


```
1 v def example(v):  
2     ...
```

example

Zadanie przykładowe nie jest rozwiązane.

```
1 mo.md(f'Zadanie przykładowe {" " if test_example(example) else "nie"} jest rozwiązane.')
```

Komórki mogą również wymagać uzupełnienia wcześniejszych komórek. Jest to używane najczęściej w finalnej komórce składającej wszystkie elementy ćwiczenia w działający program.

Zadanie przykładowe nie jest rozwiązane.

```
1 mo.stop(not test_example(example), mo.md('Zadanie przykładowe nie jest rozwiązane.'))  
2 a # = 3
```

✓ Asynchroniczność

Marimo jest środowiskiem asynchronicznym, w związku z czym w niektórych miejscach mogą pojawić się słowa kluczowe `async` i `await`. W ćwiczeniach często pojawiać się będą dwa asynchroniczne sformułowania.

- `async def` w przeciwieństwie do `def` tworzy funkcję asynchroniczną, opcja ta będzie używana w kilku miejscach do utworzenia funkcji, które mogą wykorzystywać inne funkcje asynchroniczne aby uruchomić funkcje asynchroniczne należy oczekiwać na ich rezultat przy użyciu słowa `await`
- `asyncio.sleep` to asynchroniczny odpowiednik `time.sleep`, który na określony czas uśpi kod, jednak nie usypiając również całego środowiska marimo - będzie używany wszędzie tam, gdzie poza notatnikiem pojawiłoby się `time.sleep`

```
def funkcja_synchroniczna():  
    # nie można wywołać funkcji asynchronicznej ze zwykłej funkcji  
    await asyncio.sleep(1) # BŁĄD  
  
async def funkcja_asynchroniczna():  
    # w laboratoriach asyncio.sleep będzie zaimportowane pod nazwą sleep  
    await sleep(1)  
  
# użycie funkcji zdefiniowanej przez nas również wymaga słowa await  
await funkcja_asynchroniczna()
```

Ostatnia ukryta komórka definiuje większość kodu pomocniczego do zadań, nie należy zmieniać jej zawartości.

```
1 from inspect import signature as _sig
```

Laboratorium 1

Wykorzystanie Raspberry Pi z Sense HAT jako kompasu

Raspberry Pi to mały komputer, posiadający dodatkowo interfejs pozwalający na podłączenie modułów *HAT* spełniających rozmaite funkcje, często zawierających różnego rodzaju sensory. Moduł *Sense HAT* zawiera wyświetlacz kolorowy 8×8, mały joystick, oraz czujniki: żyroskop, akcelerometr, magnetometr, termometr, higrometr, ciśnieniomierz, czujnik naświetlenia.

Raspberry Pi oraz moduły są stosunkowo niedrogie, a pozwalają na projektowanie własnych urządzeń, które mogą służyć jako prototyp preprodukcyjny lub spełniać unikalną funkcję. *Raspberry Pi* jest również pełnoprawnym komputerem używającym systemu Linux, ze względu na swoją wielkość bardzo przenośnym, może więc również służyć jako diagnostyczny komputer lub domowy serwer.

W tym zadaniu do płytki podłączony jest zestaw Sense HAT, dodający możliwości pomiaru orientacji (żyroskop), przyspieszenia (akcelerometr), pola magnetycznego (magnetometr), ciśnienia atmosferycznego (barometr), temperatury (termometr) i wilgotności (higrometr). Zestaw zawiera również matrycę 8×8 pikseli oraz mały joystick.

Aby wykorzystać sensory dostępne w urządzeniu można przeczytać ich dokumentację i użyć ich bezpośrednio, ale istnieją również pomocnicze biblioteki, które upraszczają ich wykorzystanie w kilku popularnych językach programowania - w tym przypadku w języku *Python*. Poniżej znajdują się zadania obrazujące kroki niezbędne do ich wykorzystania przy użyciu tego języka.

Z1. Kalibracja IMU

Aby użyć czujników inercyjnych (**IMU**), do których zalicza się żyroskop, akcelerometr i magnetometr, należy najpierw je skalibrować. Można to zrobić przy użyciu (już zainstalowanej) biblioteki systemowej *octave*.

Otwórz terminal i skopiuj domyślne pliki konfiguracyjne sensora do folderu studenta: `cp -a /usr/share/librtimulib-utils/RTellipsoidFit ~`

Przejdź do skopiowanego folderu: `cd ~/RTellipsoidFit`

W tym folderze znajdują się pliki niezbędne programowi kalibracyjnemu do przetworzenia danych otrzymanych podczas kalibracji na konfigurację prawidłowo dostosowującą dane generowane w trakcie korzystania z czujników. Można dostosować te domyślne wartości dla specjalistycznych potrzeb, jednak w tym przypadku w zupełności wystarczą w niezmienionej formie.

Aby uruchomić program kalibracyjny, użyj komendy: `RTIMULibCal`

Program ten posiada prosty interfejs w języku angielskim. Wybierz opcję `m` i naciśnij dowolny przycisk, aby rozpocząć kalibrację, która odbywa się poprzez poruszanie sensorem (a zatem i płytką *Raspberry Pi*). Spróbuj obracać urządzenie w taki sposób, aby w trakcie kalibracji znalazło się w każdej możliwej orientacji. Kalibracja kończy się również poprzez naciśnięcie dowolnego przycisku.

Wybierz opcję `x` aby wyjść z programu kalibracyjnego.

Domyślnie plik konfiguracyjny jest zapisywany w lokalnym folderze użytkownika, jednak aby używać prawidłowej kalibracji we wszystkich programach, przenieś go w miejsce konfiguracji systemowej: `sudo mv ~/.config/sense_hat/RTIMULib.ini /etc`

Czujniki IMU potrzebują chwili, aby uruchomić się i skalibrować po uruchomieniu systemu. Po następnym kroku uruchamiającym system ponownie, upewnij się, że urządzenie przez 6-10 sekund po uruchomieniu (~30 sekund łącznie) nie poruszy się, aby uzyskać najdokładniejsze rezultaty.

