



ChatGPT

{ "answer": { "python": { "best_practices": [{ "pattern": "Stosuj automatyczne formatowanie i linting zgodnie z PEP 8", "why": "Ujednolicony styl kodu poprawia czytelność i utrzymanie. Automatyczne narzędzia eliminują błędy stylistyczne i typowe pomyłki 1 2 .", "how": "Skonfiguruj formatowanie (np. Black) i linter (np. Ruff) w edytorze lub CI. Narzędzia te automatycznie dostosują kod do standardów PEP 8 i zgłoszą naruszenia.", "when": "Zawsze – od początku projektu. Wprowadzenie formatowania i lintingu wcześnie zapobiega nagromadzeniu długiego technicznego związanego ze stylem.", "pitfalls": "Nie wyłączaj linterów bez powodu – mogą wykryć realne problemy. Unikaj też \"kosmetycznych\" zmian stylu w środku prac, by nie zaciemnić historii zmian.", "snippet_python": "def policz_srednia(liczby: list[float]) -> float:\n \"\"\"\n Zwraca średnią arytmetyczną listy liczb.\n \"\"\"\n assert liczby, \"Lista liczb nie może być pusta\"\n return sum(liczby) / len(liczby)\n", "refs": ["1", "2"], { "pattern": "Używaj podpowiedzi typów i analizatorów statycznych", "why": "Podpowiedzi typów ułatwiają zrozumienie kodu i wykrywanie błędów przed uruchomieniem. Narzędzia jak mypy wyłapują niezgodności typów, zapobiegając błędem wykonania 3 .", "how": "Dodawaj adnotacje typów do definicji funkcji, zmiennych i zwracanych wartości. Uruchamiaj mypy (lub inny checker) w trakcie developmentu lub w CI, aby sprawdzić zgodność typów w całym projekcie.", "when": "Przy rozbudowanych projektach, bibliotekach publicznych i wszędzie tam, gdzie błędy typów mogą być kosztowne. Można stopniowo wprowadzać type hints nawet do istniejącego kodu.", "pitfalls": "Brak uruchamiania checkera (same adnotacje nie wystarczą). Ignorowanie ostrzeżeń mypy może skutkować błędnym założeniem o typach. Należy też unikać nadużywania typów kosztem czytelności w prostych skryptach.", "snippet_python": "from typing import Optional\n\nclass User:\n def __init__(self, name: str, age: Optional[int] = None):\n self.name = str\n self.age = Optional[int] = age\n\n def przedstaw_sie(self) -> str:\n return f\"Jestem {self.name}, mam {self.age} lat.\n \" if self.age is not None else f\"Jestem {self.name}.