MicroPython

Popular Data Structures

1. Dictionary (`dict`)

- **Purpose**: Stores key-value pairs, similar to a hashmap.
- **Use Case**: Useful for representing structured data, configurations, or storing state information.

Example:

```
config = {
    "wifi_ssid": "MyNetwork",
    "wifi_password": "password123",
    "device_id": "sensor01"
}
```

Access:

```
ssid = config["wifi_ssid"]
```

2. List (`list`)

- **Purpose**: An ordered collection of items (can be of different types).
- **Use Case**: Used to store sequences of data, such as sensor readings, pin states, or configurations.

Example:

```
sensor_readings = [23, 45, 67, 89]
```

Access:
python
Copy code
```
first_reading = sensor_readings[0]
```

-

3. Tuple (`tuple`)

- **Purpose**: An immutable ordered collection of items.
- **Use Case**: Used when you need a constant set of values, like coordinates, fixed configurations, or multiple return values from a function.

Example:
```
coordinates = (10, 20)
```
Access:
```
x, y = coordinates
```

4. Set (`set`)

- **Purpose**: An unordered collection of unique items.
- **Use Case**: Useful for membership tests, eliminating duplicates, or set operations like union and intersection.

Example:
```
active_pins = {5, 12, 16}
```

Access:

```
if 12 in active_pins:
    print("Pin 12 is active")
```

5. String (`str`)

- **Purpose**: A sequence of characters.
- **Use Case**: Commonly used for storing text, messages, or any data that requires manipulation as a string.

Example:

```
device_name = "MicroController01"
```
Access:

```
print(device_name[0:5])  # Outputs "Micro"
```

10. Arrays (`array.array`)

- **Purpose**: Efficient arrays of basic types (like integers or floats).
- **Use Case**: Used for numeric data that requires efficient storage and manipulation.

Example:
```
from array import array
arr = array('i', [1, 2, 3, 4])
```
Access:
```
arr[2] = 5  # Modifies the third element
```

MicroPython syntax

In MicroPython, the structure of a program is similar to standard Python but optimized for microcontrollers. Below is a concise explanation of the structure and syntax:

1. Imports

- Import necessary modules, often specific to the hardware (e.g., `machine`, `utime`, `network`).

```
import machine
import utime
```

2. Configuration

- Set up hardware components like pins, I2C, SPI, etc.

```
led = machine.Pin(2, machine.Pin.OUT)
```

3. Functions

- Define functions for code reusability and clarity.

```
def blink_led():
    led.value(1)
    utime.sleep(0.5)
    led.value(0)
    utime.sleep(0.5)
```

4. Main Loop

- The core of many MicroPython programs, often an infinite loop to continually perform tasks.

```
while True:
    blink_led()
```

5. Conditional Statements

- Use `if`, `elif`, `else` for decision-making.

```
if button.value() == 1:
    led.on()
else:
    led.off()
```

6. Exception Handling

- Handle errors using `try`, `except`.

```
try:
    # code that may cause an error
except Exception as e:
    print(e)
```

7. Comments

- Use # for single-line comments.

```
# This is a comment
```

Example Program:

```
import machine
import utime

led = machine.Pin(2, machine.Pin.OUT)

def blink_led():
    led.value(1)
    utime.sleep(0.5)
    led.value(0)
    utime.sleep(0.5)

while True:
    blink_led()
```

This structure is typical for a MicroPython program running on microcontrollers, focusing on hardware interaction, timing, and continuous operation.

Main

In MicroPython, there isn't a built-in `main` function like in some other programming environments, but you can use the following approaches to structure your code:

1. Top-Level Code Execution

MicroPython scripts execute top-level code immediately when the script is run. You can use this feature to initialize your environment and start the main loop directly:

```
import machine

import utime

# Initialize hardware

led = machine.Pin(2, machine.Pin.OUT)

def blink_led():

    led.value(1)

    utime.sleep(0.5)

    led.value(0)

    utime.sleep(0.5)


# Main loop

while True:

    blink_led()
```

2. Using a Function as an Entry Point

Define a function, commonly named `main()`, and call it at the end of your script. This pattern helps in organizing code and can be useful for testing or modular design:

```
import machine

import utime


def main():
```

```
    led = machine.Pin(2, machine.Pin.OUT)

    def blink_led():
        led.value(1)
        utime.sleep(0.5)
        led.value(0)
        utime.sleep(0.5)

    while True:
        blink_led()

# Call the main function
main()
```

3. Using a `boot.py` and `main.py` Setup

MicroPython supports a dual-script setup where `boot.py` is run on startup, and `main.py` can be used to handle the main application logic. This is useful for initializing settings or hardware in `boot.py` and then running your main logic in `main.py`.

boot.py (Initialization):

```
import machine
# Initialize hardware settings here
# For example, set up Wi-Fi, file system, etc.
```

main.py (Main Application Logic):

```
import machine
import utime

led = machine.Pin(2, machine.Pin.OUT)

def blink_led():
    led.value(1)
    utime.sleep(0.5)
    led.value(0)
    utime.sleep(0.5)

while True:
    blink_led()
```