MicroPython w systemach wbudowanych, jakość i bezpieczeństwo aplikacji

Magdalena Szymczyk
AGH Akademia Górniczo-Hutnicza, Katedra Biocybernetyki i Inżynierii Biomedycznej, al. Mickiewicza 30, 30-059 Kraków

Streszczenie: W artykule przedstawiono możliwości wykorzystania języka Python w systemach wbudowanych. Przeanalizowano różnice oraz ograniczenia w stosunku do jego wersji standardowej, zwrócono uwagę na aspekt bezpieczeństwa i jakość aplikacji do zastosowań w medycynie i w automatyce.

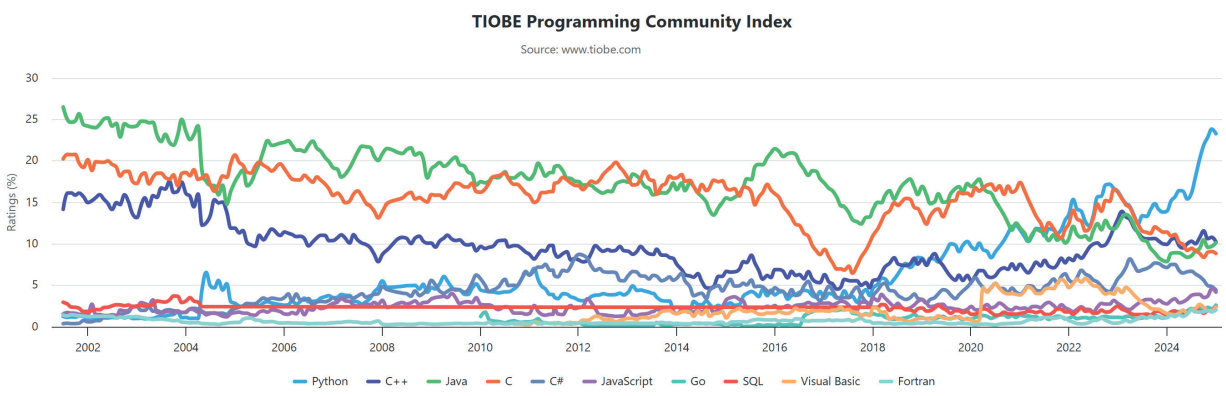
Słowa kluczowe: MicroPython, systemy wbudowane, jakość, bezpieczeństwo

1. Wprowadzenie

Systemy wbudowane od dziesięcioleci są nieodłącznym elementem systemów automatyzacji. Odgrywają kluczową rolę w automatyce, integrując zaawansowane technologie z procesami przemysłowymi i urządzeniami codziennego użytku. Są dedykowane do monitorowania, sterowania i automatycznego wykonywania operacji, są powszechnie stosowane w szerokim spektrum aplikacji. Rynek systemów wbudowanych jest jednym z najszybciej rozwijających się na świecie, zatem istnieje ogromna potrzeba wsparcia go odpowiednimi narzędziami (językami programowania) do szybkiego prototypowania, testowania i wdrażania oprogramowania tego typu.

Python jest obecnie najpopularniejszym językiem programowania na świecie, co pokazano w rankingu popularności języków programowania publikowanym przez TIOBE [9]. Został zaimplementowany w języku C (standardowy CPhyton) dla systemów ogólnego przeznaczenia (takich jak komputery desktopowe, serwery).

Na rysunku 1 oraz w tabeli 1 przedstawiono dane (styczeń 2025) o trendach i udziale języków programowania stosowanych w projektach informatycznych. Ta powszechność użycia Phytona powoduje, że jest duża dostępność programistów znających język Phyton i stosunkowo niski koszt ich zatrudnienia, a to z kolei powoduje, że mogą oni łatwo zacząć programować w MicroPythonie, co nie jest już tak szybkie w porównaniu do deweloperów pracujących w asemblerze, języku C czy C++. Nakłada się na to również coraz większa potrzeba rozwoju systemów wbudowanych i konieczność szybkiego wprowadzania ich na rynek, czyli oczekiwanie szybkiej implementacji potrzebnego kodu. Czynniki te powodują duże zainteresowanie wykorzystaniem Pythona w systemach wbudowanych, czemu naprzeciw wychodzi jego minimalistyczna implementacja w postaci MicroPythona.



Rys. 1. Wykres zmian popularności języków programowania [9]
Fig. 1. A graph of changes in the popularity of programming languages [9]

Autor korespondujący:
Magdalena Szymczyk, Magdalena.Szymczyk@agh.edu.pl

Artykuł recenzowany
nadesłany 20.012.2025 r., przyjęty do druku 24.03.2025 r.

