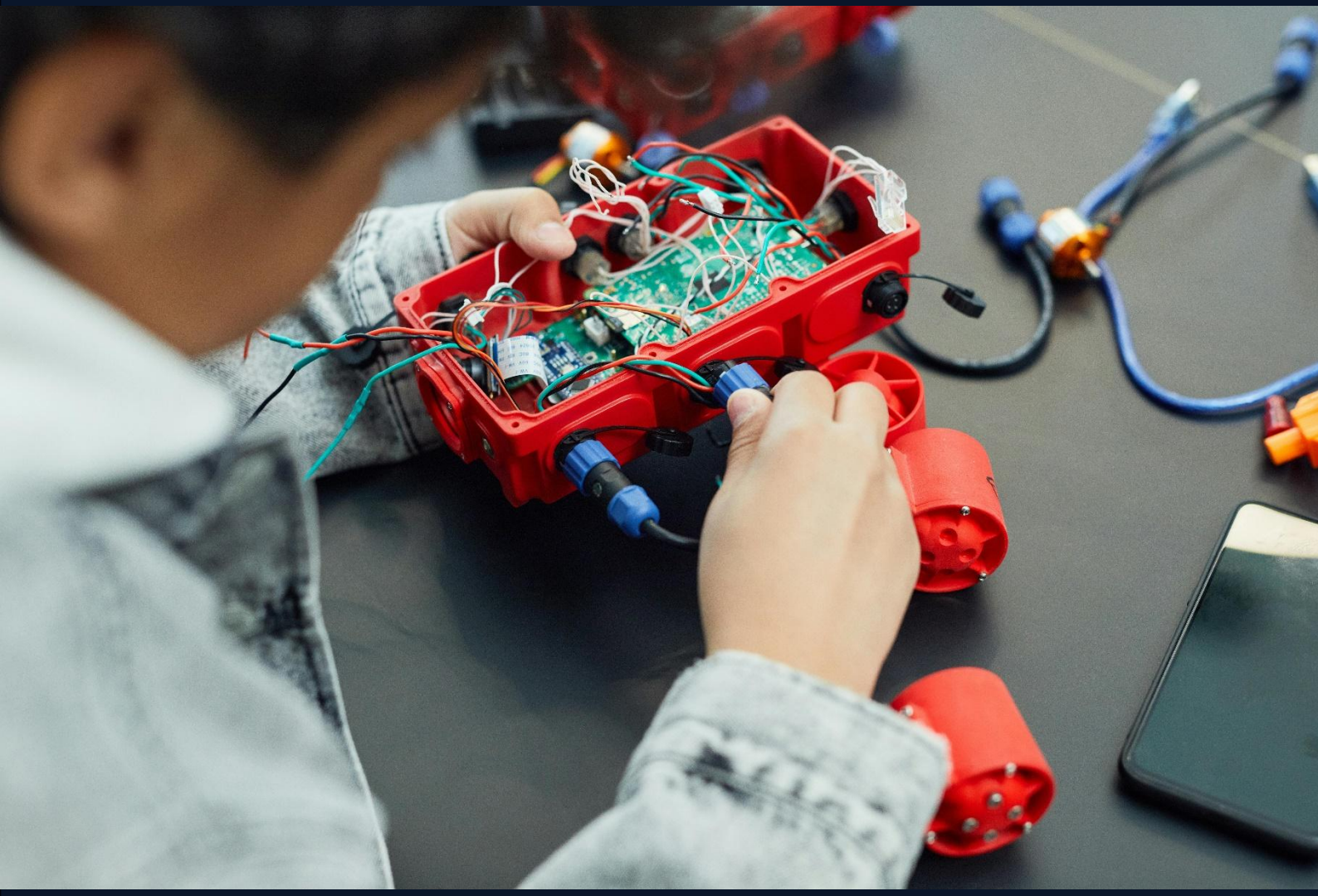
Leming



MOMENTUM-M401

User Manual

13/11/2024

Unleash your creativity, t*he future is yours to build*.

Table of Contents

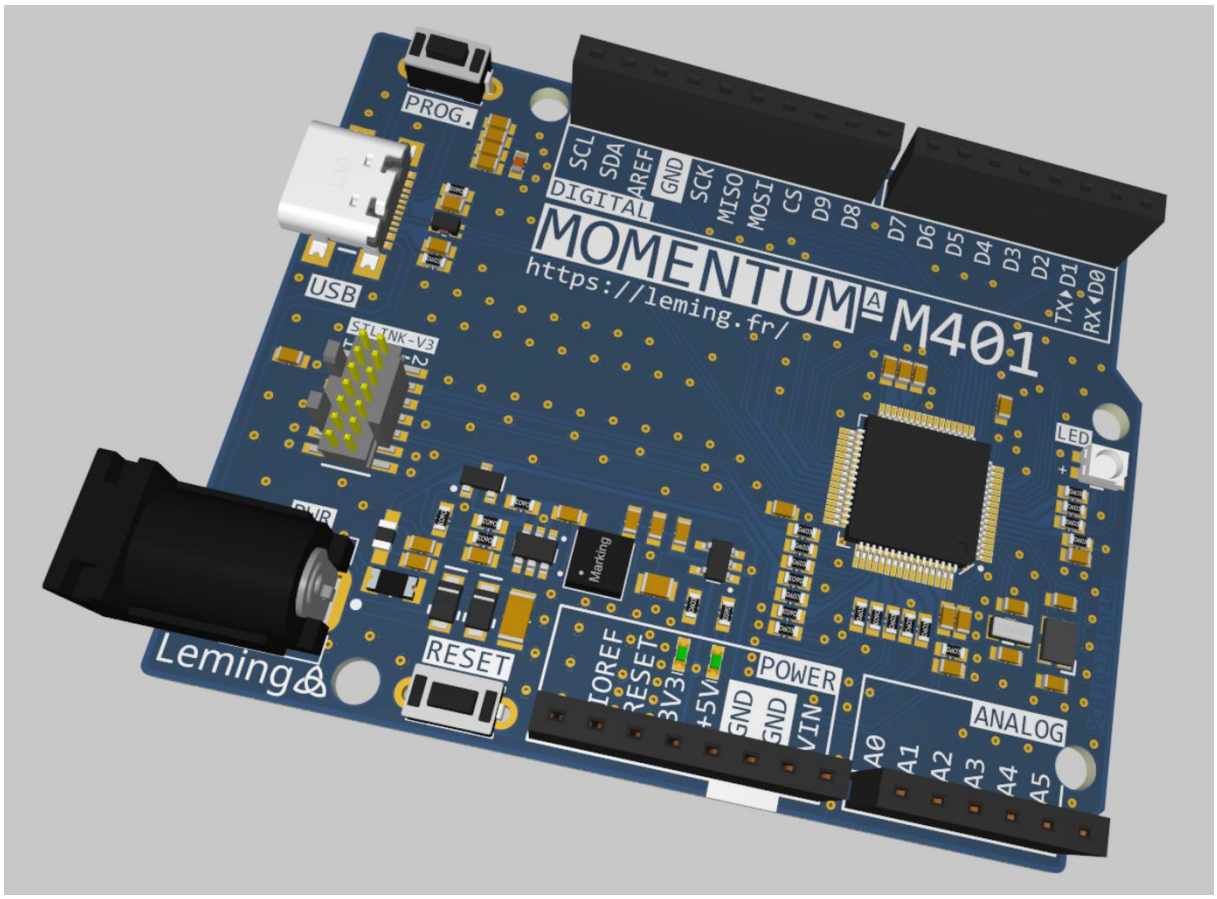
Introduction

Description

The MOMENTUM-M401 board is a versatile and high-performance development platform built around the STM32G474RET6 microcontroller. This microcontroller features a 170 MHz ARM Cortex-M4F core, offering ideal processing power for applications requiring high performance and low power consumption. Compatible with the Arduino ecosystem, it simplifies prototyping and integration with standard shields and peripherals.

Preloaded with MicroPython, the MOMENTUM-M401 enables fast and intuitive development with an accessible and widely adopted programming language. It offers rich connectivity options with UART, SPI, I2C, CAN interfaces, as well as advanced features like PWM capture, ADC/DAC conversion, and motor control.

With its compact dimensions, optimized pinout, and compatibility with a wide range of sensors and actuators, the MOMENTUM-M401 is suited for both technology enthusiasts and professionals seeking a robust and flexible solution for IoT, automation, or robotics projects.