Po tym kroku należy zrestartować programy używające kalibracji, w tym również środowisko tego laboratorium, więc zapisz zmiany wykonane w tym pliku, a następnie użyj komendy do uruchomienia systemu ponownie: `sudo reboot`

Dzięki prawidłowo skalibrowanym sensorom odczyty będą bardziej odporne na wpływ magnetyzowanych materiałów w środowisku, choć mocne zakłócenia nadal mogą spowodować błędy w odczycie informacji. Staraj się trzymać sensor z dala od urządzeń elektronicznych i elementów ferromagnetycznych w trakcie kalibracji i obsługi czujników.

Wykorzystaj [dokumentację](#) API Sense HAT do prawidłowego wykonania kolejnych kroków.

Z2. Konfiguracja sensorów

Ustaw czujnik inercyjny IMU do wykorzystania żyroskopu i magnetometru, ale nie akcelerometru. Odpowiedzią na zadanie jest uzupełnienie funkcji wykonującej tę czynność. Obiekt czujnika jest dostępny jako zmienna o nazwie `sense`.

```
1 def set_imu():
2     ...
```

set_imu

Zadanie Z2. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z2. {" " if test_z2(set_imu) else "nie"} jest rozwiązane.')
```

✓ Z3. Zbieranie danych

Znajdź odpowiednią funkcję w dokumentacji, aby uzyskać wartość orientacji czujnika IMU jako radiany. Wyłuskaj z rezultatu funkcji wartość `yaw`, czyli obrót wobec osi pionowej.

```
1 def get_yaw():
2     ...
```

get_yaw

Zadanie Z3. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z3. {" " if test_z3(get_yaw) else "nie"} jest rozwiązane.')
```

Z4. Wyświetl wynik na wyświetlaczu Sense. Użyj funkcji `calculate_leds(yaw_radians: float) -> list[list[int]]`, która jest już zdefiniowana.

Dostępny jako parametr jest również wynik poprzedniego zadania.

```
1 def draw_compass(yaw):
2     ...
```

draw_compass

Zadanie Z4. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z4. {" " if test_z4(draw_compass) else "nie"} jest rozwiązane.')
```

Uruchom

```
1 stp = mo.ui.run_button(label='Uruchom')
```

Zadanie 2 nie jest rozwiązane.

```
1 mo.stop(not test_z2(set_imu), mo.md('Zadanie 2 nie jest rozwiązane.'))
2 mo.stop(not test_z3(get_yaw), mo.md('Zadanie 3 nie jest rozwiązane.'))
3 mo.stop(not test_z4(draw_compass), mo.md('Zadanie 4 nie jest rozwiązane.'))
4 mo.stop(not stp.value, mo.md('Wciśnij uruchom'))
5
6 set_imu()
7 while True:
8     draw_compass(get_yaw())
9     await sleep(0.1)
```

Z5. Jakie czynniki mogą mieć wpływ na dokładność otrzymanych wyników? Poszukaj źródeł, przeprowadź dyskusję i odpowiedz na pytanie.

```
1 #import sense_hat as _sense_hat
```

Pomiar odległości przy użyciu zestawu Grove

Moduły



✓ Z2. Identyfikacja sensora

Wpisz **numer** portu, do którego podłączony jest czujnik ultradźwiękowy. Jest to liczba umieszczona na płytce nad portem, obok litery D.

```
1 Sonar_port = ...
```

Zadanie Z2. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z2. {" " if test_z2(Sonar_port) else "nie"} jest rozwiązane.')
```

✓ Z3.

Uzupełnij zmienne tworzące instancje klas, podając odpowiednie parametry do ich konstruktorów. Pozwoli to na wykorzystanie tych urządzeń w pozostałych zadaniach.

Sygnatury konstruktorów:

GroveUltrasonicRanger(port: int)

GroveDisplay(adres: int)

```
1 sonar = ...
2 display = ...
```

Zadanie Z3. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z3. {" " if test_z3(sonar, display) else "nie"} jest rozwiązane.')
```

✓ Z4. Przetwarzanie danych

Sensor ultradźwiękowy zwraca jedynie czas od wystąpienia sygnału do otrzymania echa. Uzupełnij funkcję przyjmującą czas w mikrosekundach, tak, aby zwracała odległość w centymetrach. Wykorzystaj znajomość praw fizyki (prędkość dźwięku). Pamiętaj, że fala dźwiękowa dociera do celu, odbija się, i wraca z powrotem do sensora.

```
1 ✓ def time_to_dist(t: float) -> float:
2     return t
```

time_to_dist

Zadanie Z4. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z4. {" " if test_z4(time_to_dist) else "nie"} jest rozwiązane.')
```

✓ Z5. Wyświetlanie informacji

Napisz funkcję wyświetlającą wynik pomiaru odległości na panelu LCD. Wyświetl również poprzednią wartość w drugiej linii.

Podpowiedzi:

Funkcja **display.setCursor** wyznacza miejsce, gdzie umieszczona zostanie treść wiadomości.