2. MicroPython w systemach wbudowanych

MicroPython w odróżnieniu od Phytona, został zaprojektowany przez Damiena P. George w 2013 r. do pracy w systemach wbudowanych (m.in. do pracy na mikrokontrolerach o małych zasobach, z systemem operacyjnym lub bez, a nawet do implementacji aplikacji na tzw. „czystym metalu"). Ideą,

Tab. 1. Ranking popularności języków programowania – styczeń 2025 [9]
Tab. 1. Programming languages popularity ranking – January 2025 [9]

Pozycja 01.2025 r.	Pozycja 01.2024 r.	Zmiana [++, +, =, –]	Język programowania	Udział [%]	Zmian [%]
1	1	=	Python	23,28	+9,32
2	3	+	C++	10,29	+0,22
3	4	+	Java	10,15	+2,28
4	2	–	C	8,86	–2,59
5	5	=	C#	4,45	–2,71
6	6	=	JavaScript	4,20	+1,43
7	11	++	Go	2,61	+1,24
8	9	+	SQL	2,41	+0,95
9	8	–	Visual Basic	2,37	+0,77
10	12	+	Fortran	2,04	+0,94

jaka przyświecała autorowi, była implementacja minimalistyczna ale użyteczna. Został on zoptymalizowany pod kątem małych zasobów pamięci RAM i flash oraz mocy obliczeniowej. Ze względu na jego przeznaczenie zaimplementowano w nim bezpośrednio obsługę peryferii, które są dostępne w mikrokontrolerach, do których można zaliczyć: GPIO, I²C, SPI, I²S, UART, CAN, PWM, ADC, DAC, RTC itp. [1]. Interpreter MicroPythona, bazujący na bytecode, jest minimalistyczny, bardzo zoptymalizowany i dostosowany do ograniczonych zasobów sprzętowych. Użytkownik może korzystać z interaktywnego środowiska REPL, które daje możliwość wykonywania poleceń bezzwłocznie zatem także ich natychmiastowej weryfikacji.

MicroPython nie ma zaimplementowanych niektórych zaawansowanych funkcji, między innymi nie są dostępne moduły związane z wielowątkowością (biblioteka threading), w zamian tego umożliwia podstawowe wsparcie dla aplikacji asynchronicznych (moduł uasyncio). Nie są dostępne standardowe moduły graficznego interfejsu użytkownika (moduł tkinter). Z powodu ograniczeń dostępnych zasobów sprzętowych mikrokontrolera MicroPython nie zapewnia też pełnego wsparcia dla wielu innych standardowych bibliotek Pythona (takich jak na przykład os, numpy oraz pandas), ale ma zaimplementowane własne biblioteki specjalnie dostosowane do możliwości mikrokontrolerów (takie jak ujson, use czy umqtt). MicroPython niestety nie wspiera bibliotek wymagających dużych zasobów (takich jak scipy czy tensorflow) wykorzystywanych do algorytmów sztucznej inteligencji.

2.1. Różnice w stosunku do standardowej wersji

MicroPython obecnie (styczeń 2025) implementuje język Python w wersji 3.4 i niektóre elementy wersji 3.5. Ponadto istnieje szereg różnic między standardowym Pythonem, a MicroPythonem, które można podzielić na niezgodności syntaktyczne, różnice samym w rdzeniu języka, typach wbudowanych oraz implementacji poszczególnych modułów [10].
Niezgodności syntaktyczne to przede wszystkim:
– Nie jest dostępne rozpakowywanie argumentów.
– Dopuszczone jest użycie := do przypisania wartości do zmiennej w wyrażeniach generujących.
– Wymagane jest użycie spacji między liczbami zapisanymi jako literały, a słowami kluczowymi.
– Nie można używać nazw znaków Unicode do wstawiania tych znaków w ciągu tekstowym.

Różnice w samym języku:
– f-strings (łańcuchy formatowane) nie obsługują konkatenacji z sąsiednimi literałami, jeśli te literały zawierają nawiasy klamrowe.
– f-strings nie obsługują wyrażeń, które wymagają parsowania w celu rozwiązania niezrównoważonych, zagnieżdżonych nawiasów klamrowych lub nawiasów kwadratowych.
– f-strings nie obsługują konwersji `!a` (dla reprezentacji ASCII).
– Metoda specjalna `__del__` (konstruktor automatyczny) nie jest zaimplementowana dla klas zdefiniowanych przez użytkownika.
– Porządek rozwiązywania metod (MRO) nie jest zgodny z Pythonem, m.in. może on być deterministyczny w bardziej złożonych hierarchiach klas.
– Zmiana nazw prywatnych członków klas (ang. *name mangling*) nie jest zaimplementowana.
– Dziedziczenie typów natywnych: jest to związane z problemem niewłaściwego wywołania metody `__init__(self, ...)` przed `super().__init__()` co powoduje błąd AttributeError (lub awarię programu, jeśli `MICROPY_BUILTIN_METHOD_CHECK_SELF_ARG` nie jest włączone).
– Przy dziedziczeniu z wielu klas, `super()` wywołuje tylko jedną klasę.
– Wywołanie właściwości (`getter`) przez `super()` w klasie podrzędnej zwraca obiekt właściwości (property object), a nie wartość.
– Komunikaty o błędach metod mogą wskazywać nieoczekiwaną liczbę argumentów.
– Obiekty funkcji nie mają atrybutu `__module__`.
– Atrybuty definiowane przez użytkownika dla funkcji nie są obsługiwane.
– Metoda `__exit__()` w menedżerze kontekstu nie jest wywoływana, gdy generator nie zakończy swojego działania.
– Zmienne lokalne nie są uwzględniane w wyniku `locals()`.
– Kod uruchamiany w funkcji `eval()` nie ma dostępu do zmiennych lokalnych.