Features

Microcontroller

- STM32G474RET6 : 170 MHz ARM Cortex-M4F core
- Operating voltage : 3.3 V
- Mathematical hardware accelerators
    - CORDIC for trigonometric functions acceleration
    - FMAC (Filter Mathematical Accelerator)
- True random number generator (RNG)
- CRC calculation unit, 96-bit unique ID

Memories

- 512 kB Flash memory
- 128 kB SRAM

Pins

- 14 x Digital pins (GPIO): D0-D13
- 6 x Analog input pins (ADC): A0-A5
- 3 x Analog output pins (DAC): D10, D12, D13
- 12 x PWM pins: D0-D7, D9-D12
- *For more details, refer to the pinout diagram*.

Peripherals

- Calendar Real-Time Clock (RTC) with alarm
- Up to 12bits ADC
- Up to 12bits DAC
- 17 timers

Communication

- USB 2.0 full-speed: USB Type-C connector
- 3 x I2C buses
- 1 x SPI bus
- 1 x FDCAN (TX, RX, requires an external transceiver)
- 1 x UART (TX, RX, CTS, RTS)
- *For more details, refer to the pinout diagram*.

Debug port

- STLINKV3 (14-pin connector)
- JTAG/SWD (4-pin interface)
- Additional Serial Port (VCP)
- Hardware Reset
- Probe detection by the microcontroller

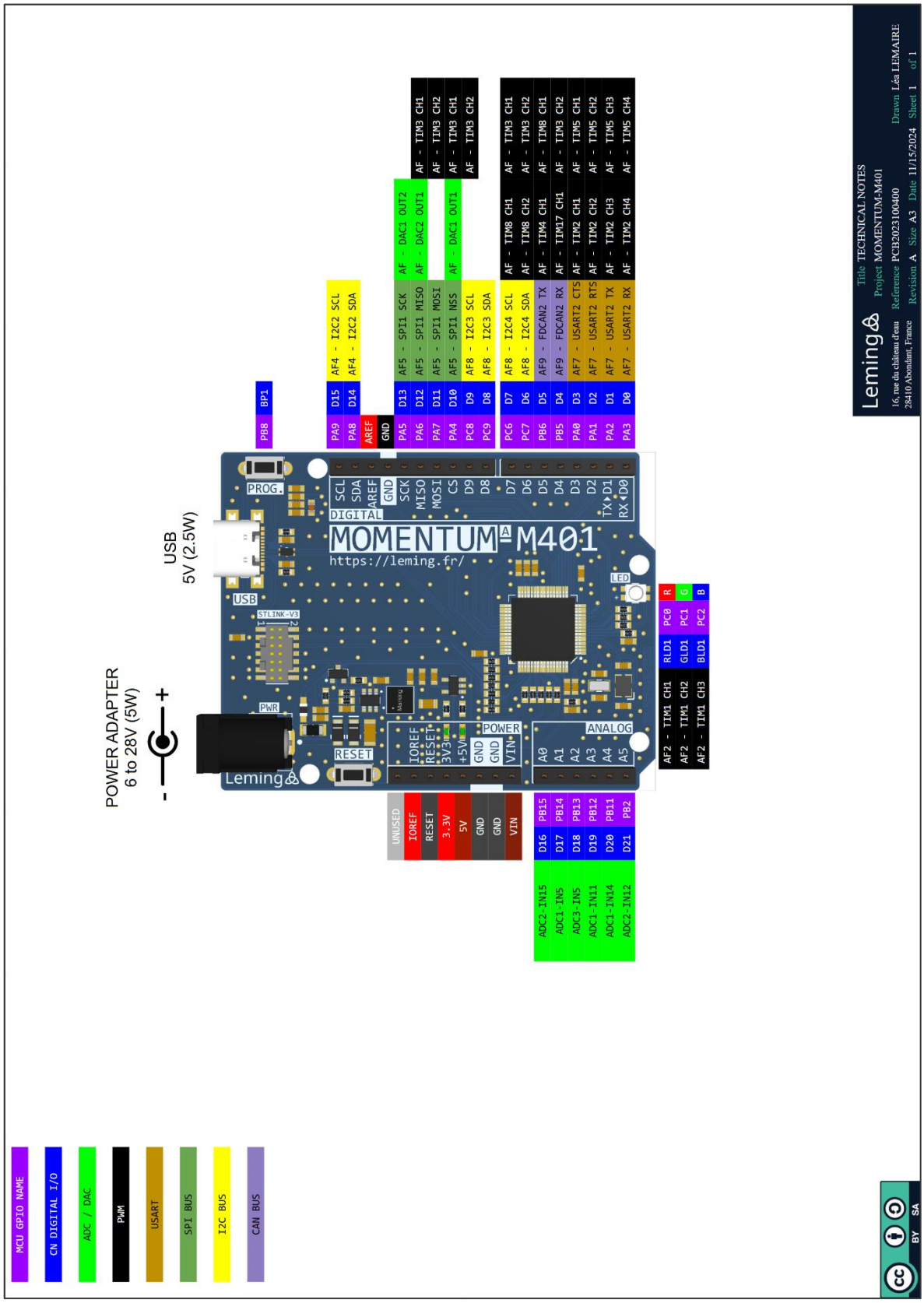
Power

- Power via USB-C® (5V, 500mA max)
- Input voltage range (from VIN or Barrel jack): 6 to 28V

User Interface

- 1 x User button (BP1), Pressing the button at startup initiates ST DFU (Device Firmware Upgrade)
- RGB LED (RLD1, GLD1, BLD1), each color can be individually dimmed for custom brightness control.

Pinout diagram



Operating conditions

Symbol	Description	Min	Typ.	Max	Unit
EXT_PWR	Input voltage from VIN pad / DC jack (5W max.)	6	9	28	V
VBUS	Input voltage from USB connector (2.5W max.)	4.8	5	5.5	V
IOREF	I/O voltage level	3.25	3.3	3.35	V
3.3V	Digital operating voltage (MCU + I/O)	3.25	3.3	3.35	V
	(3.3V) Output current			500	mA
5V	Output voltage	4.8	5	5.2	V
	(5V) Output current from VIN			1	A
	(5V) Output current from VBUS			500	mA
TA	Ambient temperature	-40	25	85	°C

Flash MicroPython Firmware

Flashing the MicroPython firmware on the MOMENTUM-M401 is a straightforward process. Follow these step-by-step instructions to prepare your board and load the firmware using STM32CubeProgrammer.

Installing STM32CubeProgrammer

To flash firmware onto the MOMENTUM-M401, you need to install the STM32CubeProgrammer tool provided by STMicroelectronics. Here's how:

Download the STM32CubeProgrammer

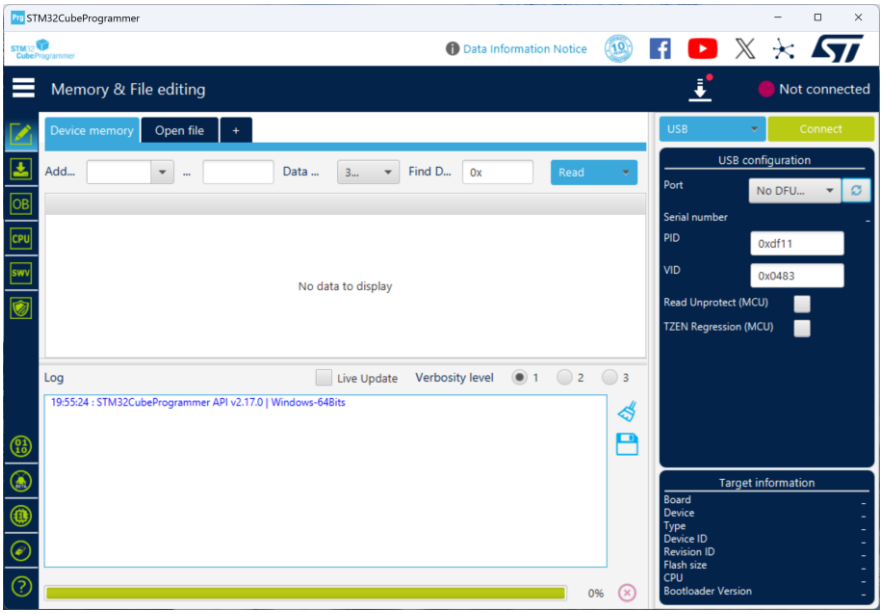
- Visit the official STMicroelectronics website: https://www.st.com
- Search for **STM32CubeProgrammer** in the tools section and download the version compatible with your operating system (Windows, macOS, or Linux).

Install the Software

- Follow the on-screen instructions provided by the installer.
- Ensure you have the necessary permissions and prerequisites (e.g., Java Runtime Environment).