Wartość zmiennych powinna zajmować zawsze tyle samo pól.

```
1 ✓ def show_measurement(val: float, prev: float):
2     ...
```

show_measurement

Zadanie Z5. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z5. {" " if test_z5(show_measurement) else "nie"} jest rozwiązane.')
```

✓ Z6. Pętla główna

Napisz pętlę zbierającą dane z sensora i wyświetlającą je co 1 sekundę. Pamiętaj o wyświetlaniu poprzedniej wartości. Użyj asynchronicznej funkcji `uśpienia` (w danych pomocniczych).

To zadanie jest sprawdzane przez prowadzącego.

Dane pomocnicze:

`sonar` - instancja klasy zbierającej dane z sensora ultradźwiękowego

`time_to_dist(t: float) -> float` - konwertuje czas w mikrosekundach na centymetry

`show_measurement(new: float, prev: float)` - wyświetla dane na ekranie

`await sleep(1)` - oczekuje przez 1 sekundę

Uruchom

```
1 | stp = mo.ui.run_button(label='Uruchom')
```

Zadanie Z1. nie jest rozwiązane

```
1 | mo.stop(not test_z1(LCD_address), mo.md('Zadanie Z1. nie jest rozwiązane'))
2 | mo.stop(not test_z2(Sonar_port), mo.md('Zadanie Z2. nie jest rozwiązane'))
3 | mo.stop(not test_z3(sonar, display), mo.md('Zadanie Z3. nie jest rozwiązane'))
4 | mo.stop(not test_z4(time_to_dist), mo.md('Zadanie Z4. nie jest rozwiązane'))
5 | mo.stop(not test_z5(show_measurement), mo.md('Zadanie Z5. nie jest rozwiązane'))
6 | mo.stop(not stp.value, mo.md('Wciśnij przycisk uruchom'))
7 |
8 | # uzupełnij kod poniżej.
9 | prv = 0
10 | while True:
11 |     cur = time_to_dist(sonar.get_time())
12 |     show_measurement(cur, prv)
13 |     prv = cur
14 |     await sleep(1)
```

✓ Z7. Doksztalcanie

Zapoznaj się z poniższym kodem, który pokazuje jak wykorzystuje się podłączone urządzenia - definiuje pomocnicze klasy wykorzystane w zadaniach.

```
1 | from grove.display.base import Display, TYPE_CHAR
2 | from time import time, sleep as tsleep
3 | from grove.gpio import GPIO
4 | from grove.i2c import Bus
5 |
6 |
7 | # Funkcja usypiająca na x mikrosekund.
8 | _usleep = lambda x: tsleep(x / 1000000)
9 |
10 |
11 | class GroveUltrasonicRanger:
12 |     TIMEOUT_SEND = 1000
13 |     TIMEOUT_RECV = 10000
14 |
15 |     def __init__(self, port: int):
16 |         # Sprawdzenie, czy wybrany jest port cyfrowy.
17 |         if port not in (5, 16, 18, 22, 24, 26):
18 |             raise ValueError(f'Nieprawidłowy port: {port}')
19 |         self._pin = port
20 |         self.dio = GPIO(port)
21 |
22 |     def _get_time(self) -> float | None:
23 |         # Wysłanie pulsu ultradźwiękowego.
24 |         self.dio.dir(GPIO.OUT)
25 |         self.dio.write(0)
26 |         _usleep(2)
27 |         self.dio.write(1)
28 |         _usleep(10)
29 |         self.dio.write(0)
30 |
31 |         self.dio.dir(GPIO.IN)
32 |
33 |         # Pomiar czasu referencyjnego.
34 |         t0 = time() * 10**6
```



```

35
36     # Odebranie pulsu wysyłającego.
37     count = 0
38     while count < self.TIMEOUT_SEND:
39         if self.dio.read():
40             break
41         count += 1
42     if count >= self.TIMEOUT_SEND:
43         return None
44     t1 = time() * 10**6
45
46     # Test, czy puls został wysłany w odpowiednim czasie od zapytania.
47     if t1 - t0 > 530:
48         return None
49
50     # Odebranie pulsu zwrotnego.
51     count = 0
52     while count < self.TIMEOUT_RECV:
53         if not self.dio.read():
54             break
55         count += 1
56     if count >= self.TIMEOUT_RECV:
57         return None
58     t2 = time() * 10**6
59
60     return t2 - t1
61
62     def get_time(self) -> float:
63         while True:
64             t = self._get_time()
65             if t:
66                 return t
67
68
69     class GroveDisplay(Display):
70         def __init__(self, address: int):
71             self._bus = Bus()
72             self._addr = address
73             if self._bus.write_byte(address, 0):
74                 raise ValueError(f'Sprawdź, czy LCD ({address}) jest podłączony')
75
76             # Konfiguracja wyświetlacza.
77             self._command(0x02)
78             tsleep(0.1)
79             self._command(0x08 | 0x04)
80             self._command(0x28)
81
82             @property
83             def name(self) -> str:
84                 return 'JHD1802'
85
86             def type(self) -> int:
87                 return TYPE_CHAR
88
89             def size(self) -> tuple[int, int]:
90                 return 2, 16 # rows, columns
91
92             def clear(self):
93                 self._command(0x01)
94
95             def home(self):
96                 self._command(0x02)
97                 tsleep(0.2)
98
99             def draw(self, data, bytes):
100                 return NotImplemented # Niedostępne dla tego rodzaju wyświetlacza.
101
102             def setCursor(self, row: int, column: int):
103                 rows, cols = self.size()
104                 if not all([0 <= row < rows, 0 <= column < cols]):
105                     raise ValueError('Poza zakresem wyświetlacza')
106                 self._command(0x80 + (row * 0x40) + (column % 0x10))
107
108             def write(self, msg: str):
109                 for c in msg:
110                     self._bus.write_byte_data(self._addr, 0x40, ord(c))
111
112             def _cursor_on(self, enable: bool):
113                 self._command(0x0E if enable else 0x0C)
114
115             def _command(self, cmd: int):
116                 self._bus.write_byte_data(self._addr, 0x80, cmd)

