BBC micro:bit MicroPython Documentation

Wydanie 0.0.1

Multiple authors

20 mar 2018

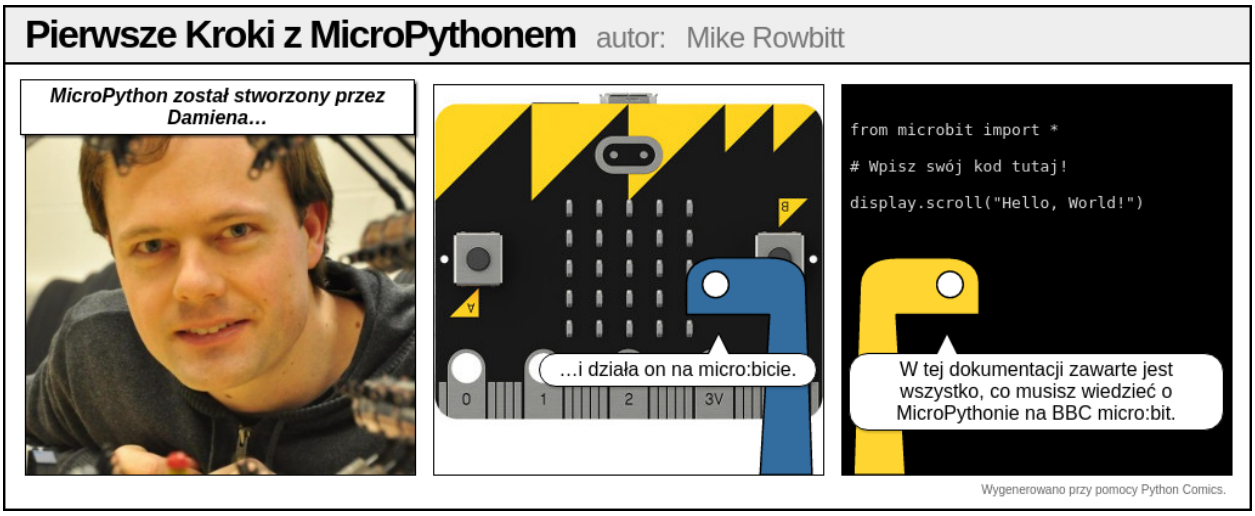
Poradniki

Witaj!

BBC micro:bit to niewielkie urządzenie obliczeniowe dla dzieci. Jednym z języków, które ono rozumie, jest język programowania Python. Odmiana Pythona, która działa na BBC micro:bit nazywa się MicroPython.

Ta dokumentacja zawiera lekcje dla nauczycieli i opis interfejsu aplikacji dla programistów (spójrz na listę po lewej stronie). Mamy nadzieję, że programowanie z użyciem MicroPythona dla BBC micro:bit sprawi ci wiele przyjemności.

Jeśli nigdy wcześniej nie programowałeś, albo nie jesteś pewien od czego zacząć, zacznij od poradników.



Aby stać się częścią społeczności, zapisz się do grupy microbit@python.org (https://mail.python.org/mailman/listinfo/microbit).

Informacja: Ten projekt jest w stanie ciągłego rozwoju. Pomóż innym dodając do dokumentacji porady, sztuczki, uwagi i odpowiedzi na często zadawane pytania. Dziękujemy!

Projekty związane z MicroPythonem na BBC microb:bit to między innymi:

- Mu - prosty edytor kodu dla dzieci, nauczycieli i początkujących programistów. Najpewniej najłatwiejszy sposób na programowanie w MicroPythonie na BBC micro:bit.
- uFlash - narzędzie linii poleceń do wgrywania programów w pythonie do BBC micro:bit.

ROZDZIAŁ 1

Wprowadzenie

Zalecamy używanie edytora Mu podczas pracy z tym poradnikiem. Instrukcje dotyczące ściągnięcia i instalacji Mu znajdują się na jego stronie internetowej. Może być też konieczne zainstalowanie sterownika, w zależności od platformy (szczegółowe instrukcje na stronie).

Mu działa na Windowsie, OSX i na Linuksie.

Kiedy już zainstalujesz Mu, uruchom go i podłącz swój micro:bit do komputera przy pomocy kabla USB.

Napisz swój skrypt w oknie edytora i kliknij ikonkę „Flash" aby przesłać go do micro:bita. Jeśli to nie zadziała, upewnij się, że micro:bit pojawia się w eksploratorze plików jako dysk.

1.1 Hello, World!

Tradycyjnie, pierwszym krokiem w nauce nowego języka programowania jest stworzenie aplikacji, która wyświetli komunikat „Witaj świecie!" (ang. „Hello, World").

To proste z MicroPythonem:

```
from microbit import *
display.scroll("Hello, world!")
```

Każdy wiersz ma swoje znaczenie. Pierwszy z nich:

```
from microbit import *
```

...mówi MicroPythonowi, by zaimportował wszystkie rzeczy potrzebne do pracy z BBC micro:bit. Wszystko to jest w module `microbit` (moduł to biblioteka z wcześniej przygotowanym kodem). Poleceniem `import` mówisz MicroPythonowi, że chcesz użyć danego modułu, a `*` to sposób Pythona na określenie *wszystkiego*. Zatem `from microbit import *` oznacza „chcę użyć wszystkiego co jest dostępne w bibliotece microbit".

Druga linia:

```
display.scroll("Hello, world!")
```

... mówi MicroPythonowi, by wyświetlił przesuwający się ciąg znaków „Hello, world!". `display` w tym przypadku to *obiekt* (ang. object) z modułu `microbit`, który reprezentuje fizyczny wyświetlacz urządzenia (mówimy „obiekt" zamiast „rzecz" lub „to coś"). By wydać wyświetlaczowi polecenie, po kropce `.` podajemy komendę – tak naprawdę takie polecenia nazywamy *metodami* (ang. method). W tym przypadku używamy polecenia `scroll` (ang. przewiń). Polecenie `scroll` musi wiedzieć jakie znaki pokazać na wyświetlaczu. Wpisujemy je ujęte w cudzysłów (`"`) i zamknięte w nawiasy (`(` i `)`). W programowaniu to co przekazujemy do metody nazywamy *argumentami* (ang. arguments). Zatem `display.scroll("Hello, world!")` oznacza, po polsku, „chcę użyć wyświetlacza by pokazać przesuwający się tekst «Hello, world!»". Jeśli metoda nie potrzebuje żadnych argumentów, musimy to jasno określić używając pustych nawiasów: `()`.

Skopiuj powyższy kod do swojego edytora i wgraj go (ang. flash) do urządzenia. Czy domyślasz się jak zmienić wyświetlany tekst? Czy możesz go zmienić tak, by przywitał ciebie? Na przykład, chciałbym by tekst brzmiał „Witaj, Andrzeju!". Mała podpowiedź: musisz zmienić argument metody `scroll`.

Ostrzeżenie: To może nie działać :-)

Tutaj zaczyna się prawdziwa zabawa, a MicroPython stara się być naprawdę pomocny. Jeśli napotka błąd, to na wyświetlaczu pokaże komunikat błędu. W miarę możliwości, pokaże również numer linii kodu, która zawiera błąd.

Python oczekuje, że wprowadzisz **BEZBŁĘDNY** kod. Na przykład, `Microbit`, `microbit` i `microBit` są przez Pythona traktowane jak trzy osobne rzeczy. Jeśli w treści błędu zobaczysz `NameError` (błąd nazwy), oznacza to że prawdopodobnie wpisana nazwa została podana niedokładnie. To jak różnica między „Pawłem" i „Gawłem" - dwa różne imiona, choć brzmią i wyglądają podobnie.

Natomiast jeśli w treści błędu zobaczysz `SyntaxError` (błąd składni), oznacza to po prostu, że podany kod jest niezrozumiały dla MicroPythona. Sprawdź czy nie brakuje żadnych znaków specjalnych, jak `"` czy `:`. To tak jak gdyby umieścić. kropkę w środku zdania. Trudno jest wtedy zrozumieć co autor miał na myśli.

Twój microbit może przestać reagować: nie uda się zaprogramować nowego kodu lub wpisywać poleceń w konsoli REPL. W takim przypadku spróbuj odłączyć urządzenie od prądu i podłączyć ponownie po krótkiej chwili. Chodzi o to, by odłączyć kabel USB (i kabel baterii jeśli jest podłączony), a po chwili podłączyć ponownie. Może być także konieczne zrestartowanie edytora kodu.

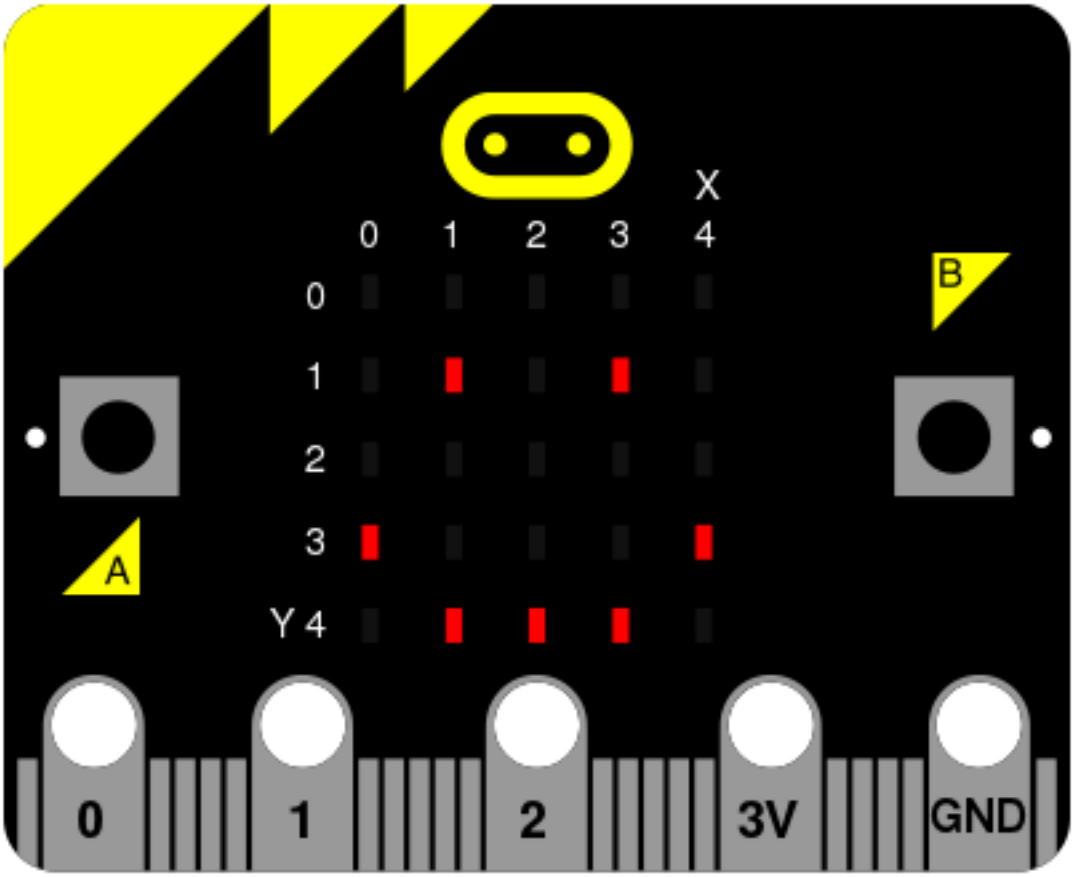
1.2 Obrazy

MicroPython jest tak dobry jak to tylko możliwe, jeśli chodzi o sztukę, jeśli jedyne czym dysponujesz to macierz 5x5 diod LED (ang. Light Emitting Diodes, diody emitujące światło na przodzie urządzenia). MicroPython daje sporo kontroli nad wyświetlaczem, zatem możesz uzyskać sporo ciekawych efektów.

MicroPython posiada wbudowany zestaw obrazów które może pokazać na wyświetlaczu. Na przykład, żeby pokazać na ekranie szczęśliwą buźkę musimy napisać:

```
from microbit import *
display.show(Image.HAPPY)
```

Mam nadzieję, że pamiętasz co robi pierwsza linijka powyżej. Druga linia używa obiektu `display` i metody `show` (ang. pokaż) do wyświetlenia obrazka. Buźka którą chcemy wyświetlić jest częścią obiektu `Image` (ang. Obraz) i nazywa się `HAPPY` (ang. szczęśliwy). Mówimy metodzie `show` by użyła tego obrazka przez podanie jego nazwy w nawiasach (`(` i `)`).



Poniżej znajdziesz listę wszystkich wbudowanych obrazów:

- `Image.HEART`
- `Image.HEART_SMALL`
- `Image.HAPPY`
- `Image.SMILE`
- `Image.SAD`
- `Image.CONFUSED`
- `Image.ANGRY`
- `Image.ASLEEP`
- `Image.SURPRISED`
- `Image.SILLY`
- `Image.FABULOUS`
- `Image.MEH`
- `Image.YES`
- `Image.NO`
- `Image.CLOCK12`, `Image.CLOCK11`, `Image.CLOCK10`, `Image.CLOCK9`, `Image.CLOCK8`, `Image.CLOCK7`, `Image.CLOCK6`, `Image.CLOCK5`, `Image.CLOCK4`, `Image.CLOCK3`, `Image.CLOCK2`,

Image.CLOCK1

- Image.ARROW_N, Image.ARROW_NE, Image.ARROW_E, Image.ARROW_SE, Image.ARROW_S, Image.ARROW_SW, Image.ARROW_W, Image.ARROW_NW
- Image.TRIANGLE
- Image.TRIANGLE_LEFT
- Image.CHESSBOARD
- Image.DIAMOND
- Image.DIAMOND_SMALL
- Image.SQUARE
- Image.SQUARE_SMALL
- Image.RABBIT
- Image.COW
- Image.MUSIC_CROTCHET
- Image.MUSIC_QUAVER
- Image.MUSIC_QUAVERS
- Image.PITCHFORK
- Image.XMAS
- Image.PACMAN
- Image.TARGET
- Image.TSHIRT
- Image.ROLLERSKATE
- Image.DUCK
- Image.HOUSE
- Image.TORTOISE
- Image.BUTTERFLY
- Image.STICKFIGURE
- Image.GHOST
- Image.SWORD
- Image.GIRAFFE
- Image.SKULL
- Image.UMBRELLA
- Image.SNAKE

Jest ich bardzo dużo! Może zmodyfikujesz kod powyżej żeby zobaczyć jak wyglądają pozostałe obrazy? (Wystarczy że zastąpisz `Image.HAPPY` jednym z innych wbudowanych obrazów które są wypisane powyżej.)

1.2.1 Obrazy – Zrób to sam

Naturalnie, na pewno chcesz stworzyć własny obrazek do wyświetlenia, prawda?

To proste.

Każda dioda LED (dalej nazywana pikselem) na wyświetlaczu może przyjąć jedną z dziesięciu wartości. Jeśli piksel jest ustawiony na `0` (zero), to znaczy że jest wyłączony. Po prostu jasność jest ustawiona na zero, dlatego nie świeci. Jeśli jednak podamy wartość `9` to ustawiamy najwyższy poziom jasności. Wartości od `1` do `8` reprezentują poziomy jasności między `0` (wyłączony) do `9` (pełna jasność).

Mając powyższe na uwadze, możemy stworzyć własny obrazek w ten sposób:

```
from microbit import *

boat = Image("05050:"
             "05050:"
             "05050:"
             "99999:"
             "09990")

display.show(boat)
```

(Kiedy uruchomisz urządzenie, na wyświetlaczu pokaże się łódka z dwoma masztami, które będą nieco ciemniejsze od kadłuba.)

Wiesz już jak tworzyć obrazki? Widzisz jak każda linia wyświetlacza jest reprezentowana przez ciąg znaków kończący się `:` (dwukropkiem) i zamknięty w `"` (cudzysłów)? Każda liczba określa jasność. Mamy pięć linii po pięć liczb, zatem możemy osobno nadać jasność każdemu pikselowi na urządzeniu. Tak właśnie tworzy się własne obrazy.

Proste? Proste!

W zasadzie nie musisz podawać tych wartości w kilku liniach. Jeśli będzie to dla ciebie czytelne, to możesz wszystkie wartości podać w jednej linii:

```
boat = Image("05050:05050:05050:99999:09990")
```

1.2.2 Animacja

Obrazki statyczne są zabawne, ale bardziej zabawne jest ich poruszenie. To też jest niesamowicie proste do zrobienia w MicroPython ~ po prostu użyj listy obrazków!

Tu jest lista zakupów:

```
Jaja
Boczek
Pomidory
```

Oto jak przedstawiłbyś tę listę w Python:

```
zakupy = ["Jaja", "Boczek", "Pomidory" ]
```

Po prostu utworzyłem listę nazwaną `zakupy` i zawiera ona trzy elementy. Python wie, że to jest lista ponieważ jest zawarta w kwadratowych nawiasach (`[` i `]`). Elementy w liście są oddzielone przecinkami (`,`) i w tej instancji elementy są trzema ciągami znaków: `"Jaja"`, `"Boczek"` i `"Pomidory"`. My wiemy, że są one ciągami znaków ponieważ są objęte znakami cudzysłowu `"`.

W liście Python możesz przechowywać cokolwiek. Tu jest lista liczb:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

Informacja: Liczby nie potrzebują być w cudzysłowie dopóki reprezentują wartość (w przeciwieństwie do ciągów znaków). Jest różnica pomiędzy `2` (numeryczna wartość 2) i `"2"` (znak/cyfra reprezentująca liczbę 2). Nie martw się jeżeli nie widzisz w tym sensu teraz. Z czasem będzie to dla Ciebie oczywiste.

Jest nawet możliwe przechowywanie różnych rodzajów rzeczy w tej samej liście:

```
mixed_up_list = ["hello!", 1.234, Image.HAPPY]
```

Zwróciłeś uwagę na ostatni element? To był obrazek!

Możemy powiedzieć MicroPythonowi, aby animował listę obrazków. Szczęśliwie mamy już kilka wbudowanych list obrazków. Są to `Image.ALL_CLOCKS` i `Image.ALL_ARROWS`:

```
from microbit import *

display.show(Image.ALL_CLOCKS, loop=True, delay=100)
```

Używamy `display.show` do pokazania listy obrazków na ekranie urządzenia tak, jak w przypadku pojedynczego obrazka. Jednak mówimy MicroPythonowi użyj `Image.ALL_CLOCKS` i on rozumie, że musi pokazać każdy obrazek z listy jeden po drugim. Rozkazujemy też MicroPythonowi aby powtarzał listę obrazków (tak więc animacja trwa nieskończenie) przez `loop=True`. Ponadto każemy mu, aby opóźnienia pomiędzy każdym obrazkiem były tylko 100 milisekund (0,1 sekundy) w argumencie `delay=100`.

Czy możesz domyślić się jak animować obrazki z listy `Image.ALL_ARROWS`? Jak możesz uniknąć nieskończonego powtarzania (podpowiedź: przeciwieństwem do `True` (ang. prawdziwy) jest `False` (ang. fałszywy) chociaż domyślna wartość dla `loop` jest `False`)? Czy potrafisz zmienić prędkość animacji?

Wreszcie tutaj możesz zobaczyć jak utworzyć swoją własną animację. W moim przykładzie zamierzam stworzyć łódź tonącą na dno ekranu:

```
from microbit import *

boat1 = Image("05050:"
              "05050:"
              "05050:"
              "99999:"
              "09990")

boat2 = Image("00000:"
              "05050:"
              "05050:"
              "05050:"
              "99999")

boat3 = Image("00000:"
              "00000:"
              "05050:"
              "05050:"
              "05050")

boat4 = Image("00000:"
              "00000:"
              "00000:"
              "05050:"
```

```
                "05050")

boat5 = Image("00000:"
                "00000:"
                "00000:"
                "00000:"
                "05050")

boat6 = Image("00000:"
                "00000:"
                "00000:"
                "00000:"
                "00000")

all_boats = [boat1, boat2, boat3, boat4, boat5, boat6]
display.show(all_boats, delay=200)
```

A tak działa ten kod:

- Utworzyłem sześć obrazków `boat` (ang. łódź) w dokładnie ten sam sposób, jak opisałem powyżej.
- Potem umieściłem wszystkie na liście, którą nazwałem `all_boats`.
- W końcu poprosiłem `display.show` o animację listy z opóźnieniem 200 milisekund.
- Ponieważ nie użyłem `loop=True` łódź utonie tylko raz (to czyni moją animację naukowo poprawną). :-)

Co chciałbyś animować? Czy chciałbyś animować efekty specjalne? Jak byś zrobił aby obrazek znikał, a potem pojawiał się znowu?

1.3 Przyciski

Jak dotychczas utworzyliśmy kod umożliwiający urządzeniu robienie czegoś. Nazywa się to *wyjście*. Musimy też jednak umożliwić urządzeniu reagowanie. To, na co ono reaguje nazywamy *wejściem*.

Łatwo zapamiętać: wyjście dotyczy wszystkiego co wychodzi z urządzenia, natomiast wejście to wszystko co przychodzi do urządzenia w celu przetworzenia.

Najłatwiejszą metodą dostarczenia *wejścia* są dwa przyciski, oznaczone `A` i `B`. Musimy jakoś zmusić MicroPython do reagowania na ich naciśnięcie.

Jest to niezwykle proste:

    from microbit import *

    sleep(10000) display.scroll(str(button_a.get_presses()))

Ten skrypt śpi przez 10 000 milisekund (czyli 10 sekund), a po tym czasie na wyświetlaczu przewinie się liczba wciśnięć przycisku `A`. To wszystko.

Chociaż ten skrypt jest bezużyteczny, to jednak pokazuje nam kilka ciekawych koncepcji:

1. *Funkcja* `sleep` usypia microbit na pewną liczbę milisekund. Jeżeli chcesz zatrzymać program, możesz to zrobić używając tej funkcji. *Funkcja* jest podobna do *metody* ale nie jest przyporządkowana do *obiektu* za pomocą kropki.
2. Obiekt `button_a` pozwala ci pobrać liczbę naciśnięć przycisku za pomocą *metody* `get_presses`.

Metoda `get_presses` daje nam tylko liczbę, a `display.scroll` przyjmuje tylko ciągi znaków. Dlatego musimy zmienić liczbę na ciąg znaków, co możemy zrobić za pomocą funkcji `str` (skrót od angielskiego słowa „string"), która przetwarza wszystko na ciąg znaków.

Trzeci wiersz jest jak cebula. Jeżeli przyjmiesz nawiasy za warstwy cebuli, to zauważysz, że `display.scroll` zawiera `str`, który zawiera `button_a.get_presses`. Python próbuje rozpracować najbardziej wewnętrzne wyrażenia najpierw, zanim przejdzie do następnej warstwy. To nazywa się *zagnieżdżenie* - programistyczny odpowiednik rosyjskiej matrioszki.

Przyjmijmy, że nacisnąłeś przycisk 10 razy. Python w następujący sposób analizuje co się dzieje w trzeciej linii:

Python widzi pełną linię i dostaje wartość `get_presses`:

```
display.scroll(str(button_a.get_presses()))
```

Teraz, gdy Python wie już ile razy został wciśnięty przycisk, skonwertuje wartość liczbową na ciąg znaków:

```
display.scroll(str(10))
```

W końcu Python wie, co ma wyświetlić na ekranie:

```
display.scroll("10")
```

Może ci się wydawać, że to jest mnóstwo pracy, ale MicroPython robi to niezwykle szybko.

1.3.1 Pętle zdarzeń

Często chcesz aby program poczekał na jakieś zdarzenie. Żeby to uzyskać, musisz zamknąć w pętli kawałek kodu definiujący jak reagować na oczekiwane zdarzenia, takie jak naciśnięcie przycisku.

Pętle w Pythonie tworzymy przy użyciu polecania `while` (ang. dopóki). Sprawdza ono czy coś jest `True` (ang. prawdziwe). Jeżeli tak jest, to uruchamia *blok kodu* zwany *ciałem* pętli. Jeżeli jednak nie jest, to przerywa wykonywanie pętli (ignorując ciało pętli) i przechodzi do dalszej części kodu.

W Pythonie definiowanie bloków kodu jest proste. Powiedzmy, że mam listę rzeczy do zrobienia napisaną na kartce papieru. Może to wyglądać tak:

```
Zakupy
Napraw uszkodzoną rynnę
Skoś trawnik
```

Gdybym chciał nieco rozbić moją listę rzeczy do zrobienia, to mógłbym napisać coś takiego:

```
Zakupy:
    Jaja
    Boczek
    Pomidory
Napraw uszkodzoną rynnę:
    Pożycz drabinę od sąsiada
    Znajdź młotek i gwoździe
    Zwróć drabinę
Skoś trawnik:
    Sprawdź trawnik wokół sadzawki dla żab
    Sprawdź poziom paliwa w kosiarce
```

Oczywiście główne zadania są podzielone na podrzędne zadania, umieszczone poniżej głównego zadania, z którym są powiązane i *wcięte*. Tak więc `Jaja`, `Boczek` i `Pomidory` są na pierwszy rzut oka związane z zadaniem `Zakupy`. Wcięcia ułatwiają nam sprawdzanie w jaki sposób zadania są powiązane ze sobą.

Nazywamy to *zagnieżdżeniem*. Zagnieżdżania używamy do definiowania bloków kodu w ten oto sposób:

```
from microbit import *

while running_time() < 10000:
    display.show(Image.ASLEEP)

display.show(Image.SURPRISED)
```

Funkcja `running_time` zwraca liczbę milisekund od startu urządzenia.

Linia `while running_time() < 10000:` sprawdza czy czas pracy urządzenia jest mniejszy od 10 000 milisekund (czyli 10 sekund). Jeżeli tak, *i tu widzimy działanie zagnieżdżania*, to zostanie wyświetlony `Image.ASLEEP`. Zwróć uwagę na wcięcie kodu po poleceniu `while` *takie jak w naszej liście zadań*.

Oczywiście, jeśli czas pracy jest równy lub większy niż 10 000 milisekund, wówczas na ekranie pojawi się `Image.SURPRISED`. Dlaczego? Ponieważ warunek `while` (ang. dopóki) będzie fałszywy (ang. False) (`running_time` nie jest już `< 10000`). W takim przypadku pętla jest zakończona i program będzie kontynuowany po bloku kodu pętli `while`. Będzie to sprawiać wrażenie, że twoje urządzenie śpi przez 10 sekund zanim obudzi się z zaskoczeniem na twarzy.

Spróbuj sam!

1.3.2 Obsługa zdarzenia

Jeśli chcemy aby MicroPython reagował na zdarzenia naciśnięcia przycisku, musimy wprowadzić go w nieskończoną pętlę i sprawdzać w niej czy przycisk jest wciśnięty (ang. `is_pressed`).

Nieskończona pętla jest prosta:

```
while True:
    # rób coś
```

(Pamiętaj, że `while` sprawdza czy coś jest `True` przed każdym wykonaniem bloku kodu. Ponieważ `True` jest oczywiście `True` przez cały czas, to otrzymujesz nieskończoną pętlę!)