Verify Installation

- Launch STM32CubeProgrammer to ensure it was installed correctly.
- You should see the interface with options to connect to a device, load firmware, and program it.

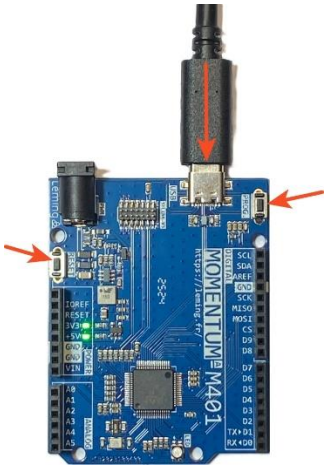


Preparing the MOMENTUM-M401

Before flashing, you must prepare your MOMENTUM-M401 board.

Connect the USB Cable

Use a USB-C cable to connect the MOMENTUM-M401 to your computer.

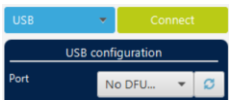


Enter Bootloader Mode

1. Press and hold the **PROG** button on the board.
2. While holding the **PROG** button, press and release the **RESET** button.
3. Release the **PROG** button. The board is now in bootloader mode and ready to receive the firmware.

Flashing the Firmware

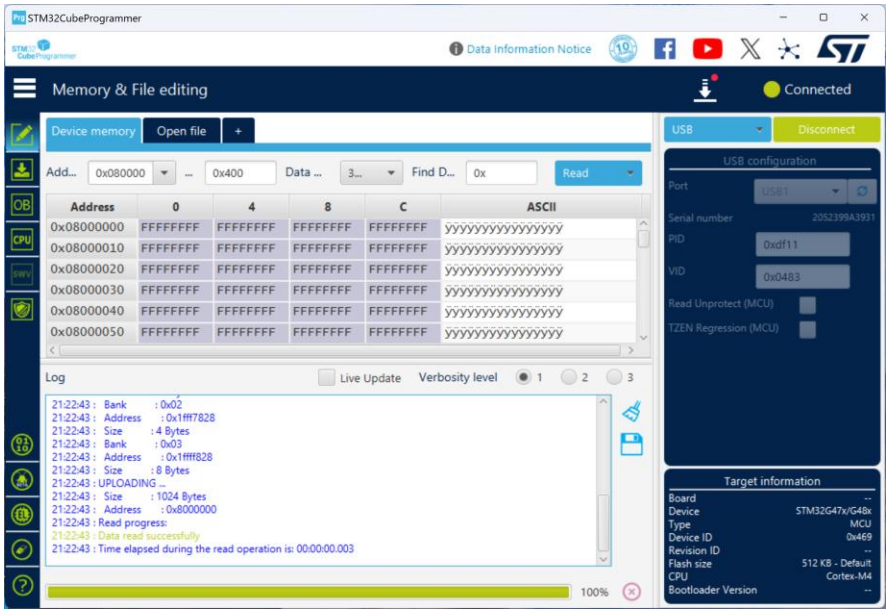
1. Download the zip file containing the MicroPython firmware binaries for the MOMENTUM-M401 from https://leming.fr/fr/microcontroleurs/
2. Launch STM32CubeProgrammer
   Open the STM32CubeProgrammer tool on your computer
3. Connect to the Board:
   - In STM32CubeProgrammer, select the USB connection option.

   - Click on refresh button

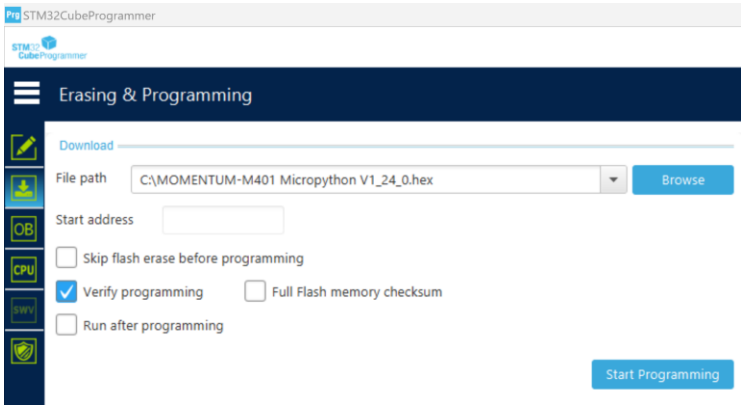
     

- Click the **Connect** button to establish a connection with the MOMENTUM-M401.



4. Load the Firmware:

   - Navigate to the **Erasing & Programming** tab.
   - Click on the **Browse** button and select the downloaded **.hex** file.

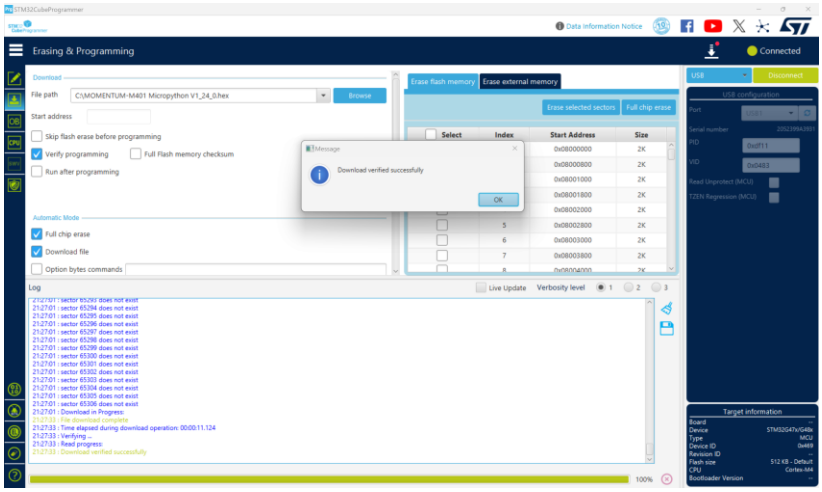


5. Flash the Firmware:

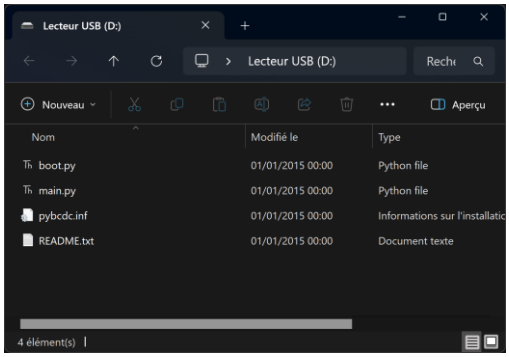
   - Click the **Start Programming** button to begin flashing.
   - Wait for the process to complete. A progress bar will indicate the status.

6. Verify Success:

- After programming is complete, check the log in STM32CubeProgrammer for confirmation of success.



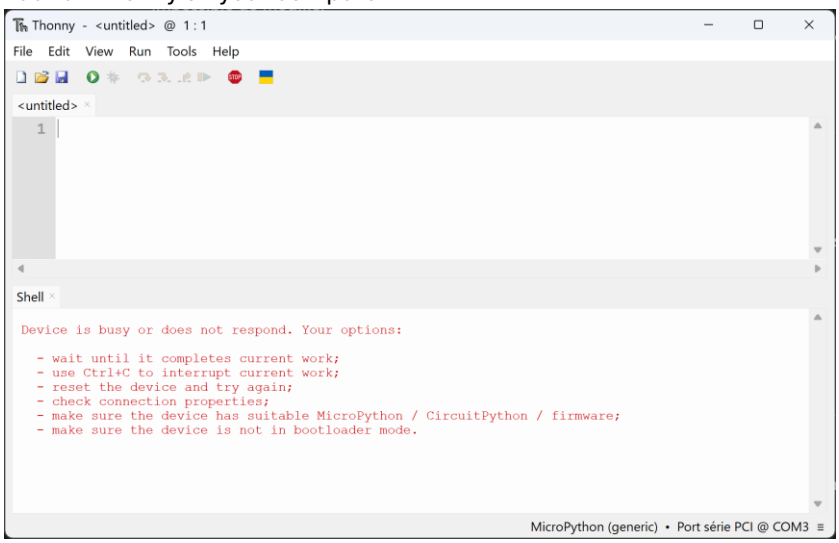
- Disconnect the board and restart it.
- A new USB drive should appear on your computer.