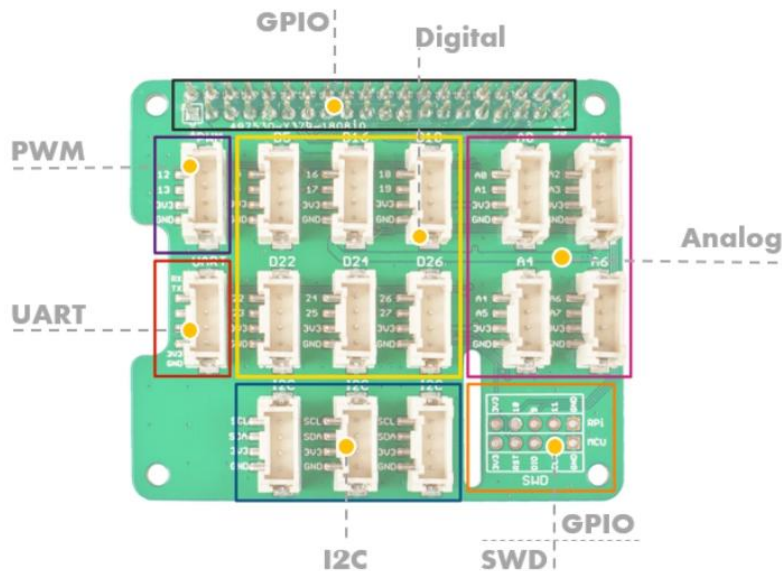
```

```
1 from asyncio import sleep
```


Laboratorium 3

Symulacja alarmu przy użyciu zestawu Grove

Zestaw Grove został opisany w poprzednim laboratorium, co ważne zasilany jest napięciem **3.3V**.



Moduły

W tym ćwiczeniu zostaną wykorzystane moduły czujnika ruchu oraz przycisku z diodą LED.

Mini czujnik ruchu PIR S16-L221D pozwala wykrywać ruch w odległości do 5 metrów (rekomendowane do 2 metrów) z kątem widzenia 110 na 90 stopni. Wykorzystuje światło podczerwone. Korzysta z łącza **cyfrowego**. Współpracuje z 3.3V lub 5V.

Przycisk z diodą komunikuje się przy użyciu łącznika **cyfrowego**. Współpracuje z dowolnym z napięć 3.3V, 5V. Przełącznik jest domyślnie ustawiony w **stan wysoki**, podpięty do **wewnętrznej linii** sygnałowej, a dioda w **stan niski** na **linii zewnętrznej**.

Z1. Identyfikacja portów

Uzupełnij poniższe zmienne numerami identyfikującymi połączenia urządzeń. Numer portu znajduje się na płytce nad portem, obok litery **D**. Numer pinu można odczytać po lewej stronie portu, a następnie użyć informacji o module do prawidłowego przypisania urządzeń.

```
1 sensor_port = ...
2 led_pin = ...
3 btn_pin = ...
```

Zadanie Z1. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z1. {" " if test_z1(sensor_port, led_pin, btn_pin) else "nie"} jest rozwiązane.')
```

Obiekty pomocnicze `motion` i `btn` pozwalają na dostęp do sensorów. Wymagają one sprecyzowania portu podłączenia w interfejsie GPIO.

Zadanie Z1. nie jest rozwiązane.

```
1 mo.stop(not test_z1(sensor_port, led_pin, btn_pin), mo.md('Zadanie Z1. nie jest rozwiązane.'))
2
3 motion = GroveMiniPIR(sensor_port)
4 btn = GroveLEDButton(led_pin)
```

✓ Z2. Wykrycie ruchu

Alarm w momencie wykrycia ruchu powiadamia użytkownika włączając diodę na przycisku. Uzupełnij funkcję, wykorzystując obiekty `motion` i `btn`.

Klasa `GroveMiniPIR` wykorzystuje mechanizm flagi, w momencie wykrycia ruchu flaga zostaje ustawiona na `True`, a odczytanie wartości flagi zmienia jej wartość na `False`. Wykorzystaj ten często używany mechanizm, aby ułatwić rozwiązanie zadania.

Dane pomocnicze:

`motion.flag` - flaga wykrycia ruchu

`btn.led` - stan diody przycisku, można go zmieniać nadając wartość tej zmiennej

```
1 def motion_detected():
2     ...
```

motion_detected

Zadanie Z2. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z2. {" " if test_z2(motion_detected) else "nie"} jest rozwiązane.')
```

✓ Z3. Wyłączenie alarmu

Wciśnięcie przycisku przez użytkownika powinno wyłączyć alarm.

Przycisk nie wykorzystuje mechanizmu flagi, natomiast daje dostęp do obecnego stanu wciśnięcia.

Dane pomocnicze:

`btn.pressed` - stan wciśnięcia przycisku

`btn.led` - stan diody przycisku, można go zmieniać nadając wartość tej zmiennej

```
1 def btn_pressed():
2     ...
```

btn_pressed

Zadanie Z3. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z3. {" " if test_z3(btn.pressed) else "nie"} jest rozwiązane.')
```

✓ Z4. Pętla główna

Napisz pętlę aktualizującą diodę przycisku co 0.1 sekundy. Przetestuj działanie alarmu, następnie zmień opóźnienie na 3 sekundy i przetestuj ponownie.

Jakie różnice istnieją między mechanizmem flagi a bezpośrednim odczytem stanu?

To zadanie jest sprawdzane przez prowadzącego.