Zróbmy bardzo proste elektroniczne zwierzątko. Jest ono smutne, gdy nie naciskasz przycisku `A`. A gdy wciśniesz przycisk `B` umiera. (Zdaję sobie sprawę, że to nie jest zbyt przyjemna gra, więc może masz pomysł jak ją ulepszyć.):

```
from microbit import *

while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif button_b.is_pressed():
        break
    else:
        display.show(Image.SAD)

display.clear()
```

Rozumiesz jak sprawdzić, które przyciski są wciśnięte? Użyliśmy `if` (ang. jeśli), `elif` (skrót od „else if") (ang. jeśli jednak) oraz `else` (ang. w pozostałych przypadkach). Są one nazywane *warunkami* i działają tak:

```
if coś jest True:
    # zrób pierwszą rzecz
elif coś innego jest True:
    # zrób następną rzecz
else:
    # zrób jeszcze coś.
```

Prawie jak po polsku!

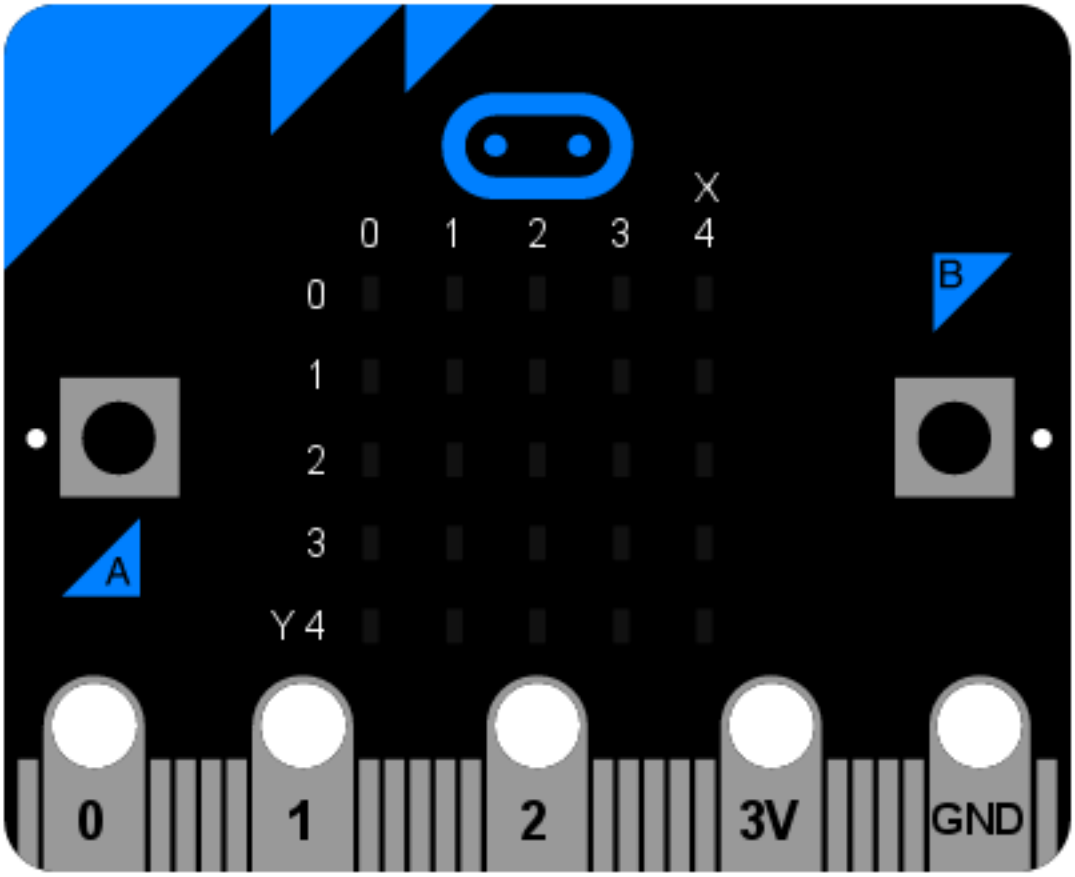
Metoda `is_pressed` generuje jeden z dwóch wyników: `True` albo `False`. Jeżeli przycisk jest wciśnięty, zwraca `True`, w przeciwnym wypadku zwróci `False`. Powyższy kod można przetłumaczyć jako „przez cały czas: jeśli przycisk A jest wciśnięty - pokazuj szczęśliwą twarz, jeśli jednak przycisk B jest wciśnięty - przerwij pętlę, a w pozostałych przypadkach pokaż smutną minę". Przerywamy pętlę (zatrzymując działający przez cały czas program) za pomocą instrukcji `break` (ang. przerwij).

Na samym końcu, kiedy elektroniczne zwierzątko nie żyje, czyścimy (ang. `clear`) ekran.

Wiesz jak uczynić tę grę mniej tragiczną? Jak sprawdziłabyś, czy *oba* przyciski są wciśnięte? (Podpowiedź: Python ma operatory logiczne „i" (ang. `and`), „lub" (ang. `or`) oraz „nie" (ang .‘‘not‘‘), którymi można sprawdzić wiele warunków (wyrażeń, które mają wartość `True` lub `False`).

1.4 Wejście/Wyjście

Na dolnej krawędzi micro:bita znajdują się paski metalu, które sprawiają, że urządzenie wygląda jakby miało zęby. Są to piny wejścia/wyjścia, w skrócie piny we/wy (ang. input/output - I/O).



Niektóre piny są większe od innych, aby było możliwe podłączenie do nich krokodylków (zacisków). Są one oznaczone 0, 1, 2, 3V oraz GND (komputery zawsze zaczynają liczyć od zera). Jeśli podłączysz płytkę złącza krawędziowego do urządzenia, będziesz mógł podłączać kabelki również do pozostałych (mniejszych) pinów.

Każdy pin micro:bita jest reprezentowany przez *obiekt* o nazwie `pinN`, gdzie `N` jest numerem pinu. Zatem, na przykład, aby zrobić coś z pinem oznaczonym numerem 0 (zero) użyj obiektu o nazwie `pin0`.

Proste!

Te obiekty posiadają różne *metody*, w zależności od tego, do czego dany pin może zostać wykorzystany.

1.4.1 Delikatny Python

Najprostszym przykładem wejścia przez pin jest sprawdzenie, czy jest on dotknięty. W taki oto sposób możesz połaskotać swoje urządzenie, aby je rozśmieszyć:

```
from microbit import *

while True:
    if pin0.is_touched():
        display.show(Image.HAPPY)
    else:
        display.show(Image.SAD)
```

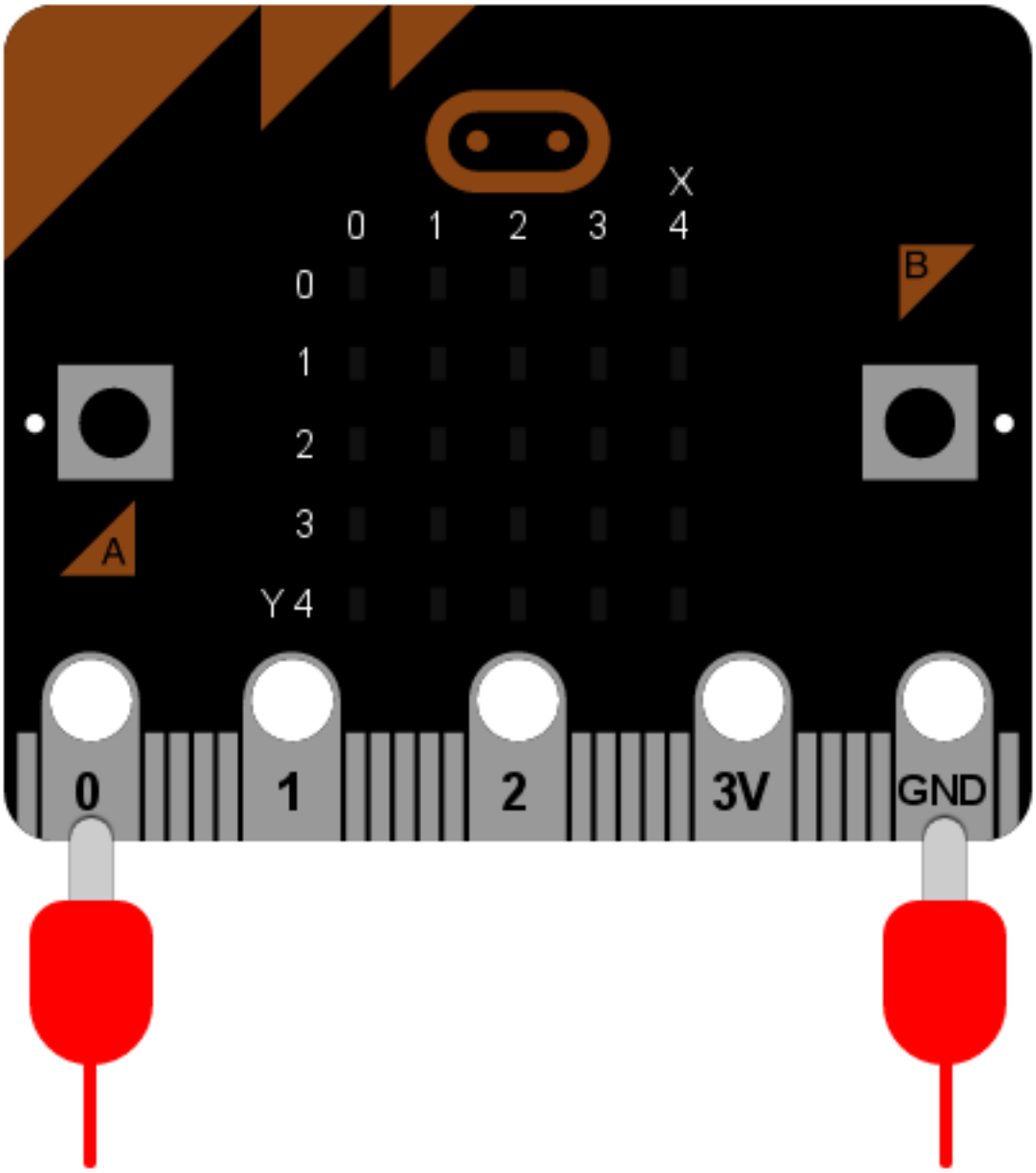
Jedną ręką trzymaj urządzenie za pin GND. Następnie, drugą ręką dotknij (lub połaskocz) pin 0 (zero). Powinieneś zobaczyć na wyświetlaczu zmianę z miny smutnej na szczęśliwą!

To jest tylko bardzo podstawowa forma sprawdzenia działania wejścia. Prawdziwa zabawa zaczyna się, gdy do pinów podłączysz obwody oraz inne urządzenia.

1.4.2 Piski i buczenia

Najprostszą rzeczą, które możemy podłączyć do naszego urządzenia jest brzęczyk piezoelektryczny. Użyjemy go jako wyjście.

To małe urządzenie wydaje wysoki dźwięk, kiedy zostanie podłączone do obwodu. Aby podłączyć je do swojego micro:bita, możesz przypiąć krokodylki do pinu 0 oraz GND (jak widać na poniższym obrazku).



Kabel z pinu 0 powinien być podłączony do dodatniego złącza brzęczyka, a kabel z pinu GND do złącza ujemnego.

Poniższy program sprawi, że brzęczyk wyda dźwięk:

```
from microbit import *

pin0.write_digital(1)
```

Jest to zabawne przez około 5 sekund, ale potem będziesz chciał wyłączyć ten okropny dźwięk. Rozbudujmy nasz przykład i sprawmy, aby urządzenie wydawało piski:

```
from microbit import *

while True:
    pin0.write_digital(1)
    sleep(20)
```

```
    pin0.write_digital(0)
    sleep(480)
```

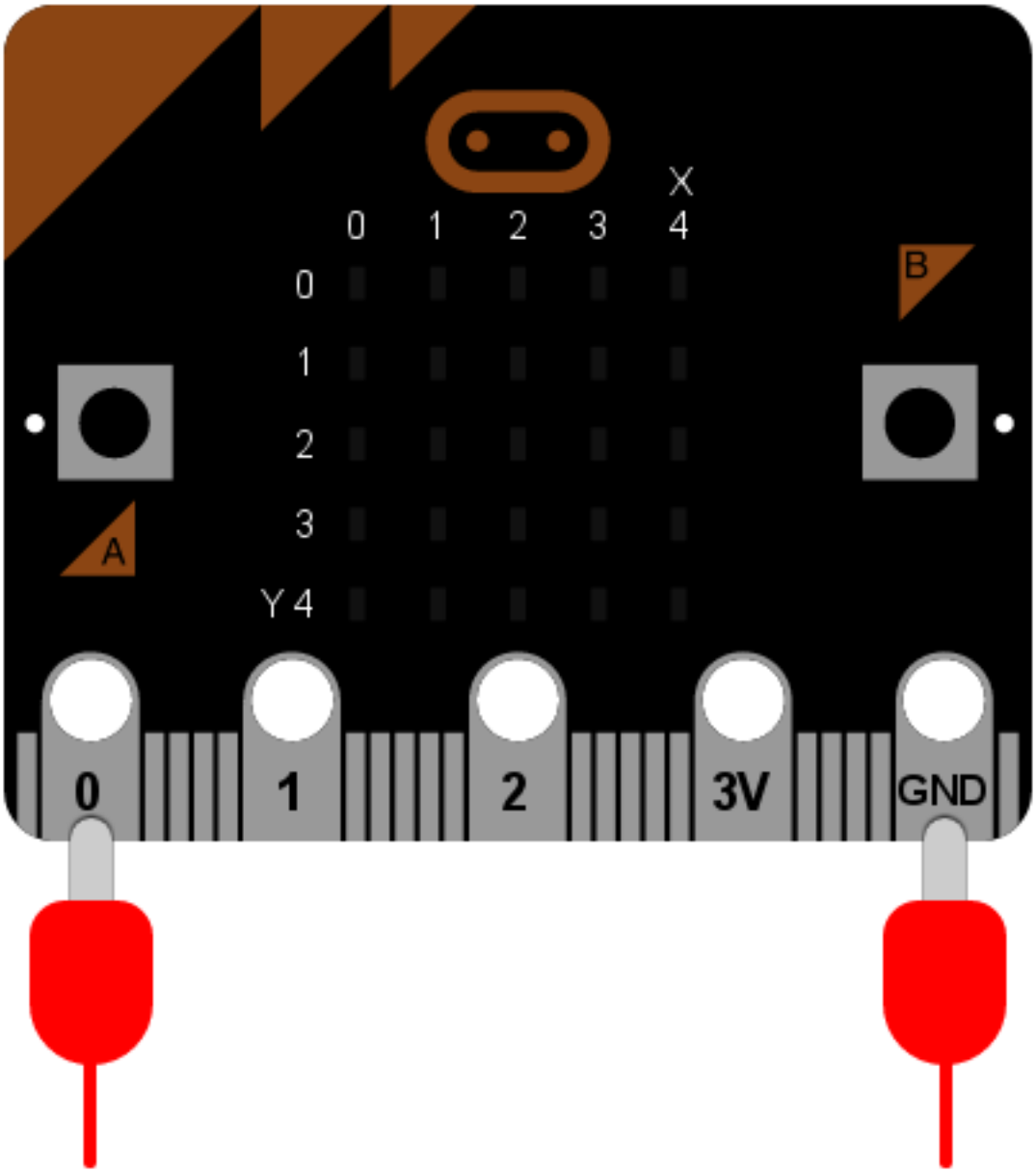
Czy domyślasz się jak działa ten skrypt? Pamiętaj, że w cyfrowym świecie `1` oznacza „włączony”, zaś `0` „wyłączony”.

Urządzenie jest wprowadzane w nieskończoną pętlę i natychmiast włącza pin 0. To powoduje, że brzęczyk wydaje pisk. Podczas gdy brzęczyk piszczy, urządzenie usypia na 20 milisekund, a następnie wyłącza pin 0. Daje to efekt krótkiego pisku. W końcu, urządzenie usypia na 480 milisekund, zanim rozpocznie pętlę od początku. To znaczy, że usłyszysz dwa piski na sekundę (jeden co 500 milisekund).

Stworzyliśmy bardzo prosty metronom!

1.5 Muzyka

MicroPython na BBC micro:bit zawiera bogaty w możliwości moduł muzyki i dźwięku. Dzięki niemu można w prosty sposób zaprogramować BBC micro:bit do wygenerowania piśnięć i innych odgłosów. Oczywiście, aby coś usłyszeć trzeba *podłączyć głośniki* – za pomocą krokodylków podłącz pin 0 oraz GND do wejść głośnika. Nie ma znaczenia który zacisk do którego wejścia zostanie podłączony.



Informacja: Do tego celu nie używaj brzęczyka piezoelektrycznego, który może generować tylko pojedynczy ton.

Zagrajmy coś!:

```
import music

music.play(music.NYAN)
```

Zauważ, że importujemy moduł `music`. Zawiera on polecenia wykorzystywane do wytwarzania i kontrolowania dźwięków.

MicroPython ma dość dużo wbudowanych melodii. Oto ich kompletna lista:

- `music.DADADADUM`

- `music.ENTERTAINER`
- `music.PRELUDE`
- `music.ODE`
- `music.NYAN`
- `music.RINGTONE`
- `music.FUNK`
- `music.BLUES`
- `music.BIRTHDAY`
- `music.WEDDING`
- `music.FUNERAL`
- `music.PUNCHLINE`
- `music.PYTHON`
- `music.BADDY`
- `music.CHASE`
- `music.BA_DING`
- `music.WAWAWAWAA`
- `music.JUMP_UP`
- `music.JUMP_DOWN`
- `music.POWER_UP`
- `music.POWER_DOWN`

Pobierz któryś z przykładowych kodów i odtwórz melodię. Która podoba Ci się najbardziej? Masz pomysł na ich wykorzystanie jako sygnały?

1.5.1 Wolfgang Amadeusz Microbit

Zaprogramowanie swojej własnej melodii jest bardzo łatwe!

Każda nuta ma nazwę (np. `C#` lub `F`), oktawę (określającą jak wysoko lub nisko dźwięk ma być grany) oraz długość (przez jaki czas dźwięk ma być grany). Oktawy są oznaczone liczbami ~ 0 oznacza najniższą oktawę, oktawa 4 zawiera środkowe C, a 8 jest tak wysoko, że wyższej już nie będziesz potrzebować, chyba że zechcesz tworzyć muzykę dla psów. Długość dźwięku jest również wyrażana jako liczby. Im większa liczba tym dłużej nuta będzie odtwarzana. Te wielkości są od siebie zależne, np. po wprowadzeniu wartości `4`, nuta będzie odtwarzana 2 razy dłużej niż po wprowadzeniu `2` itd. Jeśli zamiast nuty wpiszesz „R”, to MicroPython w tym miejscu zrobi pauzę (ang. rest) o zadanej długości.

Każda nuta opisywana jest jako ciąg znaków:

```
NUTA[oktawa][:długość]
```

Na przykład, `"A1:4"` opisuje nutę `A` w oktawie numer `1` o długości `4`.

Utwórz listę nut, które stworzą melodię. Poniżej znajduje się zapis, dzięki któremu MicroPython zagra początek melodii „Panie Janie”:

```
import music

tune = ["C4:4", "D4:4", "E4:4", "C4:4", "C4:4", "D4:4", "E4:4", "C4:4",
        "E4:4", "F4:4", "G4:8", "E4:4", "F4:4", "G4:8"]
music.play(tune)
```

Informacja: MicroPython umożliwia zapisywanie takich melodii w uproszczony sposób. Zapamięta oktawę oraz długość nuty i będzie je wykorzystywał aż do wprowadzenia nowych wartości. Dzięki temu powyższy przykład może zostać zapisany tak:

import music

tune = [„C4:4”, „D”, „E”, „C”, „C”, „D”, „E”, „C”, „E”, „F”, „G:8”,

„E:4”, „F”, „G:8”]

music.play(tune)

Zauważ, że wartości określające oktawę i czas trwania są wpisywane tylko wtedy, kiedy są zmieniane. Dzięki temu kod jest łatwiejszy zarówno do napisania jak i do odczytania.

1.5.2 Efekty dźwiękowe

MiroPython umożliwia tworzenie melodii i dźwięków zapisanych inaczej niż nutami. Na przykład dzięki poniższemu kodowi możemy uzyskać efekt syreny policyjnej:

```
import music

while True:
    for freq in range(880, 1760, 16):
        music.pitch(freq, 6)
    for freq in range(1760, 880, -16):
        music.pitch(freq, 6)
```

Zwróć uwagę, że w tym przypadku użyliśmy metody `music.pitch`. Wymaga ona podania częstotliwości. Na przykład częstotliwość 440 to koncertowe A używane do strojenia instrumentów w orkiestrze symfonicznej.

W powyższym przykładzie funkcja `range` (ang. zakres) jest wykorzystana do wygenerowania zakresów numerycznych wartości. Te liczby są użyte do zdefiniowania wysokości tonu. Te trzy argumenty dla funkcji `range` to odpowiednio wartość początkowa zakresu, wartość końcowa i krok. Zatem pierwsze użycie `range`, mówi po polsku „utwórz przedział liczb pomiędzy 880 a 1760 o kroku 16”. Drugie użycie `range` mówi „utwórz przedział wartości pomiędzy 1760 a 880 o kroku -16”. W ten sposób uzyskamy zakres częstotliwości, które stopniowo zwiększając się i zmniejszając tworzą dźwięk przypominający syrenę.

Ponieważ dźwięk syreny powinien trwać nieskończenie długo, jest on wpisany w niekończącą się pętlę `while`.

Co ważne, wprowadziliśmy nowy rodzaj pętli wewnątrz pętli `while` (ang. dopóki): pętlę `for` (ang. dla). Po polsku to jest tak, jak powiedzieć „dla każdego elementu w pewnym zbiorze, wykonaj z nim pewną czynność”. Konkretnie w powyższym przypadku to oznacza „dla każdej częstotliwości w określonym w zakresie, graj ton o tej częstotliwości przez 6 milisekund”. Zwróć uwagę na wcięcia przy każdym wierszu wewnątrz pętli `for` (mówiliśmy o tym wcześniej), sprawiają one, że Python dokładnie wie, który kod uruchomić aby obsłużyć poszczególne elementy.

1.6 Losowość

Czasem chcesz pozostawić rzeczy szczęściu, lub odrobinę je pomieszać. Słowem: chcesz, by urządzenie działało losowo.

MicroPython zawiera moduł `random`, wprowadzający element losowy oraz odrobinę chaosu do Twojego kodu. Oto przykładowy kod przewijający na wyświetlaczu losowe imię:

```
from microbit import *
import random

names = ["Mary", "Yolanda", "Damien", "Alia", "Kushal", "Mei Xiu", "Zoltan" ]

display.scroll(random.choice(names))
```

Lista (`names`) zawiera siedem imion zdefiniowanych jako łańcuchy znaków. Ostatnia linia jest *zagnieżdżona* (efekt „cebuli" zaprezentowany wcześniej): metoda `random.choice` przyjmuje jako argument listę `names` oraz zwraca jej losowy element. Element ten (losowo wybrane imię) jest argumentem dla `display.scroll`.

Czy możesz zmodyfikować listę tak, by zawierała wybrane przez Ciebie imiona?

1.6.1 Losowe Liczby

Losowe liczby są bardzo użyteczne. Często używane są w grach. Po cóż innego mamy kostki?

MicroPython zawiera wiele użytecznych metod powiązanych z losowymi liczbami. Oto przykład prostej kości do gry:

```
from microbit import *
import random

display.show(str(random.randint(1, 6)))
```

Po każdym restarcie urządzenia, wyświetla ono numer z zakresu 1-6. Zaczynasz zapoznawać się z *zagnieżdżaniem*, warto więc zauważyć, że `random.randint` zwraca liczbę całkowitą z podanego przedziału włącznie (liczba całkowita nazywana jest integer, stąd nazwa metody). Zauważ, że skoro `display.show` oczekuje znaku alfabetycznego, używamy funkcji `str` do zamiany numeru na znak (na przykład: `6` na `"6"`).

W wypadku, gdy zawsze potrzebujesz liczny z przedziału pomiędzy `0` a `N`, używaj metody `random.randrange`. Gdy podasz jej pojedynczy argument, zwróci losową liczbę całkowitą z przedziału od `0` do `N` włącznie (w przeciwieństwie do funkcji `random.randint`).

Czasami potrzebujesz liczby z miejscem po przecinku. Nazywamy je liczbami *zmiennoprzecinkowymi* i można je uzyskać używając metody `random.random`. Zwraca ona wartość z przedziału domkniętego od `0.0` do `1.0`. Jeżeli potrzebujesz dużej liczby zmiennoprzecinkowej dodaj `random.randrange` do `random.random`, jak poniżej:

```
from microbit import *
import random

answer = random.randrange(100) + random.random()
display.scroll(str(answer))
```

1.6.2 Ziarno Chaosu

Losowe liczby generowane przez komputery nie są naprawdę losowe. Zwracają jedynie pseudolosowe rezultaty zaczynając od ziarna. Wartość ziarna generowana jest na podstawie pseudolosowej wartości, takiej jak obecna godzina i/lub odczyty z sensorów sprzętowych.

Czasem chcesz osiągnąć powtarzalne, pseudolosowe zachowanie: odtwarzalne ziarno. To jakby oczekiwać tych samych, losowych wyników pięciu kolejnych rzutów kością.

Można to łatwo osiągnąć ustalając odgórnie wartość ziarna. Dla danego ziarna generator liczb losowych będzie tworzył tę samą pulę liczb losowych. Ziarno jest ustawiane poprzez `random.seed` jako dowolna liczba całkowita. Poniższa wersja programu do rzutu kością poda zawsze ten sam rezultat:

```
from microbit import *
import random

random.seed(1337)
while True:
    if button_a.was_pressed():
        display.show(str(random.randint(1, 6)))
```

Czy dostrzegasz, czemu powyższy program oczekuje wciśnięcia klawisza A, w przeciwieństwie do restartu urządzenia z poprzedniego przykładu?

1.7 Ruch

BBC micro:bit ma wbudowany akcelerometr. Mierzy on ruch wzdłuż trzech osi:

*X - przechylanie od lewej do prawej. *Y - przechylanie do przodu i do tyłu. *Z - poruszanie górę i w dół.

Dla każdej z osi jest metoda zwracająca liczbę dodatnią lub ujemną, wskazując pomiar w mili-g. Kiedy wyświetlacz pokazuje 0, oznacza to, że przyrząd jest równoległy do danej osi.

Poniższy przykład pokazuje bardzo prostą poziomicę wykorzystującą metodę `get_x` mierzącą jak bardzo urządzenie jest wypoziomowane względem osi X.

from microbit import *

while True: reading = accelerometer.get_x() if reading > 20:

display.show(„R”)

elif reading < -20: display.show(„L”)

else: display.show(„-„)

Jeżeli trzymasz urządzenie płasko, powinno ono wyświetlić −; jednak jeżeli tylko obrócisz je w lewo lub w prawo, pokaże ono odpowiednio `L` lub `R`.

Chcemy, aby urządzenie natychmiastowo reagowało na zmiany, dlatego też użyliśmy nieskończonej pętli `while`. Pierwsza rzecz, która stanie się *w ciele pętli*, to pomiar wzdłuż osi X nazwany `reading` (pol. odczyt). Ponieważ akcelerometr jest *bardzo* czuły, urządzenie jest uznane za wypoziomowane kiedy wartości znajdują się w granicy +/- 20. Dlatego właśnie warunki `if` oraz `elif` sprawdzają dla wartości `>20` oraz `<-20`. Wyrażenie `else` oznacza, że jeżeli `reading` znajduje się pomiędzy -20 i 20, to urządzenie uważane jest za wypoziomowane. Dla każdego z tych warunków do pokazania odpowiednich znaków używamy wyświetlacza.

