Finger Exercises 23

Randomness & Simulation

≈ 70 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for Lecture 23. You will roll Python's dice in your head, fix broken simulations, play turns of Roll or Stop with dice we give you, and count chances by hand to check the computer.

How to do it. On **paper**, **no computer**, in one sitting. If an item takes you more than a minute, skip it and come back. Parts A–P are for *everyone*; Part Q is a **stretch**. The Console Quests are for tonight, *at* a computer. The answer key is on the last pages — do not peek until you are done.

Predict first, then check. Write your answer *before* you type anything. A wrong guess teaches you more than an answer you only watched the computer give.

Dice we give you. Paper cannot roll dice, so many items say which numbers `random.randint(1, 6)` gives, in order: ⚀ ⚄ ⚀ ⚅ means the first call gives 4, the next 3, then 1, then 6. A red die is a 1. Everywhere on this sheet, `import random` has already run.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. `random.randint(1, 6)` gives a whole number from _____ to _____. `range(1, 6)` gives the numbers from _____ to _____.

A2. Ulam's recipe: play the game _____ times, count the games that _____, then _____ by the number of games.

A3. Numbers that only *look* random but come from a fixed rule are called _____. The start number of that rule is the _____.

A4. In Roll or Stop, a turn ends when you choose _____ or roll a ____. With a **1**, the turn gives _____ points.

A5. 100 times more tries give about _____ more correct digit(s).

A6. Circle **True** or **False**:

(a) A computer can roll a real die. True / False

(b) `random.choice('HT')` picks `'H'` or `'T'`. True / False

(c) After `random.seed(7)`, the next rolls are the same on every run. True / False

(d) A simulation with 10 tries gives the exact answer. True / False

(e) We tested the recipe on two dice because there we knew the answer. True / False

(f) In the lecture, the best h for points per turn was 20. True / False

(g) `turn(10)` always gives at least 10 points. True / False

(h) A bug that crashes only sometimes is easy to find. True / False

Part B • Can It Ever Give That?

CLASSIFY ≈ 5 min

Can the line **ever** give the value? Write **YES** or **NO**. If the line always stops with an error, write the error's name.

	Line	Value	Ever?
1	`random.randint(1, 6)`	`6`	
2	`random.randint(1, 6)`	`0`	
3	`random.randint(1, 6)`	`3.5`	
4	`random.randint(0, 1)`	`1`	
5	`random.randint(5, 5)`	`5`	
6	`random.randint(6, 1)`	`3`	
7	`random.choice(['ROLL', 'STOP'])`	`'R'`	
8	`random.randint(1, 6) + random.randint(1, 6)`	`1`	
9	`random.randint(1, 6) + random.randint(1, 6)`	`12`	
10	`random.randint(1, 6) - random.randint(1, 6)`	`0`	
11	`2 * random.randint(1, 3)`	`3`	
12	`random.choice('HT')`	`'HT'`	
13	`random.choice('ROLL')`	`'L'`	
14	`random.choice([])`	`0`	

Part C • Who Goes First?

FIX ≈ 5 min

Three friends play: `names = ['Ali', 'Sana', 'Omar']`. Each line tries to pick who goes first. For each line, tick **one** column. In the middle column, also write *who* is never picked.

	Line	fine	never picks ...	can crash
1	`names[random.randint(0, len(names))]`	☐	☐ _____	☐
2	`names[random.randint(0, len(names) - 1)]`	☐	☐ _____	☐
3	`names[random.randint(1, len(names))]`	☐	☐ _____	☐
4	`names[random.randint(1, len(names) - 1)]`	☐	☐ _____	☐
5	`names[random.randint(1, 3) - 1]`	☐	☐ _____	☐
6	`names[random.randint(-3, -1)]`	☐	☐ _____	☐
7	`random.choice(names)`	☐	☐ _____	☐
8	`random.choice(names[1:])`	☐	☐ _____	☐

C1. Line 1 ran 8 times with no crash. Is it correct? Why, or why not?

_____

Part D • Fake Dice by Hand

TRACE ≈ 5 min

Von Neumann's middle-square method: write the square with **8** digits (put zeros in front if it is shorter), then keep the **middle four**. That is the next number. Example: $1234^2 = 1522756$, as 8 digits `01522756`, so the next number is `5227`.