Dane pomocnicze:

`motion_detected()` - aktualizacja diody przy wykryciu ruchu

`btn.pressed()` - aktualizacja diody przy wciśnięciu przycisku

`await sleep(delay: float)` - oczekiwanie przez `delay` sekund

Uruchom

```
1 stp = mo.ui.run_button(label='Uruchom')
```

Zadanie Z1. nie jest rozwiązane.

```
1 mo.stop(not test_z1(sensor_port, led_pin, btn_pin), mo.md('Zadanie Z1. nie jest rozwiązane.'))
2 mo.stop(not test_z2(motion_detected), mo.md('Zadanie Z2. nie jest rozwiązane.'))
3 mo.stop(not test_z3(btn.pressed), mo.md('Zadanie Z3. nie jest rozwiązane.'))
4 mo.stop(not stp.value, mo.md('Wciśnij uruchom'))
5
6 # Uzupełnij kod poniżej.
```

▼ Z5. Doszkalanie

Zapoznaj się z kodem definiującym klasy pomocnicze użyte w zadaniach.

```
1 # Ustawienie trybu adresacji portów GPIO.
2 GPIO.setmode(GPIO.BCM)
3
4
5 class GroveMiniPIR:
6     def __init__(self, port: int):
7         self.port = port
8         self._flag = False
9         # Wyczyszczenie poprzednio zdefiniowanych funkcji obsługujących zdarzenia.
10        GPIO.remove_event_detect(port)
11        # Przygotowanie pinu GPIO.
12        GPIO.setup(port, GPIO.IN)
13        GPIO.add_event_detect(port, GPIO.BOTH, self._handle)
14
15    # Funkcja wywoływana dla uzyskania wartości atrybutu flag.
16    @property
17    def flag(self):
18        # Zwrot obecnego stanu flagi.
19        f = bool(self._flag)
20        # Wyczyszczenie flagi podczas odczytu.
21        self._flag = False
22        return f
23
24    def _handle(self, port: int):
25        # Ustawienie flagi przy wykryciu ruchu.
26        if GPIO.input(port):
27            self._flag = True
28
29
30 class GroveLEDButton:
31     def __init__(self, port: int):
32         self.port = port
33         self._led = False
34         self.pressed = False
35         # Wyczyszczenie poprzednio zdefiniowanych funkcji obsługujących zdarzenia.
36         GPIO.remove_event_detect(port + 1)
37         # Przygotowanie pinu GPIO diody.
38         GPIO.setup(port, GPIO.OUT)
39         # Przygotowanie pinu GPIO przycisku.
40         GPIO.setup(port + 1, GPIO.IN, pull_up_down=GPIO.PUD_UP)
41         GPIO.add_event_detect(port + 1, GPIO.BOTH, self._handle, 100)
42
43    @property
44    def led(self):
45        # Odczytanie stanu ze zmiennej.
46        return self._led
47
48    # Funkcja wywoływana dla ustalenia wartości atrybutu led.
49    @led.setter
50    def led(self, on: bool):
51        # Zapis stanu do zmiennej.
52        self._led = on
53        # Ustawienie stanu diody.
54        GPIO.output(self.port, on)
55
56    def _handle(self, pin: int):
57        # Zapis obecnego stanu przycisku.
58        self.pressed = not GPIO.input(pin)
```

```
1 from inspect import signature as _sig
```



Laboratorium 4

Wykorzystanie nakładki Enviro jako stacji kontroli warunków pomieszczenia

Nakładka Enviro jest wyposażona w zestaw czujników, którego celem jest monitorowanie wnętr pomieszczeń. Zestaw składa się z czujników wilgotności, mikrofonu, światła, zbliżeniowego, oraz wyświetlacza LCD. Dzięki pełnej integracji komponentów, nie wymaga skomplikowanej konfiguracji.

Wykorzystany również zostanie darmowy serwis Pub/Sub **ntfy.sh** do wysłania powiadomień o niestandardowych warunkach otoczenia (w tym przypadku zakrycia czujnika zbliżeniowego).

Z1. Aktualizacja sensorów

Sensory wykorzystane w Enviro podzielone są na dwa podzespoły. **BME280 (kod)** zawiera czujniki temperatury, ciśnienia, wilgotności, a **LTR559 (kod)** czujnik zbliżeniowy oraz światła.

Aby pobrać jakiegokolwiek dane najpierw trzeba powiadomić sensor, aby zaktualizował kanały danych, gdzie dane zostaną udostępnione. W związku z brakiem dokumentacji bibliotek wyżej dostępne linki odnoszą się do miejsca w kodzie, gdzie utworzona jest funkcja o takim właśnie działaniu.

Dostępne są obiekty **bme280** i **ltr559** reprezentujące sensory, zgodne z powyższym kodem. Uzupełnij funkcję aktualizującą dane dla obu tych sensorów. Mikrofon **sph0645lm4h_b** nie wymaga aktualizacji.

```
1 def update_sensors():
2     ...
```

update_sensors

Zadanie Z1. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z1. {" " if test_z1(update_sensors) else "nie"} jest rozwiązane.')
```

Z2. Definiowanie metryk informacji

Po aktualizacji danych dostępnych na wyjściach sensorów można zaczynać zbieranie tych dane. W celu ułatwienia tego zadania utworzony jest mechanizm pozwalający na opis informacji jako metryki. Metryki są zbiorem metainformacji opisującym w głębszy sposób zebrane dane, w tym przypadku zawierającym mechanizm zbierania informacji, nazwę wyświetlaną na ekranie, oraz jednostkę w jakiej dane są zbierane.

Format tych metryk to: **Metric(nazwa: str, jednostka: str, funkcja_zbierająca: Callable)**.

Poniżej znajduje się już dodana przykładowa metryka CPU (nie usuwaj jej). W podobny sposób dodaj resztę informacji na temat otoczenia. Dane będą wyświetlane na monitorze płytki, w związku z czym wykorzystaj krótkie nazwy. Wykorzystaj funkcje pomocnicze oraz obiekty **bme280**, **ltr559** oraz **sph0645lm4h_b** do uzupełnienia poniższego kodu. Zbieraj dane w sposób **pasywny** (bez wywoływania dodatkowych aktualizacji).

Podpowiedź: Każda z metryk może wykorzystać funkcje jednolinijkowe lambda (nie będą za długie).

Dostępne są funkcje pomocnicze:

compensate_temperature(float) -> **float** - termometr znajduje się blisko procesora generującego ciepło, funkcja ta próbuje skompensować dane z sensora, aby pomiar temperatury pomieszczenia był mniej niedokładny

proximity_to_mm(int) -> **float** - sensor zbliżeniowy zbiera dane jako wartość bezjednostkowa (2047 = blisko, 0 = daleko), funkcja ta przelicza tą wartość na milimetry (w przybliżeniu)

