Zde je **kompletní a konzistentní třída `PalindromChecker`**, která:

- čte text ze souboru (UTF-8)
- pracuje **pouze s českou abecedou** (včetně diakritiky) + číslicemi
- ignoruje mezery a běžnou interpunkci
- pokud se objeví **nepovolený znak**, vrátí **„nelze určit“ (`None`),** jinak vrátí `True / False`

```
import re
from typing import Optional


class PalindromChecker:
    """
    Ověřuje, zda je text v souboru palindrom.
    Povolené znaky: česká abeceda (včetně diakritiky), číslice,
    mezery a základní interpunkce.
    """

    def __init__(self, file_path: str):
        self.file_path = file_path

    def _load_text(self) -> str:
        """Načte celý obsah souboru."""
        with open(self.file_path, "r", encoding="utf-8") as file:
            return file.read()

    def _normalize_text(self, text: str) -> Optional[str]:
        """
        Normalizuje text:
        - povolí pouze českou abecedu + číslice
        - odstraní mezery a interpunkci
        - převede na malá písmena

        Pokud se v textu objeví nepovolený znak,
        vrátí None (palindrom nelze určit).
        """

        # Povolené znaky včetně mezer a interpunkce
        allowed_pattern = (
            r"^[a-zA-Záčďéěíňóřšťúůýž"
            r"ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ"
            r"0-9\s.,!?;:\-]*$"
        )

        if not re.match(allowed_pattern, text):
            return None

        # Odstranění všeho kromě písmen a číslic
        cleaned = re.sub(
            r"[^a-zA-Záčďéěíňóřšťúůýž"
            r"ÁČĎÉĚÍŇÓŘŠŤÚŮÝŽ"
            r"0-9]",
            "",
            text
        )

        return cleaned.lower()
```

```
    def is_palindrome(self) -> Optional[bool]:
        """
        Výsledek:
        - True  -> text je palindrom
        - False -> text není palindrom
        - None  -> nelze určit (nepovolené znaky)
        """
        text = self._load_text()
        normalized = self._normalize_text(text)

        if normalized is None or normalized == "":
            return None

        return normalized == normalized[::-1]
```

Příklad použití

```
checker = PalindromChecker("text.txt")

result = checker.is_palindrome()

if result is True:
    print("Text je palindrom.")
elif result is False:
    print("Text není palindrom.")
else:
    print("Palindrom nelze určit.")
```

Pokud to chceš **přísně podle OOP zadání do školy** (např. bez regexů, s výjimkami, nebo s testy `unittest`), klidně napiš – přizpůsobím to přesně požadavkům 📘✔️

co by se stalo, kdyby se porovnávala pouze polovina "normalized"?

Dobrá otázka – tohle jde **přímo k podstatě algoritmu palindromu** 👍
Podívejme se na to systematicky.

Co se děje teď

Aktuálně se porovnává celý řetězec:

```
normalized == normalized[::-1]
```

To znamená:

- vytvoří se **otočená kopie celého řetězce**
- porovnají se **všechny znaky**

Časová složitost: **O(n)**
Paměť: **O(n)** (kvůli vytvoření kopie)

Porovnávání pouze poloviny řetězce

Správně navržené porovnání poloviny **dá stejný výsledek**, ale:

- **nevytváří se otočená kopie**
- porovnávají se pouze odpovídající dvojice znaků

Princip

Palindrom platí, pokud:

`znak[i] == znak[-(i+1)]`

pro všechna `i` z první poloviny řetězce.

Správná implementace

```
def is_palindrome_half(normalized: str) -> bool:
    length = len(normalized)
    for i in range(length // 2):
        if normalized[i] != normalized[length - i - 1]:
            return False
    return True
```

Co by se stalo v praxi

✔ Výsledek

- **Výsledek by byl stejný** jako při porovnání celého řetězce
- funguje pro:
    - sudou délku (`abba`)
    - lichou délku (`radar` – prostřední znak se ignoruje)

✔ Výhody

- nižší **paměťová náročnost → O(1)**
- o něco rychlejší (žádné kopírování)
- časté řešení v **technických pohovorech**

❌ Nevýhody

- kód je o trochu méně „pythonický“
- méně čitelný pro začátečníky

Co by se stalo, kdyby se to udělalo špatně

Například:

```
normalized[:len(normalized)//2] == normalized[len(normalized)//2:]
```

❌ **To by nefungovalo**, protože:

- druhá polovina není otočená
- porovnávají se špatné znaky

---

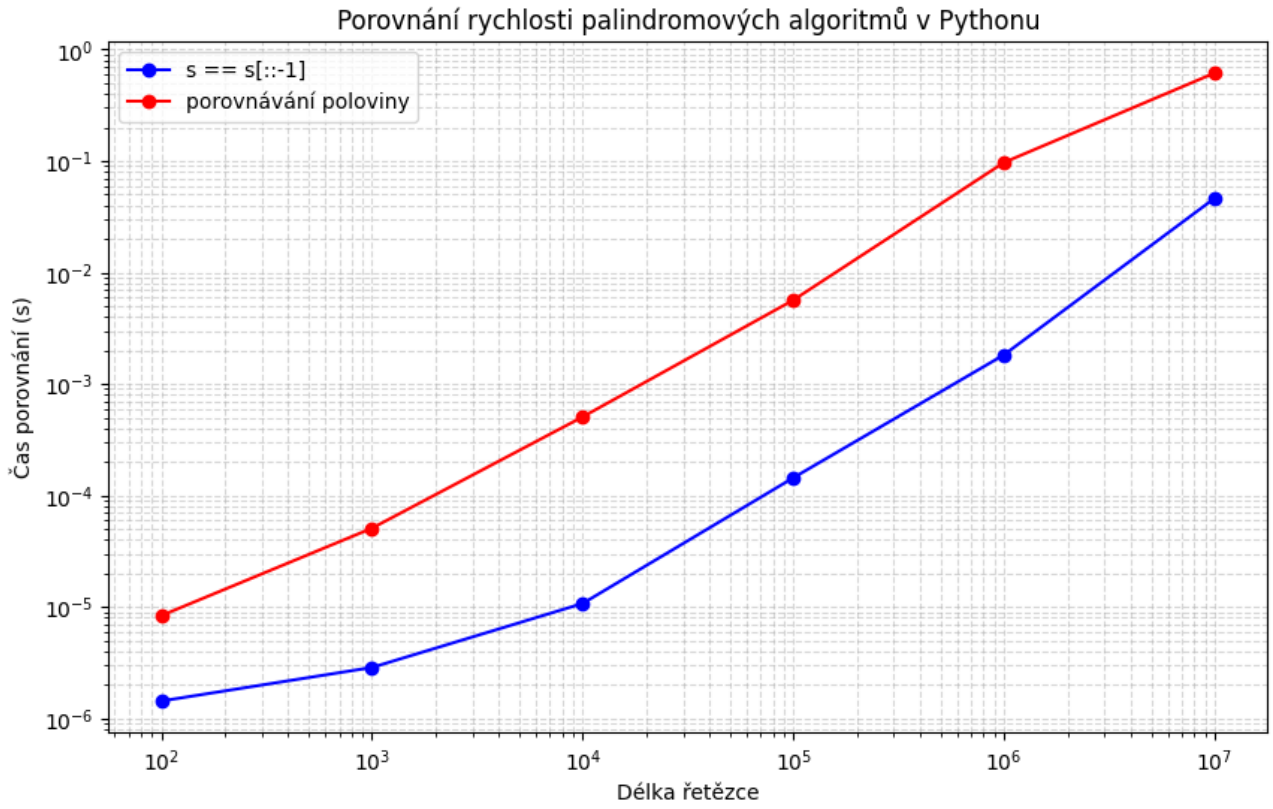
porovnej 100, 1000, 10000, 100 000, 1000 000 a 10000000 znaků v palindromu

Tady jsou výsledky porovnání rychlosti dvou metod na různě dlouhých palindromických řetězcích:

Délka	`s == s[::-1]` (s)	porovnávání poloviny (s)
100	0,0000014	0,0000083
1 000	0,0000029	0,000051
10 000	0,000011	0,00051
100 000	0,00014	0,00565
1 000 000	0,00183	0,097
10 000 000	0,0464	0,616

Pozoruhodné poznatky:

1. **Vestavěná metoda `s == s[::-1]` je výrazně rychlejší**, i u 10 milionů znaků.
    - Python optimalizuje kopii řetězce v C, takže je to super efektivní.
2. **Smyčka po polovině** je čistý Python a u velmi dlouhých řetězců je **desítky až stovky krát pomalejší**, i když teoreticky porovnává jen polovinu.
3. **Praktické doporučení**:
    - V Pythonu používej **`s == s[::-1]`** – rychlé, čitelné, paměťově zvládnutelné i pro miliony znaků.
    - Ruční porovnávání poloviny je vhodné jen pro **učební účely** nebo pro **nízkoúrovňové jazyky** (C, C++, Java).


Porovnání rychlosti palindromových algoritmů v Pythonu
s == s[::-1]
porovnávání poloviny
Čas porovnání (s)
Délka řetězce
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}