Number	Square	As 8 digits	Next number (middle four)
2500	6250000	_____	_____
4100	16810000	_____	_____
8100	65610000	_____	_____
6100	37210000	_____	_____
2100	4410000	_____	_____
1000	1000000	_____	_____

D1. Start at `2500`. Write the next five numbers: _____

D2. Start at `4100`. After how many steps do you see `4100` again? _____ Then what happens? _____

D3. Start at `1000`. What comes out after that, for ever? _____

D4. Two students both start at `1234`. Do they get the same numbers? _____
Why? _____

Part E • What Does the Seed Print?

TRACE ≈ 4 min

In our Python, `random.seed(7)` and then eight calls of `random.randint(1, 6)` give 3, 2, 4, 6, 1, 1, 5, 1. Use that fact. If you cannot know the output without a computer, write **cannot know**.

E1.

```
1 random.seed(7)
2 for i in range(4):
3     print(random.randint(1, 6), end=' ')
4 print()
5 for i in range(4):
6     print(random.randint(1, 6), end=' ')
```

First line: _____

Second line: _____

E2.

```
1 for i in range(5):
2     random.seed(7)
3     print(random.randint(1, 6), end=' ')
```

Output: _____

E3.

```
1 random.seed(8)
2 print(random.randint(1, 6))
```

Output: _____

E4. Sana runs `random.seed(7)` and then four rolls; Omar does the same on his own laptop. Same numbers? _____

Part F • Count by Hand

COUNT ≈ 6 min

Each box is one of the **36** pairs of two dice; the number in it is the total. Ring the boxes you need, then write each chance as a fraction of 36.

die b \ die a	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

F1. total is 7 _____ / 36

F2. a double (`a == b`) _____ / 36

F3. total is 12 _____ / 36

F4. total is 10 or more _____ / 36

F5. die `a` is bigger than `b` _____ / 36

F6. total is 1 _____ / 36

F7. total is even _____ / 36

F8. The most likely total is _____. Why does `random.randint(2, 12)` *not* act like two dice?

_____

Part G • The Recipe, Filled In

FILL IN ≈ 5 min

G1. Fill in the blanks so that `chance_of_doubles(n)` rolls two dice `n` times and gives the share of doubles.

```
def chance_of_doubles(n):
    hits = ______
    for i in range(______):
        a = random.randint(1, 6)
        b = random.randint(______, ______)
        if ______________:
            hits += 1
    return ______________
```

With a big `n`, it prints about _____ (use Part F).

G2. In `chance_of_seven` (Part I), which line must change to give the chance of a total of **10 or more**? Write the new line: _____

G3. Fill in the one blank so that this gives the chance that a turn (Part J) loses everything:

```
def chance_of_zero(h, n):
    hits = 0
    for i in range(n):
        if ____________________:
            hits += 1
    return hits / n
```

Part H • Is That Result Possible?

CLASSIFY ≈ 5 min

H1. `chance_of_seven(n)` gives `hits / n`. Can it give this value? Write **YES**, **NO** or the error.

`chance_of_seven(10)`	`0.4`	_____	`chance_of_seven(4)`	`0.25`	_____
`chance_of_seven(10)`	`0.45`	_____	`chance_of_seven(4)`	`0.3`	_____
`chance_of_seven(10)`	`1.0`	_____	`chance_of_seven(1)`	`0.0`	_____
`chance_of_seven(10)`	`1.2`	_____	`chance_of_seven(0)`	`0.0`	_____

H2. Fill in the table. Use "100 times more tries, about 1 more correct digit".

tries		correct digits
100	→	1
10,000	→	_____
_____	→	3
_____	→	4

H3. Python does about 1,000,000 tries a second. About how long for **4** correct digits? _____ For 6? _____

Part I • Broken Recipes

FIX ≈ 7 min

The working code (its answer is about 0.167):

```
1 def chance_of_seven(n):
2     hits = 0
3     for i in range(n):
4         a = random.randint(1, 6)
5         b = random.randint(1, 6)
6         if a + b == 7:
7             hits += 1
8     return hits / n
```

Each version changes the code in **one** way. What does `chance_of_seven(1000)` print? Write a number (or "about ..."), or the two values it can print.