```
1 metrics = [
2     Metric('CPU', '°C', lambda: get_cpu_temperature()),
3     # dodaj linijki metryk dla temperatury, ciśnienia, wilgotności, bliskości, jasności, głośności SPL
4 ]
```

Zadanie Z2. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z2. {" " if test_z2(metrics) else "nie"} jest rozwiązane.')
```

✓ Z3. Publikowanie danych

Zgodnie z wcześniejszym opisem, dane będą również wysyłane przy użyciu serwisu `ntfy.sh` ([dokumentacja](#)), co pozwoli na otrzymywanie powiadomień. Uzupełnij funkcję, aby wysłać najnowsze dane.

Dane znajdują się w słowniku o nazwie `values`. Kluczami słownika są **nazwy metryk**. Wyślij najnowszą wartość każdej metryki jako typ danych `float`. Wykorzystaj parametr `json` funkcji `requests.post` (już zaimportowanej pod nazwą `post`) i zmiennej `ntfy_topic`.

Wysyłaj wiadomości tylko jeśli użytkownik dotyka sensora zbliżeniowego.

Twoja nazwa tematu (zmienna `ntfy_topic`): `lab-e45f01472278`.

```
1 | mo.md(f'Twoja nazwa tematu (zmienna `ntfy_topic`): `{ntfy_topic}`')
```

```
1 | def send_message():
2 |     ...
```

send_message

Zadanie Z3 nie jest rozwiązane.

```
1 | mo.md(f'Zadanie Z3 {" " if test_z3(send_message) else "nie"} jest rozwiązane.')
```

✓ Z4. Pętla główna

Zbiór, przetwarzanie, wysyłanie i wyświetlanie informacji powinno być statym procesem, który nie odbywa się jedynie jeden raz.

Napisz pętlę główną programu, która odbywa się co sekundę. Jako opóźnienie wykorzystaj `await sleep(1)`.

Funkcje pomocnicze:

`await sleep(1)` - 1 sekunda opóźnienia (w trybie asynchronicznym, który jest używany w tym środowisku laboratoryjnym)

`data_loop(list[Metric], times: int = 5) -> Metric` - jest to nieskończony iterator, która zwraca kolejne metryki po `times` razy, co pomaga w wygodnym wyświetleniu ich na ekranie

`insert_vals(list[Metric])` - funkcja zbierająca i dopisująca wartości metryk do słownika `values`

`display_text(Metric)` - funkcja wyświetlająca na ekranie dane metryki

`update_sensors()` - wcześniej utworzona funkcja uaktualniająca dane sensorów, pamiętaj, że pierwsze wywołanie zwraca nieprawidłowe dane

`send_message()` - wcześniej utworzona funkcja wysyłająca dane

```
1 | async def loop():
2 |     ...
```

loop

Zadanie Z4. nie jest rozwiązane.

```
1 | mo.md(f'Zadanie Z4. {" " if await test_z4(loop) else "nie"} jest rozwiązane.')
```

✓ Z5. Kalibracja poziomu głośności

Mikrofony MEMS zbierają informacje jako poziom głośności w odniesieniu do maksymalnego możliwego do odbioru, wzorując się wartością napięcia odbieranych danych. Nie jest to poziom głośności w decybelach, a tylko wartość mająca znaczenie po kalibracji otoczenia.

Kalibracja wykorzystuje bardzo prosty wzór, do którego niezbędne są dwie wartości.

`dbms_base` - pomiar nieskalibrowanego mikrofonu `spl_base` - pomiar prawdziwej głośności otoczenia w tym samym momencie

Wykorzystaj mikrofon w swoim telefonie do pomiaru głośności pomieszczenia (np. [link](#)) i uzupełnij wartości funkcji, aby skalibrować odczyt płytki.

Stan `spl_base=0`, `dbms_base=0` to pomiar nieskalibrowany, potrzebny do określenia prawidłowej wartości `dbms_base`.

To zadanie nie jest sprawdzane.

```
1 | sph0645lm4h_b = SPH0645LM4H_B(spl_base=0, dbms_base=0)
```

✓ Z6. Odbieranie informacji

Bez tworzenia konta można otworzyć stronę internetową `ntfy.sh` ([tutaj](#)) i zasubskrybować temat (o nazwie użytej wcześniej). Po uruchomieniu aplikacji zaistnienie czujnika zbliżeniowego spowoduje pojawienie się wiadomości w przeglądarce.

Istnieje wiele aplikacji, które są w stanie odbierać wiadomości tego rodzaju, co pozwala wygodnie łączyć je ze sobą bez tworzenia infrastruktury.

To zadanie jest sprawdzane przez prowadzącego.