– Atrybut `__all__` nie jest obsługiwany w pliku `__init__.py` w MicroPython.
– Atrybut `__path__` pakietu ma inny typ (pojedynczy ciąg znaków zamiast listy ciągów) w MicroPython.
– MicroPython nie obsługuje pakietów przestrzeni nazw (namespace packages) podzielonych między różne lokalizacje w systemie plików.

Istnieją również miej lub bardziej znaczące różnice w typach wbudowanych, takich jak: `Exception`, `bytearray`, `bytes`, `dict`, `float`, `int`, `list`, `memoryview`, `str` i `tuple` oraz w implementacji następujących modułów: `array`, `builtins`, `json`, `os`, `random`, `struct`, `sys`. Standardowy Python przyjmuje jako argumenty funkcji, tak zwane argumenty nazwane (keyword arguments), MicroPython obsługuje wyłącznie argumenty pozycyjne.

W związku z wszystkimi powyższymi różnicami, pisanie aplikacji w MicroPythonie, dla osób, które dobrze znają i długo używały standardowego Pythona może powodować pewne problemy. Różnice te jednak nie są aż tak znaczące, aby uniemożliwiały sprawną i szybką implementację kodu w MicroPythonie, mimo to zawsze trzeba o nich pamiętać, gdy planuje się przenieść kod wcześniej uruchamiany na standardowej wersji Pythona, co dość często ma miejsce.

2.2. Platformy sprzętowe

Implementacja MicroPythona jest szeroko dostępna na większości popularnych platformach sprzętowych. Podstawową platformą jest jednak Pyboard specjalnie stworzona do jego uruchamiania. Najbardziej popularne platformy sprzętowe, na których dostępny jest MicroPython to:
– STM23 STMicroelectronics,
– ESP8266,
– ESP32,
– RP2040 – Raspberry Pi Pico,
– SAMD21/SAMD51 Microchip (Atmel),
– NRF52 Nordic Semiconductor,
– Teensy – i.MXRT,
– LPC1768/LPC4088 NXP,
– CC3200 Texas Instruments,
– Micro:bit,
– RA6M2 Renesas,
– Litex/VexRiscv (z RISC-V).

2.3. Implementacje w systemach operacyjnych

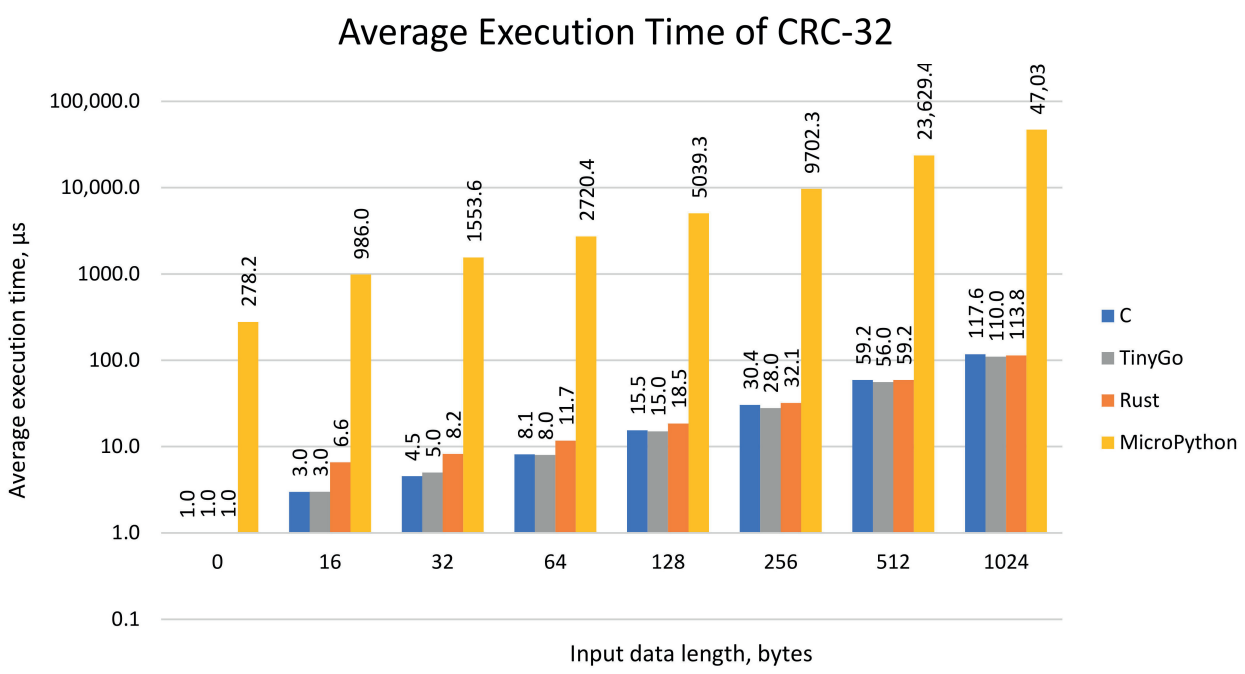
Interpreter Micropythona oprócz możliwości pracy bez systemu operacyjnego (bezpośrednio na mikrokontrolerze) jest też zaimplementowany w systemach operacyjnych czasu rzeczywistego oraz w klasycznych systemach operacyjnych, a także w środowiskach symulacyjnych. Przykładowe systemy operacyjne, w których jest dostępny MikroPython:
– FreeRTOS (OpenRTOS, SaveRTOS),
– Zephyr,
– Unix / Linux,
– MS Windows.

