Python and SQL: intro / SQL platforms

Ewa Weychert

Class 3: Loops

What is a loop?

- A **loop** repeats a block of code multiple times.
- Python has two primary loops:
  - `for` loop: iterate over items of an *iterable*.
  - `while` loop: repeat *while* a condition is true.
- Use loops to avoid duplication and express iteration clearly.

for-loop basics

```
for item in iterable:
    # do something with item
```

- `iterable`: objects you can loop over (lists, tuples, strings, dicts, files, generators, etc.).
- The loop assigns each element to `item` in order.

Iterating with `range()`

```
for i in range(5):          # 0..4
    print(i)

for i in range(2, 7):       # 2..6
    print(i)

for i in range(10, 0, -2): # 10,8,6,4,2
    print(i)
```

- `range(start, stop, step)` generates integers lazily.
- Common for counted loops and indexing (though `enumerate` is often better).

Iterating sequences

```
nums = [3, 1, 4]
for n in nums:
    print(n)

text = "loop"
for ch in text:
    print(ch.upper())
```

- Loop directly over items instead of indices when possible.
- Works for lists, tuples, sets (unordered), strings, and more.

`enumerate()`: index + value

```
words = ["spam", "eggs", "ham"]
for i, w in enumerate(words, start=1):
    print(i, w)
```

- Adds a running index without manual counters.
- `start=` lets you choose the initial index.

`zip()`: iterate in parallel

```
names = ["Ada", "Linus", "Guido"]
langs = ["Python", "C", "Python"]

for name, lang in zip(names, langs):
    print(f"{name} -> {lang}")
```

- `zip` stops at the shortest input.
- Use `itertools.zip_longest` to fill to the longest.

Iterating dictionaries

```
grades = {"Ana": 5, "Bob": 4, "Cid": 3}

for key in grades:
    print(key)                  # keys (default)

for key, val in grades.items():
    print(key, val)             # key-value pairs

for val in grades.values():
    print(val)                  # values only
```

while-loop basics

```
n = 1
while n*n <= 1000:
    print(n)
    n += 1
```

- Use when the number of iterations is not known in advance.
- Ensure the loop variable changes so the loop eventually ends.

break, continue, pass

```
for x in range(10):
    if x == 5:
        break       # exit loop
    if x % 2 == 0:
        continue    # skip even numbers
    # placeholder you can fill later
    pass
    print(x)        # prints 1,3
```

Loop `else` clause

```
for n in range(2, 10):
    for d in range(2, n):
        if n % d == 0:
            break             # found a divisor
    else:
        print(n, "is prime")# no break => else runs

# while ... else works similarly
```

- `else` runs only if the loop wasn't exited via `break`.
- Great for search tasks (found vs. not found).

Nested loops

```
for r in range(1, 4):
    for c in range(1, 4):
        print(r, c, end=" ")
    print()
```

- Useful for grids, matrices, and combinations.
- Watch out for time complexity: $O(n^2)$ or worse.

List comprehensions vs. loops

```
# Loop
squares = []
for x in range(10):
    squares.append(x*x)

# Comprehension
squares2 = [x*x for x in range(10)]
```

- Comprehensions are concise and often faster.
- Prefer for simple map/filter; use loops for complex logic.

Generator expressions & iterators

```
# generator expression (lazy)
total = sum(x*x for x in range(10))

# custom iterator
class CountDown:
    def __init__(self, start): self.cur = start
    def __iter__(self): return self
    def __next__(self):
        if self.cur <= 0: raise StopIteration
        self.cur -= 1
        return self.cur + 1
```

Looping over files (streaming)

```
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:
        line = line.strip()
        if not line:
            continue
        # process the line
```

- File objects are iterables that yield lines lazily.
- Use `with` to ensure the file is closed.

Handling errors inside loops

```
data = ["42", "x", "100"]
nums = []
for s in data:
    try:
        nums.append(int(s))
    except ValueError:
        # log and continue
        continue
```

- Use `try/except` to skip or correct bad items.

Progress monitoring & limits

- For long loops, show progress (e.g., `tqdm`) and consider timeouts.

```
# pip install tqdm
from tqdm import tqdm

for i in tqdm(range(10_000_000)):
    ...  # work
```

- Use counters, early exits, and checkpoints for robustness.

Performance tips & pitfalls

- Minimize work inside tight loops; hoist invariants.
- Prefer local variables; avoid repeated attribute lookups.
- Use built-ins (`sum`, `min`, `any`, `all`).
- Consider vectorization (NumPy) for numeric heavy loops.