\"\n", "refs": ["3"], { "pattern": "Optymalizuj wydajność poprzez właściwe struktury danych i profilowanie", "why": "Wybór odpowiednich struktur (listy vs sety, itp.) i algorytmów znaczco wpływa na szybkość. Profilowanie ujawnia realne wąskie gardła, dzięki czemu optymalizujesz tylko to, co trzeba 4 .", "how": "Używaj wbudowanych typów i bibliotek (np. kolekcji jak deque, set) – są napisane w C i często szybsze. Mierz czas wykonania modułem cProfile lub timeit przed i po zmianach. Złożone operacje na dużych danych przenieś do numpy/C, jeśli to krytyczne.", "when": "Gdy aplikacja zaczyna zwalniać lub wymaga obsługi dużych zbiorów danych. Najpierw upewnij się, że działa poprawnie, potem profiluj i usprawniaj najwolniejsze fragmenty.", "pitfalls": "Przedwczesna optymalizacja bez danych – można skomplikować kod bez realnych korzyści. Nieodpowiedni dobór struktury (np. linearne wyszukiwanie w liście zamiast set) bywa ukrytym problemem, stąd znaczenie profilowania.", "snippet_python": "import math\n\n def znajdz_pierwsze(n: int) -> list[int]:\n \"\"\"\n Zwraca listę liczb pierwszych mniejszych od n.\n \"\"\"\n primes: list[int] = []\n sieve = [True] * n\n\n for p in range(2, int(math.sqrt(n)) + 1):\n if sieve[p]:\n for x in range(p*p, n, p):\n sieve[x] = False\n\n return primes\n", "refs": ["10"], { "pattern": "Stosuj wielowątkowość lub asynchroniczność dla zadań wejścia/wyjścia", "why": "Python umożliwia równoległe wykonywanie zadań I/O (sieć, plik) dzięki asyncio lub wątkom, co znacznie przyspiesza obsługę wielu żądań jednocześnie. Asyncio eliminuje blokowanie na I/O, zwiększając przepustowość serwisów.", "how": "Dla zadań I/O (np. zapytania HTTP, operacje plikowe) użyj asyncio i await, by wykonywać wiele operacji naraz. Przy CPU-bound użyj multiprocessing lub zewnętrznych bibliotek (NumPy, Cython). Zawsze ograniczaj liczbę wątków lub zadań asynchronicznych do poziomu, który system może obsłużyć.", "when": "Aplikacje serwerowe (serwisy web, API), scrapery sieciowe, przetwarzanie wielu plików – wszędzie tam, gdzie program czeka na I/O. Nie stosuj wątków/async bez potrzeby przy prostych skryptach jednorazowych.", "pitfalls": "Złożoność współprzeźności – debugowanie bywa trudne. Wątki nie przyspieszą operacji CPU (ograniczenie GIL). Z kolei asynchroniczność wymaga, by biblioteki I/O jeju używały – mieszanie synchronicznych wywołań z async spowoduje blokady.", "snippet_python": "import asyncio\nimport time\n\n @asyncio.coroutine\n def pobierz_dane(url: str) -> str:\n \"\"\"\n Symulacja opóźnionego pobierania\n \"\"\"\n await asyncio.sleep(1)\n return f\"<dane z {url}>\"\n\n @asyncio.coroutine\n def pobierz_wszystko(urls: list[str]) -> list[str]:\n \"\"\"\n Symulacja opóźnionego pobierania\n \"\"\"\n tasks = [\n pobierz_dane(url)\n for url in urls\n]\n return await asyncio.gather(*tasks)\n", "refs": ["11"] } } }