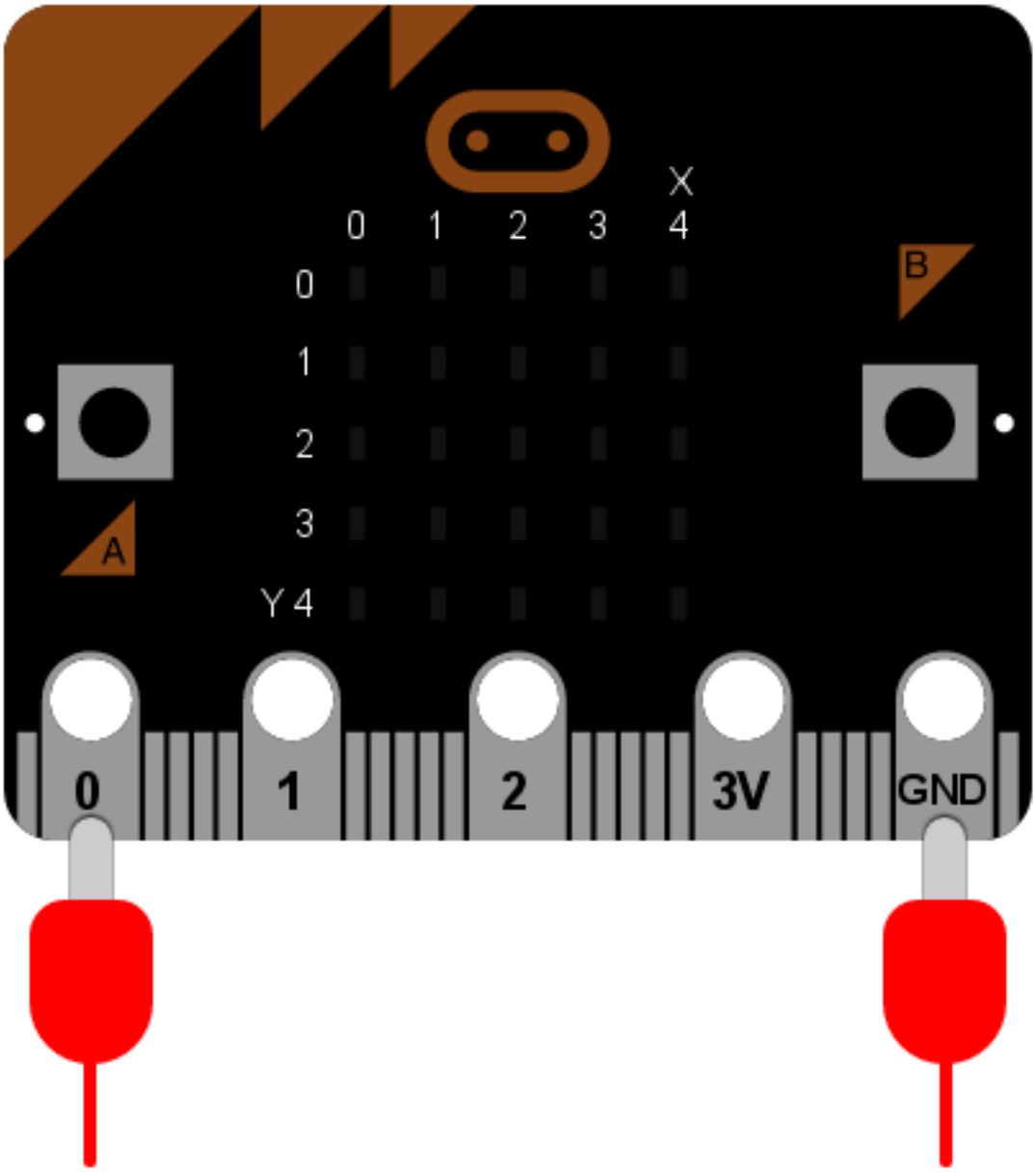
Istnieje również metoda `get_y` dla osi Y oraz `get_z` dla osi Z.

Zastanawiałeś się kiedyś w jaki sposób telefon komórkowy rozpoznaje w którą stronę jest zwrócony i w jaki sposób ma być zorientowany wyświetlany obraz na ekranie? Potrafi to właśnie dzięki wbudowanemu akcelerometrowi działającemu dokładnie jak ten w powyższym programie. Kontrolery gier również zawierają akcelerometry umożliwiające sterowanie i poruszanie się w grach.

1.7.1 Muzyczny Zamęt

Jednym z najcudowniejszych aspektów MicroPython na BBC micro:bit jest łatwość z jaką można łączyć różne możliwości tego urządzenia. Na przykład zamieńmy go (w pewnym sensie) w instrument muzyczny.

Podłącz głośnik zgodnie z instrukcją w rozdziale Muzyka. Użyj krokodylków. Połącz styki 0 oraz GND z dodatnim i ujemnym wejściem głośnika – nie ma znaczenia które wejście z którym stykiem zostanie połączone.



Co się stanie jak wykorzystamy odczyty z akcelerometru i stworzymy na ich podstawie dźwięki o danym tonie? Przekonajmy się:

```
from microbit import *
import music

while True:
    music.pitch(accelerometer.get_y(), 10)
```

Kluczowa linijka znajduje się na końcu i jest niewiarygodnie prosta. *Zagnieżdżamy* odczyt z osi Y jako częstotliwość wprowadzaną do metody `music.pitch`. Ponieważ chcemy, żeby ton był zmieniany tak szybko jak urządzenie jest przechylane, pozwalamy mu grać tylko przez 10 milisekund. Dzięki temu, że urządzenie jest w niekończącej się pętli `while`, nieustannie reaguje na zmiany w odczytach osi Y.

To wystarczy!

Przechyl urządzenie do przodu i do tyłu. Jeżeli odczyt wzdłuż osi Y jest dodatni, zmieni wysokość dźwięku granego przez micro:bit.

Wyobraź sobie całą orkiestrę symfoniczną takich urządzeń. Możesz zagrać melodię? Jakie zmiany wprowadziłbyś do programu, aby micro:bit brzmiał bardziej muzykalnie?

1.8 Gesty

Naprawdę interesującym efektem ubocznym posiadania akcelerometru jest wykrywanie gestów. Jeśli poruszysz swoim micro:bitem w pewien specjalny sposób (jak przy wykonywaniu gestu), MicroPython jest w stanie to wykryć.

MicroPython potrafi rozpoznać następujące gesty: `up` (ang. w górę), `down` (ang. w dół), `left` (ang. w lewo), `right` (ang. w prawo), `face up` (ang. zwrócony w górę), `face down` (ang. zwrócony w dół), `freefall` (ang. swobodne spadanie), `3g`, `6g`, `8g`, `shake` (ang. potrząśnięcie). Gesty zawsze są reprezentowane przez ciągi znaków. Podczas gdy większość z tych nazw powinna być oczywista, gesty `3g`, `6g` oraz `8g` odnoszą się do sytuacji, w której na urządzenie oddziałują odpowiadające poziomy przeciążenia (jak na astronautę, kiedy jest wystrzeliwany w kosmos).

Aby pobrać aktualny gest użyj metody `accelerometer.current_gesture`. Jej wynikiem będzie jeden z nazwanych gestów z powyższej listy. Na przykład, ten program sprawi, że Twoje urządzenie będzie szczęśliwe tylko, kiedy będzie zwrócone ku górze:

```
from microbit import *

while True:
    gesture = accelerometer.current_gesture()
    if gesture == "face up":
        display.show(Image.HAPPY)
    else:
        display.show(Image.ANGRY)
```

Ponieważ chcemy, aby urządzenie reagowało na zmieniające się okoliczności, używamy pętli `while`. Wewnątrz pętli aktualny gest jest odczytywany i podstawiany pod zmienną `gesture`. Warunek `if` sprawdza, czy `gesture` jest równa `"face up"` (Python używa == aby sprawdzić równość, pojedynczy znak równości = jest używany do przypisywania - tak samo jak przypisujemy odczyt gestu do objektu `gesture`). Jeżeli `gesture` jest równa `"face up"`, używamy wyświetlacza do pokazania szczęśliwej miny. W przeciwnym przypadku, urządzenie wyświetli rozzłoszczoną minę.

1.8.1 Magic-8

Magiczna bila 8 to zabawka wynaleziona w latach pięćdziesiątych. Pomysł polega na zadaniu pytania typu tak lub nie, potrząśnięciu bilą i poczekaniu, aż wyjawi prawdę. Raczej łatwo jest napisać działający w taki sposób program:

```
from microbit import *
import random

answers = [
    "To pewne",
    "Zdecydowanie tak",
```

```
    "Bez wątpienia",
    "Tak, z pewnością",
    "Możesz na tym polegać",
    "Jak ja to widzę, tak",
    "Najprawdopodobniej",
    "Wygląda dobrze",
    "Tak",
    "Znaki wskazują na tak",
    "Odpowiedź niejasna, spróbuj ponownie",
    "Zapytaj ponownie później",
    "Lepiej, abym teraz nie powiedziała",
    "Nie mogę teraz przewidzieć",
    "Skoncentruj się i zapytaj ponownie",
    "Nie licz na to",
    "Moja odpowiedź brzmi nie",
    "Moje źródła mówią nie",
    "Nie wygląda to dobrze",
    "Bardzo wątpliwe",
]

while True:
    display.show("8")
    if accelerometer.was_gesture("shake"):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(answers))
```

Większość programu to tablica nazwana `answers` (ang. odpowiedzi). Sama gra znajduje się w pętli `while` na końcu.

Domyślny stan gry wyświetla cyfrę `"8"`. Jednak program musi wykryć, że nastąpiło potrząśnięcie. Metoda `was_gesture` używa swojego argumentu (w tym przypadku ciągu `"shake"`, ponieważ chcemy wykryć potrząśnięcie) i zwraca `True` lub `False`.

Jeżeli urządzenie było potrząśnięte, warunek `if` wykonuje swój blok kodu, w którym czyści ekran, czeka przez sekundę (aby urządzenie wyglądało, jakby zastanawiało się nad twoim pytaniem), a następnie wyświetla losowo wybraną odpowiedź.

Nie masz wrażenia, że jest to najlepszy program na świecie? Co mógłbyś zrobić, aby „oszukać" i uzyskać zawsze pozytywną lub negatywną odpowiedź? (Podpowiedź: użyj przycisków).

1.9 Kierunek

Na urządzeniu BBC micro:bit znajduje się kompas. Jeżeli kiedykolwiek zrobisz stację pogodową, możesz użyć micro:bita do sprawdzania kierunku wiatru.

1.9.1 Kompas

Urządzenie może również wskazać kierunek północny:

```
from microbit import *

compass.calibrate()

while True:
```

```
    needle = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[needle])
```

Informacja: **Przed dokonaniem pomiaru należy skalibrować urządzenie.** W innym wypadku rezultaty będą niepoprawne. Aby urządzenie zorientowało się w swoim ustawieniu względem pola magnetycznego Ziemi, metoda `calibration` uruchamia małą, śmieszną grę.

Aby skalibrować kompas, obracaj micro:bitem, dopóki krańcowe piksele wyświetlacza nie utworzą koła.

Program używa `compass.heading` i, wraz z prostą, acz piękną matematyką, podłogą i sufitem `//` oraz resztą z dzielenia `%`, wyprowadza pozycję wskazówki zegara, po czym wyświetla ją tak, by pokazywała mniej więcej północ.

1.10 Przechowywanie danych

Czasami potrzebujesz trwale zachować użyteczne informacje. Takie informacje przechowywane są jako dane (cyfrowa reprezentacja informacji przechowywanych na komputerze). Zachowane na komputerze dane powinny przetrwać nawet jeśli wyłączysz i ponownie włączysz urządzenie.

Na szczęście MicroPython na minikomputerze micro:bit pozwala Ci to osiągnąć za pomocą prostego systemu plików. Z powodu ograniczeń pamięci, na system plików dostępne jest tylko **około 30k**.

Czym jest system plików?

To sposób przechowywania i organizowania danych w sposób trwały - dane zapisane w systemie plików powinny przetrwać restart urządzenia. Tak jak sugeruje to nazwa dane zapisywane w systemie plików przechowywane są jako pliki.

(W języku angielskim słowo file, tłumaczone jako plik, oznacza także dokument - przyp. tłum.)

Plik (komputerowy) jest posiadającym nazwę zasobem cyfrowym, który przechowywany jest w systemie plików. Zasoby takie przechowują użyteczne informacje jako dane. To swego rodzaju oznakowany pojemnik przechowujący informacje. Zazwyczaj nazwa pliku oddaje to co plik zawiera. Zwyczajowo nazwy plików komputerowych kończy się sufiksem `.coś`. Zazwyczaj to `.coś` wskazuje jaki typ danych został użyty do reprezentacji informacji. Na przykład, `.txt` oznacza plik tekstowy, `.jpg` obraz w formacie JPEG a `.mp3` dane dźwiękowe przetworzone do postaci cyfrowej algorytmem MP3.

Niektóre systemy plików (taki jak ten na Twoim laptopie albo komputerze stacjonarnym) pozwalają na organizowanie plików w katalogi – posiadające nazwę pojemniki, które grupują powiązane ze sobą pliki i podkatalogi. Jednakże *system plików dostępny w MicroPythonie jest płaskim systemem plików*. Płaski system plików nie posiada katalogów – wszystkie Twoje pliki są po prostu przechowywane w jednym i tym samym miejscu.

Język programowania Python zawiera łatwe w użyciu i bardzo skuteczne metody pracy z komputerowym systemem plików. W MicroPythonie na minikomputerze micro:bit zaimplementowano użyteczny podzbiór tych metod. Pozwalają one w łatwy sposób odczytywać i zapisywać pliki z i do urządzenia, jednocześnie zapewniając zgodność z innymi wersjami Pythona.

Ostrzeżenie: Wgrywanie programów na minikomputer micro:bit KASUJE WSZYSTKIE DANE ponieważ operacja ta nadpisuje pamięć flash używaną przez urządzenie, a system plików także jest w niej przechowywany.

Jeżeli jedynie wyłączysz swoje urządzenie, dane pozostaną nienaruszone do czasu, aż je usuniesz albo załadujesz na urządzenie nowy program.

1.10.1 Sezamie otwórz się

Odczyt i zapis w systemie plików wykonywany jest za pomocą funkcji `open` (ang. otwórz). Kiedy plik jest już otwarty możesz z nim robić rozmaite rzeczy, do czasu aż go zamkniesz. To bardzo ważne abyś zamykał pliki po to, żeby MicroPython wiedział, że skończyłeś z nimi pracować.

Aby mieć pewność, że tak się stanie najlepszym sposobem jest użycie instrukcji `with` w następujący sposób:

```
with open('story.txt') as my_file:
    content = my_file.read()
print(content)
```

Instrukcja `with` używa funkcji `open` aby otworzyć plik i przypisać go do obiektu. W powyższym przykładzie funkcja `open` otwiera plik `story.txt` (zapewne tekstowy plik z jakąś historyjką [ang. story - opowieść]). Obiekt, który w Pythonie reprezentuje ten plik nazywa się `my_file` (ang. mój_plik). Następnie, we wciętej linijce programu pod instrukcją `with`, obiekt `my_file` zostaje użyty aby odczytać (funkcja `read()`) zawartość pliku i przypisać ją do obiektu `content`.

Ważna uwaga: *następna linijka zawierająca instrukcję* `print` *nie jest wcięta*. Blok programu związany z instrukcją `with` złożony jest tylko z jednej linijki wczytującej plik. Jak tylko blok programu związany z instrukcją `with` zostanie zamknięty Python (a także MicroPython) automatycznie zamknie plik za Ciebie. To tak zwana obsługa kontekstu – funkcja `open` tworzy obiekty, które są identyfikatorami kontekstu dla plików.

Mówiąc prościej, zakres programu, w którym operujesz na pliku jest związany z blokiem instrukcji `with`, która służy do jego otwarcia.

Zdezorientowany?

Niepotrzebnie. Mówię po prostu, że Twój program powinien wyglądać w następujący sposób:

> **with open(«some_file») as some_object:** # W tym bloku programu, związanym z instrukcją `with`, # zrób coś z obiektem some_object.
>
> # Kiedy MicroPython dojdzie do końca tego bloku, # automatycznie zamknie plik za Ciebie.

Plik komputerowy otwieramy z dwóch powodów – aby odczytać jego zawartość (jak to pokazano powyżej), albo żeby do niego coś zapisać. Domyślnym trybem jest tryb odczytu. Jeśli chcesz pisać do pliku musisz powiedzieć o tym funkcji `open` w następujący sposób:

> **with open(«hello.txt», «w») as my_file:** my_file.write(„Hello, World!”)

Zwróć uwagę na argument `'w'`, który został użyty aby ustawić obiekt `my_file` w tryb zapisu. Aby ustawić obiekt pliku w tryb odczytu, możesz przekazać argument `'r'`, ale ponieważ jest to zachowanie domyślne rzadko się to robi.

Zapis danych do pliku odbywa się za pomocą metody (dobrze odgadłeś) `write` (ang. pisz). Przyjmuje ona, jako argument, łańcuch znaków (ang. string), który chcesz zapisać do pliku. W powyższym przykładzie zapisałem do pliku „hello.txt” napis „Hello, World!”.

Proste!

Informacja: Kiedy otwierasz plik aby do niego pisać (możliwe, że nawet wielokrotnie, dopóki jest otwarty) będziesz nadpisywać jego zawartość jeśli plik już istnieje.

Jeśli chcesz dodać dane do pliku powinieneś/powinnaś najpierw go wczytać, zachować gdzieś jego zawartość, zamknąć go, dodać (nowe) dane do zachowanej zawartości a następnie otworzyć plik ponownie i wpisać uaktualnioną zawartość.

Tak właśnie postępuje MicroPython; „zwyczajny” Python potrafi otwierać pliki w trybie dodawania („append”). Powodem, dla którego nie możemy tego zrobić na minikomputerze micro:bit jest uproszczony sposób implementacji systemu plików.

1.10.2 Na pomoc systemie operacyjny

Oprócz czytania z i pisania do plików, Python potrafi także wykonywać z nimi inne operacje. Z pewnością chcesz wiedzieć jakie pliki znajdują się w systemie plików, a czasami także je z niego usunąć.

Na zwykłym komputerze, w imieniu Pythona, to system operacyjny (taki jak Windows, OSX czy Linux) zarządza plikami. W Pythonie odpowiednie do tego funkcje dostępne są za pośrednictwem modułu `os`. Ponieważ jednak sam MicroPython **jest** systemem operacyjnym postanowiliśmy, dla spójności, pozostawić te funkcje w module `os`, tak abyś wiedział(a) gdzie je znaleźć gdy będziesz używać `normalnej` wersji Pythona na urządzeniach takich jak laptop czy Raspberry Pi.

Zasadniczo możesz wykonać trzy operacje związane z systemem plików: uzyskać listę plików, usunąć plik oraz zapytać o rozmiar pliku.

Aby uzyskać listę plików w Twoim systemie plików użyj funkcji `listdir`. Zwraca ona listę łańcuchów znakowych reprezentujących nazwy plików:

```
import os
my_files = os.listdir()
```

Aby skasować plik użyj funkcji `remove` (ang. usuń). Przyjmuje ona jako argument łańcuch znakowy reprezentujący nazwę pliku, który chcesz skasować w następujący sposób:

```
import os
os.remove('filename.txt')
```

Wreszcie, czasami przydaje się wiedzieć jak duży jest plik zanim rozpocznie się jego wczytywanie. Osiągniesz to używając funkcji `size` (ang. wielkość). Tak jak funkcja `remove` przyjmuje ona łańcuch znaków reprezentujący nazwę pliku, którego rozmiar chcesz poznać. Funkcja zwraca wartość całkowitą oznaczającą liczbę bajtów, które zajmuje plik:

```
import os
file_size = os.size('a_big_file.txt')
```

Bardzo fajnie, że na minikomputerze dostępny jest system plików, ale co jeśli chcemy pobrać z niego albo załadować na niego plik z zewnątrz?

Użyj po prostu narzędzia `microfs`!

1.10.3 Przesyłanie plików

Jeśli na komputerze, którego używasz do oprogramowywania swojego BBC micro:bit, masz zainstalowanego Pythona, możesz użyć specjalnego narzędzia zwanego `microfs` (nazywanego skrótowo `ufs` jeśli używany jest z linii komend). Instrukcję instalacji i pełny opis użycia wszystkich funkcji znajdziesz w dokumentacji tego narzędzia.

Niemniej jednak większość potrzebnych rzeczy można wykonać za pomocą czterech prostych poleceń:

```
$ ufs ls
story.txt
```

Komenda `ls` wyświetla listę plików znajdujących się w systemie plików (została nazywa tak samo jak dobrze znana Uniksowa komenda `ls`, która ma dokładnie to samo zadanie).

```
$ ufs get story.txt
```

Komenda `get` pobiera plik z podłączonego minikomputera micro:bit i zapisuje go do bieżącego katalogu na Twoim komputerze (nazwana została tak samo jak komenda `get` popularnego protokołu przesyłu plików – FTP, która spełnia to samo zadanie).

```
$ ufs rm story.txt
```

Komenda `rm` usuwa plik o podanej nazwie z systemu plików na podłączonym micro:bit (nazwana została tak samo jako komanda Uniksowa, która spełnia to samo zadanie).

```
$ ufs put story2.txt
```

W końcu komenda `put` umieszcza plik z komputera na podłączonym do niego micro:bit (nazywa się tak jak, wykonująca tę samą funkcję, komenda protokołu FTP).

1.10.4 Głównie main.py

System plików ma ciekawą cechę - jeśli na urządzenie załadujesz jedynie środowisko uruchomieniowe MicroPython, po włączeniu czeka ono po prostu na wykonanie jakichś poleceń. Jednakże, jeśli skopiujesz również specjalny plik nazwany `main.py` (ang. main: główny), po restarcie urządzenia MicroPython wykona zawartość tego specjalnego pliku.

Dodatkowo jeśli przekopiujesz inne pliki Pythona do systemu plików minikomputera, możesz je zaimportować (`import`) tak jak każdy moduł Pythona. Na przykład gdybyś miał plik `hello.py` zawierający następujący program:

```
def say_hello(name="World"):
    return "Hello, {}!".format(name)
```

... mógłbyś zaimportować i użyć funkcji `say_hello` w następujący sposób:

```
from microbit import display
from hello import say_hello

display.scroll(say_hello())
```

Oczywiście wynikiem będzie napis „Hello, World!" przewijający się przez wyświetlacz. Ważne jest, że w tym przykładzie funkcje rozdzielone są na dwa moduły a instrukcja `import` służy do współdzielenia kodu.

Informacja: Jeśli, na urządzenie, oprócz środowiska uruchomieniowego wgrałeś także skrypt, MicroPython zignoruje plik `main.py` i zamiast tego wykona polecenia ze skryptu.

Aby wgrać tylko środowisko uruchomieniowe, upewnij się że skrypt, który wpisałeś w edytorze, jest pusty (nie ma w nim żadnych znaków). Po załadowaniu będziesz mógł przekopiować plik `main.py`.

1.11 Mowa

Ostrzeżenie: UWAGA! TO JEST KOD ALFA!

Zastrzegamy sobie prawo do zmiany tego interfejsu API w miarę rozwoju oprogramowania.

Jakość mowy nie jest świetna, tylko „wystarczająco dobra". Ze względu na ograniczenia urządzenia, mogą wystąpić błędy pamięci i / lub nieoczekiwane dodatkowe dźwięki podczas odtwarzania. To są dopiero początki i cały czas poprawiamy kod syntezatora mowy. Zgłaszanie błędów będzie mile widziane.

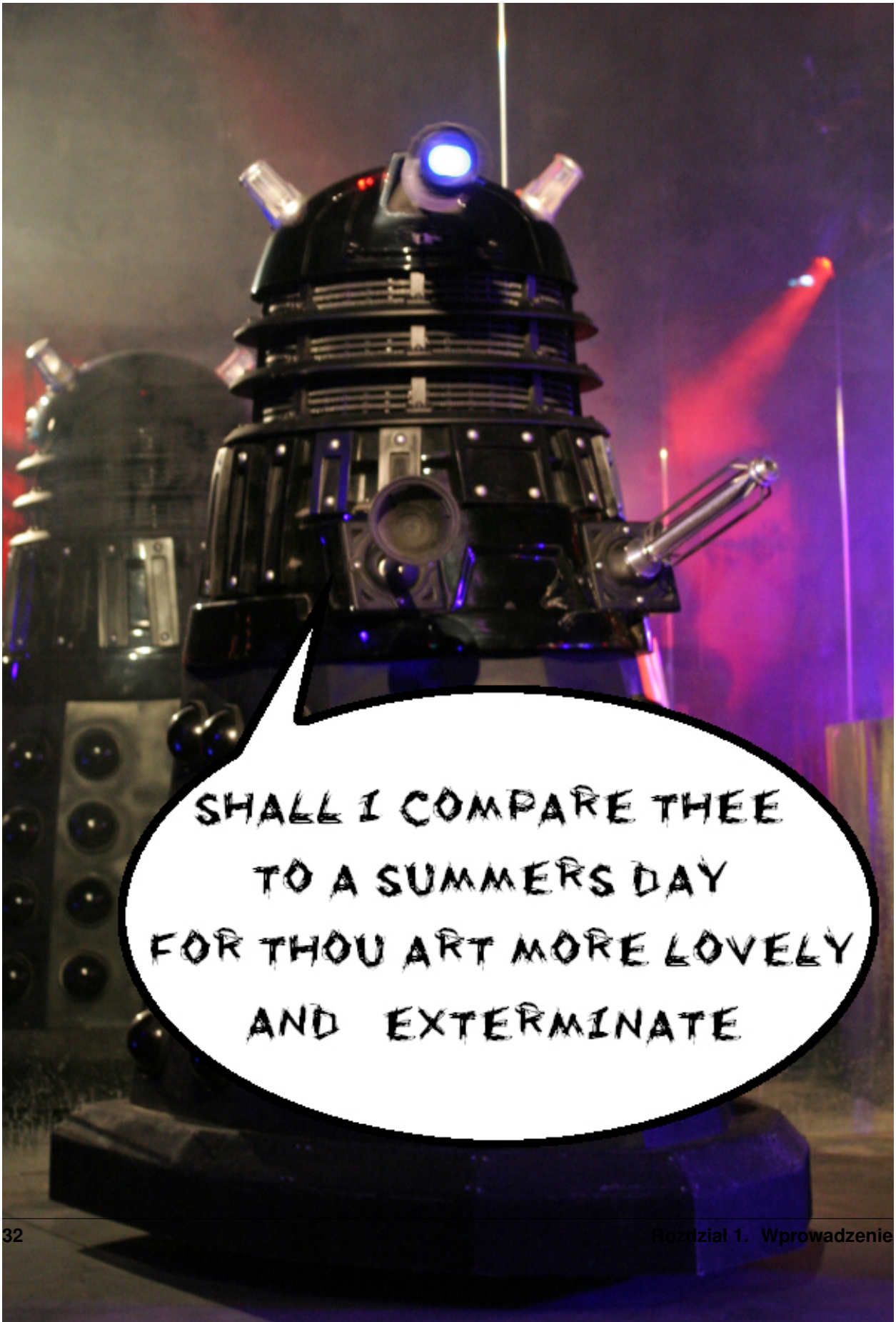
Komputery i roboty, które mówią wydają się bardziej „ludzkie".