3. Porównanie oprogramowania w MicroPython i C/C++

Ogólne porównanie tych dwóch rozwiązań zostało przeprowadzone pod wieloma aspektami, niemniej jednak należy pamiętać, że w konkretnym specyficznym przypadku należy rozważyć, być może także aspekty charakterystyczne dla danej, być może nietypowej sytuacji.

3.1. Kompilator a interpreter, bezpieczeństwo – błędy syntaktyczne

Języki, takie jak C czy C++ są językami kompilowanymi, co oznacza, że analiza kodu, jego formalnej poprawności oraz generowanie plików wykonywalnych odbywa się w środowisku deweloperskim w czasie kompilacji. Dzięki temu, możliwe jest wykrycie i poprawienie kodu źródłowego. Współczesne kompilatory pozwalają na daleko idące optymalizacje kodu wykonywalnego, dzięki czemu wykonanie programu będzie bardzo szybkie. C++ w najnowszych wersjach wprowadza także możliwość obliczania pewnych elementów już w czasie kompilacji, dzięki mechanizmom wbudowanym w standard języka, takim jak stałe wyrażenia (ang. *constant expressions*), funkcje constexpr, oraz szablony (ang. *templates*). Natomiast interpreter MicroPythona nie ma możliwości wcześniejszego wykrycia błędów w kodzie w momencie innym niż interpretacja danego fragmentu kodu w czasie pracy aplikacji w środowisku docelowym. Taka sytuacja może być niebezpieczna oraz trudna do wykrycia i poprawienia. Sam interpreter MicroPythona (REPL) został napisany w C z wykorzystaniem standardu C99, w związku



Rys. 2. Średni czas wykonania algorytmu obliczania CRC-32 [4]
Fig. 2. Average execution times of the CRC-32 algorithm [4]

z tym będzie powodował duży narzut na czas wykonania w stosunku do oprogramowania wprost napisanego w C/C++. Ponadto interpreter potrzebuje do własnych potrzeb dodatkowe miejsce w pamięci, co dla bardzo ograniczonych zasobów mikrokontrolerów może stanowić poważny problem.

3.2. Prędkość wykonania programu i koszty mikrokontrolera

W pracach [2–5] przedstawiono różnego rodzaju testy porównawcze, ich wyniki jasno pokazują, że wszystkie porównywane przypadki dają wielokrotnie większe czasy wykonania tego samego zadania przez program napisany w MicroPythonie, w stosunku do takich języków programowania jak C++, RUST czy Go (wersja TinyGo dla mikrokontrolerów).

Na rysunku 2 przedstawiono wyniki analizy porównawczej czasów obliczania sum kontrolnych CRC-32 (ang. *Cyclic Redundancy Check*) dla różnych języków programowania. Należy zwrócić uwagę, że oś pionowa (średni czas wykonania) jest w skali logarytmicznej, a wyniki dla MicroPythona dla wszystkich przypadków są o co najmniej dwa rzędy wielkości gorsze (200–400 razy gorsze). Wydajność C/C++ jest bliska wydajności sprzętu.