	The change	Prints
1	Lines 4 and 5 move up, above line 3 (the dice are rolled once).	_____
2	Line 2 moves down, into the loop (it is the loop's first line).	_____
3	Line 8 moves into the loop, under line 7's `if` (same indent as `if`).	_____
4	Line 7 becomes `hits + 1`	_____
5	Line 8 becomes `return hits`	_____
6	Lines 4 and 5: `random.randint(1, 7)`	_____
7	Line 4 becomes `a = random.randint(2, 12)`, line 5 goes, line 6 becomes `if a == 7:`	_____
8	Line 4 becomes `a = 2 * random.randint(1, 6)` and line 5 `b = 0`	_____

Part J • One Turn by Hand

TRACE ≈ 6 min

```
1 def turn(h):
2     total = 0
3     while total < h:
4         roll = random.randint(1, 6)
5         if roll == 1:
6             return 0
7         total += roll
8     return total
```

J1. Fill in the table. "Rolls used" is how many dice the call took.

Call	Dice given	Gives	Rolls used
`turn(10)`	⚂⚃⚄⚁	_____	_____
`turn(10)`	⚅⚁⚀⚄	_____	_____
`turn(10)`	⚁⚃⚃⚅	_____	_____
`turn(20)`	⚀⚅⚅⚅	_____	_____
`turn(20)`	⚅⚅⚅⚅	_____	_____
`turn(6)`	⚅⚀⚀⚀	_____	_____
`turn(0)`	⚀⚀⚀⚀	_____	_____

J2. Python's dice give, in order: ⚄⚁⚁⚅⚅⚁⚀⚁⚃⚃⚂⚀. We type `turn(10)` four times. What does each call give?

_____ _____ _____ _____

Part K • Broken Turns

FIX ≈ 7 min

Each version changes `turn` (Part J) in **one** way. Run it as `turn(10)` on each stream of dice: **A** is ⚃⚂⚀⚅⚅⚄ and **B** is ⚃⚂⚂⚁⚅⚅. The working `turn` gives **0** on A and **10** on B. Tick **same** if the version *always* gives what the working one gives.

	The change	A gives	B gives	same?
1	Line 6 becomes `break`	_____	_____	☐
2	Line 6 becomes `return total`	_____	_____	☐
3	Line 6 becomes `total = 0`	_____	_____	☐
4	Line 3 becomes `while total <= h:`	_____	_____	☐
5	Line 7 moves up, above line 5 (add first, then check for a 1).	_____	_____	☐
6	Line 4 moves up, above line 3 (the die is rolled once).	_____	_____	☐
7	Line 8 moves into the loop, under line 7 (same indent).	_____	_____	☐
8	Line 5 becomes `if roll == 6:`	_____	_____	☐

Part L • Many Turns and the Hill

TRACE ≈ 5 min

```
1 def average_points(h, n):
2     total = 0
3     for i in range(n):
4         total += turn(h)
5     return total / n
```

L1. `average_points(10, 4)`: the four turns give 15, 0, 10, 0. What is `total` after each turn?

_____ _____ _____ _____

What does it return? _____

L2. Line 4 becomes `total = turn(h)`. Same four turns: it returns _____

L3. `average_points(10, 0)` gives _____

L4. 100,000 turns for each h gave:

h	5	10	15	20	25	30	35	40
points	4.81	6.86	7.82	8.18	7.95	7.58	7.00	6.35

(a) Which is better, $h = 10$ or $h = 30$? _____

(b) Why does a very big h give *fewer* points? _____

(c) The exact answer for $h = 20$ is 8.14, not 8.18. Why the difference? _____

Part M • Roll Once More?

EXPLAIN ≈ 4 min

You have t points this turn. One more roll: 1 time in 6 you **lose** t; 5 times in 6 you gain 4 on average (the average of 2 to 6). So the loss is $\frac{t}{6}$ and the gain is $\frac{20}{6}$.

M1. Write **ROLL** or **STOP** (or **either**, when loss and gain are equal): $t = 6$ _____ $t = 19$ _____ $t = 20$ _____ $t = 27$ _____

M2. A new die has faces 1 to **8**. A 1 still loses. The gain is $\frac{2+3+\cdots+8}{8} = \frac{35}{8}$ and the loss is $\frac{t}{8}$. Keep rolling while $t <$ _____

M3. Back to a normal die, but now a 1 **or** a 2 loses. The gain is $\frac{3+4+5+6}{6} = \frac{18}{6}$; the loss is $\frac{2t}{6}$. Keep rolling while $t <$ _____

Part N • Darts and a Frog

TRACE ≈ 5 min

N1. A dart lands at `(x, y)`. It is **inside** the quarter circle when `x*x + y*y <= 1`. Write **in** or **out**.