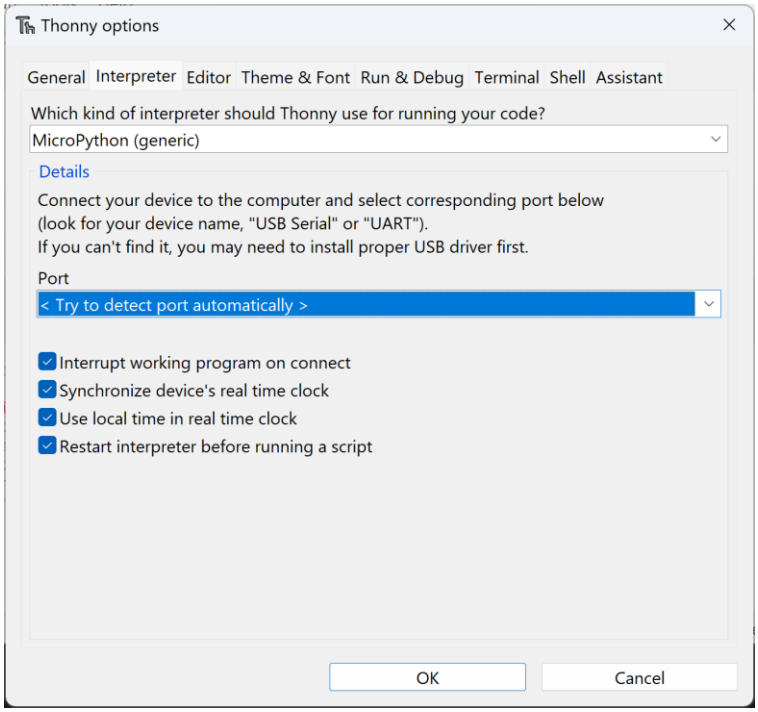
Testing the Firmware

After flashing, the MOMENTUM-M401 should boot with the new MicroPython firmware.

1. Open the Thonny IDE:
    a. Download and install Thonny from https://thonny.org if you don't already have it installed.
    b. Launch Thonny on your computer.



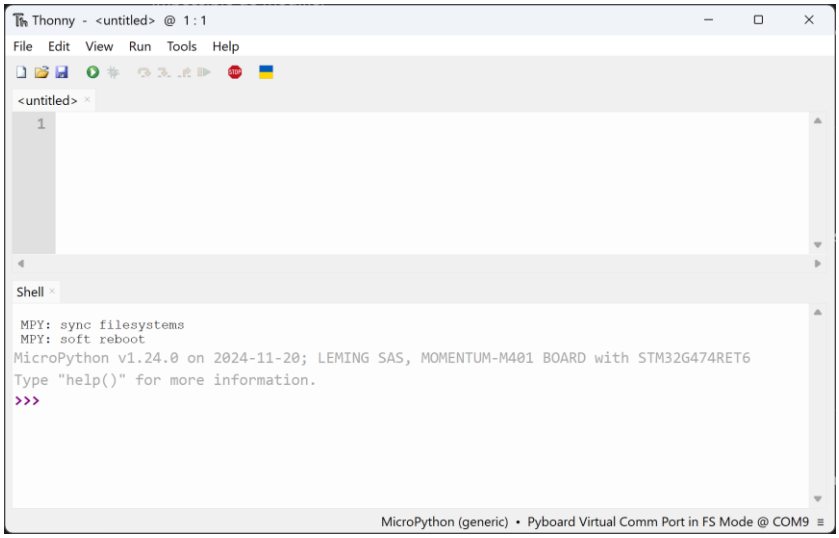
2. Select the MicroPython Interpreter:
    a. Go to **Tools > Options > Interpreter**.
    b. Select **MicroPython (generic)** as the interpreter.
    c. Choose the appropriate COM port for the MOMENTUM-M401, or select **<Try to detect port automatically>**



3. Test the REPL:

    a. Click the **Stop/Restart** button in Thonny to connect to the board.

    b. You should see the MicroPython REPL prompt “**>>>**” appear in the shell.



Congratulations! Your MOMENTUM-M401 is now running MicroPython and ready for development.

Using MicroPython

The MOMENTUM board is fully compatible with MicroPython, allowing you to program the board easily and quickly using an interpreted language. This chapter will guide you through the initial setup, running your first scripts and utilizing the main features of the board.

What is MicroPython ?

MicroPython is a lightweight implementation of the Python programming language designed for microcontrollers. With MicroPython, you can:

- Directly control GPIO pins.
- Communicate with sensors and peripherals via I2C, SPI, UART or CAN.
- Rapidly prototype IoT, home automation and other projects.

The MOMENTUM-M401 comes preloaded with MicroPython, enabling you to get started immediately.

Installing tools

Step 1 - Download and install a MicroPython-compatible IDE:

- Thonny IDE (built-in support for MicroPython), download here: https://thonny.org

Step 2 – Connect the MOMENTUM board to your computer:

- Plug in the board via USB
- The board should be automatically detected as a serial (COM) port.

Step 3 – Access the REPL (Read-Eval-Print-Loop):

- Launch Thonny IDE.
- In the Tools > Options > Interpreter, select MicroPython (Generic) as the interpreter.
- Choose **Try to detect port automatically** in the **Port** field, or choose the serial port corresponding to your board (usually COMx on Windows or /dev/ttyUSBx on Linux/macOS).

Let's start

Open Thonny IDE and connect to your board.

```
TODO add some visual
```

Indentation in Python

What is indentation ?

In Python, indentation means adding spaces at the beginning of a line. It is used to show which lines of code belong together. Unlike other programming languages that use braces ({ }) to define code blocks, Python uses indentation.

Python is designed for readability and simplicity. Proper indentation ensures that the code is both easy to read and executable.

Why do we need indentation ?

Indentation is used to organize your code into blocks. Blocks are groups of lines that work together, like:

- Code inside an if statement
- Code inside a loop (e.g., **for** or **while**)
- Code inside a function

```
Correct indentations
if 7 > 3:
    print("7 is greater than 3")  # This line is indented
    print("This is part of the same block")  # Same block
```

```
Incorrect indentations
if 7 > 3:
print("7 is greater than 3")  # Error: No indentation
```

In the following example, the indentation is technically correct, but consistency is not maintained. The first block uses 2 spaces for indentation, while the second block uses 4 spaces. This inconsistency can lead to confusion and errors, especially in larger projects.

```
if 7 > 3:
  print("7 is greater than 3") # This line is indented
if 5 < 20:
    print("5 is lower than 20") # This line is indented
```

Rule Regarding Indentation:

- Python relies on **consistent indentation** to define blocks of code. It is crucial to use the same number of spaces for all indented lines in a block.
- **PEP 8**, the official style guide for Python, recommends using **4 spaces** per indentation level. Mixing different numbers of spaces or tabs can result in errors and make your code harder to read. Stick to one indentation style throughout your code.

```
example in a loop
for i in range(3):
    print("Iteration:", i)  # Indented inside the loop
print("Loop finished")  # Not part of the loop
```

```
example in a function
def greet(name):
    print(f"Hello, {name}!")  # Indented inside the function

greet("Léa")  # Calls the function
```

Adding comments in Python

In Python, comments are lines in the code that the interpreter ignores. They are used to add explanations, notes, or reminders for yourself or others reading the code. Comments improve readability and help clarify complex or important sections.

Single-line comments

A single-line comment starts with the **#** symbol. Anything written after **#** on the same line is ignored by the interpreter.

```
# This line displays a welcome message
print("Welcome to our program!")
print("Welcome to our program!")  # This is also a single-line comment
```

Multi-line comments

Python doesn't have a dedicated syntax for multi-line comments, but you can use multiple single-line comments or enclose text in triple quotes (**"""**).

```
Using multiple single-line comments
# This is a comment
# written across
# several lines
print("Welcome to our program!")
```

```
Using triple quotes
"""
This is a comment
written across
several lines
"""
print("Welcome to our program!")
```

Keep in mind that while triple quotes can be used as comments, they are technically treated as multi-line strings that aren't assigned to a variable. This means they won't affect your program's execution but might not be ideal for all commenting scenarios.

By using comments effectively, you can make your code easier to understand and maintain, especially when working with others or revisiting your project later.

Understanding variables

In Python, a variable is like a container that holds data. You can use variables to store information that you might need to work with later in your program. A variable is given a name, and you can assign different values to it during the execution of your code.

What is a variable ?

A variable is a name that refers to a value stored in the computer's memory. You can think of a variable like a label or a box where you store something.

```
age = 15
name = "Leny"
```

In this example, we have two variables: **age** and **name**. The variable **age** holds the number **15**, and the variable **name** holds the string **"Leny"**.

How to create a variable ?