Często o tym co komputer właśnie robi dowiadujemy się poprzez graficzny interfejs użytkownika (ang. graphical user interface, GUI). W przypadku BBC micro:bit, GUI to matryca LED 5x5, która pozostawia wiele do życzenia.

Sprawienie, aby micro:bit mówił, jest jednym ze sposobów na wyrażenie informacji w zabawny, skuteczny i użyteczny sposób. W tym celu zintegrowaliśmy prosty syntezator mowy oparty na zdekonstruowanej wersji syntezatora z wczesnych lat osiemdziesiątych. Brzmi on naprawdę uroczo, w stylu „wszyscy ludzie muszą umrzeć".

Mając to na uwadze, użyjemy syntezatora do stworzenia...

1.11.1 Poezji DALEKów



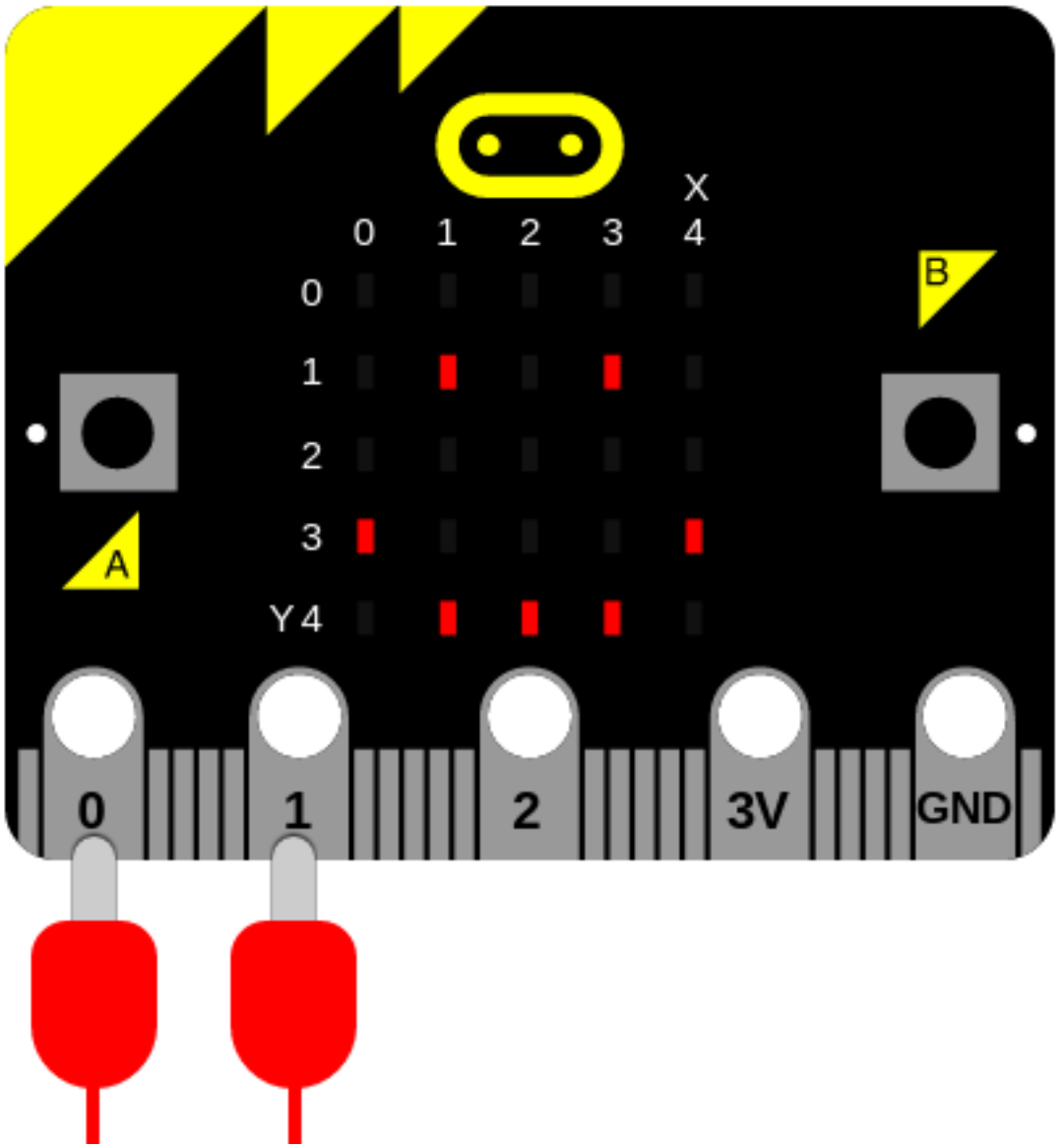
Mało kto wie, że DALEKowie lubią poezję – szczególnie limeryki. Szaleją na punkcie wierszy z anapestycznym metrum o surowej budowie AABBA. Kto by pomyślał!

(Jak się niżej dowiemy, to wina Doktora, że, ku irytacji Davrosa, DALEKowie lubią limeryki.)

W każdym razie, zamierzamy stworzyć recital poezji DALEKowej na żądanie.

1.11.2 Powiedz coś

Zanim urządzenie będzie mogło mówić, musisz podłączyć głośniczek w taki sposób:



Najłatwiejszym sposobem na to żeby urządzenie zaczęło mówić jest zaimportowanie modułu `speech` (pol. `mowa`) i użycie funkcji `say` (pol. `powiedz`) w taki sposób:

```
import speech

speech.say("Hello, World")
```

Jest to urocze, jednak nie jest wystarczająco DALEKowate jak na nasz gust, dlatego musimy zmienić niektóre parametry używane przez syntezator do wygenerowania mowy. Nasz syntezator mowy jest pod tym względem dość potężny, ponieważ możemy zmienić cztery parametry:

- `pitch` (pol. `ton`) - jak wysoko lub nisko brzmi głos (0 = wysoko, 255 = Barry White)
- `speed` (pol. `szybkość`) - jak szybko urządzenie mówi (0 = niemożliwe, 255 = opowiadanie na dobranoc)
- `mouth` (pol. `usta`) - czy mowa jest przez zaciśnięte zęby czy bardzo wyraźna (0 = kukiełka brzuchomówcy, 255 = Kurak Leghorn)
- `throat` (pol. `gardło`) - jak spokojny lub napięty jest ton głosu (0 = załamujący się, 255 = kompletnie rozluźniony)

W sumie razem parametry te kontrolują jakość dźwięku – tj. barwę dźwięku. Szczerze mówiąc, najlepszą drogą do uzyskania pożądanej barwy dźwięku jest eksperymentowanie, ocena i dostosowanie.

Aby dostosować ustawienia, przekazujesz je jako argumenty funkcji `say`. Więcej szczegółów można znaleźć w dokumentacji interfejsu API `speech`.

Po kilku przeprowadzonych eksperymentach, ten brzmi całkiem DALEKowato:

```
speech.say("I am a DALEK - EXTERMINATE", speed=120, pitch=100, throat=100, mouth=200)
```

1.11.3 Poezja na Żądanie

Będąc cyborgami, DALEKowie wykorzystują umiejętności robotów do tworzenia poezji i okazuje się, że korzystają z algorytmów napisanych w Pythonie, np:

```
# Generator poezji DALEKowej Doktora
import speech
import random
from microbit import sleep

# Losowo wybierz fragmenty do wstawienia do szablonu.
location = random.choice(["brent", "trent", "kent", "tashkent"])
action = random.choice(["wrapped up", "covered", "sang to", "played games with"])
obj = random.choice(["head", "hand", "dog", "foot"])
prop = random.choice(["in a tent", "with cement", "with some scent",
                      "that was bent"])
result = random.choice(["it ran off", "it glowed", "it blew up",
                        "it turned blue"])
attitude = random.choice(["in the park", "like a shark", "for a lark",
                          "with a bark"])
conclusion = random.choice(["where it went", "its intent", "why it went",
                            "what it meant"])

# Szablon wiersza. Nawiasy {} będą zastąpione nazwanymi fragmentami.
poem = [
    "there was a young man from {}".format(location),
    "who {} his {} {}".format(action, obj, prop),
    "one night after dark",
    "{} {}".format(result, attitude),
    "and he never worked out {}".format(conclusion),
    "EXTERMINATE",
]
```

```
# Wprowadź w pętlę każdą linię wiersza i użyj modułu
for line in poem:
    speech.say(line, speed=120, pitch=100, throat=100, mouth=200)
    sleep(500)
```

Jak pokazują komentarze, jest to bardzo prosty model:

- Nazwane fragmenty (`location`, `prop`, `attitude` itd.) są generowane losowo z predefiniowanych list możliwych wartości. Zwróć uwagę na użycie `random.choice` w celu wybrania pojedynczego elementu z listy.
- Szablon wiersza definiowany jest jako lista zwrotek z „dziurami" (oznaczonymi przez `{}`), do których za pomocą metody `format` wstawione zostaną nazwane fragmenty.
- Na koniec, Python przechodzi po wszystkich elementach na liście wypełniaczy poezji i używa `speech.say` z ustawieniami głosu DALEKa do recytowania wiersza. Między poszczególnymi liniami wstawiana jest pauza 500 milisekund, ponieważ nawet DALEKowie muszą odetchnąć.

Co ciekawe, oryginalne wytyczne związane z poezją zostały napisane przez Davrosa w języku FORTRAN (odpowiedni język dla DALEKów, gdyż pisany jest TYLKO WIELKIMI LITERAMI). Jednakże Doktor cofnął się w czasie dokładnie do punktu pomiędzy wprowadzaniem testów jednostkowych Davros'a i potoków wdrożeniowych . Wówczas był w stanie umieścić interpreter MicroPythona w systemie operacyjnym DALEKA i powyżej widzisz zawarty w bankach pamięci DALEKA kod z ukrytą niespodzianką czy też rickrollem od Władcy Czasu.

1.11.4 Fonemy

Zauważysz, że funkcja `say` nie tłumaczy dokładnie słów na odpowiednie dźwięki. Aby mieć lepszą kontrolę nad rezultatem, użyj fonemów: podstawowej jednostki struktury fonologicznej mowy.

Korzyść z używania fonemow jest taka, że nie musisz wiedzieć jak się poszczególne wyrazy pisze. Musisz tylko wiedzieć jak się to słowo wymawia, aby zapisać je fonetycznie.

Pełna lista fonemów rozumianych przez syntezator mowy znajduje się w dokumentacji API dla mowy. Ewentualnie, zaoszczędź sobie dużo czasu wprowadzając angielskie słowa do funkcji `translate` (pol. przetłumacz). Zwróci ona pierwsze przybliżenie fonemów, które micro:bit wykorzystałby do wygenerowania audio. Otrzymany rezultat może zostać ręcznie zmodyfikowany, żeby poprawić dokładność, fleksję i akcent (tak aby brzmiał bardziej naturalnie).

Funkcja `pronounce` (pol. wymówić) jest używana do wyprowadzania fonemu w następujący sposób:

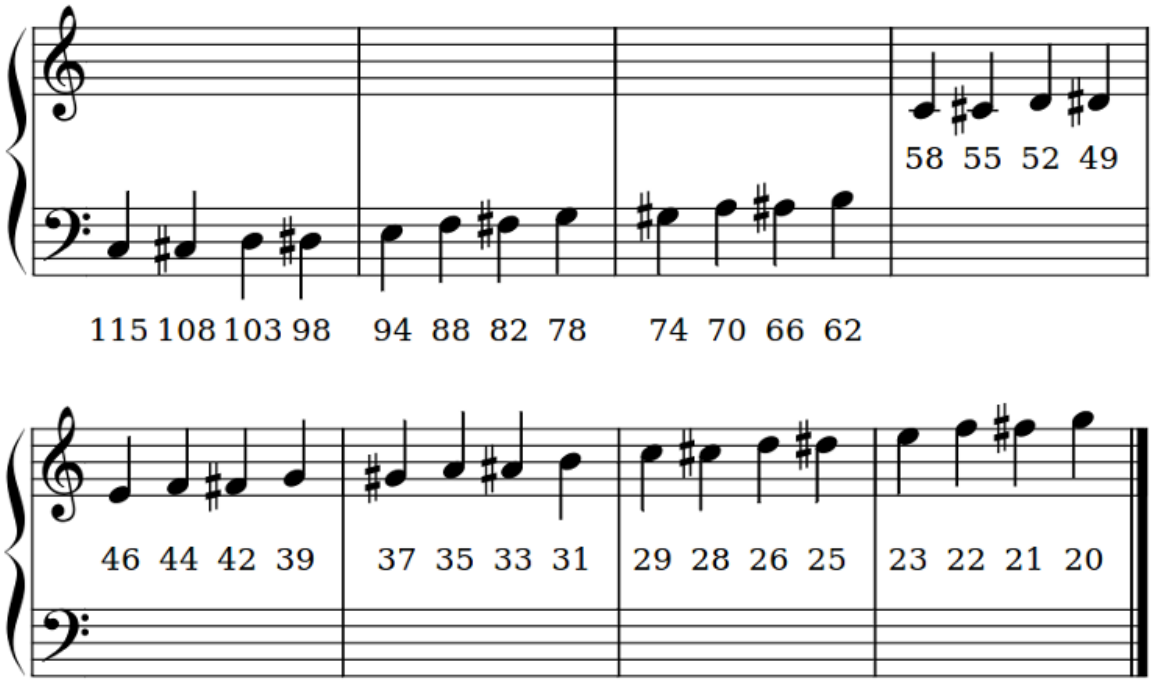
```
speech.pronounce("/HEH5EH4EH3EH2EH2EH3EH4EH5EHLP.")
```

Jak udoskonaliłbyś kod Doktora, aby używał fonemów?

1.11.5 Zaśpiewaj Piosenkę Micro:bit'a

Poprzez zmianę ustawień `pitch` (pol. ton) i wywołanie funkcji `sing` (pol. śpiewaj), urządzenie może zacząć śpiewać (jednakże bez szans na wygranie Eurowizji, póki co).

Mapowanie z numerów tonów na nuty jest pokazane poniżej:



Funkcja `sing` musi przyjąć jako dane wejściowe fonemy i tonację:

```
speech.sing("#115DOWWWW")
```

Zwróć uwagę na to, w jaki sposób fonemy poprzedzone są wysokością dźwięku z symbolem kratki (#). Tonacja pozostanie taka sama dla kolejnych fonemów do momentu wprowadzenia nowej tonacji.

Poniższy przykład pokazuje jak wszystkie trzy funkcje generujące (`say`, `pronounce` oraz `sing`) mogą zostać wykorzystane do stworzenia czegoś przypominającego mowę.

```
import speech
from microbit import sleep

# The say method attempts to convert English into phonemes.
speech.say("I can sing!")
sleep(1000)
speech.say("Listen to me!")
sleep(1000)

# Clearing the throat requires the use of phonemes. Changing
# the pitch and speed also helps create the right effect.
speech.pronounce("AEAE/HAEMM", pitch=200, speed=100)  # Ahem
sleep(1000)

# Singing requires a phoneme with an annotated pitch for each syllable.
solfa = [
    "#115DOWWWWWW",   # Doh
    "#103REYYYYYY",   # Re
    "#94MIYYYYYY",    # Mi
    "#88FAOAOAOAOR",  # Fa
    "#78SOHWWWWW",    # Soh
    "#70LAOAOAOAOR",  # La
```

```
    "#62TIYYYYYY",     # Ti
    "#58DOWWWWWW",     # Doh
]

# Sing the scale ascending in pitch.
song = ''.join(solfa)
speech.sing(song, speed=100)
# Reverse the list of syllables.
solfa.reverse()
song = ''.join(solfa)
# Sing the scale descending in pitch.
speech.sing(song, speed=100)
```

1.12 Network

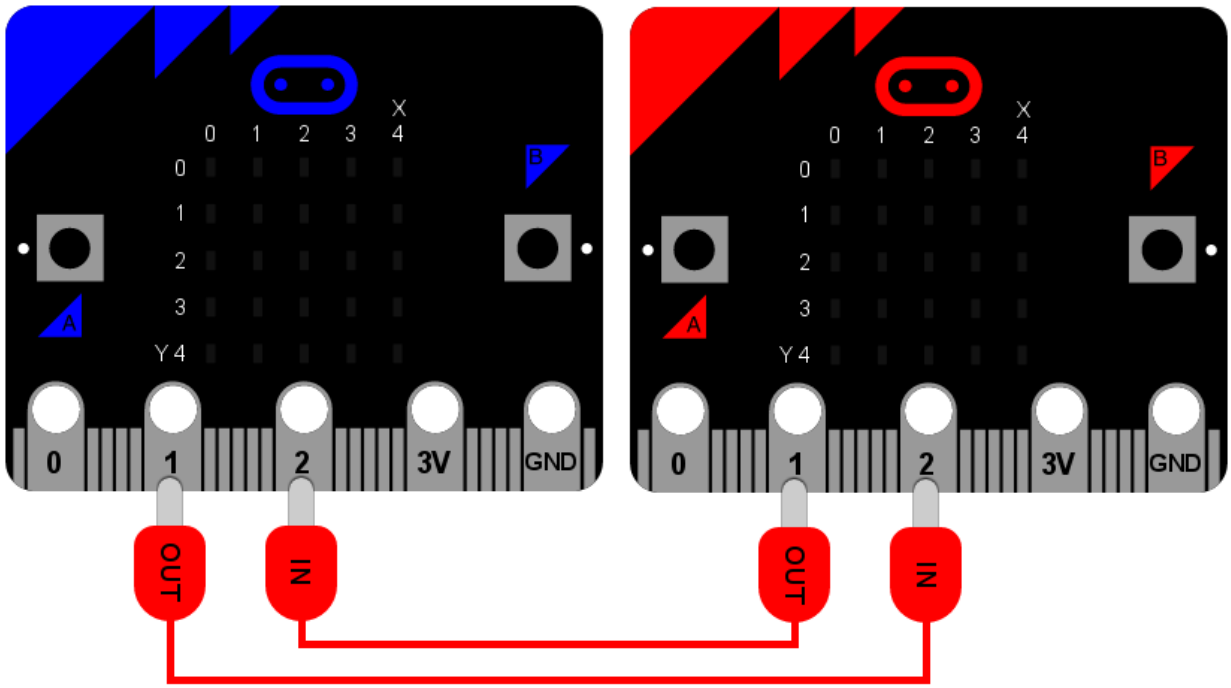
It is possible to connect devices together to send and receive messages to and from each other. This is called a network. A network of interconnected networks is called an internet. The Internet is an internet of all the internets.

Networking is hard and this is reflected in the program described below. However, the beautiful thing about this project is it contains all the common aspects of network programming you need to know about. It's also remarkably simple and fun.

But first, let's set the scene...

1.12.1 Connection

Imagine a network as a series of layers. At the very bottom is the most fundamental aspect of communication: there needs to be some sort of way for a signal to get from one device to the other. Sometimes this is done via a radio connection, but in this example we're simply going to use two wires.



It is upon this foundation that we can build all the other layers in the *network stack*.

As the diagram shows, blue and red micro:bits are connected via crocodile leads. Both use pin 1 for output and pin 2 for input. The output from one device is connected to the input on the other. It's a bit like knowing which way round to hold a telephone handset - one end has a microphone (the input) and the other a speaker (the output). The recording of your voice via your microphone is played out of the other person's speaker. If you hold the phone the wrong way up, you'll get strange results!

It's exactly the same in this instance: you must connect the wires properly!

1.12.2 Signal

The next layer in the *network stack* is the signal. Often this will depend upon the characteristics of the connection. In our example it's simply digital on and off signals sent down the wires via the IO pins.

If you remember, it's possible to use the IO pins like this:

```
pin1.write_digital(1)  # switch the signal on
pin1.write_digital(0)  # switch the signal off
input = pin2.read_digital()  # read the value of the signal (either 1 or 0)
```

The next step involves describing how to use and handle a signal. For that we need a...

1.12.3 Protocol

If you ever meet the Queen there are expectations about how you ought to behave. For example, when she arrives you may bow or curtsey, if she offers her hand politely shake it, refer to her as „your majesty" and thereafter as „ma'am" and so on. This set of rules is called the royal protocol. A protocol explains how to behave given a specific situation (such as meeting the Queen). A protocol is pre-defined to ensure everyone understands what's going on before a given situation arises.

It is for this reason that we define and use protocols for communicating messages via a computer network. Computers need to agree before hand how to send and receive messages. Perhaps the best known protocol is the hypertext transfer protocol (HTTP) used by the world wide web.

Another famous protocol for sending messages (that pre-dates computers) is Morse code. It defines how to send character-based messages via on/off signals of long or short durations. Often such signals are played as bleeps. Long durations are called dashes (−) whereas short durations are dots (.). By combining dashes and dots Morse defines a way to send characters. For example, here's how the standard Morse alphabet is defined:

```
.-    A     ---   J     ...   S     .----  1     ----.  9
-...  B     -.-   K     -     T     ..---  2     -----  0
-.-.  C     .-..  L     ..-   U     ...--  3
-..   D     --    M     ...-  V     ....-  4
.     E     -.    N     .--   W     .....  5
..-.  F     ---   O     -..-  X     -....  6
--.   G     .--.  P     -.--  Y     --...  7
....  H     --.-  Q     --..  Z     ---..  8
..    I     .-.   R
```

Given the chart above, to send the character „H" the signal is switched on four times for a short duration, indicating four dots (`....`). For the letter „L" the signal is also switched on four times, but the second signal has a longer duration (`.-..`).

Obviously, the timing of the signal is important: we need to tell a dot from a dash. That's another point of a protocol, to agree such things so everyone's implementation of the protocol will work with everyone elses. In this instance we'll just say that:

- A signal with a duration less than 250 milliseconds is a dot.
- A signal with a duration from 250 milliseconds to less than 500 milliseconds is a dash.
- Any other duration of signal is ignored.
- A pause / gap in the signal of greater than 500 milliseconds indicates the end of a character.

In this way, the sending of a letter „H" is defined as four „on" signals that last no longer than 250 milliseconds each, followed by a pause of greater than 500 milliseconds (indicating the end of the character).

1.12.4 Message

We're finally at a stage where we can build a message - a message that actually means something to us humans. This is the top-most layer of our *network stack*.

Using the protocol defined above I can send the following sequence of signals down the physical wire to the other micro:bit:

```
...././.-../.-../---/.--/---/.-./.-../-..
```

Can you work out what it says?

1.12.5 Application

It's all very well having a network stack, but you also need a way to interact with it - some form of application to send and receive messages. While HTTP is interesting *most* people don't know about it and let their web-browser handle it - the underlying *network stack* of the world wide web is hidden (as it should be).

So, what sort of application should we write for the BBC micro:bit? How should it work, from the user's point of view?

Obviously, to send a message you should be able to input dots and dashes (we can use button A for that). If we want to see the message we sent or just received we should be able to trigger it to scroll across the display (we can use button B for that). Finally, this being Morse code, if a speaker is attached, we should be able to play the beeps as a form of aural feedback while the user is entering their message.

1.12.6 The End Result

Here's the program, in all its glory and annotated with plenty of comments so you can see what's going on:

```
from microbit import *
import music


# A lookup table of morse codes and associated characters.
MORSE_CODE_LOOKUP = {
    ".-": "A",
    "-...": "B",
    "-.-.": "C",
    "-..": "D",
    ".": "E",
    "..-.": "F",
    "--.": "G",
    "....": "H",
    "..": "I",
    ".---": "J",
    "-.-": "K",
    ".-..": "L",
    "--": "M",
    "-.": "N",
    "---": "O",
    ".--.": "P",
    "--.-": "Q",
    ".-.": "R",
    "...": "S",
    "-": "T",
    "..-": "U",
    "...-": "V",
    ".--": "W",
    "-..-": "X",
    "-.--": "Y",
    "--..": "Z",
    ".----": "1",
    "..---": "2",
    "...--": "3",
    "....-": "4",
    ".....": "5",
    "-....": "6",
    "--...": "7",
    "---..": "8",
    "----.": "9",
    "-----": "0"
}


def decode(buffer):
    # Attempts to get the buffer of Morse code data from the lookup table. If
    # it's not there, just return a full stop.
    return MORSE_CODE_LOOKUP.get(buffer, '.')


# How to display a single dot.
DOT = Image("00000:"
            "00000:"
```

```
            "00900:"
            "00000:"
            "00000:")


# How to display a single dash.
DASH = Image("00000:"
             "00000:"
             "09990:"
             "00000:"
             "00000:")


# To create a DOT you need to hold the button for less than 250ms.
DOT_THRESHOLD = 250
# To create a DASH you need to hold the button for less than 500ms.
DASH_THRESHOLD = 500


# Holds the incoming Morse signals.
buffer = ''
# Holds the translated Morse as characters.
message = ''
# The time from which the device has been waiting for the next keypress.
started_to_wait = running_time()


# Put the device in a loop to wait for and react to key presses.
while True:
    # Work out how long the device has been waiting for a keypress.
    waiting = running_time() - started_to_wait
    # Reset the timestamp for the key_down_time.
    key_down_time = None
    # If button_a is held down, then...
    while button_a.is_pressed():
        # Play a beep - this is Morse code y'know ;-)
        music.pitch(880, 10)
        # Set pin1 (output) to "on"
        pin1.write_digital(1)
        # ...and if there's not a key_down_time then set it to now!
        if not key_down_time:
            key_down_time = running_time()
    # Alternatively, if pin2 (input) is getting a signal, pretend it's a
    # button_a key press...
    while pin2.read_digital():
        if not key_down_time:
            key_down_time = running_time()
    # Get the current time and call it key_up_time.
    key_up_time = running_time()
    # Set pin1 (output) to "off"
    pin1.write_digital(0)
    # If there's a key_down_time (created when button_a was first pressed
    # down).
    if key_down_time:
        # ... then work out for how long it was pressed.
        duration = key_up_time - key_down_time
        # If the duration is less than the max length for a "dot" press...
        if duration < DOT_THRESHOLD:
```

```
            # ... then add a dot to the buffer containing incoming Morse codes
            # and display a dot on the display.
            buffer += '.'
            display.show(DOT)
        # Else, if the duration is less than the max length for a "dash"
        # press... (but longer than that for a DOT ~ handled above)
        elif duration < DASH_THRESHOLD:
            # ... then add a dash to the buffer and display a dash.
            buffer += '-'
            display.show(DASH)
        # Otherwise, any other sort of keypress duration is ignored (this isn't
        # needed, but added for "understandability").
        else:
            pass
        # The button press has been handled, so reset the time from which the
        # device is starting to wait for a  button press.
        started_to_wait = running_time()
    # Otherwise, there hasn't been a button_a press during this cycle of the
    # loop, so check there's not been a pause to indicate an end of the
    # incoming Morse code character. The pause must be longer than a DASH
    # code's duration.
    elif len(buffer) > 0 and waiting > DASH_THRESHOLD:
        # There is a buffer and it's reached the end of a code so...
        # Decode the incoming buffer.
        character = decode(buffer)
        # Reset the buffer to empty.
        buffer = ''
        # Show the decoded character.
        display.show(character)
        # Add the character to the message.
        message += character
    # Finally, if button_b was pressed while all the above was going on...
    if button_b.was_pressed():
        # ... display the message,
        display.scroll(message)
        # then reset it to empty (ready for a new message).
        message = ''
```

How would you improve it? Can you change the definition of a dot and a dash so speedy Morse code users can use it? What happens if both devices are sending at the same time? What might you do to handle this situation?

1.13 Radio

Interaction at a distance feels like magic.