`(0.5, 0.5)` _____ `(0.9, 0.6)` _____ `(0.6, 0.8)` _____ `(0.3, 0.9)` _____

`(1.0, 0.1)` _____ `(0.0, 0.0)` _____ `(0.7, 0.7)` _____ `(0.8, 0.7)` _____

N2. Those 8 darts: $4 \times \frac{\text{inside}}{8} =$ _____. Far from 3.14. Why? _____

N3. Work out $4 \times \frac{\text{inside}}{\text{all}}$: 8 of 10 _____ 39 of 50 _____ 157 of 200 _____

N4. The frog starts at home (0). H: one step right (+1); T: one step left (−1). The coins come up `H H T H T T T`. Where is the frog? _____ How far from home? _____

N5. After **3** steps, where can the frog be? Circle all: −3 −2 −1 0 1 2 3

Part O • Hangman by Hand

TRACE ≈ 6 min

The computer player guesses in this fixed order, most common letter first, and never looks at the dashes: `order = 'esiarntoldcugpm...'`. Assignment 2's rules: 10 guesses; a wrong **vowel** costs 2, a wrong **consonant** costs 1, a right letter costs nothing. **Won**: every letter of the word is guessed. **Lost**: 0 guesses left.

O1. Play `secret = 'rain'`. Fill in each row until the game ends; leave the rest empty.

letter	e	s	i	a	r	n	t	o
in the word?	____	____	____	____	____	____	____	____
guesses left	____	____	____	____	____	____	____	____

Won or lost? _____ After which letter? _____

O2. Play `secret = 'dog'` the same way. Won or lost? _____ After which letter? _____

O3. The most-common player won **22,223** of 100,000 games; the random player won **1,484**. Write each chance as a decimal: _____ and _____. About how many times more wins? _____

Part P • One Sentence Each

EXPLAIN ≈ 3 min

P1. Why did Ulam play solitaire many times instead of working out his chance?

_____

P2. Why test the recipe on two dice before using it on Roll or Stop?

_____

P3. How does `random.seed` help you fix a bug that crashes only sometimes?

_____

Part Q • Write the Code

STRETCH ≈ 10 min, optional

Q1. Write `rolls_to_six()`: roll one die until a 6 comes up and return how many rolls it took (the 6 counts). Then write `average_rolls(n)`, which gives the average of `n` calls. Guess its answer first.

Q2. The frog's walk is below. Write `chance_home(steps, n)`: the chance that the frog is back at home (0) after `steps` steps.

```
def walk(steps):
    x = 0
    for i in range(steps):
        x += random.choice([-1, 1])
    return x
```

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type. Start from `lec23.py`.

1. Type `random.seed(7)` and then `random.randint(1, 6)` eight times. Is it `3 2 4 6 1 1 5 1`? Now try `random.seed(102)`.
2. Run `chance_of_seven(10)` five times. How far from 0.1667 is the worst one? Then run `chance_of_seven(100000)` five times.
3. Run `average_turn(turn, h, 1000000)` for h = 18, 19, 20, 21 and 22. Can you see that 20 and 21 tie? (It may take a minute.)

Stuck on more than two items? Re-read the Lecture 23 slides "Two Tools", "Ulam's Recipe, in Python", "Two Dice, Simulated", "One Turn in Python" and "Many Turns in Python" before the next class — then redo Parts B, I and K cold.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. **1** to **6** (both ends in); **1** to **5** (`range` leaves out its end).

A2. **many** (`n`) times; that **worked** (the hits); **divide**. Chance ≈ hits ÷ games.

A3. **pseudo-random**; the **seed** (`random.seed(7)` sets it to 7).

A4. **STOP**; a **1**; **0** (you lose the whole turn's points).

A5. **1**. 100 tries: 1 digit; 10,000: 2 digits; 1,000,000: 3 digits.

A6. (a) **False** — it only does maths; its numbers are pseudo-random. (b) **True** — a string is a sequence of letters. (c) **True** — that is what a seed is for. (d) **False** — 10 tries are far too few; the answer is only close after many tries. (e) **True** — $6/36 = 0.1667$, worked out by hand. (f) **True** — about 8.14 points a turn. (g) **False** — a 1 gives 0. (h) **False** — it hides on most runs; a seed makes it come back every run.