To create a variable, you simply choose a name and use the assignment operator (**=**) to store a value in it. The syntax is:

```
variable_name = value
```

- The **variable name** is the label or identifier you give to the variable.
- The **value** is the data that the variable will hold.

Naming variables

When naming variables, you need to follow a few rules:

1. A variable name must begin with a letter (a-z, A-Z) or an underscore (_).
2. The rest of the name can include letters, numbers (0-9), or underscores.
3. You cannot use Python keywords (like **if**, **else**, **print**) as variable names.
4. Variable names are case-sensitive, meaning **age** and **Age** are two different variables.

```
my_age = 30
your_age = 25
print(my_age)  # This will print 30
```

Changing the value of a variable

You can change the value stored in a variable at any time. Just assign a new value to the variable:

```
age = 30
print(age)  # This will print 30

age = 35  # Change the value of age
print(age)  # This will print 35
```

In this example, the value of **age** changes from **30** to **35**.

Types of data stored in variables

Python allows you to store different types of data in variables, including:

- **Numbers**: Whole numbers (**int**) or decimal numbers (**float**).
- **Strings**: Text, enclosed in quotation marks (" " or ' ').
- **Booleans**: True or False.
- **Lists**: Ordered collections of items.
- **Dictionaries**: Collections of key-value pairs.

```
examples
age = 25  # Integer
height = 5.8  # Float
name = "Pixel"  # String
is_student = True  # Boolean
```

Reassigning variables

Once you create a variable, you can change its value as many times as needed.

```
x = 10
print(x)  # Prints 10

x = 20
print(x)  # Prints 20
```

You can also use variables to perform arithmetic operations. For instance:

```
x = 10
y = 5
sum = x + y  # Adding x and y
print(sum)  # Prints 15
```

In this example, the variable **sum** stores the result of adding **x** and **y**.

Casting

In Python, **casting** refers to the process of converting one data type into another. This is often necessary when you want to perform operations between different types, such as adding a string and a number, or when you need to work with specific data formats. Python provides several built-in functions for casting:

```
int(): Converts a value to an integer (if possible)
x = "10"
y = int(x)  # Converts the string "10" into the integer 10
```

```
float(): Converts a value to a floating-point number
x = "3.14"
y = float(x)  # Converts the string "3.14" into the float 3.14
```

```
str(): Converts a value to a string
x = 25
y = str(x)  # Converts the integer 25 into the string "25"
```

```
bool(): Converts a value to a boolean (True or False)
x = 0
y = bool(x)  # Converts 0 into False
```

Any non-zero number or non-empty string is considered True, while zero and None are considered False.

```
Here is a simple example demonstrating multiple types of casting
x = str(7)     # x will be ‘7’
y = int(7)     # y will be 7
z = float(7)   # z will be 7.0
```

Casting is useful when you need to ensure that data types match for operations or when handling user input. However, it's important to be cautious of errors, especially when attempting to cast incompatible types, such as trying to convert a non-numeric string to an integer.

Get the type

In Python, it's often important to know the type of a variable or value to ensure you are working with the correct data type for your operations. The built-in function **type()** is used to retrieve the type of an object.

Explanation

The type() function returns the type of the object passed to it. This can help you verify whether the value stored in a variable is of the expected type, and can be especially useful when working with dynamic data or user input.

Examples

```
Check the type of an integer
x = 10
print(type(x))  # Output: <class 'int'>
```

```
Check the type of a float
x = 3.14
print(type(x))  # Output: <class 'float'>
```

```
Check the type of a string
x = "Hello"
print(type(x))  # Output: <class 'str'>
```

```
Check the type of a boolean
x = True
print(type(x))  # Output: <class 'bool'>
```

```
Check the type of a list
x = [1, 2, 3]
print(type(x))  # Output: <class 'list'>
```

```
Check the type of a dictionary
x = {"name": "Léa", "age": 22}
print(type(x))  # Output: <class 'dict'>
```

Sometimes, it’s useful to check the type of an object before performing operations, especially when working with different data inputs. For example, you might want to make sure a variable is an integer before performing mathematical operations:

```
x = "10"
if type(x) == int:
    print(x * 2)
else:
    print("Not an integer!")
```

Using **type()** helps ensure your code handles the right types and avoids errors during execution.

Scope of a variable

The **scope** of a variable refers to the region of the program where the variable can be accessed or modified. There are two main types of scope in Python: **local** and **global**.

Local variable

A **local variable** is a variable that is defined within a function or block of code. It can only be used inside that function or block. Once the function or block finishes executing, the local variable is destroyed and no longer accessible.

```
def greet():
    name = "Leny"  # Local variable
    print("Hello, " + name)

greet()  # This will work and print "Hello, Leny"
print(name)  # This will raise an error because 'name' is not accessible outside the function
```

Global variables

A **global variable** is a variable that is defined outside of any function and can be accessed from any part of the program, including inside functions.

```
name = "Leny"  # Global variable

def greet():
    print("Hello, " + name)  # Can access the global variable

greet()  # This will print "Hello, Leny"
print(name)  # This will print "Leny"
```

Modifying Global Variables Inside a Function

If you want to modify a global variable inside a function, you need to use the **global** keyword. Without it, the function will create a local variable with the same name, and the global variable will remain unchanged.

```
Example
counter = 0  # Global variable

def increment():
    global counter  # Use the global variable
    counter += 1

increment()
print(counter)  # This will print 1
```

Without the **global** keyword, the function would not modify the global counter variable but would instead create a new local variable.

Conclusion

Variables are essential in Python because they allow you to store and manipulate data. By using variables, you can create programs that remember and process information efficiently. Understanding the scope of variables is important to control where and how they are accessed in your code.

Key Points

- Variables are used to store values that can be used later in your program.
- You can change the value of a variable at any time.
- Different types of data can be stored in variables, such as numbers, strings, and booleans.
- **Scope** refers to where a variable can be accessed: inside a function (local) or throughout the program (global).
- **Global variables** can be accessed and modified anywhere, while **local variables** are confined to the function or block in which they are created.
- Consistent naming and good understanding of variable scope help in writing clean, readable, and error-free code.

Strings

In Python, strings are sequences of characters used to store and manipulate text. You can define a string using either single quotes (') or double quotes ("), as both are functionally identical. The choice between them depends on personal preference or specific coding needs.

Examples

```
Using double quotes
name = "Leny"
print(name)  # Output: Leny
```

```
Using single quotes
name = 'Leny'
print(name)  # Output: Leny
```

Both examples produce the same result. The main consideration lies in how you handle strings containing quotes.

When to use single or double quotes

- **Consistency**
  To improve readability and maintainability, it's good practice to consistently use either single or double quotes throughout your code.
- **Avoiding Escape Characters**
  When a string contains double quotes, enclose it in single quotes to avoid escaping the quotes.

```
quote = 'He said, "Hello!"'
```

Similarly, if a string contains single quotes, use double quotes:

```
sentence = "It's a sunny day"
```

This approach makes your code cleaner and easier to read. While Python allows you to escape quotes with a backslash (\), this can make the code less readable:

```
# Less readable
message = "He said, \"Hello!\""

# More readable
message = 'He said, "Hello!"'
```

Key takeaways

- Strings in Python are flexible and easy to use.
- The choice between single and double quotes does not affect functionality but impacts readability.
- To write clean and consistent code:
  1. Pick one style and stick with it.
  2. Use the type of quotes that minimizes the need for escape characters.
  3. Prioritize clarity and simplicity in your code.

Multiline string assignment

When you need to assign a string that spans multiple lines, Python provides two main options:

1. **Using Triple Quotes (''' or """)**
   Triple quotes allow you to create multiline strings easily. These strings preserve line breaks, spaces, and formatting exactly as written.

```
Example
paragraph = """This is a multiline string.
It spans several lines.
Each line break is preserved."""
print(paragraph)
```

```
Output
This is a multiline string.
It spans several lines.
Each line break is preserved.
```

You can use either triple single quotes (''') or triple double quotes (""""). Both work the same way.

2. **Using Escape Characters**
   If you prefer single or double quotes but still need line breaks, use the newline escape character (**\n**).

```
Example
paragraph = "This is a multiline string.\nIt spans several lines.\nEach line break is
preserved."
print(paragraph)
```

```
Output
This is a multiline string.
It spans several lines.
Each line break is preserved.
```

While this method works, it is less readable compared to using triple quotes.

Key points to remember

- Triple quotes are the most convenient way to handle multiline strings in Python, especially when readability and formatting matter.
- Escape characters like \n are useful for dynamic strings or when precise control over line breaks is needed.
- Both methods can be combined with string variables and expressions for flexibility.