przy obróbce dużych zbiorów danych.", "Buforuj wyniki kosztownych obliczeń (memoizacja, `functools.lru_cache`) zamiast liczyć je wielokrotnie.", "Stosuj wielowątkowość/asynchroniczność dla zadań I/O i multiprocessing dla zadań CPU-bound, by lepiej wykorzystać zasoby."], "security": ["Waliduj wszystkie dane wejściowe (typ, format, zakres) – nie zakładaj, że klient przesyła poprawne dane.", "Przechowuj sekrety (hasła, klucze API) poza kodem – np. w zmiennych środowiskowych lub menedżerach sekretów.", "Ustaw minimalne uprawnienia dla procesów i operacji (zasada najmniejszych uprawnień).", "Stosuj bezpieczne funkcje wyjścia: parametryzuj zapytania SQL, escapuj dane w HTML itp., aby uniknąć wstrzyknięć.", "Loguj incydenty bezpieczeństwa (np. wielokrotne błędne logowania), ale nie loguj poufnych danych w formie jawniej."], "maintainability": ["Przestrzegaj PEP8 i jednolitego formatowania kodu w całym projekcie.", "Nazywaj zmienne, funkcje i klasy zrozumiale, zgodnie z ich przeznaczeniem.", "Unikaj \"magicznych\" stałych w kodzie – używaj nazwanych stałych lub konfiguracji.", "Pisz docstringi dla modułów, klas i funkcji, dokumentując ich działanie i oczekiwane parametry.", "Regularnie eliminuj duplikacje i dziel zbyt rozbudowane funkcje na mniejsze – ciągła refaktoryzacja utrzymuje kod czystym."], "tests": ["Pisz testy jednostkowe dla kluczowych funkcji i modułów (pokrywające główną logikę).", "Testuj przypadki brzegowe i niepoprawne dane wejściowe, nie tylko \"szczęśliwą ścieżkę\".", "Wykorzystuj pytest z parametryzacją i fiksturami, aby łatwo pokryć wiele wariantów wejść w testach.", "Włącz testy do CI/CD – każda zmiana kodu powinna automatycznie przechodzić wszystkie testy.", "Stosuj testy property-based (np. Hypothesis) dla złożonych funkcji – mogą wykryć nieoczywiste błędy."] }, "tools": ["Black", "Ruff", "mypy", "pytest", "Hypothesis", "Bandit", "Pydantic"] }, "json": { "best_practices": [{ "rule": "Stosuj jednolitą konwencję nazw kluczy", "rationale": "Użycie spójnego stylu (np. wyłącznie camelCase lub wyłącznie snake_case) ułatwia konsumentom odczyt i integrację danych. Konsekwentne nazewnictwo sprawia, że JSON wygląda jak z jednego źródła" }] }, "example_json": { "rule": "Mixowanie stylów: { \"user_name\": \"JanKowalski\", \"UserID\": 123 }", "anti_pattern": "Różne konwencje w jednym JSON.", "rationale": "Przekazuj liczby jako liczby, booleany jako true/false, a nie np. tekst \"true\". Trzymaj listy elementów w tablicach JSON, zamiast łączyć je w pojedynczy string. Dzięki temu parsery i programiści łatwiej przetwarzają dane bez dodatkowego parsowania.", "refs": ["7"] }, "example_json": { "rule": "Zachowuj poprawne typy danych w polach", "anti_pattern": "Nieprawidłowe: { \"count\": \"5\", \"active\": \"false\", \"tags\": \"news,release\" }", "rationale": "Jeśli pole może nie mieć wartości, zdecyduj czy lepiej je pominąć, czy dać null – i stosuj jedną logikę. W wielu przypadkach brak pola i null powinny znaczyć to samo, aby uprościć logikę klienta. Jeżeli null niesie specjalne znaczenie, rozważ zamiast tego użycie innego typu (np. string \"unknown\")" }, "example_json": { "rule": "Używaj null świadomie i konsekwentnie", "anti_pattern": "Interpretowanie null i braku pola jako dwóch różnych stanów (prowadzi to do niejednoznaczności i błędów po stronie odbiorcy).", "rationale": "Jeśli pole może nie mieć wartości, zdecyduj czy lepiej je pominąć, czy dać null – i stosuj jedną logikę. W wielu przypadkach brak pola i null powinny znaczyć to samo, aby uprościć logikę klienta. Jeżeli null niesie specjalne znaczenie, rozważ zamiast tego użycie innego typu (np. string \"unknown\")" }, "example_json": { "rule": "Gdy pole zawiera listę wielu elementów, użyj tablicy JSON. Nazwa takiego pola powinna być w liczbie mnogiej (np. \"items\"). To jasno wskazuje, że dane są kolekcją. Unikaj dynamicznego tworzenia wielu pól numerycznych dla podobnych danych.", "anti_pattern": "Błędne podejście: { \"item1\": {}, \"item2\": {} } (powielanie struktury kluczy zamiast tablicy) albo użycie jednego pola z rozzielaną listą ID \"items\": \"1,2,3\".", "rationale": "Wersjonuj format JSON przy znaczących zmianach", "refs": ["7"] }, "example_json": { "rule": "Wersjonuj format JSON przy znaczących zmianach", "anti_pattern": "Cicha zmiana struktury (np. zmiana znaczenia pola albo usunięcie pola) bez komunikacji – klienci odczytujący stary format zaczną działać nieprawidłowo.", "rationale": "Podczas pracy deweloperskiej formatowanie JSON z wcięciami ułatwia analizę (można łatwo znaleźć błąd). Natomiast w środowisku produkcyjnym zaleca się wysyłanie zminimalizowanego JSON (bez wcięć i nadmiarowych" }] }] }] }