Practical example

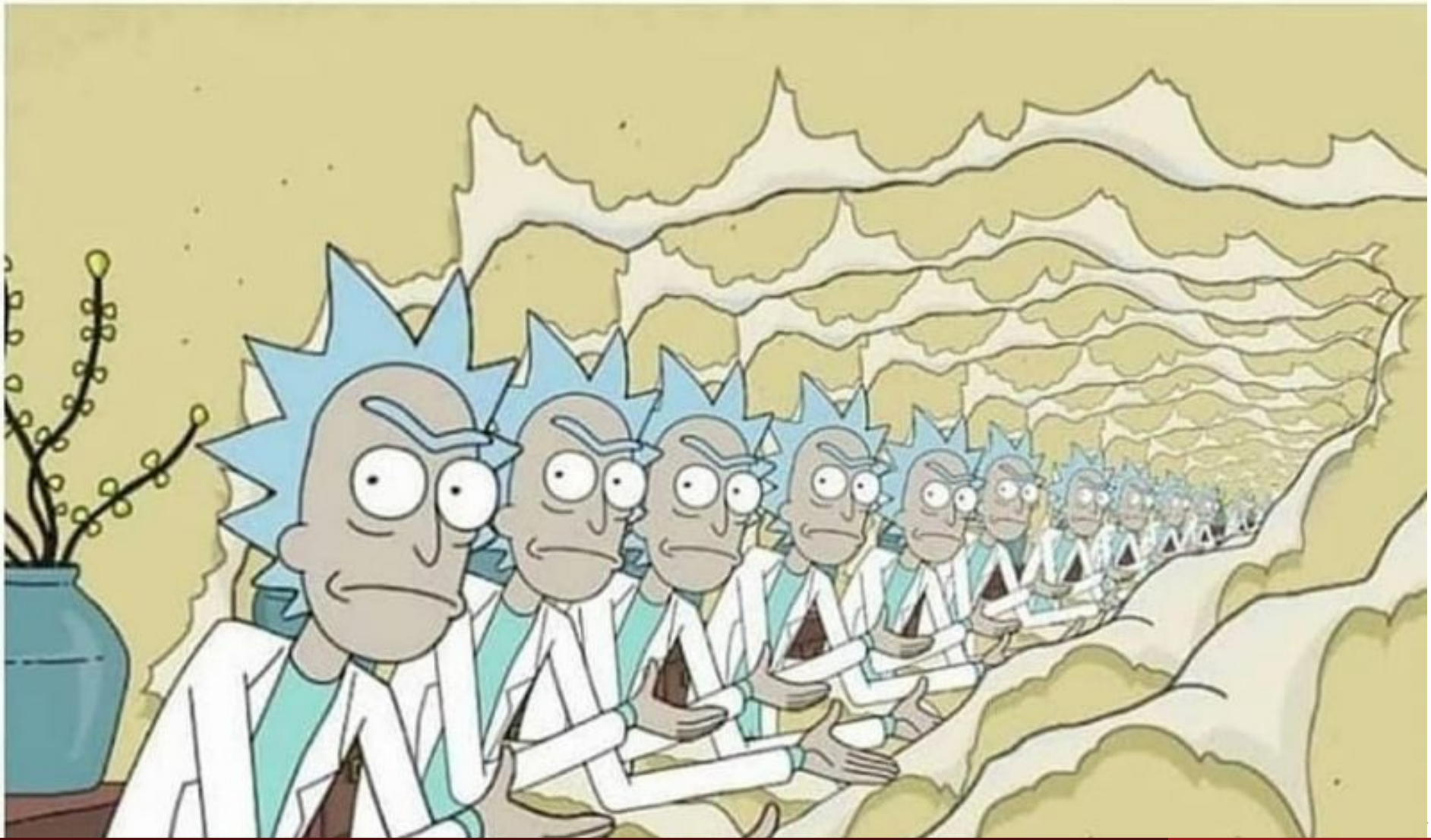
```
nums = range(1, 21)
evens = []
odds = []
for n in nums:
    (evens if n % 2 == 0 else odds).append(n)

print("Evens:", evens)
print("Odds:", odds)
```