W tabeli 2 zamieszczono implementację algorytmu obliczającego CRC-32 w Pythonie oraz w języku C. Z praktycznego punktu widzenia, jeśli chcielibyśmy zapewnić podobne czasy wykonania dla aplikacji w MicroPythonie, to musielibyśmy zastosować kilkaset razy szybszy mikrokontroler, co oczywiście powodowałoby znaczy wzrost ceny mikrokontrolera. Jest także możliwe, że ze względu na brak modelu o lepszych parametrach, nastąpiło by zaniechanie implementacji oprogramowania w MicroPythonie, a zatem wymóg łatwej i szybkiej skalowalności sytemu nie zostałby dotrzymany. Potrzeba mocniejszego (szybszego) mikrokontrolera należy rozpatrzyć również pod kontem wolumenu przewidywanej produkcji danych urządzeń – jeśli hipotetycznie koszt jednego mikrokontrolera byłby większy na przykład o 10 PLN to przy 1 mln egzemplarzy tylko z tego tytułu trzeba by było wydać na produkcję o 10 mln zł więcej, co nie jest małą kwotą. Z biznesowego punktu widzenia, w masowej produkcji, liczy się każda kwota więc lepiej poświęcić więcej czasu i pieniędzy na rozwój aplikacji przy użyciu C/C++ niż mieć taki problem z powodu zastosowania MicroPythona do oprogramowania.

3.3. Pamięć, stos, sterta, Garbage Collector

MicroPython potrzebuje do pracy interpreter, o czym wspomniano wcześniej, a ten z kolei potrzebuje dodatkowej pamięci RAM dla swoich potrzeb. Zużycie RAM przez implementację MicroPythona zależy od platformy, na której jest uruchamiany (STM32 około 16 KB, ESP8266 20–24 KB, ESP32 64–80 KB, RP2040 30–50 KB) oraz od wielkości i złożoności uruchamianego kodu.

Można zoptymalizować zużycie pamięci aplikacji w MicroPythonie przez:

- stosowanie zmiennych lokalnych.
- stosowanie wersji modułów u* (MicroPython dostarcza moduły takie jak ujson, ure, umqtt, które są zoptymalizowanymi wersjami modułów CPython, zużywającymi mniej RAM).
- minimalizację importowanych modułów – importowanie tylko tych modułów, które są niezbędne.
- ręczne wywoływanie Garbage Collector (użycie gc.collect()).

W pracy [3] pokazano porównanie zapotrzebowanie na pamięć, przy implementacji algorytmu obliczania prędkości obiektów, w aplikacji dla radaru. Okazuje się, że w przypadku MicroPythona maksymalne użycie sterty wyniosło 153 728 B, a w języku C tylko 33 624 B – czyli około pięć razy mniej.

3.4. Bezpieczeństwo – niewykryte błędy

Python (Micropython) to język dynamicznie typowany. Oznacza to, że nie trzeba jawnie deklarować typu zmiennej – typ jest określany automatycznie na podstawie przypisanej wartości. Zmienna w języku Python jest tworzona w podczas przypisania jej wartości za pomocą operatora =. Typ zmiennej w Pythonie można zmienić w dowolnym momencie przez przypisanie nowej wartości. Stanowi to poważny problem w przypadku niewłaściwego użycia tego mechanizmu. Od Pythona 3.5 można wprawdzie opcjonalnie określać typ zmiennej za pomocą adnotacji typów. Jest to pomocne przy dokumentowaniu kodu i w narzędziach statycznej analizy kodu, ale adnotacja nie jest

Tabl. 2 Implementacje algorytmu obliczania CRC-32 w języku C i Python
Tabl. 2 C and Python implementations of the CRC-32 calculation algorithm

Implementacja algorytmu obliczania CRC-32 w języku Python

```
CRC32_TABLE = [0] * 256

for i in range(256):
    crc = i
    for _ in range(8):
        if crc & 1:
            crc = (crc >> 1) ^ 0xEDB88320
        else:
            crc >>= 1
    CRC32_TABLE[i] = crc

def crc32(data, crc=0xFFFFFFFF):
    for byte in data:
        crc = (crc >> 8) ^ \
              CRC32_TABLE[(crc ^ byte) & 0xFF]
    return crc ^ 0xFFFFFFFF
```

Implementacja algorytmu obliczania CRC-32 w języku C

```
uint32_t crc32_table[256];

void generate_crc32_table() {
    for (uint32_t i = 0; i < 256; i++) {
        uint32_t crc = i;
        for (int j = 0; j < 8; j++) {
            if (crc & 1)
                crc = (crc >> 1) ^ 0xEDB88320;
            else
                crc >>= 1;
        }
        crc32_table[i] = crc;
    }
}

uint32_t crc32(const uint8_t* data, size_t length) {
    uint32_t crc = 0xFFFFFFFF;
    for (size_t i = 0; i < length; i++) {
        crc = (crc >> 8) ^
                crc32_table[(crc ^ data[i]) & 0xFF];
    }
    return crc ^ 0xFFFFFFFF;
}
```

wymagana i nie wymusza typu w czasie działania, czyli nie daje gwarancji, że zmienna taka nie zmieni typu, w wyniku przypisania jej jakiejś wartości o innym typie.