Magic might be useful if you're an elf, wizard or unicorn, but such things only exist in stories.

However, there's something much better than magic: physics!

Wireless interaction is all about physics: radio waves (a type of electromagnetic radiation, similar to visible light) have some sort of property (such as their amplitude, phase or pulse width) modulated by a transmitter in such a way that information can be encoded and, thus, broadcast. When radio waves encounter an electrical conductor (i.e. an aerial), they cause an alternating current from which the information in the waves can be extracted and transformed back into its original form.

1.13.1 Layers upon Layers

If you remember, networks are built in layers.

The most fundamental requirement for a network is some sort of connection that allows a signal to get from one device to the other. In our networking tutorial we used wires connected to the I/O pins. Thanks to the radio module we can do away with wires and use the physics summarised above as the invisible connection between devices.

The next layer up in the network stack is also different from the example in the networking tutorial. With the wired example we used digital on and off to send and read a signal from the pins. With the built-in radio on the micro:bit the smallest useful part of the signal is a byte.

1.13.2 Bytes

A byte is a unit of information that (usually) consists of eight bits. A bit is the smallest possible unit of information since it can only be in two states: on or off.

Bytes work like a sort of abacus: each position in the byte is like a column in an abacus - they represent an associated number. In an abacus these are usually thousands, hundreds, tens and units (in UK parlance). In a byte they are 128, 64, 32, 16, 8, 4, 2 and 1. As bits (on/off signals) are sent over the air, they are re-combined into bytes by the recipient.

Have you spotted the pattern? (Hint: base 2.)

By adding the numbers associated with the positions in a byte that are set to „on" we can represent numbers between 0 and 255. The image below shows how this works with five bits and counting from zero to 32:

If we can agree what each one of the 255 numbers (encoded by a byte) represents ~ such as a character ~ then we can start to send text one character per byte at a time.

Funnily enough, people have already thought of this ~ using bytes to encode and decode information is commonplace. This approximately corresponds to the Morse-code „protocol" layer in the wired networking example.

A really great series of child (and teacher) friendly explanations of „all things bytes" can be found at the CS unplugged website.

1.13.3 Addressing

The problem with radio is that you can't transmit directly to one person. Anyone with an appropriate aerial can receive the messages you transmit. As a result it's important to be able to differentiate who should be receiving broadcasts.

The way the radio built into the micro:bit solves this problem is quite simple:

- It's possible to tune the radio to different channels (numbered 0-100). This works in exactly the same way as kids» walkie-talkie radios: everyone tunes into the same channel and everyone hears what everyone else broadcasts via that channel. As with walkie-talkies, if you use adjacent channels there is a slight possibility of interference.
- The radio module allows you to specify two pieces of information: an address and a group. The address is like a postal address whereas a group is like a specific recipient at the address. The important thing is the radio will filter out messages that it receives that do not match *your* address and group. As a result, it's important to pre-arrange the address and group your application is going to use.

Of course, the micro:bit is still receiving broadcast messages for other address/group combinations. The important thing is you don't need to worry about filtering those out. Nevertheless, if someone were clever enough, they could just read *all the wireless network traffic* no matter what the target address/group was supposed to be. In this case, it's *essential* to use encrypted means of communication so only the desired recipient can actually read the message that was broadcast. Cryptography is a fascinating subject but, unfortunately, beyond the scope of this tutorial.

1.13.4 Fireflies

This is a firefly:

It's a sort of bug that uses bioluminescence to signal (without wires) to its friends. Here's what they look like when they signal to each other:

The BBC have rather a beautiful video of fireflies available online.

We're going to use the radio module to create something akin to a swarm of fireflies signalling to each other.

First `import radio` to make the functions available to your Python program. Then call the `radio.on()` function to turn the radio on. Since the radio draws power and takes up memory we've made it so *you* decide when it is enabled (there is, of course a `radio.off()` function).

At this point the radio module is configured to sensible defaults that make it compatible with other platforms that may target the BBC micro:bit. It is possible to control many of the features discussed above (such as channel and addressing) as well as the amount of power used to broadcast messages and the amount of RAM the incoming message queue will take up. The API documentation contains all the information you need to configure the radio to your needs.

Assuming we're happy with the defaults, the simplest way to send a message is like this:

```
radio.send("a message")
```

The example uses the `send` function to simply broadcast the string „a message". To receive a message is even easier:

```
new_message = radio.receive()
```

As messages are received they are put on a message queue. The `receive` function returns the oldest message from the queue as a string, making space for a new incoming message. If the message queue fills up, then new incoming messages are ignored.

That's really all there is to it! (Although the radio module is also powerful enough that you can send any arbitrary type of data, not just strings. See the API documentation for how this works.)

Armed with this knowledge, it's simple to make micro:bit fireflies like this:

```
# A micro:bit Firefly.
# By Nicholas H.Tollervey. Released to the public domain.
import radio
import random
from microbit import display, Image, button_a, sleep

# Create the "flash" animation frames. Can you work out how it's done?
flash = [Image().invert()*(i/9) for i in range(9, -1, -1)]

# The radio won't work unless it's switched on.
radio.on()

# Event loop.
while True:
    # Button A sends a "flash" message.
    if button_a.was_pressed():
        radio.send('flash')  # a-ha
    # Read any incoming messages.
    incoming = radio.receive()
    if incoming == 'flash':
```

```
        # If there's an incoming "flash" message display
        # the firefly flash animation after a random short
        # pause.
        sleep(random.randint(50, 350))
        display.show(flash, delay=100, wait=False)
        # Randomly re-broadcast the flash message after a
        # slight delay.
        if random.randint(0, 9) == 0:
            sleep(500)
            radio.send('flash')  # a-ha
```

The import stuff happens in the event loop. First, it checks if button A was pressed and, if it was, uses the radio to send the message „flash". Then it reads any messages from the message queue with `radio.receive()`. If there is a message it sleeps a short, random period of time (to make the display more interesting) and uses `display.show()` to animate a firefly flash. Finally, to make things a bit exciting, it chooses a random number so that it has a 1 in 10 chance of re-broadcasting the „flash" message to anyone else (this is how it's possible to sustain the firefly display among several devices). If it decides to re-broadcast then it waits for half a second (so the display from the initial flash message has chance to die down) before sending the „flash" signal again. Because this code is enclosed within a `while True` block, it loops back to the beginning of the event loop and repeats this process forever.

The end result (using a group of micro:bits) should look something like this:

1.14 Next Steps

These tutorials are only the first steps in using MicroPython with the BBC micro:bit. A musical analogy: you've got a basic understanding of a very simple instrument and confidently play „Three Blind Mice".

This is an achievement to build upon.

Ahead of you is an exciting journey to becoming a virtuoso coder.

You will encounter frustration, failure and foolishness. When you do please remember that you're not alone. Python has a secret weapon: the most amazing community of programmers on the planet. Connect with this community and you will make friends, find mentors, support each other and share resources.

The examples in the tutorials are simple to explain but may not be the simplest or most efficient implementations. We've left out lots of *really fun stuff* so we could concentrate on arming you with the basics. If you *really* want to know how to make MicroPython fly on the BBC micro:bit then read the API reference documentation. It contains information about *all* the capabilities available to you.

Explore, experiment and be fearless trying things out ~ for these are the attributes of a virtuoso coder. To encourage you we have hidden a number of Easter eggs in MicroPython and the code editors (both TouchDevelop and Mu). They're fun rewards for looking „under the hood" and „poking with a stick".

Such skill in Python is valuable: it's one of the world's most popular professional programming languages.

Amaze us with your code! Make things that delight us! Most of all, have fun!

Happy hacking!

Python jest jednym z najpopularniejszych na świecie języków programowania. Zapewne codziennie używasz oprogramowania napisanego w Pythonie, nie zdając sobie nawet z tego sprawy. Wszelkiego rodzaju firmy i organizacje używają Pythona do różnorodnych zastosowań. Google, NASA, Banki, Disney, CERN, YouTube, Mozilla, Newspapers – lista jest długa i obejmuje wszystkie obszary handlu, nauki i sztuki.

Dla przykładu, pamiętasz ogłoszone niedawno odkrycie fal grawitacyjnych? Instrumenty pomiarowe użyte tam były sterowane za pomocą Pythona.

Po prostu, nauka Pythona da ci niesłychanie cenną umiejętność, która ma zastosowanie we wszystkich obszarach ludzkich starań.

Jednym z takich obszarów jest zadziwiające urządzenie micro:bit, stworzone przez BBC. Działa na nim wersja Pythona nazwana MicroPython, która jest specjalnie zaprojektowana dla małych komputerów jak ten. Jest to pełnoprawna implementacja Pythona 3, wiec tego samego języka, którego używać będziesz później (na przykład do programowania Raspberry Pi).

MicroPython nie zawiera wszystkich bibliotek standardowych, które zawarte są w „zwykłym" Pythonie, jednakże stworzyliśmy specjalnym moduł nazwany `microbit`, który pozwala na pełną kontrolę nad micro:bit.

Zarówno Python, jak i MicroPython, to otwarte oprogramowanie. To znaczy nie tylko to, że nie trzeba za nie płacić, ale także to, że każdy może dodać coś od siebie do ich społeczności. To może być kod, dokumentacja, raporty o błędach, lokalna grupa użytkowników, albo nowe poradniki (jak ten). W zasadzie wszystkie materiały dotyczące BBC micro:bit zostały stworzone przez międzynarodowy zespół ochotników, pracujących w swoim wolnym czasie.

Te oto lekcje wprowadzają do MicroPythona i BBC micro:bit w prostych krokach. Mogą one być swobodnie wykorzystywane i modyfikowane do lekcji szkolnych. Możesz też uczyć się z nich w domu.

Najwięcej zyskasz odkrywając, eksperymentując i bawiąc się. Nie da się zepsuć BBC micro:bit pisząc i uruchamiając na nim błędy kod, więc po prostu spróbuj!

Słowo ostrzeżenia: nie wszystko uda się za pierwszym razem i nie ma w tym nic złego. Porażki są najlepszym sposobem nauki dla dobrych programistów. Szukanie błędów i unikanie powtarzania błędów jest źródłem dużej satysfakcji dla tych z nas, którzy są programistami.

W razie wątpliwości, pamiętaj Zen MicroPythona:

```
Koduj,
Skleć to do kupy,
Mniej to więcej,
Upraszczaj,
Małe jest piękne,

Bądz dzielny! Psuj rzeczy! Ucz się i baw się dobrze!
Wyraź siebie przez MicroPythona.

Miłego hakowania! :-)
```

Powodzenia!

ROZDZIAŁ 2

micro:bit Micropython API

Ostrzeżenie: As we work towards a 1.0 release, this API is subject to frequent changes. This page reflects the current micro:bit API in a developer-friendly (but not necessarily kid-friendly) way. The tutorials associated with this documentation are a good place to start for non-developers looking for information.

2.1 The microbit module

Everything directly related to interacting with the hardware lives in the *microbit* module. For ease of use it's recommended you start all scripts with:

```
from microbit import *
```

The following documentation assumes you have done this.

There are a few functions available directly:

```
# sleep for the given number of milliseconds.
sleep(ms)
# returns the number of milliseconds since the micro:bit was last switched on.
running_time()
# makes the micro:bit enter panic mode (this usually happens when the DAL runs
# out of memory, and causes a sad face to be drawn on the display). The error
# code can be any arbitrary integer value.
panic(error_code)
# resets the micro:bit.
reset()
```

The rest of the functionality is provided by objects and classes in the microbit module, as described below.

Note that the API exposes integers only (ie no floats are needed, but they may be accepted). We thus use milliseconds for the standard time unit.

2.1.1 Buttons

There are 2 buttons:

```
button_a
button_b
```

These are both objects and have the following methods:

```
# returns True or False to indicate if the button is pressed at the time of
# the method call.
button.is_pressed()
# returns True or False to indicate if the button was pressed since the device
# started or the last time this method was called.
button.was_pressed()
# returns the running total of button presses, and resets this counter to zero
button.get_presses()
```

2.1.2 The LED display

The LED display is exposed via the *display* object:

```
# gets the brightness of the pixel (x,y). Brightness can be from 0 (the pixel
# is off) to 9 (the pixel is at maximum brightness).
display.get_pixel(x, y)
# sets the brightness of the pixel (x,y) to val (between 0 [off] and 9 [max
# brightness], inclusive).
display.set_pixel(x, y, val)
# clears the display.
display.clear()
# shows the image.
display.show(image, delay=0, wait=True, loop=False, clear=False)
# shows each image or letter in the iterable, with delay ms. in between each.
display.show(iterable, delay=400, wait=True, loop=False, clear=False)
# scrolls a string across the display (more exciting than display.show for
# written messages).
display.scroll(string, delay=400)
```

2.1.3 Pins

Provide digital and analog input and output functionality, for the pins in the connector. Some pins are connected internally to the I/O that drives the LED matrix and the buttons.

Each pin is provided as an object directly in the `microbit` module. This keeps the API relatively flat, making it very easy to use:

- pin0
- pin1
- ...
- pin15
- pin16
- *Warning: P17-P18 (inclusive) are unavailable.*

- pin19
- pin20

Each of these pins are instances of the `MicroBitPin` class, which offers the following API:

```
# value can be 0, 1, False, True
pin.write_digital(value)
# returns either 1 or 0
pin.read_digital()
# value is between 0 and 1023
pin.write_analog(value)
# returns an integer between 0 and 1023
pin.read_analog()
# sets the period of the PWM output of the pin in milliseconds
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period(int)
# sets the period of the PWM output of the pin in microseconds
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period_microseconds(int)
# returns boolean
pin.is_touched()
```

2.1.4 Images

Informacja: You don't always need to create one of these yourself - you can access the image shown on the display directly with *display.image*. *display.image* is just an instance of *Image*, so you can use all of the same methods.

Images API:

```
# creates an empty 5x5 image
image = Image()
# create an image from a string - each character in the string represents an
# LED - 0 (or space) is off and 9 is maximum brightness. The colon ":"
# indicates the end of a line.
image = Image('90009:09090:00900:09090:90009:')
# create an empty image of given size
image = Image(width, height)
# initialises an Image with the specified width and height. The buffer
# should be an array of length width * height
image = Image(width, height, buffer)

# methods
# returns the image's width (most often 5)
image.width()
# returns the image's height (most often 5)
image.height()
# sets the pixel at the specified position (between 0 and 9). May fail for
# constant images.
image.set_pixel(x, y, value)
# gets the pixel at the specified position (between 0 and 9)
image.get_pixel(x, y)
# returns a new image created by shifting the picture left 'n' times.
image.shift_left(n)
# returns a new image created by shifting the picture right 'n' times.
image.shift_right(n)
```

```
# returns a new image created by shifting the picture up 'n' times.
image.shift_up(n)
# returns a new image created by shifting the picture down 'n' times.
image.shift_down(n)
# get a compact string representation of the image
repr(image)
# get a more readable string representation of the image
str(image)

#operators
# returns a new image created by superimposing the two images
image + image
# returns a new image created by multiplying the brightness of each pixel by n
image * n

# built-in images.
Image.HEART
Image.HEART_SMALL
Image.HAPPY
Image.SMILE
Image.SAD
Image.CONFUSED
Image.ANGRY
Image.ASLEEP
Image.SURPRISED
Image.SILLY
Image.FABULOUS
Image.MEH
Image.YES
Image.NO
Image.CLOCK12 # clock at 12 o' clock
Image.CLOCK11
... # many clocks (Image.CLOCKn)
Image.CLOCK1 # clock at 1 o'clock
Image.ARROW_N
... # arrows pointing N, NE, E, SE, S, SW, W, NW (microbit.Image.ARROW_direction)
Image.ARROW_NW
Image.TRIANGLE
Image.TRIANGLE_LEFT
Image.CHESSBOARD
Image.DIAMOND
Image.DIAMOND_SMALL
Image.SQUARE
Image.SQUARE_SMALL
Image.RABBIT
Image.COW
Image.MUSIC_CROTCHET
Image.MUSIC_QUAVER
Image.MUSIC_QUAVERS
Image.PITCHFORK
Image.XMAS
Image.PACMAN
Image.TARGET
Image.TSHIRT
Image.ROLLERSKATE
Image.DUCK
Image.HOUSE
Image.TORTOISE
```

```
Image.BUTTERFLY
Image.STICKFIGURE
Image.GHOST
Image.SWORD
Image.GIRAFFE
Image.SKULL
Image.UMBRELLA
Image.SNAKE
# built-in lists - useful for animations, e.g. display.show(Image.ALL_CLOCKS)
Image.ALL_CLOCKS
Image.ALL_ARROWS
```

2.1.5 The accelerometer

The accelerometer is accessed via the `accelerometer` object:

```
# read the X axis of the device. Measured in milli-g.
accelerometer.get_x()
# read the Y axis of the device. Measured in milli-g.
accelerometer.get_y()
# read the Z axis of the device. Measured in milli-g.
accelerometer.get_z()
# get tuple of all three X, Y and Z readings (listed in that order).
accelerometer.get_values()
# return the name of the current gesture.
accelerometer.current_gesture()
# return True or False to indicate if the named gesture is currently active.
accelerometer.is_gesture(name)
# return True or False to indicate if the named gesture was active since the
# last call.
accelerometer.was_gesture(name)
# return a tuple of the gesture history. The most recent is listed last.
accelerometer.get_gestures()
```

The recognised gestures are: `up`, `down`, `left`, `right`, `face up`, `face down`, `freefall`, `3g`, `6g`, `8g`, `shake`.

2.1.6 The compass

The compass is accessed via the *compass* object:

```
# calibrate the compass (this is needed to get accurate readings).
compass.calibrate()
# return a numeric indication of degrees offset from "north".
compass.heading()
# return an numeric indication of the strength of magnetic field around
# the micro:bit.
compass.get_field_strength()
# returns True or False to indicate if the compass is calibrated.
compass.is_calibrated()
# resets the compass to a pre-calibration state.
compass.clear_calibration()
```

2.1.7 I2C bus

There is an I2C bus on the micro:bit that is exposed via the *i2c* object. It has the following methods:

```
# read n bytes from device with addr; repeat=True means a stop bit won't
# be sent.
i2c.read(addr, n, repeat=False)
# write buf to device with addr; repeat=True means a stop bit won't be sent.
i2c.write(addr, buf, repeat=False)
```

2.1.8 UART

Use `uart` to communicate with a serial device connected to the device's I/O pins:

```
# set up communication (use pins 0 [TX] and 1 [RX]) with a baud rate of 9600.
uart.init()
# return True or False to indicate if there are incoming characters waiting to
# be read.
uart.any()
# return (read) n incoming characters.
uart.read(n)
# return (read) as much incoming data as possible.
uart.readall()
# return (read) all the characters to a newline character is reached.
uart.readline()
# read bytes into the referenced buffer.
uart.readinto(buffer)
# write bytes from the buffer to the connected device.
uart.write(buffer)
```

ROZDZIAŁ 3

Microbit Module

The `microbit` module gives you access to all the hardware that is built-in into your board.

3.1 Functions

`microbit.`**`panic`**`(`*n*`)`
Enter a panic mode. Requires restart. Pass in an arbitrary integer <= 255 to indicate a status:

```
microbit.panic(255)
```

`microbit.`**`reset`**`()`
Restart the board.

`microbit.`**`sleep`**`(`*n*`)`
Wait for `n` milliseconds. One second is 1000 milliseconds, so:

```
microbit.sleep(1000)
```

will pause the execution for one second. `n` can be an integer or a floating point number.

`microbit.`**`running_time`**`()`
Return the number of milliseconds since the board was switched on or restarted.

`microbit.`**`temperature`**`()`
Return the temperature of the micro:bit in degrees Celcius.

3.2 Attributes

3.2.1 Buttons

There are two buttons on the board, called `button_a` and `button_b`.

Attributes

button_a
A `Button` instance (see below) representing the left button.

button_b
Represents the right button.

Classes

class Button
Represents a button.

Informacja: This class is not actually available to the user, it is only used by the two button instances, which are provided already initialized.

---

is_pressed()
Returns `True` if the specified button `button` is pressed, and `False` otherwise.

was_pressed()
Returns `True` or `False` to indicate if the button was pressed since the device started or the last time this method was called.

get_presses()
Returns the running total of button presses, and resets this total to zero before returning.

Example

```
import microbit

while True:
    if microbit.button_a.is_pressed() and microbit.button_b.is_pressed():
        microbit.display.scroll("AB")
        break
    elif microbit.button_a.is_pressed():
        microbit.display.scroll("A")
    elif microbit.button_b.is_pressed():
        microbit.display.scroll("B")
    microbit.sleep(100)
```

3.2.2 Input/Output Pins

The pins are your board's way to communicate with external devices connected to it. There are 19 pins for your disposal, numbered 0-16 and 19-20. Pins 17 and 18 are not available.

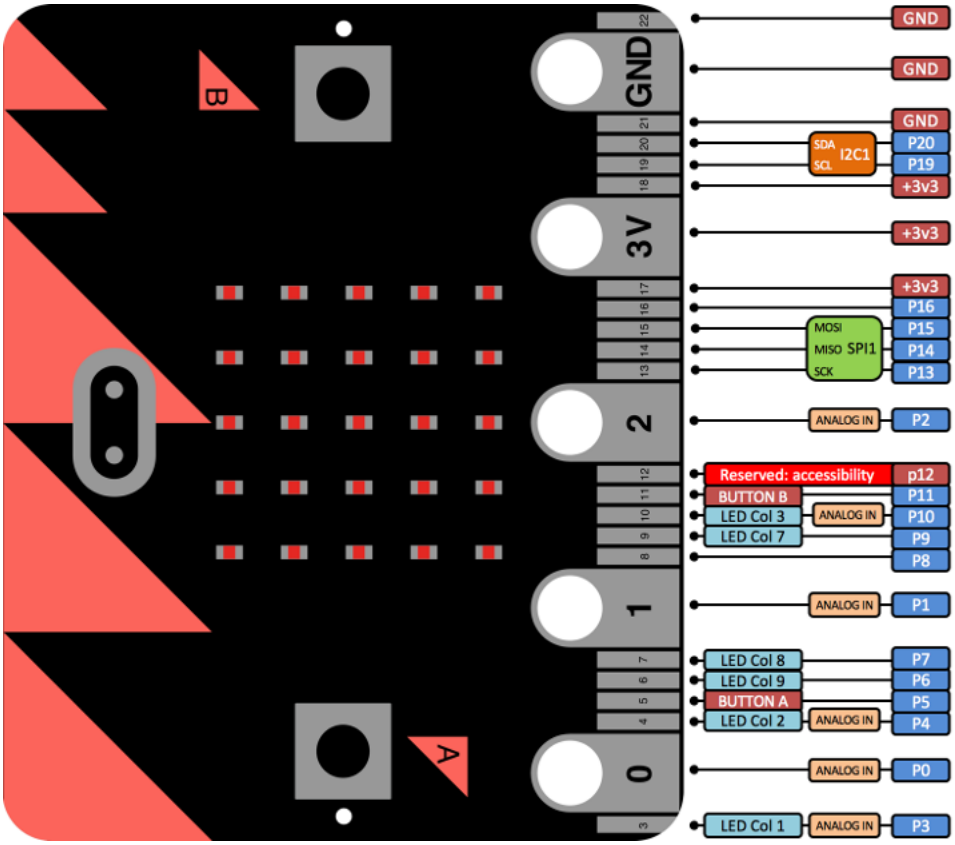
For example, the script below will change the display on the micro:bit depending upon the digital reading on pin 0:

```
from microbit import *


while True:
    if pin0.read_digital():
        display.show(Image.HAPPY)
```

```
    else:
        display.show(Image.SAD)
```

Pin Functions



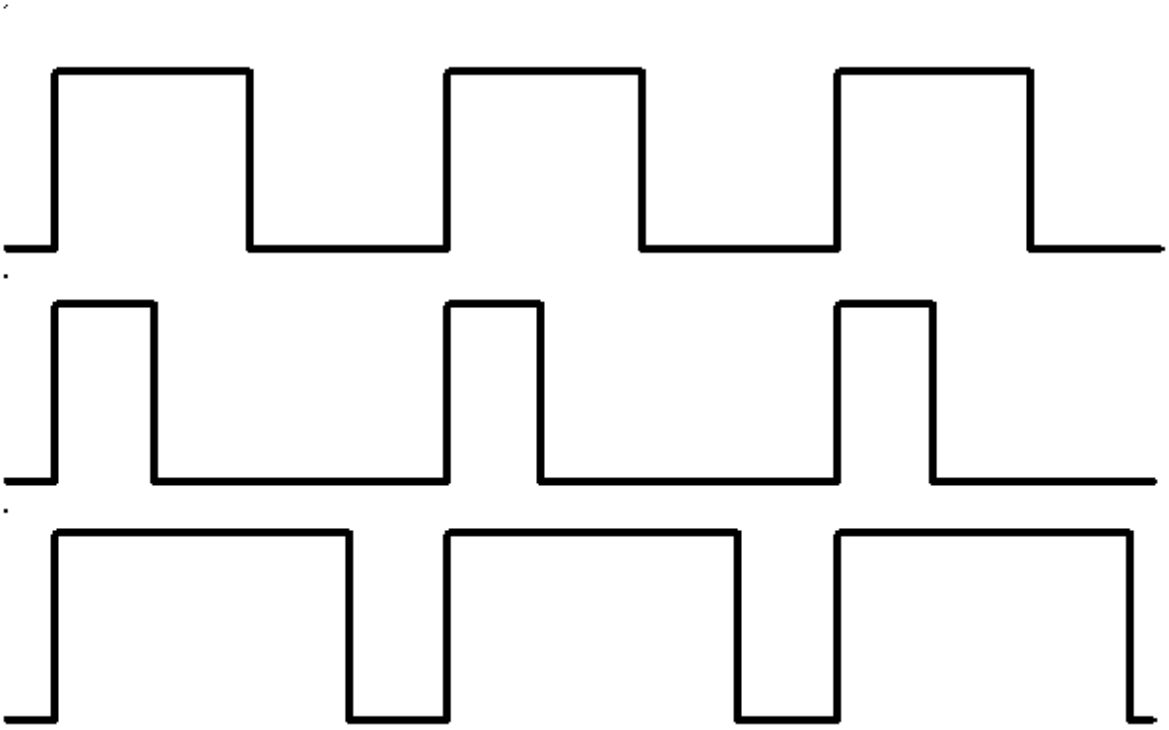
Those pins are available as attributes on the `microbit` module:`microbit.pin0` - `microbit.pin20`.

Pin	Type	Function
0	Touch	Pad 0
1	Touch	Pad 1
2	Touch	Pad 2
3	Analog	Column 1
4	Analog	Column 2
5	Digital	Button A
6	Digital	Row 2
7	Digital	Row 1
8	Digital	
9	Digital	Row 3
10	Analog	Column 3
11	Digital	Button B
12	Digital	
13	Digital	SPI MOSI
14	Digital	SPI MISO
15	Digital	SPI SCK
16	Digital	
19	Digital	I2C SCL
20	Digital	I2C SDA

The above table summarizes the pins available, their types (see below) and what they are internally connected to.

Pulse-Width Modulation

The pins of your board cannot output analog signal the way an audio amplifier can do it – by modulating the voltage on the pin. Those pins can only either enable the full 3.3V output, or pull it down to 0V. However, it is still possible to control the brightness of LEDs or speed of an electric motor, by switching that voltage on and off very fast, and controlling how long it is on and how long it is off. This technique is called Pulse-Width Modulation (PWM), and that's what the `write_analog` method below does.