Booleans

Booleans represent one of the simplest data types in Python. They can only have one of two possible values: **True** or **False**. These values are used to represent truthiness in programming and are especially helpful for controlling the flow of code using conditional statements.

Key Points:

- Booleans are capitalized in Python: **True** and **False**.
- Booleans are often the result of comparison operations or logical expressions.
- In Python, other values can implicitly evaluate to **True** or **False** in a Boolean context (e.g., in **if** statements).

Examples

```
Basic Booleans
# Assigning Boolean values
a = True
b = False

# Printing Booleans
print(a)  # Output: True
print(b)  # Output: False
```

we assign the Boolean values **True** and **False** to variables **a** and **b**, then print them to demonstrate how they are displayed.

```
Booleans from Comparisons
# Comparison operations result in Boolean values
x = 10
y = 20

print(x > y)   # Output: False
print(x == y)  # Output: False
print(x < y)   # Output: True
```

shows how comparison operators like **>**, **==**, and **<** return Boolean values based on the relationship between the variables **x** and **y**.

```
Using Booleans in Conditional Statements
# Conditional statements use Booleans to decide the flow
is_raining = True

if is_raining:
    print("Take an umbrella!")
else:
    print("Enjoy the sunshine!")

# Output: Take an umbrella!
```

The Boolean value **is_raining** is used in an **if** statement to determine which message to print. This illustrates how Booleans control program flow.

```
Booleans from String and Numeric Values
# Strings and numbers can be evaluated as Booleans
y = ""
x = "Hello"

print(bool(y))  # Output: False (empty string is False)
print(bool(x))  # Output: True (non-empty string is True)

a = 0
b = 15

print(bool(a))  # Output: False (0 is False)
print(bool(b))  # Output: True (non-zero numbers are True)
```

This example demonstrates how Python evaluates strings and numbers in a Boolean context. **Empty strings** and **0** evaluate to **False**, while non-empty strings and non-zero numbers evaluate to **True**.

```
Booleans from Common Objects
# Some objects evaluate to False in a Boolean context
print(bool(False))  # Output: False
print(bool(None))   # Output: False
print(bool(0))      # Output: False
print(bool(""))     # Output: False (empty string)
print(bool(()))     # Output: False (empty tuple)
print(bool([]))     # Output: False (empty list)
print(bool({}))     # Output: False (empty dictionary)

# Non-empty or non-zero objects evaluate to True
print(bool("Hello"))   # Output: True
print(bool(123))       # Output: True
print(bool([1, 2, 3])) # Output: True
```

Various objects like **None**, **0**, and empty collections (e.g., lists, dictionaries) evaluate to **False**, while non-empty or non-zero objects evaluate to **True**.

```
Using a Function with Booleans
# Functions can return Boolean values
def is_even(number):
    return number % 2 == 0

print(is_even(4))  # Output: True (4 is even)
print(is_even(7))  # Output: False (7 is odd)

# Using the function in a conditional statement
number = 10
if is_even(number):
    print(f"{number} is even.")
else:
    print(f"{number} is odd.")

# Output: 10 is even.
```

This example shows how a function can return a Boolean value based on a condition. The function **is_even** checks if a number is divisible by 2 and returns **True** or **False**. This is then used in an **if** statement.

```
Checking if an Object is a String
# A function to check if an object is a string
def is_string(obj):
    return isinstance(obj, str)

print(is_string("Hello"))   # Output: True
print(is_string(123))       # Output: False
print(is_string([1, 2, 3])) # Output: False

# Using the function in a conditional statement
obj = "Python"
if is_string(obj):
    print(f"{obj} is a string.")
else:
    print(f"{obj} is not a string.")

# Output: Python is a string.
```

The function **is_string** uses the **isinstance** method to check if an object is of type **str**. This can be helpful to validate inputs or perform type-specific operations.

Summary

Booleans are fundamental to programming logic. They help determine the behavior of a program through conditions and control flow. Mastering how to use **True** and **False**, along with Boolean operations, is a key step in learning Python.

Operators

Operators are special symbols or keywords in Python used to perform operations on variables and values. Python supports several types of operators that allow you to perform computations, comparisons, and other tasks.

Types of Python Operators

1. **Arithmetic Operators**: Perform mathematical operations.
2. **Comparison Operators**: Compare two values and return a Boolean result.
3. **Logical Operators**: Combine Boolean values and return a Boolean result.
4. **Bitwise Operators**: Perform operations on binary representations of integers.
5. **Assignment Operators**: Assign values to variables.
6. **Membership Operators**: Test if a value is in a sequence.
7. **Identity Operators**: Compare the identity of two objects.

Examples

Arithmetic Operators

```
# Arithmetic operations
x = 10
y = 3

print(x + y)  # Output: 13 (Addition)
print(x - y)  # Output: 7 (Subtraction)
print(x * y)  # Output: 30 (Multiplication)
print(x / y)  # Output: 3.333... (Division)
print(x % y)  # Output: 1 (Modulus)
print(x ** y) # Output: 1000 (Exponentiation)
print(x // y) # Output: 3 (Floor division)
```

Arithmetic operators like +, -, *, /, %, **, and // allow you to perform basic mathematical calculations.

Comparison Operators

```
# Comparison operations
x = 10
y = 20

print(x == y)  # Output: False (Equality)
print(x != y)  # Output: True (Inequality)
print(x > y)   # Output: False (Greater than)
print(x < y)   # Output: True (Less than)
print(x >= y)  # Output: False (Greater than or equal to)
print(x <= y)  # Output: True (Less than or equal to)
```

Comparison operators compare two values and return a Boolean result (True or False).

Logical Operators

```
# Logical operations
a = True
b = False

print(a and b)  # Output: False (Logical AND)
print(a or b)   # Output: True (Logical OR)
print(not a)    # Output: False (Logical NOT)
```

Logical operators like and, or, and not combine or invert Boolean values.

```
Bitwise Operators
# Bitwise operations
x = 5   # Binary: 0101
y = 3   # Binary: 0011

print(x & y)  # Output: 1 (Bitwise AND)
print(x | y)  # Output: 7 (Bitwise OR)
print(x ^ y)  # Output: 6 (Bitwise XOR)
print(~x)     # Output: -6 (Bitwise NOT)
print(x << 1) # Output: 10 (Left shift)
print(x >> 1) # Output: 2 (Right shift)
```

Bitwise operators perform operations on binary representations of integers.

```
Assignment Operators
# Assignment operations
x = 10

x += 5  # Equivalent to x = x + 5
print(x)  # Output: 15

x -= 3  # Equivalent to x = x - 3
print(x)  # Output: 12

x *= 2  # Equivalent to x = x * 2
print(x)  # Output: 24

x /= 4  # Equivalent to x = x / 4
print(x)  # Output: 6.0

x %= 5  # Equivalent to x = x % 5
print(x)  # Output: 1.0

x **= 3  # Equivalent to x = x ** 3
print(x)  # Output: 1.0

x //= 2  # Equivalent to x = x // 2
print(x)  # Output: 0.0

# Bitwise assignment operators
y = 5

y &= 3  # Equivalent to y = y & 3
print(y)  # Output: 1

y |= 2  # Equivalent to y = y | 2
print(y)  # Output: 3

y ^= 3  # Equivalent to y = y ^ 3
print(y)  # Output: 0

y <<= 1  # Equivalent to y = y << 1
print(y)  # Output: 0

y >>= 1  # Equivalent to y = y >> 1
print(y)  # Output: 0

# Walrus operator
z = 10
print(result := z > 5)  # Output: True (Assigns and evaluates in one step)
```

Assignment operators include arithmetic, modulus, exponentiation, floor division, bitwise operations, and the walrus operator (:=), which assigns a value and returns it in a single step.

```
Membership Operators
# Membership operations
my_list = [1, 2, 3, 4, 5]

print(3 in my_list)   # Output: True (3 is in the list)
print(6 not in my_list) # Output: True (6 is not in the list)
```

Membership operators in and not in check if a value exists in a sequence (e.g., list, string, tuple).

```
Identity Operators
# Identity operations
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)   # Output: True (x and y reference the same object)
print(x is z)   # Output: False (x and z reference different objects)
print(x is not z) # Output: True (x and z are not the same object)
```

Identity operators is and is not check whether two variables reference the same object in memory.

Summary

Python operators are essential for performing various computations, comparisons, and logical operations. Understanding how to use each type of operator effectively is key to writing efficient and clear Python code.

Lists

Overview

Lists are one of the most versatile and widely used data structures in Python. They are mutable, ordered collections of items that can hold elements of any data type. This flexibility makes lists an essential tool for Python developers.