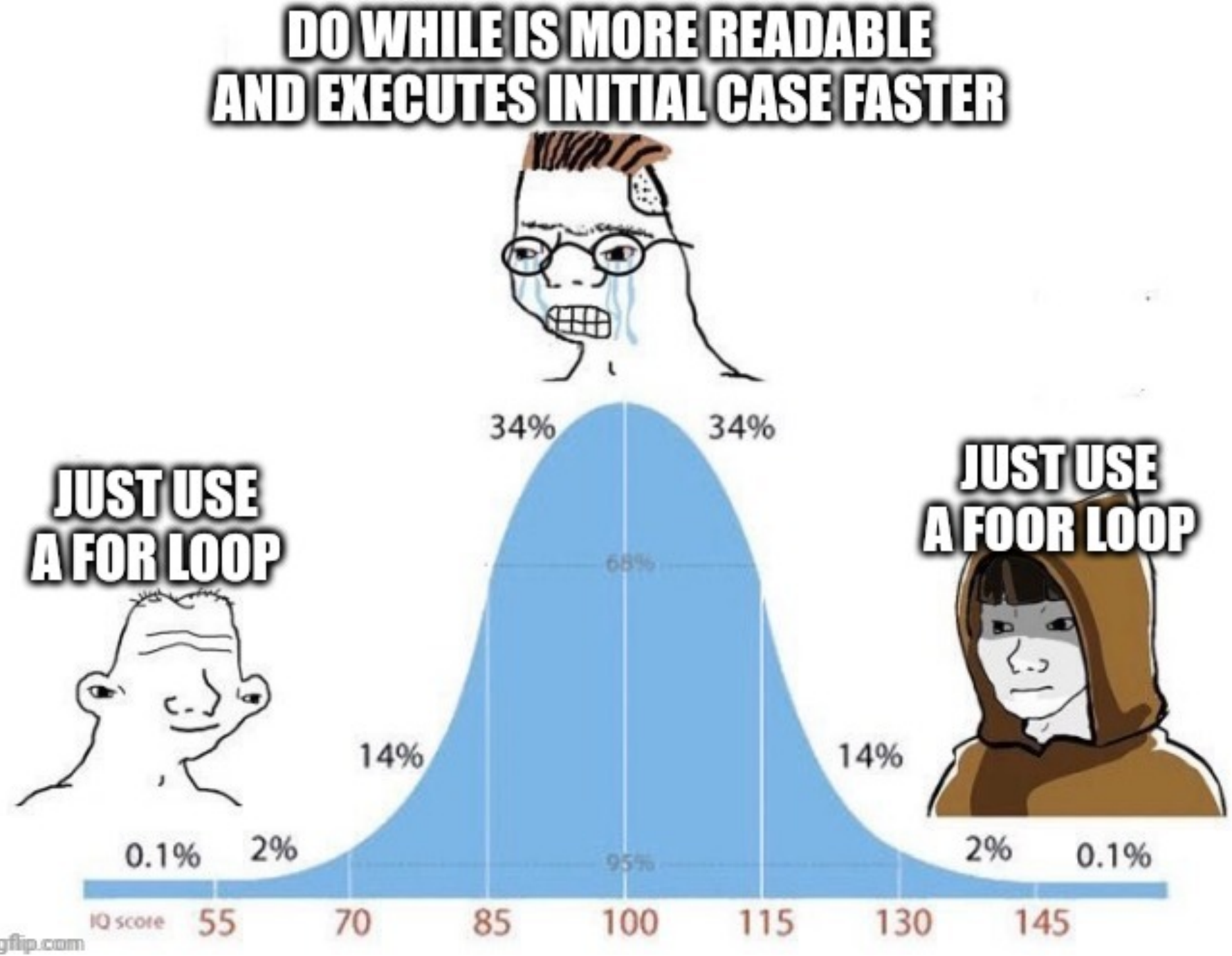
- Demonstrates conditional logic inside a loop.

Recap & cheat sheet

- `for x in iterable: ...`, `while cond: ...`
- Helpers: `range`, `enumerate`, `zip`
- Controls: `break`, `continue`, `else`
- Patterns: dict iteration, nested loops, file loops
- Alternatives: comprehensions, generators, vectorization

When you forget to break out of the while loop




DO WHILE IS MORE READABLE
AND EXECUTES INITIAL CASE FASTER
JUST USE
A FOR LOOP
JUST USE
A FOOR LOOP
34%
34%
14%
14%
68%
95%
0.1%
2%
2%
0.1%
IQ score
55
70
85
100
115
130
145
imgflip.com

Thank you!

See you next week

e.weychert@uw.edu.pl

UNIWERSYTET WARSZAWSKI | WYDZIAŁ NAUK EKONOMICZNYCH