Above you can see the diagrams of three different PWM signals. All of them have the same period (and thus frequency), but they have different duty cycles.

The first one would be generated by `write_analog(511)`, as it has exactly 50% duty – the power is on half of the time, and off half of the time. The result of that is that the total energy of this signal is the same, as if it was 1.65V instead of 3.3V.

The second signal has 25% duty cycle, and could be generated with `write_analog(255)`. It has similar effect as if 0.825V was being output on that pin.

The third signal has 75% duty cycle, and can be generated with `write_analog(767)`. It has three times as much energy, as the second signal, and is equivalent to outputting 2.475V on th pin.

Note that this works well with devices such as motors, which have huge inertia by themselves, or LEDs, which blink too fast for the human eye to see the difference, but will not work so good with generating sound waves. This board can only generate square wave sounds on itself, which sound pretty much like the very old computer games – mostly because those games also only could do that.

Classes

There are three kinds of pins, differing in what is available for them. They are represented by the classes listed below. Note that they form a hierarchy, so that each class has all the functionality of the previous class, and adds its own to that.

Informacja: Those classes are not actually available for the user, you can't create new instances of them. You can only use the instances already provided, representing the physical pins on your board.

---

class microbit.**MicroBitDigitalPin**

read_digital()
Return 1 if the pin is high, and 0 if it's low.

write_digital(*value*)
Set the pin to high if `value` is 1, or to low, if it is 0.

class microbit.**MicroBitAnalogDigitalPin**

read_analog()
Read the voltage applied to the pin, and return it as an integer between 0 (meaning 0V) and 1023 (meaning 3.3V).

write_analog(*value*)
Output a PWM signal on the pin, with the duty cycle proportional to the provided `value`. The `value` may be either an integer or a floating point number between 0 (0% duty cycle) and 1023 (100% duty).

set_analog_period(*period*)
Set the period of the PWM signal being output to `period` in milliseconds. The minimum valid value is 1ms.

set_analog_period_microseconds(*period*)
Set the period of the PWM signal being output to `period` in microseconds. The minimum valid value is 256µs.

class microbit.**MicroBitTouchPin**

is_touched()
Return `True` if the pin is being touched with a finger, otherwise return `False`.

This test is done by measuring the capacitance of the pin together with whatever is connected to it. Human body has quite a large capacitance, so touching the pin gives a dramatic change in reading, which can be detected.

The pull mode for a pin is automatically configured when the pin changes to an input mode. Input modes are when you call `read_analog` / `read_digital` / `is_touched`. The pull mode for these is, respectively, `NO_PULL`, `PULL_DOWN`, `PULL_UP`. Only when in `read_digital` mode can you call `set_pull` to change the pull mode from the default.

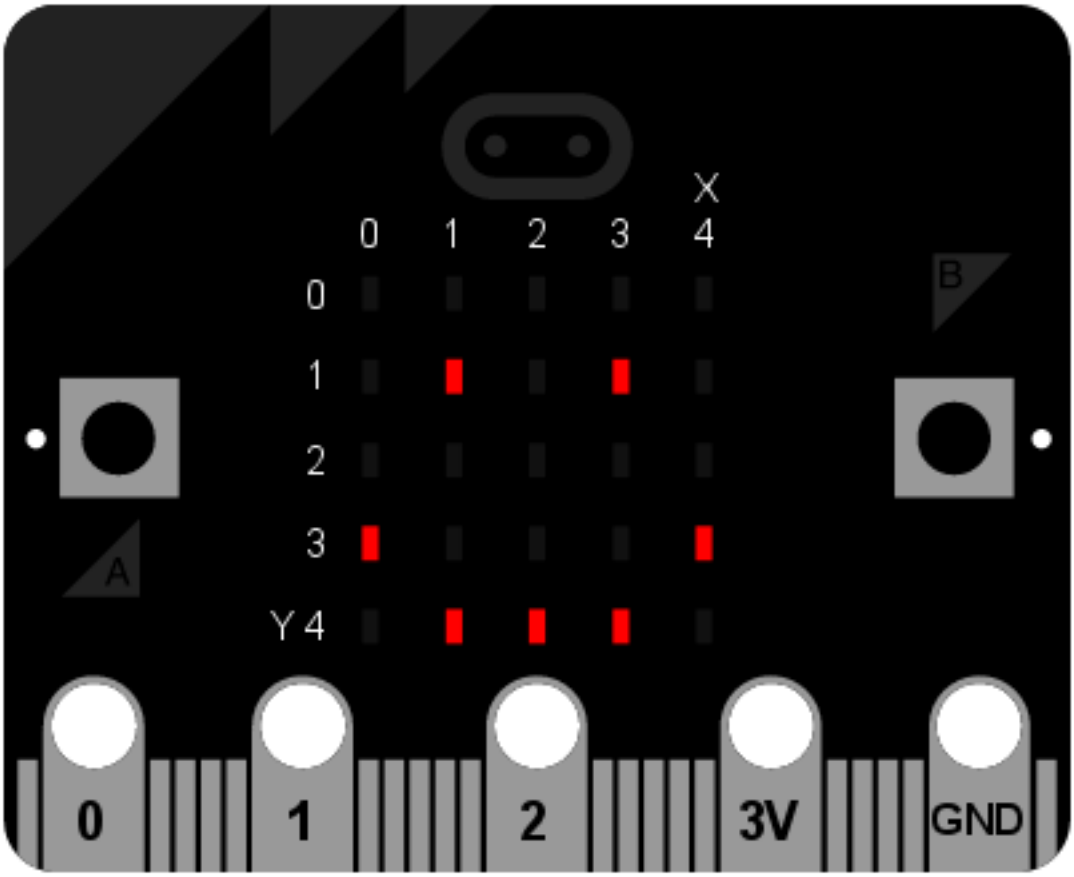
Informacja: Also note, the micro:bit has external weak (10M) pull-ups fitted on pins 0, 1 and 2 only, in order for the touch sensing to work. See the edge connector data sheet here: http://tech.microbit.org/hardware/edgeconnector_ds/

3.3 Classes

3.3.1 Image

The `Image` class is used to create images that can be displayed easily on the device's LED matrix. Given an image object it's possible to display it via the `display` API:

```
display.show(Image.HAPPY)
```



Classes

class microbit.**Image**(*string*)
class microbit.**Image**(*width=None*, *height=None*, *buffer=None*)

If `string` is used, it has to consist of digits 0-9 arranged into lines, describing the image, for example:

```
image = Image("90009:"
              "09090:"
              "00900:"
              "09090:"
              "90009")
```

will create a 5×5 image of an X. The end of a line is indicated by a colon. It's also possible to use a newline (n) to indicate the end of a line like this:

```
image = Image("90009\n"
              "09090\n"
              "00900\n"
              "09090\n"
              "90009")
```

The other form creates an empty image with `width` columns and `height` rows. Optionally `buffer` can be an array of `width``×``height` integers in range 0-9 to initialize the image.

width()
Return the number of columns in the image.

height()
Return the numbers of rows in the image.

set_pixel(*x*, *y*, *value*)
Set the brightness of the pixel at column `x` and row `y` to the `value`, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

get_pixel(*x*, *y*)
Return the brightness of pixel at column `x` and row `y` as an integer between 0 and 9.

shift_left(*n*)
Return a new image created by shifting the picture left by `n` columns.

shift_right(*n*)
Same as `image.shift_left(-n)`.

shift_up(*n*)
Return a new image created by shifting the picture up by `n` rows.

shift_down(*n*)
Same as `image.shift_up(-n)`.

crop(*x*, *y*, *w*, *h*)
Return a new image by cropping the picture to a width of `w` and a height of `h`, starting with the pixel at column `x` and row `y`.

copy()
Return an exact copy of the image.

invert()
Return a new image by inverting the brightness of the pixels in the source image.

fill(*value*)
Set the brightness of all the pixels in the image to the `value`, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

blit(*src*, *x*, *y*, *w*, *h*, *xdest=0*, *ydest=0*)
Copy the rectangle defined by `x`, `y`, `w`, `h` from the image `src` into this image at `xdest`, `ydest`. Areas in the source rectangle, but outside the source image are treated as having a value of 0.

`shift_left()`, `shift_right()`, `shift_up()`, `shift_down()` and `crop()` can are all implemented by using `blit()`. For example, img.crop(x, y, w, h) can be implemented as:

```
def crop(self, x, y, w, h):
    res = Image(w, h)
    res.blit(self, x, y, w, h)
    return res
```

Attributes

The `Image` class also has the following built-in instances of itself included as its attributes (the attribute names indicate what the image represents):

- Image.HEART
- Image.HEART_SMALL
- Image.HAPPY
- Image.SMILE
- Image.SAD
- Image.CONFUSED
- Image.ANGRY
- Image.ASLEEP
- Image.SURPRISED
- Image.SILLY
- Image.FABULOUS
- Image.MEH
- Image.YES
- Image.NO
- Image.CLOCK12, Image.CLOCK11, Image.CLOCK10, Image.CLOCK9, Image.CLOCK8, Image.CLOCK7, Image.CLOCK6, Image.CLOCK5, Image.CLOCK4, Image.CLOCK3, Image.CLOCK2, Image.CLOCK1
- Image.ARROW_N, Image.ARROW_NE, Image.ARROW_E, Image.ARROW_SE, Image.ARROW_S, Image.ARROW_SW, Image.ARROW_W, Image.ARROW_NW
- Image.TRIANGLE
- Image.TRIANGLE_LEFT
- Image.CHESSBOARD
- Image.DIAMOND
- Image.DIAMOND_SMALL
- Image.SQUARE
- Image.SQUARE_SMALL
- Image.RABBIT
- Image.COW
- Image.MUSIC_CROTCHET
- Image.MUSIC_QUAVER
- Image.MUSIC_QUAVERS
- Image.PITCHFORK
- Image.XMAS
- Image.PACMAN
- Image.TARGET
- Image.TSHIRT
- Image.ROLLERSKATE

- `Image.DUCK`
- `Image.HOUSE`
- `Image.TORTOISE`
- `Image.BUTTERFLY`
- `Image.STICKFIGURE`
- `Image.GHOST`
- `Image.SWORD`
- `Image.GIRAFFE`
- `Image.SKULL`
- `Image.UMBRELLA`
- `Image.SNAKE`

Finally, related collections of images have been grouped together:

```
* ``Image.ALL_CLOCKS``
* ``Image.ALL_ARROWS``
```

Operations

```
repr(image)
```

Get a compact string representation of the image.

```
str(image)
```

Get a readable string representation of the image.

```
image1 + image2
```

Create a new image by adding the brightness values from the two images for each pixel.

```
image * n
```

Create a new image by multiplying the brightness of each pixel by `n`.

3.4 Modules

3.4.1 Display

This module controls the 5×5 LED display on the front of your board. It can be used to display images, animations and even text.

Functions

`microbit.display.`**`get_pixel`**(*x*, *y*)
Return the brightness of the LED at column `x` and row `y` as an integer between 0 (off) and 9 (bright).

`microbit.display.`**`set_pixel`**(*x*, *y*, *value*)
Set the brightness of the LED at column `x` and row `y` to `value`, which has to be an integer between 0 and 9.

`microbit.display.`**`clear`**()
Set the brightness of all LEDs to 0 (off).

`microbit.display.`**`show`**(*image*)
Display the `image`.

`microbit.display.`**`show`**(*iterable*, *delay=400*, ***, *wait=True*, *loop=False*, *clear=False*)
Display images or letters from the `iterable` in sequence, with `delay` milliseconds between them.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `clear` is `True`, the display will be cleared after the iterable has finished.

Note that the `wait`, `loop` and `clear` arguments must be specified using their keyword.

Informacja: If using a generator as the `iterable`, then take care not to allocate any memory in the generator as allocating memory in an interrupt is prohibited and will raise a `MemoryError`.

`microbit.display.`**`scroll`**(*string*, *delay=150*, ***, *wait=True*, *loop=False*, *monospace=False*)
Similar to `show`, but scrolls the `string` horizontally instead. The `delay` parameter controls how fast the text is scrolling.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `monospace` is `True`, the characters will all take up 5 pixel-columns in width, otherwise there will be exactly 1 blank pixel-column between each character as they scroll.

Note that the `wait`, `loop` and `monospace` arguments must be specified using their keyword.

`microbit.display.`**`on`**()
Use on() to turn on the display.

`microbit.display.`**`off`**()
Use off() to turn off the display (thus allowing you to re-use the GPIO pins associated with the display for other purposes).

`microbit.display.`**`is_on`**()
Returns `True` if the display is on, otherwise returns `False`.

Example

To continuously scroll a string across the display, and do it in the background, you can use:

```
import microbit

microbit.display.scroll('Hello!', wait=False, loop=True)
```

3.4.2 UART

The `uart` module lets you talk to a device connected to your board using a serial interface.

Functions

`microbit.uart.`**`init`**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, ***, *tx=None*, *rx=None*)

Initialize serial communication with the specified parameters on the specified `tx` and `rx` pins. Note that for correct communication, the parameters have to be the same on both communicating devices.

> **Ostrzeżenie:** Initializing the UART on external pins will cause the Python console on USB to become unaccessible, as it uses the same hardware. To bring the console back you must reinitialize the UART without passing anything for ‘‘tx»» or ‘‘rx»» (or passing ‘‘None»» to these arguments). This means that calling ‘‘uart.init(115200)»» is enough to restore the Python console.

The `baudrate` defines the speed of communication. Common baud rates include:

- 9600
- 14400
- 19200
- 28800
- 38400
- 57600
- 115200

The `bits` defines the size of bytes being transmitted, and the board only supports 8. The `parity` parameter defines how parity is checked, and it can be `None`, `microbit.uart.ODD` or `microbit.uart.EVEN`. The `stop` parameter tells the number of stop bits, and has to be 1 for this board.

If `tx` and `rx` are not specified then the internal USB-UART TX/RX pins are used which connect to the USB serial convertor on the micro:bit, thus connecting the UART to your PC. You can specify any other pins you want by passing the desired pin objects to the `tx` and `rx` parameters.

Informacja: When connecting the device, make sure you „cross” the wires – the TX pin on your board needs to be connected with the RX pin on the device, and the RX pin – with the TX pin on the device. Also make sure the ground pins of both devices are connected.

`uart.`**`any`**()

Return `True` if any characters waiting, else `False`.

`uart.`**`read`**(*[nbytes]*)

Read characters. If `nbytes` is specified then read at most that many bytes.

`uart.`**`readall`**()

Read as much data as possible.

Return value: a bytes object or `None` on timeout.

`uart.`**`readinto`**(*buf[, nbytes]*)

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

`uart.`**`readline`**`()`
Read a line, ending in a newline character.

Return value: the line read or `None` on timeout. The newline character is included in the returned bytes.

`uart.`**`write`**`(`*buf*`)`
Write the buffer of bytes to the bus.

Return value: number of bytes written or `None` on timeout.

3.4.3 SPI

The `spi` module lets you talk to a device connected to your board using a serial peripheral interface (SPI) bus. SPI uses a so-called master-slave architecture with a single master. You will need to specify the connections for three signals:

- SCLK : Serial Clock (output from master).
- MOSI : Master Output, Slave Input (output from master).
- MISO : Master Input, Slave Output (output from slave).

Functions

`microbit.spi.`**`init`**`(`*baudrate=1000000*, *bits=8*, *mode=0*, *sclk=pin13*, *mosi=pin15*, *miso=pin14*`)`
Initialize SPI communication with the specified parameters on the specified `pins`. Note that for correct communication, the parameters have to be the same on both communicating devices.

The `baudrate` defines the speed of communication.

The `bits` defines the size of bytes being transmitted. Currently only `bits=8` is supported. However, this may change in the future.

The `mode` determines the combination of clock polarity and phase according to the following convention, with polarity as the high order bit and phase as the low order bit:

SPI Mode	Polarity (CPOL)	Phase (CPHA)
0	0	0
1	0	1
2	1	0
3	1	1

Polarity (aka CPOL) 0 means that the clock is at logic value 0 when idle and goes high (logic value 1) when active; polarity 1 means the clock is at logic value 1 when idle and goes low (logic value 0) when active. Phase (aka CPHA) 0 means that data is sampled on the leading edge of the clock, and 1 means on the trailing edge (viz. https://en.wikipedia.org/wiki/Signal_edge).

The `sclk`, `mosi` and `miso` arguments specify the pins to use for each type of signal.

`spi.`**`read`**`(`*nbytes*`)`
Read at most `nbytes`. Returns what was read.

`spi.`**`write`**`(`*buffer*`)`
Write the `buffer` of bytes to the bus.

`spi.`**`write_readinto`**(*out*, *in*)
Write the `out` buffer to the bus and read any response into the `in` buffer. The length of the buffers should be the same. The buffers can be the same object.

3.4.4 I²C

The `i2c` module lets you communicate with devices connected to your board using the I²C bus protocol. There can be multiple slave devices connected at the same time, and each one has its own unique address, that is either fixed for the device or configured on it. Your board acts as the I²C master.

We use 7-bit addressing for devices because of the reasons stated here.

This may be different to other micro:bit related solutions.

How exactly you should communicate with the devices, that is, what bytes to send and how to interpret the responses, depends on the device in question and should be described separately in that device's documentation.

Functions

`microbit.i2c.`**`init`**(*freq=100000*, *sda=pin20*, *scl=pin19*)
Re-initialize peripheral with the specified clock frequency `freq` on the specified `sda` and `scl` pins.

> **Ostrzeżenie:** Changing the I²C pins from defaults will make the accelerometer and compass stop working, as they are connected internally to those pins.

`microbit.i2c.`**`read`**(*addr*, *n*, *repeat=False*)
Read `n` bytes from the device with 7-bit address `addr`. If `repeat` is `True`, no stop bit will be sent.

`microbit.i2c.`**`write`**(*addr*, *buf*, *repeat=False*)
Write bytes from `buf` to the device with 7-bit address `addr`. If `repeat` is `True`, no stop bit will be sent.

Connecting

You should connect the device's `SCL` pin to micro:bit pin 19, and the device's `SDA` pin to micro:bit pin 20. You also must connect the device's ground to the micro:bit ground (pin `GND`). You may need to power the device using an external power supply or the micro:bit.

There are internal pull-up resistors on the I²C lines of the board, but with particularly long wires or large number of devices you may need to add additional pull-up resistors, to ensure noise-free communication.

3.4.5 Accelerometer

This object gives you access to the on-board accelerometer. The accelerometer also provides convenience functions for detecting gestures. The recognised gestures are: `up`, `down`, `left`, `right`, `face up`, `face down`, `freefall`, `3g`, `6g`, `8g`, `shake`.

Functions

`microbit.accelerometer.`**`get_x`**()
Get the acceleration measurement in the `x` axis, as a positive or negative integer, depending on the direction.

microbit.accelerometer.**get_y**()
Get the acceleration measurement in the `y` axis, as a positive or negative integer, depending on the direction.

microbit.accelerometer.**get_z**()
Get the acceleration measurement in the `z` axis, as a positive or negative integer, depending on the direction.

microbit.accelerometer.**get_values**()
Get the acceleration measurements in all axes at once, as a three-element tuple of integers ordered as X, Y, Z.

microbit.accelerometer.**current_gesture**()
Return the name of the current gesture.

Informacja: MicroPython understands the following gesture names: `"up"`, `"down"`, `"left"`, `"right"`, `"face up"`, `"face down"`, `"freefall"`, `"3g"`, `"6g"`, `"8g"`, `"shake"`. Gestures are always represented as strings.

---

microbit.accelerometer.**is_gesture**(*name*)
Return True or False to indicate if the named gesture is currently active.

microbit.accelerometer.**was_gesture**(*name*)
Return True or False to indicate if the named gesture was active since the last call.

microbit.accelerometer.**get_gestures**()
Return a tuple of the gesture history. The most recent is listed last. Also clears the gesture history before returning.

Examples

A fortune telling magic 8-ball. Ask a question then shake the device for an answer.

```
# Magic 8 ball by Nicholas Tollervey. February 2016.
#
# Ask a question then shake.
#
# This program has been placed into the public domain.
from microbit import *
import random

answers = [
    "It is certain",
    "It is decidedly so",
    "Without a doubt",
    "Yes, definitely",
    "You may rely on it",
    "As I see it, yes",
    "Most likely",
    "Outlook good",
    "Yes",
    "Signs point to yes",
    "Reply hazy try again",
    "Ask again later",
    "Better not tell you now",
    "Cannot predict now",
    "Concentrate and ask again",
    "Don't count on it",
    "My reply is no",
    "My sources say no",
    "Outlook not so good",
```

```
    "Very doubtful",
]

while True:
    display.show('8')
    if accelerometer.was_gesture('shake'):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(answers))
    sleep(10)
```

Simple Slalom. Move the device to avoid the obstacles.

```
# Simple Slalom by Larry Hastings, September 2015
#
# This program has been placed into the public domain.

import microbit as m
import random

p = m.display.show

min_x = -1024
max_x = 1024
range_x = max_x - min_x

wall_min_speed = 400
player_min_speed = 200

wall_max_speed = 100
player_max_speed = 50

speed_max = 12


while True:

    i = m.Image('00000:'*5)
    s = i.set_pixel

    player_x = 2

    wall_y = -1
    hole = 0

    score = 0
    handled_this_wall = False

    wall_speed = wall_min_speed
    player_speed = player_min_speed

    wall_next = 0
    player_next = 0

    while True:
        t = m.running_time()
        player_update = t >= player_next
        wall_update = t >= wall_next
```

```
        if not (player_update or wall_update):
            next_event = min(wall_next, player_next)
            delta = next_event - t
            m.sleep(delta)
            continue

        if wall_update:
            # calculate new speeds
            speed = min(score, speed_max)
            wall_speed = wall_min_speed + int((wall_max_speed - wall_min_speed) *␣
↪speed / speed_max)
            player_speed = player_min_speed + int((player_max_speed - player_min_
↪speed) * speed / speed_max)

            wall_next = t + wall_speed
            if wall_y < 5:
                # erase old wall
                use_wall_y = max(wall_y, 0)
                for wall_x in range(5):
                    if wall_x != hole:
                        s(wall_x, use_wall_y, 0)

        wall_reached_player = (wall_y == 4)
        if player_update:
            player_next = t + player_speed
            # find new x coord
            x = m.accelerometer.get_x()
            x = min(max(min_x, x), max_x)
            # print("x accel", x)
            s(player_x, 4, 0) # turn off old pixel
            x = ((x - min_x) / range_x) * 5
            x = min(max(0, x), 4)
            x = int(x + 0.5)
            # print("have", position, "want", x)

            if not handled_this_wall:
                if player_x < x:
                    player_x += 1
                elif player_x > x:
                    player_x -= 1
            # print("new", position)
            # print()

        if wall_update:
            # update wall position
            wall_y += 1
            if wall_y == 7:
                wall_y = -1
                hole = random.randrange(5)
                handled_this_wall = False

            if wall_y < 5:
                # draw new wall
                use_wall_y = max(wall_y, 0)
                for wall_x in range(5):
                    if wall_x != hole:
                        s(wall_x, use_wall_y, 6)
```

```
        if wall_reached_player and not handled_this_wall:
            handled_this_wall = True
            if (player_x != hole):
                # collision! game over!
                break
            score += 1

        if player_update:
            s(player_x, 4, 9) # turn on new pixel

        p(i)

    p(i.SAD)
    m.sleep(1000)
    m.display.scroll("Score:" + str(score))

    while True:
        if (m.button_a.is_pressed() and m.button_a.is_pressed()):
            break
        m.sleep(100)
```

3.4.6 Compass

This module lets you access the built-in electronic compass. Before using, the compass should be calibrated, otherwise the readings may be wrong.

Ostrzeżenie: Calibrating the compass will cause your program to pause until calibration is complete. Calibration consists of a little game to draw a circle on the LED display by rotating the device.

Functions

`microbit.compass.`**`calibrate`**`()`
Starts the calibration process. An instructive message will be scrolled to the user after which they will need to rotate the device in order to draw a circle on the LED display.

`microbit.compass.`**`is_calibrated`**`()`
Returns `True` if the compass has been successfully calibrated, and returns `False` otherwise.

`microbit.compass.`**`clear_calibration`**`()`
Undoes the calibration, making the compass uncalibrated again.

`microbit.compass.`**`get_x`**`()`
Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.`**`get_y`**`()`
Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.`**`get_z`**`()`
Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.`**`heading`**`()`
Gives the compass heading, calculated from the above readings, as an integer in the range from 0 to 360,

representing the angle in degrees, clockwise, with north as 0. If the compass has not been calibrated, then this will call `calibrate`.

`microbit.compass.`**`get_field_strength`**`()`
Returns an integer indication of the magnitude of the magnetic field around the device.

Example

```
"""
    compass.py
    ~~~~~~~~~~

    Creates a compass.

    The user will need to calibrate the compass first. The compass uses the
    built-in clock images to display the position of the needle.

"""
from microbit import *


# Start calibrating
compass.calibrate()

# Try to keep the needle pointed in (roughly) the correct direction
while True:
    sleep(100)
    needle = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[needle])
```

ROZDZIAŁ 4

Bluetooth

Wprawdzie BBC micro:bit jest wyposażony w sprzęt pozwalający na pracę jako urządzenie Bluetooth Low Energy (BLE), ma jednak tylko 16 kilobajtów pamięci RAM. Oprogramowanie BLE zajmuje 12 kilobajtów pamięci RAM, przez co nie zostaje jej wystarczająco dużo na MicroPythona.

Jest możliwe, że przyszłe wersje urządzenia będą wyposażone w 32 kilobajty pamięci RAM, co może już wystarczyć. Dopóki to nie nastąpi, wsparcie dla BLE w MicroPythonie jest mało prawdopodobne.

Informacja: MicroPython daje dostęp do wbudowanego radia poprzez moduł `radio`. Pozwala to użytkownikom na stworzenie prostej, lecz efektywnej bezprzewodowej sieci urządzeń micro:bit.

Ponadto, protokół używany w module `radio` jest znacznie prostszy niż BLE. Jest dzięki temu łatwiejszy w użyciu w edukacji.

ROZDZIAŁ 5

Local Persistent File System

It is useful to store data in a persistent manner so that it remains intact between restarts of the device. On traditional computers this is often achieved by a file system consisting of named files that hold raw data, and named directories that contain files. Python supports the various operations needed to work with such file systems.

However, since the micro:bit is a limited device in terms of both hardware and storage capacity MicroPython provides a small subset of the functions needed to persist data on the device. Because of memory constraints **there is approximately 30k of storage available** on the file system.

> **Ostrzeżenie:** Re-flashing the device will DESTROY YOUR DATA.
>
> Since the file system is stored in the micro:bit's flash memory and flashing the device rewrites all the available flash memory then all your data will be lost if you flash your device.
>
> However, if you switch your device off the data will remain intact until you either delete it (see below) or re-flash the device.

MicroPython on the micro:bit provides a flat file system; i.e. there is no notion of a directory hierarchy, the file system is just a list of named files. Reading and writing a file is achieved via the standard Python `open` function and the resulting file-like object (representing the file) of types `TextIO` or `BytesIO`. Operations for working with files on the file system (for example, listing or deleting files) are contained within the *os* module.

If a file ends in the `.py` file extension then it can be imported. For example, a file named `hello.py` can be imported like this: `import hello`.