B. 1–7: **YES** (both ends in); **NO**; **NO** (whole numbers only); **YES** (0 or 1); **YES** (always 5: the only number from 5 to 5); **ValueError** (the first number must not be bigger: an empty range); **NO** (it picks a whole item of the list, `'ROLL'` or `'STOP'`). 8–14: **NO** (the smallest total is $1 + 1 = 2$); **YES** ($6 + 6$); **YES** (any two equal dice; the values go from -5 to 5); **NO** (only 2, 4 or 6); **NO** (one letter at a time); **YES** (a string is a sequence of letters, so any of R, O, L, L); **IndexError** (nothing to choose from).

C. 1 **can crash** (it can give 3: there is no `names[3]`, an IndexError). 2 **fine** (0 to 2). 3 **can crash**, and it never picks Ali (place 0). 4 never picks **Ali** (1 or 2 only), no crash. 5 **fine** (1 to 3, minus 1, is 0 to 2). 6 **fine** (places -3, -2, -1 are Ali, Sana, Omar, counted from the end). 7 **fine**, and the simplest. 8 never picks **Ali** (`names[1:]` leaves him out).

C1. **Not for sure.** `randint(0, 3)` gives 3 only about 1 time in 4, so 8 runs often miss it. The bug is still there: it *hides*. Run it with `random.seed(1)` and the crash comes every time.

D. `06250000` → `2500`; `16810000` → `8100`; `65610000` → `6100`; `37210000` → `2100`; `04410000` → `4100`; `01000000` → `0000`.

D1. `2500 2500 2500 2500 2500`: it is **stuck** (the middle four of `06250000` is `2500` again).

D2. After **4** steps: 4100, 8100, 6100, 2100, 4100. Then the same four numbers come round **for ever**: a loop.

D3. `0000`, and then `0` for ever ($0^2 = 0$).

D4. **Yes**: the rule has no luck in it. The same start always gives the same numbers. That is why they are only *pseudo*-random.

E1. `3 2 4 6`, then `1 1 5 1`. The seed is set once; the second loop goes on from where the first stopped.

E2. `3 3 3 3 3`. The seed goes back to 7 before every roll, so every roll is the *first* roll after `seed(7)`. Never put the seed inside the loop.

E3. **Cannot know** without running it: seed 8 is a new start. But every run with seed 8 prints the same number.

E4. **Yes** (with the same Python): `3 2 4 6` for both. A seed makes a run you can share and repeat.

F1. **6/36** = 0.1667 (the lecture's answer).

F2. **6/36**: the diagonal. Same chance as a 7.

F3. **1/36**: only 6 and 6.

F4. **6/36**: three 10s, two 11s, one 12.

F5. **15/36**: 36 pairs, minus 6 doubles, is 30; half of them have `a` bigger.

F6. **0/36**: it can never happen.

F7. **18/36** = 0.5.

F8. **7** (6 boxes). `randint(2, 12)` gives each of 2 to 12 the *same* chance, 1/11; with two dice, 12 is 1 box but 7 is 6 boxes.

G1. `0`; `n`; `1, 6`; `a == b`; `hits / n`. About **0.1667** (6/36).

G2. Line 6: `if a + b >= 10:`. About 0.1667 again (6/36, Part F4).

G3. `turn(h) == 0`. "Play one game" and "it worked" are one line here. (For $h = 20$ it is about 0.62: most turns lose!)

H1. Left: **YES** (4 sevens in 10 can happen); **NO** (10 tries give tenths only); **YES** (10 sevens in a row: very rare, but possible); **NO** (hits can never be more than `n`). Right: **YES** (1 in 4); **NO** (quarters only); **YES** (one try, not a 7); **ZeroDivisionError** (`0 / 0`).

H2. **2**; **1,000,000**; **100,000,000**.

H3. 100,000,000 tries: about **100 seconds**. 6 digits need 10,000 times more: 1,000,000 seconds, about **12 days**. Each digit costs 100 times more.

