a) It prints 6.
(b) It prints 8.
(c) Python stops: `add_two` is given too many arguments.
(d) Python stops: `job` holds the number 6, and a number cannot be called.

Key 9.6 The user inputs 1 and 5. Predict the number the program should print after `Tennis ball:`.

Answer: .....

Key 9.7 The drop height is 80. Work out the smallest number of bounces for which the program prints a number below 10 after `Tennis ball:`.

Answer: .....

TASK 10 Galileo's Seconds

EASY `task10.py` ~7 min

Around 1604, Galileo timed balls rolling down a ramp. He found that a falling object moves farther in each second than in the second before. A stone that has been falling for t seconds has fallen $5 \times t \times t$ metres. A cyclist at steady speed is different: after t seconds she has gone $6 \times t$ metres.

Part 1, the function: `moved(position, t)`. `position` is a function: it returns the distance travelled after a number of seconds. `moved` returns how far the thing moves during the next second: its distance after $t + 1$ seconds, minus its distance after t seconds. It prints nothing. In the Shell, `moved(lambda s: 6 * s, 0)` gives 6.

Part 2, the program: Read t. Print how far the stone moves during the next second, then the cyclist. Write each position rule as a `lambda`, inside the call.

Input: one line: the time t in seconds, a whole number, 0 or more.

Output: two lines.
Line 1: `Stone: X m`.
Line 2: `Cyclist: Y m`.

Sample 1 (the checker uses this one)

Input
```
3
```

Output
```
Stone: 35 m
Cyclist: 6 m
```

Why: The stone has fallen 45 m after 3 seconds and 80 m after 4 seconds: $80 - 45$ is 35.

Sample 2

Input	Output
`0`	`Stone: 5 m` `Cyclist: 6 m`

Why: In its first second the stone falls from 0 m to 5 m. The cyclist goes from 0 m to 6 m.

Hint: write each rule as a `lambda` with one parameter, in the place of the first argument. A `lambda` has no `return`.

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

Key 10.1 The user inputs 2. Predict the number the program should print after `Stone:`.

Answer: .....

Key 10.2 Predict what this code prints.

```
def both(job, a, b):
    return job(a) + job(b)

print(both(lambda n: n * n, 2, 3))
```

Answer: .....

Key 10.3 Which line makes a function that gives back double its input?

(a) `lambda n: return n * 2`
(b) `lambda n n * 2`
(c) `lambda n: n * 2`
(d) `lambda: n * 2`

Key 10.4 The program printed `Stone: 55 m`. Work out the number the user input.

Answer: .....

Key 10.5 Which line can the program never print?

(a) `Stone: 20 m`
(b) `Stone: 5 m`
(c) `Stone: 45 m`
(d) `Cyclist: 6 m`

Self-Assessment Checklist

Before you leave, tick what you can now do:

- ☐ I can write a `def` line with a name, parameters and a colon, and a body that returns a value.
- ☐ I can use the value a call returns: in a sum, in a comparison and inside an f-string.
- ☐ I can tell `print` from `return`, and I know a function with no `return` gives `None`.
- ☐ I can shrink a failing input until the bug is easy to see.
- ☐ I can write a function with several `return` lines, and I know the first one reached ends it.
- ☐ I can return from inside a loop, and return a default after the loop.
- ☐ I can split a job into two functions and have one call the other.

☐ I can give a parameter a default, and call by position and by keyword.

☐ I can say which frame a name lives in, and why a function cannot change a variable outside it.

☐ I can update a variable by storing what a function returns.

☐ I can store a function in a variable and pass it to another function, with no brackets.

☐ I can write a `lambda` for a one-line rule.