spacji), aby zmniejszyć zużycie pasma i opóźnienia.", "example_json": "{ \"id\": 1, \"name\": \"Test\" } // sformatowany\n{\"id\":1,\"name\":\"Test\"} // skompaktowany", "anti_pattern": "Wysyłanie dużych plików JSON z niepotrzebnymi znakami nowej linii i spacjami w API produkcyjnym (zwiększyły rozmiar bez korzyści). Lub odwrotnie – analiza ogromnego jednolinijkowego JSON podczas debugowania lokalnego.", "refs": [] }, "schema_tips": ["Używaj pola `$schema` w definicji – określa ono wersję standardu JSON Schema, której używasz ¹³ .", "Dodaj pole `$id` z unikalnym identyfikatorem URI dla każdego schematu – ułatwia to referencje i utrzymanie wielu schematów jednocześnie.", "Dokładnie zdefiniuj strukturę obiektu: użyj `properties` z okresemieniem `type` dla każdego pola i listą `required` dla wymaganych pól.", "Wykorzystuj współdzielenie definicji: zagnieżdżaj powtarzalne struktury w `definitions` i odwołuj się do nich przez `$ref` zamiast kopiować fragmenty.", "Dodawaj `description` do pól i schematów – opisuje znaczenie danych i pomaga innym deweloperom zrozumieć Twoje API (oraz generować dokumentację).", "Zmiany schematu projektuj z myślą o kompatybilności: nowe pola dodawaj opcjonalnie, a usunięcie lub zmianę znaczenia pola traktuj jako zmianę wersji schematu (wersjonuj schemat)."], "security_tips": ["Nigdy nie ufaj danym wejściowym – zawsze waliduj format, typy i zakres wartości zanim użyjesz JSON w swojej aplikacji.", "Ogranicz rozmiar przyjmowanego JSON (liczbę elementów, głębokość) i odrzucaj zbyt duże ładunki (np. HTTP 413 dla nadmiarowego body) ¹⁴ .", "Nie umieszczaj w JSON danych wrażliwych (hasał, tokenów) w postaci jawnej – jeśli musisz przekazać takie informacje, to je maskuj lub szyfruj.", "Przy przetwarzaniu bardzo dużych dokumentów JSON używaj API strumieniowego (stream) lub iteracyjnego – unikniesz zużycia całej pamięci jednorazowo (ataki DoS).", "Escape'uj dane z JSON, gdy osadzasz je w innych kontekstach (np. HTML) – zabezpieczysz się przed wstrzyknięciem szkodliwego kodu (XSS, SQL injection)."], "json_pseudocode": { "prompt_patterns": [{ "name": "Sformalizowane polecenie w formacie JSON", "when": "Gdy chcesz jednoznacznie przekazać zadanie, dane i ograniczenia do LLM w ustrukturyzowany sposób.", "json_template": "\n\"task\": \"Przetłumacz tekst na język angielski\", \n\"data\": {\"tekst\": \"Cześć świecie\"}, \n\"constraints\": {\"formality\": \"informal\"}, \n\"output_schema\": {\"type\": \"object\", \"properties\": {\"translation\": {\"type\": \"string\"}}}\n", "pseudocode": "PLAN: Odczytaj polecenie z pola 'task'. Użyj danych z 'data'. Uwzględnij ograniczenia z 'constraints'. Zwróć wynik w formacie 'output_schema'.", "test_case": "", "refs": [] }, { "name": "Planowanie rozwiązania przed odpowiedzią", "when": "Przy złożonych problemach wymagających rozumowania krok po kroku (Chain-of-Thought).", "json_template": "\n\"steps\": [\"Zidentyfikuj liczby.\", \"Dodaj je.\"], \n\"answer\": 5\n", "pseudocode": "PLAN: Najpierw wygeneruj plan rozwiązania krok po kroku zamiast od razu końcowej odpowiedzi. Następnie przedstaw finalny wynik.", "test_case": "", "refs": ["5"] }, { "name": "Wbudowane testy w treści polecenia", "when": "Gdy chcesz upewnić się, że wynik spełnia określone warunki lub przykłady (np. generowanie kodu, obliczenia).", "json_template": "\n\"task\": \"Napisz funkcję obliczającą kwadrat liczby.\", \n\"tests\": [\"input\": 2, \"expected\": 4}, \n\"input\": -3, \"expected\": 9}\n]", "pseudocode": "PLAN: Przeanalizuj dostarczone testy. Wygeneruj rozwiązanie, które dla każdego `input` zwróci oczekiwany wynik.", "test_case": "Dla powyższego polecenia model najpierw zrozumie zadanie, następnie napisze funkcję `f(x)` i zweryfikuje ją na danych testowych $2 \rightarrow 4$ i $-3 \rightarrow 9$.", "refs": [] }, { "name": "lint_rules", "when": "Generuj wyłącznie poprawny składniowo JSON (żadnych dopisków poza strukturą JSON).", "json_template": "\n\"task\": \"Stosuj podwójne cudzysłówki do nazw pól i ciągów tekstowych (zgodnie ze specyfikacją JSON).\", \n\"validation_rules\": [\"Sprawdź wyniki za pomocą parsera JSON – odpowiedź powinna się parsować bez błędów.\", \"Zweryfikuj obecność wszystkich wymaganych pól wyjściowych i typów ich wartości.\", \"Porównaj strukturę i typy danych z oczekiwany schematem (jeśli podano output_schema).\", \"Przetestuj odpowiedź na przykładowych danych (jeśli w prompt były testy lub znane oczekiwania).\", \"Upewnij się, że model nie pominął żadnego kroku wymaganego w poleceniu i że odpowiedź jest zgodna z wszystkimi constraintami.\"]}], "references": [{ "id": "1", "title": "Python Code Quality: Best Practices and Tools", "url": "https://realpython.com/python-code-quality/", "publisher_or_author": "Real Python (L. Pozo Ramos)", "published_date": "2025-03-24", "accessed_date": "" }] }] }