I. 1: **0.0 or 1.0** — all 1,000 tries use the same two dice (the lecture's Bug Hunt). 2: **0.0 or 0.001** — `hits` goes back to 0 on every try, so only the last try counts. 3: **0.0 or 0.001** — it returns after the first try. 4: **0.0** — `hits + 1` works out a number and throws it away; `hits` stays 0. 5: **about 167** — the hits, not the chance (no divide). 6: **about 0.122** — still 6 pairs make 7, but out of $7 \times 7 = 49$. 7: **about 0.091** — every total 2 to 12 gets 1/11 (Part F8). 8: **0.0** — the total is always even, never 7.

J1. `12`, 3 rolls ($3 + 4 + 5$; the 2 is never rolled). `0`, 3 rolls (lost the 8). `10`, 3 rolls (exactly 10: `10 < 10` is False). `0`, 1 roll. `24`, 4 rolls. `6`, 1 roll (one roll reaches 6; the 1s are never rolled). `0`, **0** rolls: `0 < 0` is False, so the loop never runs and line 8 gives 0.

J2. **15** (5 2 2 6), **0** (6 2 1), **10** (2 4 4), **0** (3 1). Each call goes on from the dice the last one left: the lecture's three calls, and one more.

K. 1: A **7**, B **10** — `break` leaves the loop but keeps the points (the lecture's Bug Hunt). 2: A **7**, B **10** — the same bug in other words. 3: A **13**, B **10** — a 1 sets `total` to 0, but line 7 still adds the 1, then $1 + 6 + 6$; the turn never loses. 4: A **0**, B **12** — it rolls again at exactly 10: a different plan ("more than h"). 5: A **0**, B **10** — **same**: the 1 is added, but then the turn gives 0 anyway. The only harmless change. 6: A **12**, B **12** — one die, 4, used again and again: 4, 8, 12. 7: A **4**, B **4** — it returns after one roll. 8: A **0**, B **10** — right here by luck: on A the 1 is added (8), then the 6 loses. A 6 now ends the turn, a 1 does not.

L1. `15, 15, 25, 25`; it returns $25/4 =$ **6.25**.

L2. **0.0**: `total` keeps only the last turn (0), then 0/4.

L3. **ZeroDivisionError**: no turns, `0 / 0`.

L4. (a) **30** (7.58 against 6.86). (b) The turn goes on longer, so a **1** is more likely to come and wipe it out. (c) **Luck**: 100,000 turns give about 2 or 3 correct digits, not the exact answer.

M1. **ROLL** ($1 < 3.3$); **ROLL** ($19 < 20$); **either** (20/6 against 20/6; the lecture stops); **STOP**.

M2. **35**. Loss $t/8$ is less than gain 35/8 while $t < 35$.

M3. **9**: $2t < 18$ means $t < 9$. More danger, so stop much sooner.

N1. Top: **in** (0.5), **out** (1.17), **in** (exactly 1.0, and `<=` lets it in), **in** (0.90). Bottom: **out** (1.01), **in** (0), **in** (0.98), **out** (1.13).

N2. 5 inside: $4 \times 5/8 =$ **2.5**. Only 8 darts: far too few tries.

N3. **3.2**; **3.12**; **3.14**.

N4. At -1 (4 Ts, 3 Hs): **1** step from home, on the left.

N5. -3, -1, 1, 3. Three ± 1 steps always add up to an **odd** number: it cannot be home after 3 steps.

O1. e no 8, s no 7, i yes 7, a yes 7, r yes 7, n yes 7: **won** after **n** (r, a, i, n all guessed). `t` and `o` are never tried.

O2. e 8, s 7, i 5, a 3, r 2, n 1, t 0: **lost** after **t**. The `o` and `d` were still to come. A short word can be hard.

O3. **0.22** (22%) and **0.015** (1.5%); about **15** times ($22{,}223 \div 1{,}484 \approx 15$).

P1. There are 52! ways to deal the cards: far too many to list one by one, so he played and counted.

P2. With two dice we know the answer (6/36), so we can check that the code is right before we trust it where nobody knows the answer.

P3. With a seed the same numbers come every run, so the crash comes every run too, and you can see that your fix works.

Q1. A `while` loop: `count = 0`, then roll and add 1 to `count` until the roll is 6, then return `count`. `average_rolls` is `average_points` with `rolls_to_six()` in place of `turn(h)`. About **6**: a 6 comes 1 time in 6.

Q2. The recipe with `if walk(steps) == 0:` as "it worked". For 10 steps it is about **0.25**; for any odd number of steps it is exactly **0.0** (Part N5).