Key Features of Lists

1. **Ordered**: The elements in a list maintain their order, and you can access them using their index.
2. **Mutable**: Lists can be modified after their creation by adding, removing, or changing elements.
3. **Heterogeneous**: A single list can contain elements of different data types, such as integers, strings, floats, or even other lists.
4. **Dynamic**: The size of a list can change dynamically, as Python allows you to add or remove elements at any time.

Creating Lists

To create a list, use square brackets **[]** and separate the elements with commas.

```
# Examples of lists
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed = ["Python", 3.14, True]
nested = [[1, 2], [3, 4]]
```

Using the List Constructor

Python provides a built-in **list()** constructor to create lists from other iterable objects, such as strings, tuples, or ranges.

```
# Examples using list()
from_string = list("hello")   # ['h', 'e', 'l', 'l', 'o']
from_tuple = list((1, 2, 3))  # [1, 2, 3]
from_range = list(range(5))   # [0, 1, 2, 3, 4]
```

Accessing List Elements

By Index

Python uses zero-based indexing. To access an element, specify its position inside square brackets:

```
my_list = ["apple", "banana", "cherry"]
print(my_list[0])  # Output: apple
print(my_list[2])  # Output: cherry
```

Negative Indexing

Negative indices allow you to access elements from the end of the list:

```
print(my_list[-1])  # Output: cherry
print(my_list[-2])  # Output: banana
```

Slicing

Retrieve a subset of elements using slicing:

```
print(my_list[0:2])  # Output: ['apple', 'banana']
print(my_list[1:])   # Output: ['banana', 'cherry']
```

Modifying Lists

Adding Elements

```
Append: Add a single element to the end of the list.
my_list.append("date")
print(my_list)  # Output: ['apple', 'banana', 'cherry', 'date']
```

```
Extend: Add multiple elements to the end.
my_list.extend(["elderberry", "fig"])
print(my_list)  # Output: ['apple', 'banana', 'cherry', 'date', 'elderberry', 'fig']
```

```
Insert: Add an element at a specific index.
my_list.insert(1, "blueberry")
print(my_list)  # Output: ['apple', 'blueberry', 'banana', 'cherry', 'date', 'elderberry',
'fig']
```

Removing Elements

```
Remove: Remove the first occurrence of a specific value.
my_list.remove("banana")
print(my_list)  # Output: ['apple', 'blueberry', 'cherry', 'date', 'elderberry', 'fig']
```

```
Pop: Remove an element by index (default is the last element).
my_list.pop(2)
print(my_list)  # Output: ['apple', 'blueberry', 'date', 'elderberry', 'fig']
```

```
Clear: Remove all elements.
my_list.clear()
print(my_list)  # Output: []
```

Iterating Over Lists

Use loops to iterate through elements of a list.

```
# Example: Iterating through a list
for fruit in ["apple", "banana", "cherry"]:
    print(fruit)
```

Common List Methods

- **len(my_list)**: Returns the number of elements in the list.
- **sorted(my_list)**: Returns a sorted version of the list (original list remains unchanged).
- **my_list.reverse()**: Reverses the elements of the list in place.
- **my_list.index(value)**: Returns the index of the first occurrence of a value.
- **my_list.count(value)**: Counts the number of occurrences of a value.

Use Cases

Lists are used in various scenarios, such as:

1. Storing and manipulating data collections.
2. Implementing stacks and queues.
3. Managing dynamic data sets in applications.
4. Iterating over elements for computations.

By mastering lists, you unlock a powerful tool in Python capable of handling a diverse range of programming challenges. With their flexibility and ease of use, lists are an indispensable component of Python programming, suited for both beginners and experienced developers alike.

Tuples

Overview

Tuples are an essential data structure in Python. They are immutable, ordered collections of items that can store elements of any data type. While they share similarities with lists, tuples are designed to be unchangeable after their creation, offering a reliable way to store fixed collections of data.

Key Features of Tuples

1. **Ordered**: The elements in a tuple maintain their order, and you can access them using their index.
2. **Immutable**: Once a tuple is created, its elements cannot be modified, added, or removed.
3. **Heterogeneous**: A tuple can contain elements of different data types, including integers, strings, floats, and other tuples.
4. **Hashable**: Tuples can be used as keys in dictionaries if all their elements are hashable.

Creating Tuples

Tuples are created using parentheses () with elements separated by commas.

```
# Examples of tuples
empty_tuple = ()
single_element_tuple = ("Python",)  # Comma is necessary for single-element tuples
tuple_of_numbers = (1, 2, 3, 4)
mixed_tuple = ("Hello", 3.14, False)
nested_tuple = ((1, 2), (3, 4))
```

Without Parentheses

Python allows tuples to be created without parentheses in many cases, relying on commas to define the tuple:

```
tuple_implicit = 1, 2, 3
print(type(tuple_implicit))  # Output: <class 'tuple'>
```

Accessing Tuple Elements

By Index

Like lists, tuples use zero-based indexing. You can access elements by their index:

```
my_tuple = ("apple", "banana", "cherry")
print(my_tuple[0])  # Output: apple
print(my_tuple[2])  # Output: cherry
```

Negative Indexing

Negative indices allow you to access elements from the end of the tuple:

```
print(my_tuple[-1])  # Output: cherry
print(my_tuple[-2])  # Output: banana
```

Slicing

You can retrieve a subset of elements using slicing:

```
print(my_tuple[0:2])  # Output: ('apple', 'banana')
print(my_tuple[1:])   # Output: ('banana', 'cherry')
```

Tuple Operations

Concatenation

You can combine two or more tuples using the + operator:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)
```

Repetition

Repeat a tuple multiple times using the * operator:

```
tuple_repeated = ("A", "B") * 3
print(tuple_repeated)  # Output: ('A', 'B', 'A', 'B', 'A', 'B')
```

Membership

Check if an element exists in a tuple using the in keyword:

```
print("apple" in my_tuple)  # Output: True
print("orange" not in my_tuple)  # Output: True
```

Immutable Nature of Tuples

Tuples cannot be modified after their creation:

```
my_tuple = (1, 2, 3)
my_tuple[1] = 10  # This will raise a TypeError
```

If modifications are required, you can convert the tuple to a list, make changes, and convert it back to a tuple:

```
temp_list = list(my_tuple)
temp_list[1] = 10
my_tuple = tuple(temp_list)
print(my_tuple)  # Output: (1, 10, 3)
```

Common Tuple Methods

- **len(my_tuple)**: Returns the number of elements in the tuple.
- **my_tuple.index(value)**: Returns the index of the first occurrence of a value.
- **my_tuple.count(value)**: Counts the number of occurrences of a value in the tuple.

Use Cases

Tuples are ideal for scenarios where data should remain constant and protected from unintentional modifications. Common use cases include:

1. Returning multiple values from a function.
2. Storing related items, such as geographic coordinates or RGB color values.
3. Using tuples as keys in dictionaries for compound lookups.

By understanding tuples, you gain access to a lightweight and efficient way to work with fixed collections of data in Python. Their immutability makes them a reliable choice for various programming tasks.

Python Sets

Overview

Sets are an unordered collection of unique elements in Python. They are mutable, making them suitable for storing and performing operations on a collection of distinct items. Sets are highly efficient for membership testing and eliminating duplicates.

Key features of sets

1. **Unordered**: The elements in a set do not maintain a specific order.
2. **Unique**: Each element in a set is unique, with duplicates automatically removed.
3. **Mutable**: Sets can be modified after their creation by adding or removing elements.
4. **Hashable Elements**: Only hashable (immutable) data types, such as numbers, strings, or tuples, can be added to a set.

Creating sets

Sets are created using curly braces **{}** or the **set()** constructor. Empty sets must be created using **set()** because **{}** creates an empty dictionary.