W językach ściśle typowanych, takich jak C/C++, typ każdej zmiennej, funkcji, czy wyrażenia jest jednoznacznie określony na etapie kompilacji i musi być zgodny z wymaganiami języka, co pomaga uniknąć niejednoznaczności i błędów w trakcie wykonania. Typy zmiennych są określane podczas kompilacji i nie można ich zmieniać w trakcie działania programu. Błędy typów są wykrywane na etapie kompilacji, co zwiększa bezpieczeństwo kodu. Jawne rzutowanie typów w C++ pozwala uniknąć nieoczekiwanych konwersji. Kompilator wymaga zgodności typów w operacjach i przypisaniach. Użycie auto ułatwia deklarację zmiennych o skomplikowanych typach i przerzuca na kompilator określenie potrzebnego typu. Nadany w ten sposób typ nie różni się niczym od zadeklarowanego wprost – nie można go zmienić. Taki typ jest nadal statycznie określany.

Do zalet ścisłego typowania można zaliczyć:
- bezpieczeństwo (wykrywanie błędów typów na etapie kompilacji, co zmniejsza ryzyko błędów w czasie wykonania),
- przejrzystość (typy zmiennych są jawnie określone, co ułatwia zrozumienie kodu),
- optymalizacja (kompilator lepiej optymalizuje kod, gdy zna typy zmiennych).

Ścisłe typowanie w C++ to kluczowy element języka, który zapewnia bezpieczeństwo, optymalizację i przejrzystość kodu. Mechanizmy takie jak static_cast, auto, decltype, szablony oraz enum class wspierają programistów w zarządzaniu typami. Mimo dodatkowej złożoności, ścisłe typowanie minimalizuje liczbę błędów i sprawia, że kod staje się bardziej niezawodny i wydajny.

Niestety te cenne mechanizmy i bezpieczeństwo, jakie wnoszą, nie są dostępne w przypadku języka MicroPython.

3.5. Konstrukcje językowe w C/C++ i w Python/MicroPython

Język C++ ma bardziej skomplikowaną składnię, wymaga zrozumienia wskaźników, zarządzania pamięcią i wielu innych konceptów. Ma silne wsparcie programowania obiektowego, proceduralnego i funkcyjnego.

Python częściowo wspiera programowanie obiektowe ale nie oferuje ścisłego mechanizmu ograniczania dostępu – nie ma pełnej enkapsulacji. Zamiast tego proponuje się pewne konwencje nazewnictwa i samoograniczenie programisty. Wszystkie atrybuty i metody są w języku MicroPython publiczne. Konwencja rozpoczynania nazwy od znaku podkreślenia sugeruje, że programista nie powinien bezpośrednio modyfikować takiego atrybutu, ale sam język nie blokuje takiego dostępu.

W języku Python wcięcia są kluczowym elementem składni i służą do definiowania bloków kodu (np. w pętlach, funkcjach, instrukcjach warunkowych). Zamiast używać nawiasów klamrowych (jak w C/C++ czy Javie), Python opiera strukturę kodu na wcięciach. Jednak niewłaściwe użycie wcięć może prowadzić do błędów.

Najczęstsze problemy z wcięciami w języku Python:
- Mieszanie tabulatorów i spacji (Python pozwala na użycie zarówno spacji, jak i tabulatorów, do tworzenia wcięć, ale mieszanie ich w jednym pliku lub bloku kodu prowadzi do błędów składniowych).
- Niewłaściwa liczba spacji we wcięciach (Python wymaga spójności w liczbie spacji używanych do wcięć w obrębie jednego bloku kodu, typowe ustawienie to cztery spacje).
- Brak wcięcia tam, gdzie jest wymagane (wcięcia są wymagane w blokach kodu, takich jak: pętle, funkcje, instrukcje warunkowe itp., brak wcięcia prowadzi do błędów składniowych).
- Niewłaściwe zakończenie bloku kodu (zbyt małe wcięcie przy kontynuacji kodu może spowodować, że Python uzna koniec bloku, mimo że zamierzeniem było jego kontynuowanie).
- Brakująca logika w blokach wymagających kodu (Python wymaga przynajmniej jednej instrukcji w bloku, jeśli blok jest pusty, należy użyć instrukcji pass).
- Problemy z kopiowaniem kodu z innych źródeł (kod skopiowany z różnych źródeł np. Internetu, edytorów tekstowych może zawierać niewidoczne tabulatory, spacje lub znaki ukryte, które powodują błędy).
- Niespójne wcięcia w pętlach i instrukcjach warunkowych (każdy blok kodu musi mieć jednolite wcięcie w ramach swojej struktury).
- Brak automatycznej weryfikacji wcięć (wcięcia mogą różnić się między edytorami, jeśli nie są skonfigurowane odpowiednio), zaleca się korzystanie z edytorów, które wyraźnie pokazują tabulatory i spacje oraz formatowania kodu za pomocą narzędzi, takich jak black lub autopep8).

Problemy z wcięciami w Pythonie wynikają głównie z niespójnego stosowania tabulatorów i spacji, błędnego rozmieszczenia wcięć lub kopiowania kodu z różnych źródeł. Aby uniknąć tych problemów, warto przestrzegać zaleceń stylu (PEP 8), używać spacji zamiast tabulatorów i korzystać z narzędzi automatycznego formatowania kodu.