Uruchom

```
1 stp = mo.ui.run_button(label='Uruchom')
```

Zadanie Z1. nie jest rozwiązane.

```
1 mo.stop(not test_z1(update_sensors), mo.md('Zadanie Z1. nie jest rozwiązane.'))
2 mo.stop(not test_z2(metrics), mo.md('Zadanie Z2. nie jest rozwiązane.'))
3 mo.stop(not test_z3(send_message), mo.md('Zadanie Z3. nie jest rozwiązane.'))
4 mo.stop(not await test_z4(loop), mo.md('Zadanie Z4. nie jest rozwiązane.'))
5 mo.stop(not stp.value, mo.md('Wciśnij uruchom'))
6
7 values.clear()
8 await loop()
```

```
1 from typing import Callable, Generator, Iterator
```

5-skrzyzowanie.py

✓

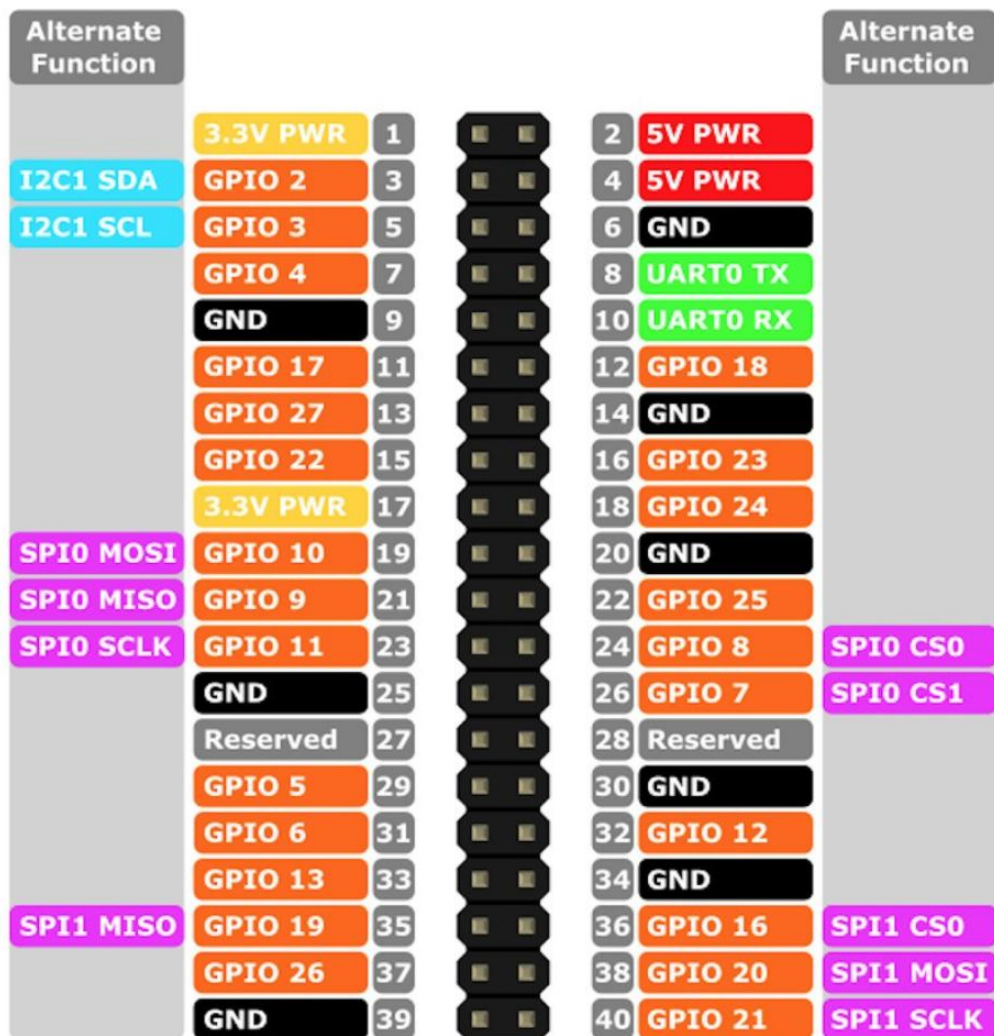
Laboratorium 5

Wykorzystanie nakładki Traffic jako zestawu świateł.

Nakładka ta wykorzystuje interfejs GPIO bezpośrednio, w związku z czym nie posiada oddzielnej biblioteki do kontroli świateł.



Diody (urządzenia **wyjściowe**) są podłączone zgodnie z **opisem** nad nimi. W związku z brakiem zewnętrznej biblioteki należy bezpośrednio ustawić porty GPIO w odpowiedni sposób, aby obsłużyć nakładkę zgodnie z poleceniami. Do prawidłowego wyboru numeru piny GPIO po jego nazwie należy postąpić się poniższym diagramem:



To zadanie wykorzystuje elementy poprzednich oraz zawiera fragment wykonywany w parach.

Z1. Konfiguracja trybu GPIO

Bezpośrednie użycie GPIO wymaga odpowiedniej konfiguracji złącz, czego pierwszym krokiem jest wybór trybu adresacji pinów. Tylko jeden z dostępnych dwóch trybów działa niezależnie od rewizji płytki Raspberry, w związku z czym to on będzie wybrany do tego zadania.

Informacje potrzebne do konfiguracji GPIO można znaleźć w [repozytorium](#). Tryb, który wybieramy ma nazwę `BOARD`. Uzupełnij poniższą funkcję konfiguracją wybierając ten tryb.

```
1 def gpio_config():
2     ...
```

gpio_config

Zadanie Z1. nie jest rozwiązane.

```
1 mo.md(f'Zadanie Z1. {" " if test_z1(gpio_config) else "nie"} jest rozwiązane.')
```

✓ Z2. Identyfikacja portów

Porty w bibliotece przypisane są do liczb, jednak użycie bezpośrednio liczb w kodzie nie jest dobrą praktyką i może prowadzić do błędów. Znajdź odpowiednie wartości i przypisz je do zmiennych, będą one dostępne w kolejnych zadaniach.

```
1 led_red = ...
2 led_yellow = ...
3 led_green = ...
```


Zadanie Z2. nie jest rozwiązane.

```
1 | mo.md(f'Zadanie Z2. {" if test_z2(led_red, led_yellow, led_green) else "nie"} jest rozwiązane.')
```

✓ Z3. Konfiguracja portów

Porty GPIO mogą zostać ustawione w tryb wejściowy lub wyjściowy. Do ich użytkowania jest niezbędna konfiguracja, domyślny jest tryb wejściowy jednak nawet użycie jako wejście **wymaga** ręcznego wyboru trybu w celu eliminacji możliwych pomyłek, jeśli na danej płytce jest utworzonych kilka programów korzystających z interfejsu GPIO.