"2025-08-26", "key_quote": "You can use linters like Pylint, code formatters like Black and Ruff, static type checkers like mypy, and security analyzers like Bandit ② ", { "id": "2", "title": "How to Write Beautiful Python Code With PEP 8", "url": "https://realpython.com/python-pep8/", "publisher_or_author": "Real Python (J. Finer)", "published_date": "2025-01-12", "accessed_date": "2025-08-26", "key_quote": "PEP 8 gives guidelines on naming conventions, code layout, and other best practices. By adhering to PEP 8, you ensure that your Python code is readable and maintainable ① " }, { "id": "3", "title": "A Complete Guide to Python Type Hints", "url": "https://betterstack.com/community/guides/scaling-python/python-type-hints/", "publisher_or_author": "Better Stack (S. Uili)", "published_date": "2025-04-11", "accessed_date": "2025-08-26", "key_quote": "If you accidentally add a string to your list of integers, Python won't complain at runtime — but mypy will catch the issue before you even run the code ③ " }, { "id": "4", "title": "Pydantic: Simplifying Data Validation in Python", "url": "https://realpython.com/python-pydantic/", "publisher_or_author": "Real Python (H. Hoffman)", "published_date": "2023", "accessed_date": "2025-08-26", "key_quote": "Pydantic is a powerful Python library that leverages type hints to help you easily validate and serialize your data schemas, making your code more robust, readable, concise, and easier to debug ④ " }, { "id": "5", "title": "An example of LLM prompting for programming", "url": "https://martinfowler.com/articles/2023-chatgpt-xu-hao.html", "publisher_or_author": "Martin Fowler", "published_date": "2023-04-13", "accessed_date": "2025-08-26", "key_quote": "His initial prompt primes the LLM with an implementation strategy (chain of thought). His prompt also asks for an implementation plan rather than code ⑯ " }, { "id": "6", "title": "Stop begging for JSON", "url": "https://www.ignorance.ai/p/stop-begging-for-json", "publisher_or_author": "Charlie Guo (Artificial Ignorance)", "published_date": "2024-12-12", "accessed_date": "2025-08-26", "key_quote": "> IMPORTANT: Return ONLY valid JSON. Do NOT include any other text or explanations. The response MUST be a valid JSON object and NOTHING else. ⑯ " }, { "id": "7", "title": "JSON Best Practices (REST API Principles)", "url": "https://schweizerischebundesbahnen.github.io/api-principles/restful/best-practices/", "publisher_or_author": "SBB API Guidelines", "published_date": "n.d.", "accessed_date": "2025-08-26", "key_quote": "Property names are restricted to ASCII strings in lower case camelCase... It's essential to establish a consistent look and feel such that JSON looks as if it came from the same hand ⑪ " }, { "id": "8", "title": "JSON Schema – The basics", "url": "https://json-schema.org/understanding-json-schema/basics.html", "publisher_or_author": "JSON Schema Documentation", "published_date": "2025", "accessed_date": "2025-08-26", "key_quote": "It's generally good practice to include the \$schema keyword to declare which version of the JSON Schema specification the schema is written to. It should always be used in the real world ⑬ " }, { "id": "9", "title": "REST Security Cheat Sheet", "url": "https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html", "publisher_or_author": "OWASP Cheat Sheet Series", "published_date": "n.d.", "accessed_date": "2025-08-26", "key_quote": "Define an appropriate request size limit and reject requests exceeding the limit with HTTP response status 413 Request Entity Too Large ⑭ " }, { "id": "10", "title": "Python Wiki – Performance Tips", "url": "https://wiki.python.org/moin/PythonSpeed/PerformanceTips", "publisher_or_author": "Python.org Wiki", "published_date": "n.d.", "accessed_date": "2025-08-26", "key_quote": "In short: 1. Get it right. 2. Test it's right. 3. Profile if slow. 4. Optimize. 5. Repeat ⑤ " }, { "id": "11", "title": "Speed Up Your Python Program With Concurrency", "url": "https://realpython.com/python-concurrency/", "publisher_or_author": "Real Python (J. Anderson)", "published_date": "2024-11-25", "accessed_date": "2025-08-26", "key_quote": "In Python, threads and asynchronous tasks facilitate concurrency on a single processor, while multiprocessing allows for true parallelism by utilizing multiple CPU cores ⑤ " }, "research_log": { "queries": ["2024 Python best practices PEP8 type hints mypy ruff black", "Python code quality tools best practices", "JSON best practices schema validation versioning security", "OWASP JSON input size limit security", "LLM prompt structured JSON output best practices", "pseudocode prompting LLM chain-of-thought plan pseudocode", "Python data validation Pydantic best practices", "Python performance optimization profiling asyncio"], "visited": ["realpython.com/python-code-quality/", "realpython.com/python-pep8/", "betterstack.com/community/guides/scaling-python/python-type-hints/", "realpython.com/python-pydantic/", "martinfowler.com/articles/2023-chatgpt-xu-hao.html", "ignorance.ai/p/stop-begging-for-json",