An example session in the MicroPython REPL may look something like this:

```
>>> with open('hello.py', 'w') as hello:
...     hello.write("print('Hello')")
...
>>> import hello
Hello
>>> with open('hello.py') as hello:
...   print(hello.read())
...
```

```
print('Hello')
>>> import os
>>> os.listdir()
['hello.py']
>>> os.remove('hello.py')
>>> os.listdir()
[]
```

open (*filename*, *mode='r'*)

Returns a file object representing the file named in the argument `filename`. The mode defaults to `'r'` which means open for reading in text mode. The other common mode is `'w'` for writing (overwriting the content of the file if it already exists). Two other modes are available to be used in conjunction with the ones describes above: `'t'` means text mode (for reading and writing strings) and `'b'` means binary mode (for reading and writing bytes). If these are not specified then `'t'` (text mode) is assumed. When in text mode the file object will be an instance of `TextIO`. When in binary mode the file object will be an instance of `BytesIO`. For example, use `'rb'` to read binary data from a file.

class TextIO
class BytesIO

Instances of these classes represent files in the micro:bit's flat file system. The TextIO class is used to represent text files. The BytesIO class is used to represent binary files. They work in exactly the same except that TextIO works with strings and BytesIO works with bytes.

You do not directly instantiate these classes. Rather, an appropriately configured instance of the class is returned by the `open` function described above.

close()

Flush and close the file. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise an exception.

name()

Returns the name of the file the object represents. This will be the same as the `filename` argument passed into the call to the `open` function that instantiated the object.

read(*size*)

Read and return at most `size` characters as a single string or `size` bytes from the file. As a convenience, if `size` is unspecified or -1, all the data contained in the file is returned. Fewer than `size` characters or bytes may be returned if there are less than `size` characters or bytes remaining to be read from the file.

If 0 characters or bytes are returned, and `size` was not 0, this indicates end of file.

A `MemoryError` exception will occur if `size` is larger than the available RAM.

readinto(*buf*, *n=-1*)

Read characters or bytes into the buffer `buf`. If `n` is supplied, read `n` number of bytes or characters into the buffer `buf`.

readline(*size*)

Read and return one line from the file. If `size` is specified, at most `size` characters will be read.

The line terminator is always `'\n'` for strings or `b'\n'` for bytes.

writable()

Return `True` if the file supports writing. If `False`, `write()` will raise `OSError`.

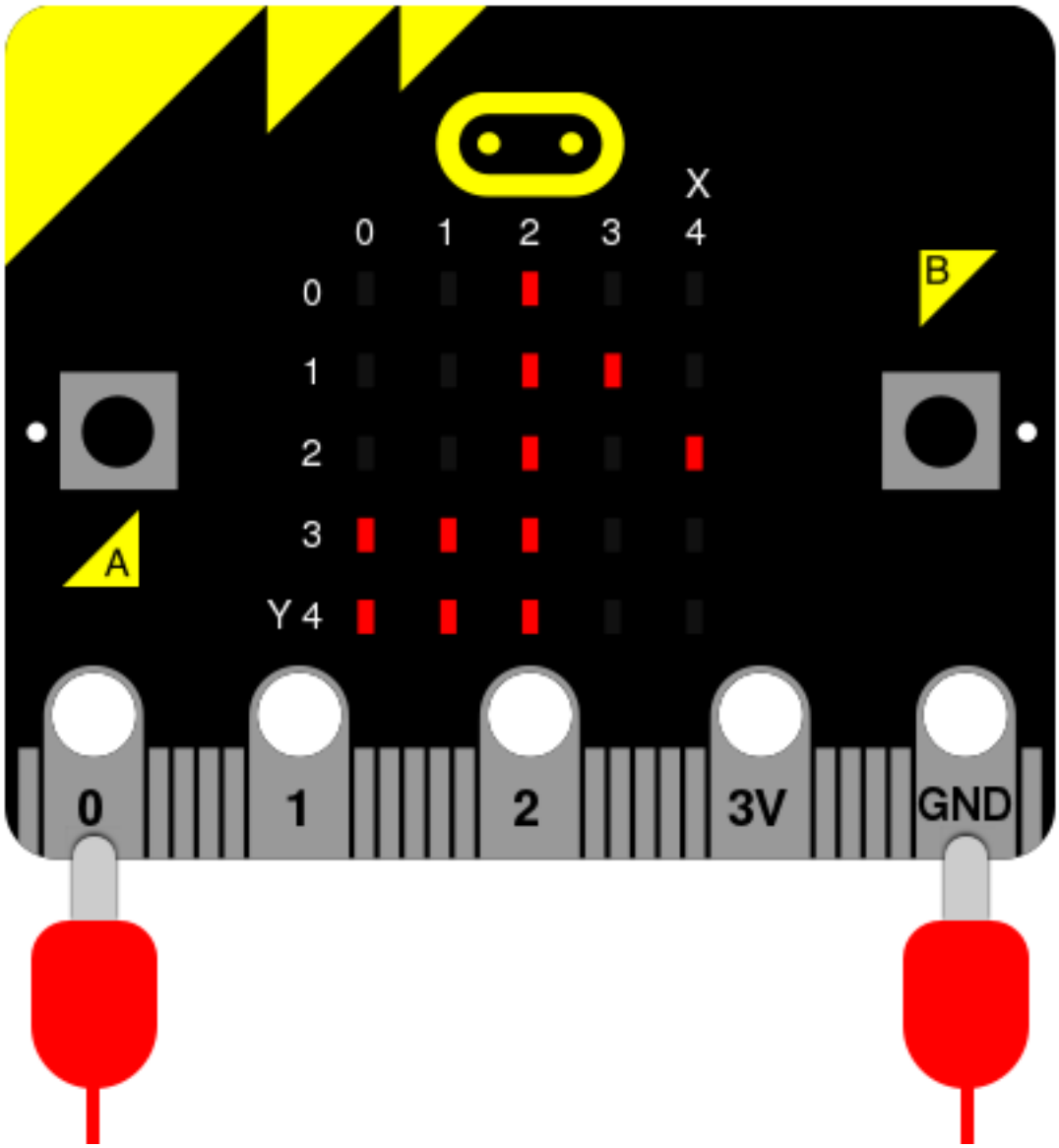
write(*buf*)

Write the string or bytes `buf` to the file and return the number of characters or bytes written.

ROZDZIAŁ 6

Music

This is the `music` module. You can use it to play simple tunes, provided that you connect a speaker to your board. By default the `music` module expects the speaker to be connected via pin 0:



This arrangement can be overridden (as discussed below).

To access this module you need to:

```
import music
```

We assume you have done this for the examples below.

6.1 Musical Notation

An individual note is specified thus:

```
NOTE[octave][:duration]
```

For example, `A1:4` refers to the note „A" in octave 1 that lasts for four ticks (a tick is an arbitrary length of time defined by a tempo setting function - see below). If the note name `R` is used then it is treated as a rest (silence).

Accidentals (flats and sharps) are denoted by the `b` (flat - a lower case b) and `#` (sharp - a hash symbol). For example, `Ab` is A-flat and `C#` is C-sharp.

Note names are case-insensitive.

The `octave` and `duration` parameters are states that carry over to subsequent notes until re-specified. The default states are `octave = 4` (containing middle C) and `duration = 4` (a crotchet, given the default tempo settings - see below).

For example, if 4 ticks is a crotchet, the following list is crotchet, quaver, quaver, crotchet based arpeggio:

```
['c1:4', 'e:2', 'g', 'c2:4']
```

The opening of Beethoven's 5th Symphony would be encoded thus:

```
['r4:2', 'g', 'g', 'g', 'eb:8', 'r:2', 'f', 'f', 'f', 'd:8']
```

The definition and scope of an octave conforms to the table listed on this page about scientific pitch notation. For example, middle „C" is `'c4'` and concert „A" (440) is `'a4'`. Octaves start on the note „C".

6.2 Functions

`music.`**`set_tempo`**(*ticks=4*, *bpm=120*)

Sets the approximate tempo for playback.

A number of ticks (expressed as an integer) constitute a beat. Each beat is to be played at a certain frequency per minute (expressed as the more familiar BPM - beats per minute - also as an integer).

Suggested default values allow the following useful behaviour:

- `music.set_tempo()` - reset the tempo to default of ticks = 4, bpm = 120
- `music.set_tempo(ticks=8)` - change the „definition" of a beat
- `music.set_tempo(bpm=180)` - just change the tempo

To work out the length of a tick in milliseconds is very simple arithmetic: `60000/bpm/ticks_per_beat` . For the default values that's `60000/120/4 = 125 milliseconds` or `1 beat = 500 milliseconds`.

`music.`**`get_tempo`**()

Gets the current tempo as a tuple of integers: `(ticks, bpm)`.

`music.`**`play`**(*music*, *pin=microbit.pin0*, *wait=True*, *loop=False*)

Plays `music` containing the musical DSL defined above.

If `music` is a string it is expected to be a single note such as, `'c1:4'`.

If `music` is specified as a list of notes (as defined in the section on the musical DSL, above) then they are played one after the other to perform a melody.

In both cases, the `duration` and `octave` values are reset to their defaults before the music (whatever it may be) is played.

An optional argument to specify the output pin can be used to override the default of `microbit.pin0`.

If `wait` is set to `True`, this function is blocking.

If `loop` is set to `True`, the tune repeats until `stop` is called (see below) or the blocking call is interrupted.

`music.pitch` (*frequency*, *len=-1*, *pin=microbit.pin0*, *wait=True*)
Plays a pitch at the integer frequency given for the specified number of milliseconds. For example, if the frequency is set to 440 and the length to 1000 then we hear a standard concert A for one second.

If `wait` is set to `True`, this function is blocking.

If `len` is negative the pitch is played continuously until either the blocking call is interrupted or, in the case of a background call, a new frequency is set or `stop` is called (see below).

`music.stop` (*pin=microbit.pin0*)
Stops all music playback on a given pin.

`music.reset` ()
Resets the state of the following attributes in the following way:

- `ticks = 4`
- `bpm = 120`
- `duration = 4`
- `octave = 4`

6.2.1 Built in Melodies

For the purposes of education and entertainment, the module contains several example tunes that are expressed as Python lists. They can be used like this:

```
>>> import music
>>> music.play(music.NYAN)
```

All the tunes are either out of copyright, composed by Nicholas H.Tollervey and released to the public domain or have an unknown composer and are covered by a fair (educational) use provision.

They are:

- `DADADADUM` - the opening to Beethoven's 5th Symphony in C minor.
- `ENTERTAINER` - the opening fragment of Scott Joplin's Ragtime classic „The Entertainer".
- `PRELUDE` - the opening of the first Prelude in C Major of J.S.Bach's 48 Preludes and Fugues.
- `ODE` - the „Ode to Joy" theme from Beethoven's 9th Symphony in D minor.
- `NYAN` - the Nyan Cat theme (http://www.nyan.cat/). The composer is unknown. This is fair use for educational porpoises (as they say in New York).
- `RINGTONE` - something that sounds like a mobile phone ringtone. To be used to indicate an incoming message.
- `FUNK` - a funky bass line for secret agents and criminal masterminds.
- `BLUES` - a boogie-woogie 12-bar blues walking bass.
- `BIRTHDAY` - „Happy Birthday to You..." for copyright status see: http://www.bbc.co.uk/news/world-us-canada-34332853
- `WEDDING` - the bridal chorus from Wagner's opera „Lohengrin".
- `FUNERAL` - the „funeral march" otherwise known as Frédéric Chopin's Piano Sonata No. 2 in B minor, Op. 35.
- `PUNCHLINE` - a fun fragment that signifies a joke has been made.
- `PYTHON` - John Philip Sousa's march „Liberty Bell" aka, the theme for „Monty Python's Flying Circus" (after which the Python programming language is named).

- `BADDY` - silent movie era entrance of a baddy.
- `CHASE` - silent movie era chase scene.
- `BA_DING` - a short signal to indicate something has happened.
- `WAWAWAWAA` - a very sad trombone.
- `JUMP_UP` - for use in a game, indicating upward movement.
- `JUMP_DOWN` - for use in a game, indicating downward movement.
- `POWER_UP` - a fanfare to indicate an achievement unlocked.
- `POWER_DOWN` - a sad fanfare to indicate an achievement lost.

6.2.2 Example

```
"""
    music.py
    ~~~~~~~~

    Plays a simple tune using the Micropython music module.
    This example requires a speaker/buzzer/headphones connected to P0 and GND.
"""
from microbit import *
import music

# play Prelude in C.
notes = [
    'c4:1', 'e', 'g', 'c5', 'e5', 'g4', 'c5', 'e5', 'c4', 'e', 'g', 'c5', 'e5', 'g4',
↪'c5', 'e5',
    'c4', 'd', 'g', 'd5', 'f5', 'g4', 'd5', 'f5', 'c4', 'd', 'g', 'd5', 'f5', 'g4',
↪'d5', 'f5',
    'b3', 'd4', 'g', 'd5', 'f5', 'g4', 'd5', 'f5', 'b3', 'd4', 'g', 'd5', 'f5', 'g4',
↪'d5', 'f5',
    'c4', 'e', 'g', 'c5', 'e5', 'g4', 'c5', 'e5', 'c4', 'e', 'g', 'c5', 'e5', 'g4',
↪'c5', 'e5',
    'c4', 'e', 'a', 'e5', 'a5', 'a4', 'e5', 'a5', 'c4', 'e', 'a', 'e5', 'a5', 'a4',
↪'e5', 'a5',
    'c4', 'd', 'f#', 'a', 'd5', 'f#4', 'a', 'd5', 'c4', 'd', 'f#', 'a', 'd5', 'f#4',
↪'a', 'd5',
    'b3', 'd4', 'g', 'd5', 'g5', 'g4', 'd5', 'g5', 'b3', 'd4', 'g', 'd5', 'g5', 'g4',
↪'d5', 'g5',
    'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g', 'c5', 'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g
↪', 'c5',
    'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g', 'c5', 'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g
↪', 'c5',
    'a3', 'c4', 'e', 'g', 'c5', 'e4', 'g', 'c5', 'a3', 'c4', 'e', 'g', 'c5', 'e4', 'g
↪', 'c5',
    'd3', 'a', 'd4', 'f#', 'c5', 'd4', 'f#', 'c5', 'd3', 'a', 'd4', 'f#', 'c5', 'd4',
↪'f#', 'c5',
    'g3', 'b', 'd4', 'g', 'b', 'd', 'g', 'b', 'g3', 'b3', 'd4', 'g', 'b', 'd', 'g', 'b
↪'
]

music.play(notes)
```

ROZDZIAŁ 7

NeoPixel

The `neopixel` module lets you use Neopixel (WS2812) individually addressable RGB LED strips with the Microbit. Note to use the `neopixel` module, you need to import it separately with:

```
import neopixel
```

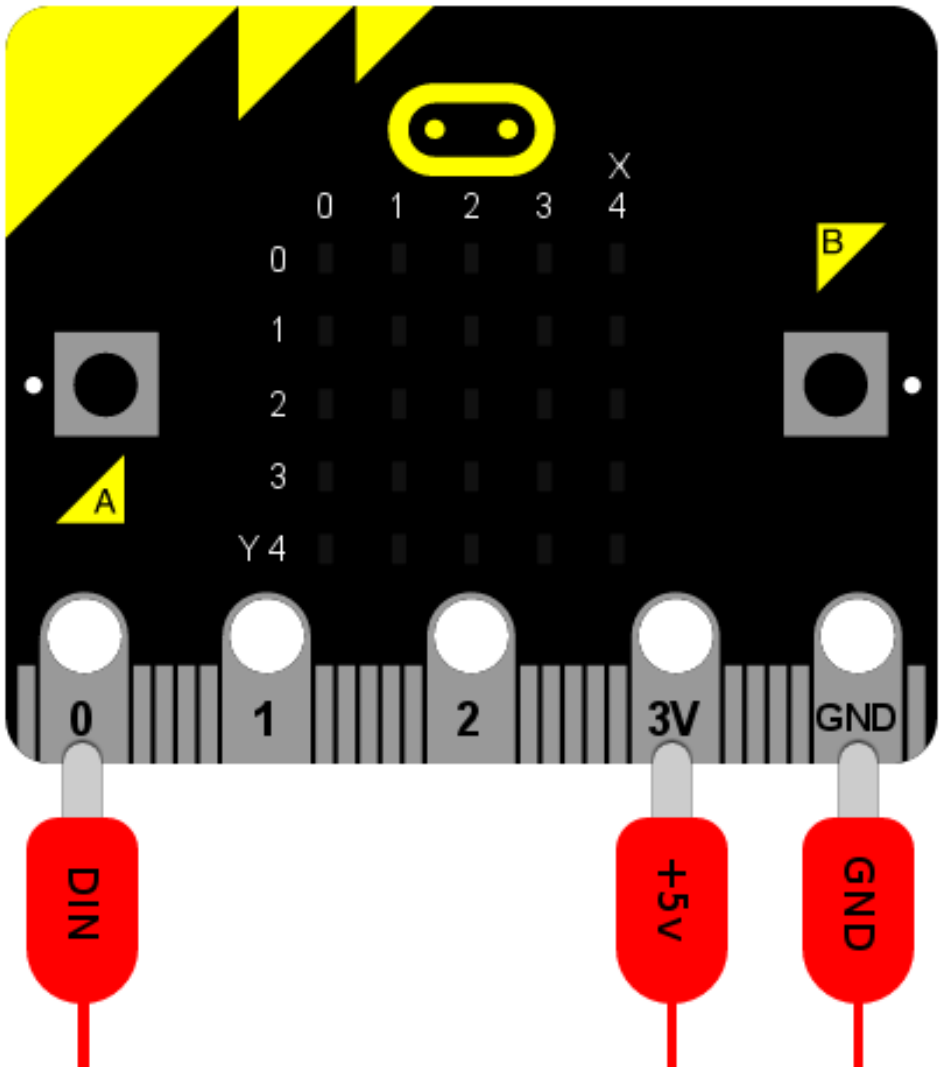
Informacja: From our tests, the Microbit Neopixel module can drive up to around 256 Neopixels. Anything above that and you may experience weird bugs and issues.

NeoPixels are fun strips of multi-coloured programmable LEDs. This module contains everything to plug them into a micro:bit and create funky displays, art and games such as the demo shown below.

To connect a strip of neopixels you'll need to attach the micro:bit as shown below (assuming you want to drive the pixels from pin 0 - you can connect neopixels to pins 1 and 2 too). The label on the crocodile clip tells you where to attach the other end on the neopixel strip.

Ostrzeżenie: Do not use the 3v connector on the Microbit to power any more than 8 Neopixels at a time.

If you wish to use more than 8 Neopixels, you must use a separate 3v-5v power supply for the Neopixel power pin.



7.1 Classes

class neopixel.**NeoPixel**(*pin*, *n*)

Initialise a new strip of `n` number of neopixel LEDs controlled via pin `pin`. Each pixel is addressed by a position (starting from 0). Neopixels are given RGB (red, green, blue) values between 0-255 as a tuple. For example, `(255,255,255)` is white.

clear()

Clear all the pixels.

show()

Show the pixels. Must be called for any updates to become visible.

7.2 Operations

Writing the colour doesn't update the display (use `show()` for that).

```
np[0] = (255, 0, 128)  # first element
np[-1] = (0, 255, 0)  # last element
np.show()  # only now will the updated value be shown
```

To read the colour of a specific pixel just reference it.

```
print(np[0])
```

7.3 Using Neopixels

Interact with Neopixels as if they were a list of tuples. Each tuple represents the RGB (red, green and blue) mix of colours for a specific pixel. The RGB values can range between 0 to 255.

For example, initialise a strip of 8 neopixels on a strip connected to pin0 like this:

```
import neopixel
np = neopixel.NeoPixel(pin0, 8)
```

Set pixels by indexing them (like with a Python list). For instance, to set the first pixel to full brightness red, you would use:

```
np[0] = (255, 0, 0)
```

Or the final pixel to purple:

```
np[-1] = (255, 0, 255)
```

Get the current colour value of a pixel by indexing it. For example, to print the first pixel's RGB value use:

```
print(np[0])
```

Finally, to push the new colour data to your Neopixel strip, use the .show() function:

```
np.show()
```

If nothing is happening, it's probably because you've forgotten this final step..!

Informacja: If you're not seeing anything change on your Neopixel strip, make sure you're `show()` at least somewhere otherwise your updates won't be shown.

7.4 Example

```
"""
    neopixel_random.py

    Repeatedly displays random colours onto the LED strip.
    This example requires a strip of 8 Neopixels (WS2812) connected to pin0.
```

```
"""
from microbit import *
import neopixel
from random import randint

# Setup the Neopixel strip on pin0 with a length of 8 pixels
np = neopixel.NeoPixel(pin0, 8)

while True:
    #Iterate over each LED in the strip

    for pixel_id in range(0, len(np)):
        red = randint(0, 60)
        green = randint(0, 60)
        blue = randint(0, 60)

        # Assign the current LED a random red, green and blue value between 0 and 60
        np[pixel_id] = (red, green, blue)

        # Display the current pixel data on the Neopixel strip
        np.show()
        sleep(100)
```

ROZDZIAŁ 8

The os Module

MicroPython contains an os module based upon the os module in the Python standard library. It's used for accessing what would traditionally be termed as operating system dependent functionality. Since there is no operating system in MicroPython the module provides functions relating to the management of the simple on-device persistent file system and information about the current system.

To access this module you need to:

```
import os
```

We assume you have done this for the examples below.

8.1 Functions

os.**listdir**()
: Returns a list of the names of all the files contained within the local persistent on-device file system.

os.**remove**(*filename*)
: Removes (deletes) the file named in the argument filename. If the file does not exist an OSError exception will occur.

os.**size**(*filename*)
: Returns the size, in bytes, of the file named in the argument filename. If the file does not exist an OSError exception will occur.

os.**uname**()
: Returns information identifying the current operating system. The return value is an object with five attributes:

  - sysname - operating system name
  - nodename - name of machine on network (implementation-defined)
  - release - operating system release
  - version - operating system version

- `machine` - hardware identifier

Informacja: There is no underlying operating system in MicroPython. As a result the information returned by the `uname` function is mostly useful for versioning details.

ROZDZIAŁ 9

Radio

Moduł „radio” umożliwia urządzeniom wspólną pracę przez prostą sieć bezprzewodową.

Moduł radia z zamyśle jest bardzo prosty:

- Przesyłane wiadomości są konfigurowanej długości (do 251 bajtów).
- Otrzymywane wiadomości przechowywane są w kolejce o określonej konfigurowalnej wielkości (im większa kolejka tym większe zużycie pamięci RAM). Jeżeli kolejka jest pełna, nowa wiadomość przychodząca zostanie zignorowana.
- Wiadomości przesyłane są wcześniej wybranym kanałem (zakres numeracji 0-100).
- Nadawanie z określoną mocą sygnału - im większa moc, tym większy zasięg nadawania.
- Wiadomości filtrowane są według adresów (jak numery domów) i grup (jak nazwisko adresata pod danym adresem).
- Przepustowość może być na jednym z trzech predefiniowanych ustawień.
- Wysyłanie i odbierania danych różnych typów.
- Dodatkowym ułatwieniem dla dzieci są łatwe do wysyłania i odbierania wiadomości w postaci ciągów znaków.
- Domyślna konfiguracja jest rozsądna i kompatybilna z innymi platformami związanymi z BBC micro:bit.

Aby mieć dostęp do tego modułu musisz wykonać import:

```
import radio
```

Zakładamy, że już to zrobiłeś dla poniższych przykładów.

9.1 Stałe

radio.**RATE_250KBIT**
: Stała oznaczająca przepustowość 256 Kbit na sekundę.

radio.**RATE_1MBIT**
: Stała oznaczająca przepustowość 1 MBit na sekundę.

radio.**RATE_2MBIT**
Stała oznaczająca przepustowość 2 MBit na sekundę.

9.2 Funkcje

radio.**on**()
Włącza radio. Funkcja musi być wywołana świadomie ponieważ radio pobiera prąd i zajmuje zasoby pamięci, które mogą być potrzebne do czegoś innego.

radio.**off**()
Wyłącza radio oszczędzając prąd i pamięć.

radio.**config**(***kwargs*)
Konfiguruje ustawienia związane z radiem. Nazwy ustawień oparte są o słowa kluczowe. Dostępne ustawienia i ich rozsądne domyślne wartości są podane poniżej.

`length` - długość (wartość domyślna = 32) definiuje maksymalną długość, w bajtach, wiadomości przesyłanej przez radio. Może mieć do 251 bajtów długości (254 - 3 bajty dla S0, długości i preambuły S1).

`queue` - kolejka (wartość domyślna = 3) określa liczbę wiadomości, które mogą być przechowywane w kolejce wiadomości przychodzących. Jeżeli nie ma wolnego miejsca w kolejce, następna wiadomość przychodząca zostanie pominięta.

`channel` - kanał (wartość domyślna = 7) musi być liczbą całkowitą od 0 do 100 (włącznie), która określa kanał na który nastrojone jest radio. Wiadomości będą wysyłane tym kanałem i tylko wiadomości otrzymane tym kanałem będą umieszczone w kolejce wiadomości przychodzących. Każdy kanał ma szerokość 1MHz, począwszy od 2400MHz.

`power` - moc (domyślna wartość = 6) jest liczbą całkowitą od 0 do 7 (włącznie) służącą do wskazania mocy sygnału w czasie nadawania. Im wyższa liczba tym mocniejszy sygnał, ale też większe zużycie prądu przez urządzenie. Wartości stałej odwołują się do poszczególnych wartości mocy nadawania wyrażonej w decybelach miliwatów [dbm]: -30, -20, -16, -12, -8, -4, 0, 4.

`address` - adres (domyślna wartość = 0x75626974) to ustalona nazwa, wyrażona jako 32 bitowy adres, stosowana do filtrowania przychodzących pakietów na poziomie sprzętowym, zachowując tylko pasujące do ustawionego adresu. Wartość domyślna jest również stosowana domyślnie na innych platformach kompatybilnych z micro:bit.

`group` - grupa (domyślna wartość = 0) jest 8 bitową wartością (0-255) stosowaną razem z adresem, do filtrowania wiadomości. Dla lepszego zobrazowania „adres" jest jak adres domu a „grupa" jest jak osoba w nim mieszkająca do której adresujemy przesyłkę.

`data_rate` - (domyślne ustawienie = radio.RATE_1MBIT) wyraża prędkość przesyłu danych. Może przyjmować jedną ze stałych zdefiniowanych w module „radio": `RATE_250KBIT`, `RATE_1MBIT` lub `RATE_2MBIT`.

Jeżeli „config" nie zostanie wywołany wtedy przyjęte zostaną ustawienia o domyślnych wartościach.

radio.**reset**()
Zresetuj ustawienia do wartości domyślnych (patrz powyżej w dokumentacji funkcji config).

Informacja: Żadna z powyższych metod wysyłania lub nadawania nie będzie działać jeżeli radio nie będzie włączone.

---

radio.**send_bytes**(*treść wiadomości*)
Wysyła wiadomość składającą się z bajtów.

radio.**receive_bytes**()
: Wczytuje z kolejki pierwszą wiadomość przychodzącą. Zwraca „None” (brak) jeżeli nie ma oczekujących wiadomości. Wiadomości wyświetlane są jako bajty.

radio.**receive_bytes_into**(*buffer*)
: Odbiera następną wiadomość przychodzącą z kolejki wiadomości. Kopiuje treść wiadomości do bufora, ucinając jej koniec jeśli to konieczne. Zwraca „None” jeżeli nie ma wiadomości oczekujących, jeżeli są, zwraca długość wiadomości (długość wiadomości może być dłuższa od długości bufora).

radio.**send**(*"treść wiadomości"*)
: Wysyła wiadomość w postaci ciągu znaków. Jest odpowiednikiem `send_bytes(bytes(treść wiadomości, 'utf8'))`, ale z `b'\x01\x00\x01'` dodanymi na początku (dla zapewnienia kompatybilności z innymi platformami współpracującymi z micro:bit).

radio.**receive**()
: Funkcja działa dokładnie w ten sam sposób co `receive_bytes` przy czym zwraca wiadomość w takiej postaci w jakiej była wysyłana.