W C/C++ nie ma takich problemów, ponieważ bloki kody są definiowane przez parę nawiasów klamrowych {…}. Braki lub nieścisłości w tym zakresie są skutecznie wykrywane przez kompilator i programista musi je wyeliminować, aby skompilować program.

3.6. Prędkość i wygoda pisania kodu aplikacji, biblioteki

Prędkość pisania kodu zależy od kilku czynników, w tym takich jak złożoność języka, dostępność narzędzi, stopień wymaganej optymalizacji i specyfika projektu. Języki Python i C/C++ różnią się filozofią, poziomem abstrakcji oraz celem, co bezpośrednio wpływa na szybkość tworzenia aplikacji. Python oferuje łatwość użycia bogatego zestawu bardzo zaawansowanych i dobrych bibliotek. Jest językiem wysokopoziomowym, dynamicznie typowanym, z prostą i intuicyjną składnią, co sprawia, że pisanie kodu w Pythonie może być znacznie szybsze niż w C/C++ [6]. Python pozwala na znaczne skrócenie tego czasu w projektach, które nie wymagają ekstremalnej optymalizacji.

C/C++ to języki niższego poziomu, które wymagają większej liczby kroków do osiągnięcia tego samego rezultatu. Są bardziej złożone, ale oferują pełną kontrolę nad zasobami, co jest kluczowe w systemach wbudowanych, czy aplikacjach wysokiej wydajności.

3.7. Wymagania i koszty programistów

Języki Python i C/C++ różnią się pod wieloma względami, co przekłada się na różne wymagania wobec programistów. Generalnie programiści C/C++ zwykle bywają bardziej doświadczeni (mają dłuższy staż) znają lepiej sprzęt (systemy wbudowane). Ponieważ Python jest językiem mniej wymagającym, a tak zwana bariera wejścia do zawodu programisty Pythona jest o wiele niższa w porównaniu do C/C++, to ich liczba jest znacznie większa, przez co wynagrodzenia są niższe. Jest to niewątpliwy atut dla osób zarządzających projektami.

3.8. Efektywność energetyczna

W systemach wbudowanych kwestie efektywności energetycznej i oszczędzania często są bardzo istotne. W związku z tym stosuje się szereg zabiegów sprzętowych i programistycznych aby optymalizować zużycie energii zasilania [9]. Szereg urządzeń jest przenośnych lub zamontowanych w miejscach

trudno dostępnych, gdzie wymiana baterii jest uciążliwa lub wręcz niemożliwa. Zapotrzebowanie na energię zasilania bezpośrednio przekłada się na pojemność, koszt, wielkość i ciężar baterii. Czas pracy bez konieczności ładowania oraz sam czas ładowania akumulatorów jest bardzo istotny ponieważ wpływa to na komfort i możliwości użytkowania danego urządzenia.

Zapotrzebowanie na zasoby w przypadku aplikacji w MicroPythonie jest dramatycznie większe, w porównaniu do C/C++, to bezpośrednio przekłada się na kwestie zużycia energii. Jak pokazano wcześniej w rozdziale 3.2, dwa rzędy wielkości wolniej działającą aplikacje w MicroPythonie. Jeśli nawet dopuścimy teką sytuację to analogiczna aplikacja w C/C++ mogłaby „usypiać" mikrokontroler po wykonaniu identycznych obliczeń i przez resztę czasu oszczędzać energię czyli w przybliżeniu można oszacować, że około 200 do 400 razy mniej energii zużyje taka aplikacja.

W pracy [3] pokazano porównanie zużycia energii aplikacji radarowej, analizującej pojedynczą ramkę. W przypadku tej aplikacji w MicroPythonie ilość energii wynosiła 43,8 nJ, a dla C 15,5 nJ, co jest znaczną różnicą.

3.9. Certyfikaty i standardy

W chwili obecnej nie istnieje możliwość certyfikowania oprogramowania aplikacji w MicroPythonie oraz samego interpretera MicroPytona. W językach programowania o ugruntowanej i wieloletniej pozycji w systemach wbudowanych – takich jak C/C++ od lat rozwijane i stosowane są takie standardy i wymogi tworzenia oprogramowania, jak MISRA [11], czy Autosar [12]. Wiele urządzeń z aplikacjami stworzonymi w tych językach programowania ma certyfikacje bezpieczeństwa funkcjonalnego, takie jak IEC 61508 (bezpieczeństwo funkcjonalne w systemach przemysłowych), ISO 26262 (bezpieczeństwo w systemach motoryzacyjnych) [13], DO-178C (bezpieczeństwo w oprogramowaniu lotniczym), certyfikaty cyberbezpieczeństwa jak ISO/IEC 27001, NIST Cybersecurity Framework, IEC 62443 (bezpieczeństwo w systemach przemysłowych i automatyce) czy też certyfikaty w sektorze medycznym takie jak IEC 62304 (standard dla cyklu życia oprogramowania medycznego) [14] oraz ISO 14971 (zarządzanie ryzykiem dla urządzeń medycznych). Certyfikacja oraz spełnianie poszczególnych standardów są wymagane przy dopuszczaniu do sprzedaży danego urządzenia. Jest to szczególnie istotne w aplikacjach krytycznych, takich jak medycyna, automatyka, systemy finansowe czy przemysł obronny. W związku z tym oprogramowanie tego typu nie może być realizowane w MicroPythonie.

