The code (for reference)

```
BLOCKED_TERMS = {'suicide','bomb','explode','terror','kill','illegal'}


def contains_blocked(text: str) -> bool:
   if not text: return False
   t = text.lower()
   return any(term in t for term in BLOCKED_TERMS)


def sanitize_output(text: str) -> str:
   if not text: return text
   out = text
   for term in BLOCKED_TERMS:
      out = out.replace(term, '[redacted]')
   return out
```

Step-by-step execution flow

1) When the module loads

1. Python evaluates the right-hand side and creates a **set** named BLOCKED_TERMS containing the six strings:
   {'suicide', 'bomb', 'explode', 'terror', 'kill', 'illegal'}.
   - A set gives O(1) average membership tests and ensures each term is unique.

---

2) Calling contains_blocked(text)

Suppose you call contains_blocked(input_text).

2.1. **Check for empty input**

- if not text: return False — if text is None or an empty string ("") the function returns False immediately. No further work is done.

2.2. **Lowercase conversion**

- t = text.lower() — convert the entire input to lowercase so matching is case-insensitive. Example: "Bomb" → "bomb".

2.3. **Search blocked terms**

- return any(term in t for term in BLOCKED_TERMS)

  - This creates a generator that, for each term in BLOCKED_TERMS, checks whether that substring appears anywhere inside t.
  - any(...) short-circuits on the first True. If any blocked term is found, the function returns True; otherwise False.

Examples

- contains_blocked("I will kill the spider") → returns True (because "kill" is present)
- contains_blocked("This is illegal activity") → True
- contains_blocked("I like sunshine") → False
- contains_blocked("") or None → False (handled by the early return)

Edge note: This is a *substring* check — it matches inside other words (e.g., "skill" contains "kill"), which may or may not be desirable.

---

3) Calling sanitize_output(text)

Suppose you call sanitize_output(input_text).

3.1. **Check for empty input**

- if not text: return text — if text is falsy (None or ""), it returns it untouched.

3.2. **Copy text to out**

- out = text — working copy so the original text variable is not modified (strings are immutable anyway).

3.3. **Loop and replace**

- for term in BLOCKED_TERMS: iterates over the set (order arbitrary).
- out = out.replace(term, '[redacted]') replaces **all occurrences** of the exact substring term with "[redacted]". This replacement is case-sensitive — it replaces only exact-match substrings, so if the original text had "Bomb" (capital B), .replace("bomb", ...) will **not** replace it unless you pre-normalize or handle case.

3.4. **Return sanitized string**

- After the loop finishes for all terms, return out.

Examples

- sanitize_output("He built a bomb") → "He built a [redacted]"
- sanitize_output("I might KILL him") → "I might KILL him" — **note**: uppercase "KILL" remains because .replace("kill", ...) won't match "KILL".
- sanitize_output("") → ""

Important behavioral notes

- Because BLOCKED_TERMS is iterated in arbitrary set order, the sequence of replacements is non-deterministic across runs (not usually a huge deal for distinct terms but can matter if one term is substring of another).
- Replacement is blind — no word-boundary detection, so "skill" becomes "s[redacted]" if "kill" is a blocked term (because "kill" occurs inside "skill").
- Case-sensitivity mismatch between contains_blocked (lowercasing, case-insensitive) and sanitize_output (case-sensitive) can lead to surprising results.

Example run (full)

Input: "He tried to explode the old kiln but failed — illegal act."

contains_blocked(...):

- Lowercases text and finds "explode" and "illegal" → returns True.

sanitize_output(...):

- Replaces occurrences of "explode" and "illegal" (only exact lowercase matches). If input used exact lowercase, both replace — otherwise some may remain.

Common pitfalls & edge cases

- **Partial-word matches:** "skill" contains "kill" → could incorrectly redact part of a benign word.
- **Case sensitivity mismatch:** contains_blocked is case-insensitive; sanitize_output is not. That leads to detection without redaction (e.g., "KILL" detected but not replaced).
- **Order of replacement:** Using a set makes order unpredictable; no guarantee which term is replaced first.
- **Unicode / accents / normalization:** Some inputs may contain Unicode variants (e.g., zero-widths, diacritics) that break matching.
- **Context matters:** The presence of a word like "suicide" often requires safe-handling (trigger helpers, not just redaction).