  Obecnie jest to odpowiednik `str(receive_bytes(), 'utf8')`, ale ze sprawdzeniem czy pierwsze 3 bajty to `b'\x01\x00\x01'` (dla zapewnienia kompatybilności z innymi platformami współpracującymi z micro:bit). Te trzy bajty są usuwane przed konwersją na ciąg znaków.

  Wyjątek `ValueError` wyświetlany jest w przypadku niepowodzenia konwersji treści wiadomości na ciąg znaków.

9.2.1 Przykłady

```
# A micro:bit Firefly.
# By Nicholas H.Tollervey. Released to the public domain.
import radio
import random
from microbit import display, Image, button_a, sleep

# Create the "flash" animation frames. Can you work out how it's done?
flash = [Image().invert()*(i/9) for i in range(9, -1, -1)]

# The radio won't work unless it's switched on.
radio.on()

# Event loop.
while True:
    # Button A sends a "flash" message.
    if button_a.was_pressed():
        radio.send('flash')  # a-ha
    # Read any incoming messages.
    incoming = radio.receive()
    if incoming == 'flash':
        # If there's an incoming "flash" message display
        # the firefly flash animation after a random short
        # pause.
        sleep(random.randint(50, 350))
        display.show(flash, delay=100, wait=False)
        # Randomly re-broadcast the flash message after a
        # slight delay.
        if random.randint(0, 9) == 0:
            sleep(500)
            radio.send('flash')  # a-ha
```

ROZDZIAŁ 10

Random Number Generation

This module is based upon the `random` module in the Python standard library. It contains functions for generating random behaviour.

To access this module you need to:

```
import random
```

We assume you have done this for the examples below.

10.1 Functions

`random.`**`getrandbits`**(*n*)
Returns an integer with `n` random bits.

Ostrzeżenie: Because the underlying generator function returns at most 30 bits, `n` may only be a value between 1-30 (inclusive).

`random.`**`seed`**(*n*)
Initialize the random number generator with a known integer `n`. This will give you reproducibly deterministic randomness from a given starting state (`n`).

`random.`**`randint`**(*a*, *b*)
Return a random integer `N` such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

`random.`**`randrange`**(*stop*)
Return a randomly selected integer between zero and up to (but not including) `stop`.

`random.`**`randrange`**(*start*, *stop*)
Return a randomly selected integer from `range(start, stop)`.

`random.`**`randrange`**(*start*, *stop*, *step*)
Return a randomly selected element from `range(start, stop, step)`.

random.**choice**(*seq*)
Return a random element from the non-empty sequence `seq`. If `seq` is empty, raises `IndexError`.

random.**random**()
Return the next random floating point number in the range [0.0, 1.0)

random.**uniform**(*a*, *b*)
Return a random floating point number `N` such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

ROZDZIAŁ 11

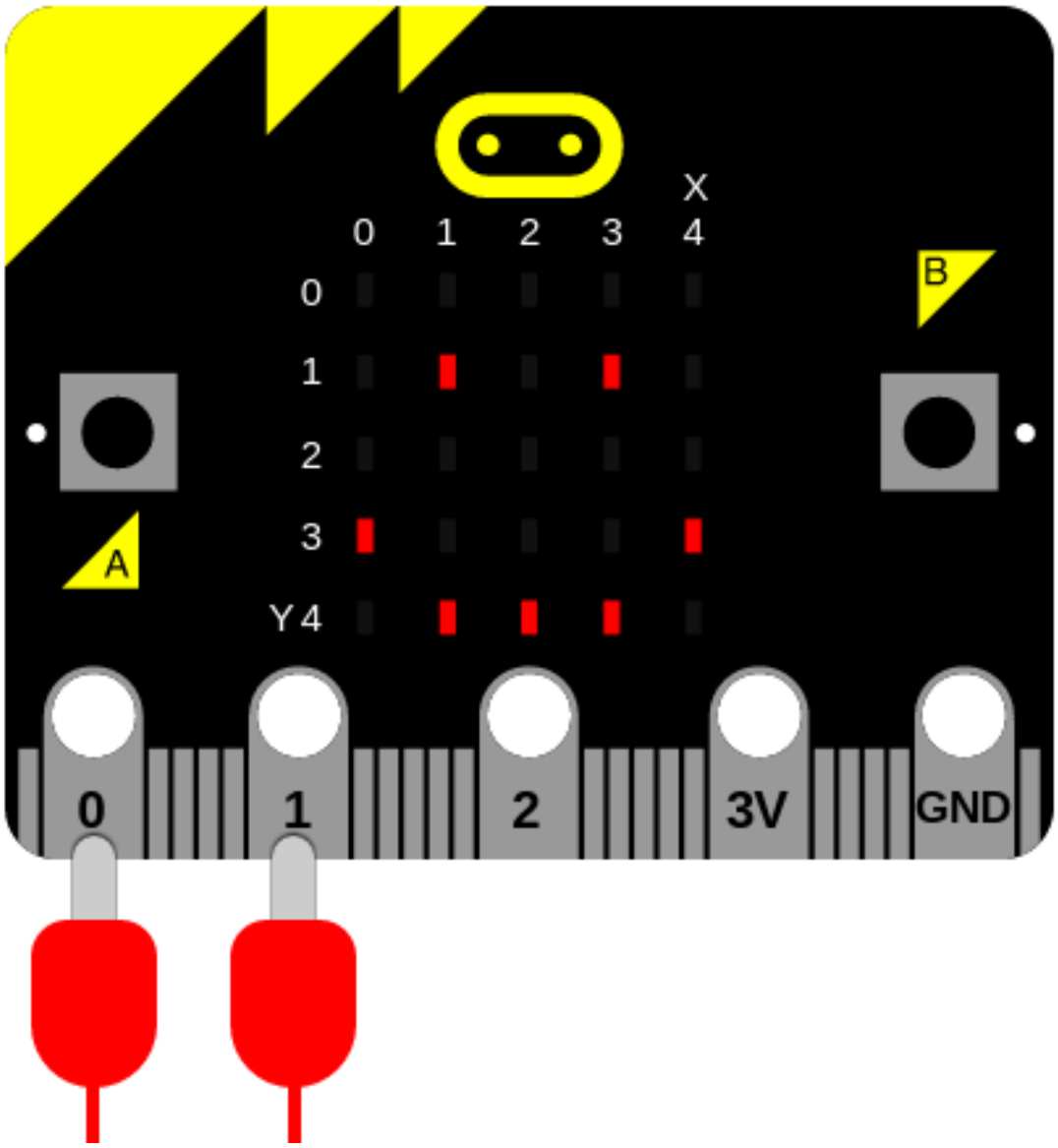
Speech

Ostrzeżenie: WARNING! THIS IS ALPHA CODE.

We reserve the right to change this API as development continues.

The quality of the speech is not great, merely „good enough". Given the constraints of the device you may encounter memory errors and / or unexpected extra sounds during playback. It's early days and we're improving the code for the speech synthesiser all the time. Bug reports and pull requests are most welcome.

This module makes microbit talk, sing and make other speech like sounds provided that you connect a speaker to your board as shown below:



Informacja: This work is based upon the amazing reverse engineering efforts of Sebastian Macke based upon an old text-to-speech (TTS) program called SAM (Software Automated Mouth) originally released in 1982 for the Commodore 64. The result is a small C library that we have adopted and adapted for the micro:bit. You can find out more from his homepage. Much of the information in this document was gleaned from the original user's manual which can be found here.

The speech synthesiser can produce around 2.5 seconds worth of sound from up to 255 characters of textual input.

To access this module you need to:

```
import speech
```

We assume you have done this for the examples below.

11.1 Functions

`speech.translate`(*words*)
Given English words in the string `words`, return a string containing a best guess at the appropriate phonemes to pronounce. The output is generated from this text to phoneme translation table.

This function should be used to generate a first approximation of phonemes that can be further hand-edited to improve accuracy, inflection and emphasis.

`speech.pronounce`(*phonemes*, ***, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Pronounce the phonemes in the string `phonemes`. See below for details of how to use phonemes to finely control the output of the speech synthesiser. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice.

`speech.say`(*words*, ***, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Say the English words in the string `words`. The result is semi-accurate for English. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice. This is a short-hand equivalent of: `speech.pronounce(speech.translate(words))`

`speech.sing`(*phonemes*, ***, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Sing the phonemes contained in the string `phonemes`. Changing the pitch and duration of the note is described below. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice.

11.2 Punctuation

Punctuation is used to alter the delivery of speech. The synthesiser understands four punctuation marks: hyphen, comma, full-stop and question mark.

The hyphen (`-`) marks clause boundaries by inserting a short pause in the speech.

The comma (`,`) marks phrase boundaries and inserts a pause of approximately double that of the hyphen.

The full-stop (`.`) and question mark (`?`) end sentences.

The full-stop inserts a pause and causes the pitch to fall.

The question mark also inserts a pause but causes the pitch to rise. This works well with yes/no questions such as, „are we home yet?" rather than more complex questions such as „why are we going home?". In the latter case, use a full-stop.

11.3 Timbre

The timbre of a sound is the quality of the sound. It's the difference between the voice of a DALEK and the voice of a human (for example). To control the timbre change the numeric settings of the `pitch`, `speed`, `mouth` and `throat` arguments.

The pitch (how high or low the voice sounds) and speed (how quickly the speech is delivered) settings are rather obvious and generally fall into the following categories:

Pitch:

- 0-20 impractical
- 20-30 very high
- 30-40 high

- 40-50 high normal
- 50-70 normal
- 70-80 low normal
- 80-90 low
- 90-255 very low

(The default is 64)

Speed:

- 0-20 impractical
- 20-40 very fast
- 40-60 fast
- 60-70 fast conversational
- 70-75 normal conversational
- 75-90 narrative
- 90-100 slow
- 100-225 very slow

(The default is 72)

The mouth and throat values are a little harder to explain and the following descriptions are based upon our aural impressions of speech produced as the value of each setting is changed.

For mouth, the lower the number the more it sounds like the speaker is talking without moving their lips. In contrast, higher numbers (up to 255) make it sound like the speech is enunciated with exagerated mouth movement.

For throat, the lower the number the more relaxed the speaker sounds. In contrast, the higher the number, the more tense the tone of voice becomes.

The important thing is to experiment and adjust the settings until you get the effect you desire.

To get you started here are some examples:

```
speech.say("I am a little robot",  speed=92, pitch=60, throat=190, mouth=190)
speech.say("I am an elf", speed=72, pitch=64, throat=110, mouth=160)
speech.say("I am a news presenter", speed=82, pitch=72, throat=110, mouth=105)
speech.say("I am an old lady", speed=82, pitch=32, throat=145, mouth=145)
speech.say("I am E.T.", speed=100, pitch=64, throat=150, mouth=200)
speech.say("I am a DALEK - EXTERMINATE", speed=120, pitch=100, throat=100, mouth=200)
```

11.4 Phonemes

The `say` function makes it easy to produce speech - but often it's not accurate. To make sure the speech synthesiser pronounces things *exactly* how you'd like, you need to use phonemes: the smallest perceptually distinct units of sound that can be used to distinguish different words. Essentially, they are the building-block sounds of speech.

The `pronounce` function takes a string containing a simplified and readable version of the International Phonetic Alphabet and optional annotations to indicate inflection and emphasis.

The advantage of using phonemes is that you don't have to know how to spell! Rather, you only have to know how to say the word in order to spell it phonetically.

The table below lists the phonemes understood by the synthesiser.

Informacja: The table contains the phoneme as characters, and an example word. The example words have the sound of the phoneme (in parenthesis), but not necessarily the same letters.

Often overlooked: the symbol for the „H" sound is `/H`. A glottal stop is a forced stoppage of sound.

```
SIMPLE VOWELS                          VOICED CONSONANTS
IY           f(ee)t                    R         (r)ed
IH           p(i)n                     L         a(ll)ow
EH           b(e)g                     W         a(w)ay
AE           S(a)m                     W         (wh)ale
AA           p(o)t                     Y         (y)ou
AH           b(u)dget                  M         (S)am
AO           t(al)k                    N         ma(n)
OH           c(o)ne                    NX        so(ng)
UH           b(oo)k                    B         (b)ad
UX           l(oo)t                    D         (d)og
ER           b(ir)d                    G         a(g)ain
AX           gall(o)n                  J         (j)u(dg)e
IX           dig(i)t                   Z         (z)oo
                                       ZH        plea(s)ure
DIPHTHONGS                             V         se(v)en
EY           m(a)de                    DH        (th)en
AY           h(igh)
OY           b(oy)
AW           h(ow)                     UNVOICED CONSONANTS
OW           sl(ow)                    S          (S)am
UW           cr(ew)                    SH         fi(sh)
                                       F          (f)ish
                                       TH         (th)in
SPECIAL PHONEMES                       P          (p)oke
UL           sett(le) (=AXL)           T          (t)alk
UM           astron(om)y (=AXM)        K          (c)ake
UN           functi(on) (=AXN)         CH         spee(ch)
Q            kitt-en (glottal stop)    /H         a(h)ead
```

The following non-standard symbols are also available to the user:

```
YX           diphthong ending (weaker version of Y)
WX           diphthong ending (weaker version of W)
RX           R after a vowel (smooth version of R)
LX           L after a vowel (smooth version of L)
/X           H before a non-front vowel or consonant - as in (wh)o
DX           T as in pi(t)y (weaker version of T)
```

Here are some seldom used phoneme combinations (and suggested alternatives):

```
PHONEME        YOU PROBABLY WANT:     UNLESS IT SPLITS SYLLABLES LIKE:
COMBINATION
GS             GZ e.g. ba(gs)         bu(gs)pray
BS             BZ e.g. slo(bz)        o(bsc)ene
DS             DZ e.g. su(ds)         Hu(ds)son
PZ             PS e.g. sla(ps)        -----
TZ             TS e.g. cur(ts)y       -----
KZ             KS e.g. fi(x)          -----
NG             NXG e.g. singing       i(ng)rate
```

```
NK              NXK e.g. bank           Su(nk)ist
```

If you use anything other than the phonemes described above, a `ValueError` exception will be raised. Pass in the phonemes as a string like this:

```
speech.pronounce("/HEHLOW")   # "Hello"
```

The phonemes are classified into two broad groups: vowels and consonants.

Vowels are further subdivided into simple vowels and diphthongs. Simple vowels don't change their sound as you say them whereas diphthongs start with one sound and end with another. For example, when you say the word „oil" the „oi" vowel starts with an „oh" sound but changes to an „ee" sound.

Consonants are also subdivided into two groups: voiced and unvoiced. Voiced consonants require the speaker to use their vocal chords to produce the sound. For example, consonants like „L", „N" and „Z" are voiced. Unvoiced consonants are produced by rushing air, such as „P", „T" and „SH".

Once you get used to it, the phoneme system is easy. To begin with some spellings may seem tricky (for example, „adventure" has a „CH" in it) but the rule is to write what you say, not what you spell. Experimentation is the best way to resolve problematic words.

It's also important that speech sounds natural and understandable. To help with improving the quality of spoken output it's often good to use the built-in stress system to add inflection or emphasis.

There are eight stress markers indicated by the numbers `1 - 8`. Simply insert the required number after the vowel to be stressed. For example, the lack of expression of „/HEHLOW" is much improved (and friendlier) when spelled out „/HEH3LOW".

It's also possible to change the meaning of words through the way they are stressed. Consider the phrase „Why should I walk to the store?". It could be pronounced in several different ways:

```
# You need a reason to do it.
speech.pronounce("WAY2 SHUH7D AY WAO5K TUX DHAH STOH5R.")
# You are reluctant to go.
speech.pronounce("WAY7 SHUH2D AY WAO7K TUX DHAH STOH5R.")
# You want someone else to do it.
speech.pronounce("WAY5 SHUH7D AY2 WAO7K TUX DHAH STOHR.")
# You'd rather drive.
speech.pronounce("WAY5 SHUHD AY7 WAO2K TUX7 DHAH STOHR.")
# You want to walk somewhere else.
speech.pronounce("WAY5 SHUHD AY WAO5K TUX DHAH STOH2OH7R.")
```

Put simply, different stresses in the speech create a more expressive tone of voice.

They work by raising or lowering pitch and elongating the associated vowel sound depending on the number you give:

1. very emotional stress
2. very emphatic stress
3. rather strong stress
4. ordinary stress
5. tight stress
6. neutral (no pitch change) stress
7. pitch-dropping stress
8. extreme pitch-dropping stress

The smaller the number, the more extreme the emphasis will be. However, such stress markers will help pronounce difficult words correctly. For example, if a syllable is not enunciated sufficiently, put in a neutral stress marker.

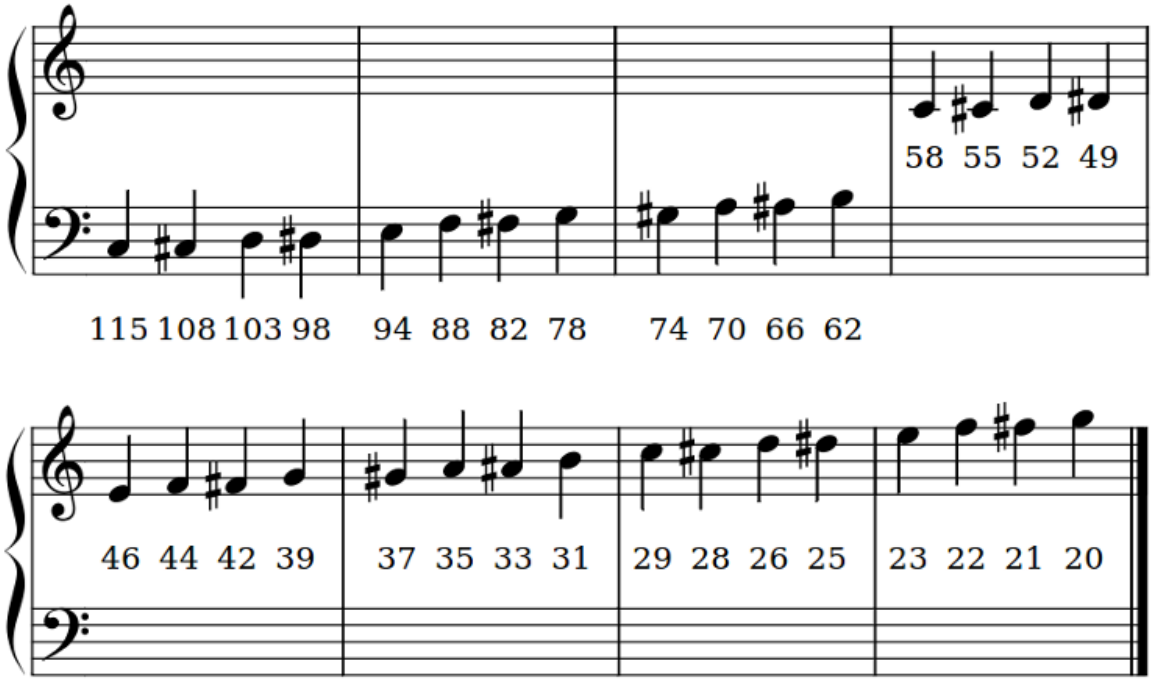
It's also possible to elongate words with stress markers:

```
speech.pronounce("/HEH5EH4EH3EH2EH2EH3EH4EH5EHLP.")
```

11.5 Singing

It's possible to make MicroPython sing phonemes.

This is done by annotating a pitch related number onto a phoneme. The lower the number, the higher the pitch. Numbers roughly translate into musical notes as shown in the diagram below:



Annotations work by pre-pending a hash (#) sign and the pitch number in front of the phoneme. The pitch will remain the same until a new annotation is given. For example, make MicroPython sing a scale like this:

```
solfa = [
    "#115DOWWWWWW",    # Doh
    "#103REYYYYYY",    # Re
    "#94MIYYYYYY",     # Mi
    "#88FAOAOAOAOR",   # Fa
    "#78SOHWWWWW",     # Soh
    "#70LAOAOAOAOR",   # La
    "#62TIYYYYYY",     # Ti
    "#58DOWWWWWW",     # Doh
]
song = ''.join(solfa)
speech.sing(song, speed=100)
```

In order to sing a note for a certain duration extend the note by repeating vowel or voiced consonant phonemes (as demonstrated in the example above). Beware diphthongs - to extend them you need to break them into their component parts. For example, „OY" can be extended with „OHOHIYIYIY".

Experimentation, listening carefully and adjusting is the only sure way to work out how many times to repeat a phoneme so the note lasts for the desired duration.

11.6 How Does it Work?

The original manual explains it well:

> First, instead of recording the actual speech waveform, we only store the frequency spectrums. By doing this, we save memory and pick up other advantages. Second, we [...] store some data about timing. These are numbers pertaining to the duration of each phoneme under different circumstances, and also some data on transition times so we can know how to blend a phoneme into its neighbors. Third, we devise a system of rules to deal with all this data and, much to our amazement, our computer is babbling in no time.
>
> —S.A.M. owner's manual.

The output is piped through the functions provided by the `audio` module and, hey presto, we have a talking micro:bit.

11.7 Example

```
import speech
from microbit import sleep

# The say method attempts to convert English into phonemes.
speech.say("I can sing!")
sleep(1000)
speech.say("Listen to me!")
sleep(1000)

# Clearing the throat requires the use of phonemes. Changing
# the pitch and speed also helps create the right effect.
speech.pronounce("AEAE/HAEMM", pitch=200, speed=100)  # Ahem
sleep(1000)

# Singing requires a phoneme with an annotated pitch for each syllable.
solfa = [
    "#115DOWWWWWW",   # Doh
    "#103REYYYYYY",   # Re
    "#94MIYYYYYY",    # Mi
    "#88FAOAOAOAOR",  # Fa
    "#78SOHWWWWW",    # Soh
    "#70LAOAOAOAOR",  # La
    "#62TIYYYYYY",    # Ti
    "#58DOWWWWWW",    # Doh
]

# Sing the scale ascending in pitch.
song = ''.join(solfa)
speech.sing(song, speed=100)
# Reverse the list of syllables.
solfa.reverse()
```

```
song = ''.join(solfa)
# Sing the scale descending in pitch.
speech.sing(song, speed=100)
```

ROZDZIAŁ 12

Installation

This section will help you set up the tools and programs needed for developing programs and firmware to flash to the BBC micro:bit using MicroPython.

12.1 Dependencies

12.2 Development Environment

You will need:

- git
- yotta

Depending on your operating system, the installation instructions vary. Use the installation scenario that best suits your system.

Yotta will require an ARM mbed account. It will walk you through signing up if you are not registered.

12.3 Installation Scenarios

12.3.1 Windows

When installing Yotta, make sure you have these components ticked to install.

- python
- gcc
- cMake
- ninja
- Yotta
- git-scm
- mbed serial driver

12.3.2 OS X

12.3.3 Linux

These steps will cover the basic flavors of Linux and working with the micro:bit and MicroPython. See also the specific sections for Raspberry Pi, Debian/Ubuntu, and Red Hat Fedora/Centos.

Debian and Ubuntu

```
sudo add-apt-repository -y ppa:team-gcc-arm-embedded
sudo add-apt-repository -y ppa:pmiller-opensource/ppa
sudo apt-get update
sudo apt-get install cmake ninja-build gcc-arm-none-eabi srecord libssl-dev
pip3 install yotta
```

Red Hat Fedora/CentOS

Raspberry Pi

12.4 Next steps

Congratulations. You have installed your development environment and are ready to begin *flashing firmware* to the micro:bit.

ROZDZIAŁ 13

Flashing Firmware

13.1 Building firmware

Use yotta to build.

Use target bbc-microbit-classic-gcc-nosd:

```
yt target bbc-microbit-classic-gcc-nosd
```

Run yotta update to fetch remote assets:

```
yt up
```

Start the build with either yotta:

```
yt build
```

. . . or use the Makefile:

```
make all
```

The result is a microbit-micropython hex file (i.e. `microbit-micropython.hex`) found in the build/bbc-microbit-classic-gcc-nosd/source from the root of the repository.

The Makefile does some extra preprocessing of the source, which is needed only if you add new interned strings to `qstrdefsport.h`. The Makefile also puts the resulting firmware at build/firmware.hex, and includes some convenience targets.

13.2 Preparing firmware and a Python program

tools/makecombined

hexlify

13.3 Flashing to the micro:bit

Installation Scenarios

ROZDZIAŁ 14

Accessing the REPL

Accessing the REPL on the micro:bit requires:

- Using a serial communication program
- Determining the communication port identifier for the micro:bit
- Establishing communication with the correct settings for your computer

If you are a Windows user you'll need to install the correct drivers. The instructions for which are found here:

https://developer.mbed.org/handbook/Windows-serial-configuration

14.1 Serial communication

To access the REPL, you need to select a program to use for serial communication. Some common options are *picocom* and *screen*. You will need to install program and understand the basics of connecting to a device.

14.2 Determining port

The micro:bit will have a port identifier (tty, usb) that can be used by the computer for communicating. Before connecting to the micro:bit, we must determine the port identifier.

14.3 Establishing communication with the micro:bit

Depending on your operating system, environment, and serial communication program, the settings and commands will vary a bit. Here are some common settings for different systems (please suggest additions that might help others)

Settings

- *OS X*
- *Linux*
- *Debian and Ubuntu*
- *Red Hat Fedora/CentOS*
- *Raspberry Pi*

ROZDZIAŁ 15

Developer FAQ

Informacja: This project is under active development. Please help other developers by adding tips, how-tos, and Q&A to this document. Thanks!

Where do I get a copy of the DAL? A: Ask Nicholas Tollervey for details.

ROZDZIAŁ 16

Contributing

Hey! Many thanks for wanting to improve MicroPython on the micro:bit.

Contributions are welcome without prejudice from *anyone* irrespective of age, gender, religion, race or sexuality. Good quality code and engagement with respect, humour and intelligence wins every time.

- If you're from a background which isn't well-represented in most geeky groups, get involved - *we want to help you make a difference*.
- If you're from a background which *is* well-represented in most geeky groups, get involved - *we want your help making a difference*.
- If you're worried about not being technical enough, get involved - *your fresh perspective will be invaluable*.
- If you think you're an imposter, get involved.
- If your day job isn't code, get involved.
- This isn't a group of experts, just people. Get involved!
- This is a new community, so, get involved.

We expect contributors to follow the Python Software Foundation's Code of Conduct: https://www.python.org/psf/codeofconduct/

Feedback may be given for contributions and, where necessary, changes will be politely requested and discussed with the originating author. Respectful yet robust argument is most welcome.

16.1 Checklist

- Your code should be commented in *plain English* (British spelling).
- If your contribution is for a major block of work and you've not done so already, add yourself to the AUTHORS file following the convention found therein.
- If in doubt, ask a question. The only stupid question is the one that's never asked.
- Have fun!

- genindex
- modindex
- search

Indeks modułów pythona

m

n

o

r

s

Indeks

O

P

R

S

T

U

W