"schweizerischebundesbahnen.github.io/api-principles/restful/best-practices/", "json-schema.org/understanding-json-schema/basics.html", "cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html"], "excluded_with_reason": ["\"21 ways to improve LLM outputs\" (Medium article) - paywalled content, unable to access full text.", "\"Learn JSON: The Complete Guide\" (Medium/JavaGuides) - mostly basic JSON intro, overlapped with official JSON docs, not used.", "Several low-authority SEO blogs on Python performance - skipped in favor of official Python Wiki and Real Python content."], "dedupe_notes": "Consolidated overlapping style and tooling tips (PEP8, Black, Ruff) from multiple sources into single best practice. Chose authoritative sources (Real Python, official docs, PEPs) where content overlapped with random blogs." }, "metrics": { "strategy": "Clustered research into Python, JSON, and prompt engineering. Prioritized official docs (PEPs, JSON Schema) and authoritative sources (Real Python, Fowler). Aggregated confirmed best practices from cross-verified references.", "confidence": 0.95, "tokens_est": 2032, "sources_used": 9 } }

1 How to Write Beautiful Python Code With PEP 8 – Real Python

<https://realpython.com/python-pep8/>

2 6 7 9 10 Python Code Quality: Best Practices and Tools – Real Python

<https://realpython.com/python-code-quality/>

3 A Complete Guide to Python Type Hints | Better Stack Community

<https://betterstack.com/community/guides/scaling-python/python-type-hints/>

4 PythonSpeed/PerformanceTips - Python Wiki

<https://wiki.python.org/moin/PythonSpeed/PerformanceTips>

5 Speed Up Your Python Program With Concurrency – Real Python

<https://realpython.com/python-concurrency/>

8 Pydantic: Simplifying Data Validation in Python – Real Python

<https://realpython.com/python-pydantic/>

11 12 Best Practices | API Principles

<https://schweizerischebundesbahnen.github.io/api-principles/restful/best-practices/>

13 JSON Schema - The basics

<https://json-schema.org/understanding-json-schema/basics>

14 REST Security - OWASP Cheat Sheet Series

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

15 An example of LLM prompting for programming

<https://martinfowler.com/articles/2023-chatgpt-xu-hao.html>

16 Stop begging for JSON - by Charlie Guo

<https://www.ignorance.ai/p/stop-begging-for-json>