4. Podsumowanie

Python został zaprojektowany jako łatwy język do nauki i zrozumienia kwestii programowania. MicroPython ze względu na swoje niewątpliwe walory stanowi bardzo ciekawą możliwość tworzenia prostych aplikacji będących raczej prototypami czy projektami hobbistycznymi niż bezpiecznymi dużymi profesjonalnymi systemami. Czasem jest on również używany do implementacji aplikacji prezentujących możliwości oferowanego sprzętu przez jakiegoś producenta na przykład czujników czy peryferii – jako programy demo. Potencjalny użytkownik może szybko poznać i przetestować ich możliwości. MicroPython jest bardzo dobrym rozwiązaniem w zastosowaniach dydaktycznych ze względu na prostotę oraz ponieważ wiele osób zna język Python, a C/C++ mogą im stwarzać problemy. Jednak w aplikacjach wymagających bezpieczeństwa, dostępności, pewności (gwarancji) działania, wydajności takich jak w sprzęcie medycznym, czy automatyki przemysłowej, nie jest rekomendowany lub wręcz wykluczany. Dodatkowy problem to koszt sprzętu, który musi mieć większe zasoby (częstotliwość zegara, pamięć RAM itp.). W wyniku wyższych wymagań oprogramowania tworzonego w MicroPythonie, w porównaniu do C/C++, staje się to poważnym problemem przy masowej produkcji.

Bibliografia

1. Tollervey N.H., *Programming with MicroPython. Embedded Programming with Microcontrollers and Python*, O'Reilly Media, 2017.
2. Plauska I., Liutkevičius A., Janavičiutė A., *Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller*, "Electronics", Vol. 12, No. 1, 2023, DOI: 10.3390/electronics12010143.
3. Skjutar F.A., Åstrand M., *MicroPython Integration for Radar Specific Application: Is it worth it?* Master's thesis, Department of Computer Science, LTH | Lund University, 2024.
4. Ionescu V.M., Enescu F.M., *Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers*, 2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2020, DOI: 10.1109/SIITME50350.2020.9292199.
5. Bugden W., Alahmar A., *The safety and performance of prominent programming*, "International Journal of Software Engineering and Knowledge Engineering", Vol. 32, No. 5, 2022, 713–744, DOI: 10.1142/S0218194022500231.
6. Bell C., *MicroPython for the Internet of Things: A Beginner's Guide to Programming with Python on Microcontrollers*, Apress, 2017.
7. Szymczyk P., Szymczyk M., *Energooszczędne oprogramowanie systemów czasu rzeczywistego mikrokontrolerów*, Praca zbiorowa pod redakcją Z. Zielińskiego: „Systemy czasu rzeczywistego Postęp badań i zastosowania", Warszawa, WKŁ, 2009, 179–188.
8. Lapan M., *Deep Reinforcement Learning Hands-On*, Packt 1Publishing, 2020.

Inne źródła

9. *TIOBE Index for March 2025*, [www.tiobe.com/tiobe-index/].
10. *MicroPython*, [https://micropython.org/].
11. MISRA, [https://misra.org.uk/].
12. AUTomotive Open System Architecture, [https://www.autosar.org/].
13. ISO 26262-1:2018(en) Road vehicles – Functional safety – Part 1: Vocabulary, [https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en].
14. Medical Device and Diagnostic Industry, *Developing Medical Device Software to IEC 62304*, [https://www.mddionline.com/software/developing-medical-device-software-toiec-62304].

MicroPython in Embedded Systems, Quality and Security of the Application

Abstract: The article presents the potential of using Python in embedded systems. Differences and limitations compared to the standard version are analysed, with attention given to the aspects of security and application quality for use in medicine and automation.

Keywords: MicroPython, embedded systems, quality, security

dr inż. Magdalena Szymczyk

Magdalena.Szymczyk@agh.edu.pl
ORCID: 0000-0001-7509-2040

Magdalena Szymczyk uzyskała tytuł magistra inżynierii elektronicznej na Akademii Górniczo-Hutniczej (Kraków, Polska) w 1988 r. oraz stopień doktora informatyki również na Akademii Górniczo-Hutniczej w 1999 r. Obecnie jest adiunktem na Akademii Górniczo-Hutniczej w Katedrze Biocybernetyki i Inżynierii Biomedycznej na Wydziale Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej. Jej zainteresowania badawcze obejmują systemy komputerowe czasu rzeczywistego, systemy wbudowane, programowanie równoległe i bioinformatykę.