Uzupełnij poniższą funkcję ustawiającą odpowiednio niezbędne do tego zadania porty. Użyj zmiennych w celu uniknięcia błędów. Nie konfigurować stanu wstępnego.

```
1 | def port_config():  
2 |     ...
```

port_config

Zadanie Z3. nie jest rozwiązane.

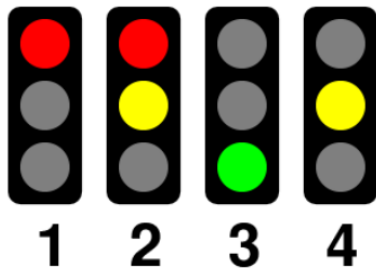
```
1 | mo.md(f'Zadanie Z3. {" if test_z3(port_config) else "nie"} jest rozwiązane.')
```

✓ Z4. System świetlny

Po skonfigurowaniu elementów GPIO mamy dostęp do użytkowania urządzenia. W związku z tematem laboratorium, niezbędne będą funkcje pokazujące odpowiednie sekwencje świetlne w celu zasymulowania działania świateł na skrzyżowaniu.

Uzupełnij funkcje zmieniając wartość wyjściową diód w odpowiednich odstępach czasowych. Czas trwania żółtego światła to standardowo **2 sekundy**.

Wykorzystaj dokumentację GPIO do ustawienia diód na włączone lub wyłączone w odpowiednim momencie. Użyj systemu czterofazowego:



Funkcja `red_to_green` rozpoczyna się w fazie 1, a ma przejść przez fazę 2 i skończyć w fazie 3. Funkcja `green_to_red` rozpoczyna się w fazie 3, a ma przejść przez fazę 4 i skończyć w fazie 1.

Test poprawności działania sprawdza stan świateł w momencie oczekania `asyncio.sleep` oraz na końcu funkcji. Te dwa momenty powinny zgadzać się z kolejnymi fazami systemu czterofazowego. Opóźnienie powinno wynosić standardowe 2 sekundy.

```
1 | async def red_to_green():  
2 |     ...  
3 |  
4 | async def green_to_red():  
5 |     ...
```

Zadanie Z4. nie jest rozwiązane.

```
1 | mo.md(f'Zadanie Z4. {" if await test_z4(red_to_green, green_to_red) else "nie"} jest rozwiązane.')
```

▼ Z5. Wysyłanie wiadomości

Dana płytka ma zamocowane tylko jedno światło. W związku z tym do przetestowania systemu sygnalizacji będzie potrzebna praca w parach, jednak można najpierw skonfigurować i przetestować komunikację przy wykorzystaniu jednego światła. Do komunikacji podobnie jak w poprzednim laboratorium użyty zostanie serwis `ntfy.sh` ([dokumentacja](#)).

Nazwa twojego kanału to: `lab-e45f01472278`. W kodzie dostępna jest zmienna `ntfy_topic`, która ją przechowuje.

W celu zaliczenia zadania usuń znak komentarza (`#`) z poniższej linijki i po uruchomieniu kodu odbierz wiadomość o treści 'zaliczono'. Wiadomość wyślij z terminala przy użyciu `curl`.

```
1 mo.md(f'''
```

```
1 #receive_task(duration=30)
```

▼ Z6. Skrzyżowanie

To zadanie będzie wymagało testowania przy użyciu dwóch płytek. Jedna przyjmie rolę nadrzędnej i będzie wysyłać wiadomości na swój kanał, a druga podrzędnej i będzie słuchać wiadomości **na kanale pierwszej**. Wiadomości powinny informować podrzędną płytkę kiedy wywoływać funkcje `red_to_green` i `green_to_red`.

Domyślny stan diód to światło zielone dla płytki nadrzędnej i czerwone dla płytki podrzędnej.

Funkcja odbierająca **musi** obsługiwać wiadomość **zakończającą pętlę**, a funkcja wysyłająca **nie może** być nieskończona, aby ułatwić sprawdzanie zadania.

Uzupełnij funkcje wykonujące obie z tych ról wspomagając się dokumentacją `ntfy.sh`. Następnie wybierz drugą osobę, z którą przetestujesz odpowiedź. Po przetestowaniu zamieńcie role płytek, przetestujcie drugą część zadania i zgłoście zakończenie ćwiczenia.

Podpowiedź: Obie części wymagają uruchomienia w pętli, która w przypadku płytki podrzędnej będzie stale nasłuchiwać i obsługiwać otrzymane wiadomości, a w przypadku płytki nadrzędnej w odpowiednich momentach wysyłać komunikaty i zmieniać stan własny, aby sygnalizacja była zsynchronizowana. Pamiętaj, że światło zielone ani żółte nie może się pojawić na obu światłach jednocześnie.

```
1 async def sender():
2     ...
3
4 async def receiver():
5     ...
```

Uruchom komunikację

☒ Ta płytka jest nadrzędna

```
1 role = mo.ui.checkbox(label="Ta płytka jest nadrzędna", value=True)
```

Zadanie Z1. nie jest rozwiązane.

```
1 mo.stop(not test_z1(gpio_config), mo.md('Zadanie Z1. nie jest rozwiązane.'))
```

```
1 import unittest.mock as _mock
```