```
# Examples of sets
empty_set = set()
fruit_set = {"apple", "banana", "cherry"}
mixed_set = {42, 3.14, "Python"}
```

Using the **set()** constructor

The **set()** constructor can create sets from other iterable objects, such as lists, strings, or tuples:

```
from_list = set([1, 2, 3, 4, 4])  # {1, 2, 3, 4}
from_string = set("hello")        # {'e', 'h', 'l', 'o'}
from_tuple = set(("a", "b", "a")) # {'a', 'b'}
```

Accessing Set Elements

Since sets are unordered, elements cannot be accessed by index or slicing. Instead, you can iterate through a set using a loop:

```
for fruit in {"apple", "banana", "cherry"}:
    print(fruit)
```

Modifying Sets

Adding Elements

```
Add: Add a single element to the set.
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

```
Update: Add multiple elements from an iterable.
my_set.update([5, 6, 7])
print(my_set)  # Output: {1, 2, 3, 4, 5, 6, 7}
```

Removing Elements

```
Remove: Remove a specific element. Raises a KeyError if the element is not found.
my_set.remove(2)
print(my_set)  # Output: {1, 3, 4, 5, 6, 7}
```

```
Discard: Remove a specific element without raising an error if it is not found.
my_set.discard(10)  # No error, even though 10 is not in the set
```

```
Pop: Remove and return an arbitrary element from the set.
element = my_set.pop()
print(element)  # Output: Randomly removed element
```

```
Clear: Remove all elements from the set.
my_set.clear()
print(my_set)  # Output: set()
```

Set Operations

Sets provide various operations for mathematical computations like union, intersection, and difference.

Union

Combine elements from two sets, keeping only unique elements:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # {1, 2, 3, 4, 5}
```

Intersection

Find common elements between two sets:

```
intersection_set = set1 & set2  # {3}
```

Difference

Find elements in one set but not the other:

```
difference_set = set1 - set2  # {1, 2}
```

Symmetric Difference

Find elements in either set but not in both:

```
symmetric_diff = set1 ^ set2  # {1, 2, 4, 5}
```

Subset and Superset

```
Check if a set is a subset of another:
print({1, 2} <= {1, 2, 3})  # Output: True
```

```
Check if a set is a superset of another:
print({1, 2, 3} >= {1, 2})  # Output: True
```

Common Set Methods

- **len(my_set)**: Returns the number of elements in the set.
- **my_set.copy()**: Creates a shallow copy of the set.
- **my_set.isdisjoint(other_set)**: Checks if two sets have no elements in common.

Use Cases

Sets are ideal for scenarios requiring uniqueness or mathematical operations. Common use cases include:

1. Eliminating duplicates from a list.
2. Checking for membership efficiently.
3. Performing set-based operations in algorithms.
4. Representing concepts like tags or categories where order doesn't matter.

By mastering sets, you add a robust tool to your Python programming toolkit. Sets excel in scenarios that demand efficiency, uniqueness, and mathematical computation.

Python Dictionaries

Overview

Dictionaries in Python are an essential and versatile data structure that allows you to store data in key-value pairs. They are mutable and unordered, making them suitable for representing and manipulating data that can be associated with unique keys.

Key features of dictionaries

1. **Key-Value Pair Structure**: Each key maps to a specific value.
2. **Keys Are Unique**: A key can only appear once in a dictionary.
3. **Mutable**: The dictionary's content can be changed after its creation.
4. **Unordered**: The order of items is not guaranteed, though Python (from version 3.7 onwards) maintains insertion order for dictionaries.
5. **Efficient Lookup**: Dictionaries provide fast access to values via keys.

Creating dictionaries

Dictionaries are created using curly braces **{}** with key-value pairs separated by a colon **:**. Alternatively, the **dict()** constructor can be used.

```
# Examples of dictionaries
empty_dict = {}
fruit_prices = {"apple": 1.2, "banana": 0.5, "cherry": 2.5}
mixed_dict = {"name": "Alice", "age": 25, "is_student": True}
```

Using the dict() constructor

The **dict()** constructor can create dictionaries from sequences of key-value pairs or keyword arguments:

```
from_tuples = dict([("a", 1), ("b", 2), ("c", 3)])  # {'a': 1, 'b': 2, 'c': 3}
from_kwargs = dict(x=10, y=20, z=30)  # {'x': 10, 'y': 20, 'z': 30}
```

Syntax of the dict() constructor

The **dict()** constructor accepts an iterable of key-value pairs, such as a list of tuples or keyword arguments:

```
# Example using list of tuples
example_dict = dict([("key1", "value1"), ("key2", "value2")])
print(example_dict)  # Output: {'key1': 'value1', 'key2': 'value2'}

# Example using keyword arguments
example_dict = dict(a=1, b=2, c=3)
print(example_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

This flexibility makes the **dict()** constructor a powerful tool for creating dictionaries programmatically.

Accessing dictionary elements

Accessing values by keys

You can access a value by specifying its corresponding key:

```
fruit_prices = {"apple": 1.2, "banana": 0.5}
print(fruit_prices["apple"])  # Output: 1.2
```

*Using **get()***

The **get()** method provides a safe way to access values without risking a **KeyError**:

```
print(fruit_prices.get("cherry", "Not found"))  # Output: Not found
```

Modifying Dictionaries

Adding or Updating Key-Value Pairs

Assign a value to a key to add or update an entry:

```
fruit_prices["cherry"] = 2.5  # Adds 'cherry'
fruit_prices["apple"] = 1.3  # Updates 'apple'
```

Removing Elements

```
pop(): Removes and returns the value for a given key.
price = fruit_prices.pop("banana")
print(price)  # Output: 0.5
```

```
del: Deletes a key-value pair.
del fruit_prices["apple"]
```

```
popitem(): Removes and returns the last inserted key-value pair (since Python 3.7).
key, value = fruit_prices.popitem()
```

```
clear(): Removes all entries from the dictionary.
fruit_prices.clear()
print(fruit_prices)  # Output: {}
```

Dictionary Methods

Commonly Used Methods

```
keys(): Returns a view object of all the keys.
print(fruit_prices.keys())
```

```
values(): Returns a view object of all the values.
print(fruit_prices.values())
```

```
items(): Returns a view object of all key-value pairs.
print(fruit_prices.items())
```

```
update(): Updates the dictionary with key-value pairs from another dictionary or iterable.
fruit_prices.update({"orange": 1.1, "grape": 3.0})
```

Iterating Through a Dictionary

```
Keys
for key in fruit_prices:
    print(key)
```

```
Values
for value in fruit_prices.values():
    print(value)
```

```
Key-Value Pairs
for key, value in fruit_prices.items():
    print(f"{key}: {value}")
```

Use Cases

Dictionaries are ideal for scenarios where data is organized in a key-value structure. Common use cases include:

1. Storing configuration settings.
2. Mapping identifiers to values (e.g., usernames to user data).
3. Implementing lookups for quick data access.
4. Storing structured data such as JSON-like formats.

By mastering dictionaries, you unlock a powerful way to manage and manipulate data efficiently in Python. Their flexibility and speed make them indispensable for various programming tasks.

If..else Statements

Introduction

In programming, conditions allow a program to make decisions based on certain criteria. By using conditional statements, you can control the flow of execution and make your program more dynamic. In Python, the most common way to implement conditions is through **if statements**. This chapter will explain how conditions work in Python, and how to use **if**, **elif**, and **else** statements to create decision-making logic.

The if statement

The **if** statement in Python is used to execute a block of code only if a specified condition evaluates to **True**. If the condition is **False**, the code inside the **if** block is skipped.

```
Syntax of if Statement
if condition:
    # Code block to execute if the condition is True
```

- **condition**: A Boolean expression that evaluates to either **True** or **False**.
- The code inside the **if** block runs only if the condition is **True**.

```
Example:
age = 20

if age >= 18:
    print("You are an adult.")
```

In this example, since the **age** is 20, which is greater than or equal to 18, the program prints "You are an adult."

The else statement

Sometimes, you need to specify an alternative block of code to execute if the condition is not met (i.e., when it evaluates to False). This can be done using the else statement.

```
Syntax of if-else Statement
if condition:
    # Code block to execute if the condition is True
else:
    # Code block to execute if the condition is False
```

```
Example:
age = 16

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

Here, the **else** block is executed because **age** is less than 18, so the program prints "You are a minor."

The elif statement

If you have multiple conditions to check, you can use the elif (short for "else if") statement. This allows you to check additional conditions if the previous ones are False.

```
Syntax of if-elif-else Statement
if condition1:
    # Code block to execute if condition1 is True
elif condition2:
    # Code block to execute if condition2 is True
else:
    # Code block to execute if none of the conditions are True
```

```
Example:
age = 70

if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior.")
```

In this example:

- If **age** is less than 18, it prints "You are a minor."
- If **age** is between 18 and 64 (inclusive), it prints "You are an adult."
- If neither of the above conditions is true (i.e., age is 65 or older), it prints "You are a senior."

Nested Conditions

Sometimes, you may need to place one condition inside another. This is called **nesting** and can be done by using an **if** statement inside another **if**, **elif**, or **else** block.

```
Syntax of Nested Conditions
if condition1:
    if condition2:
        # Code block to execute if both condition1 and condition2 are True
    else:
        # Code block to execute if condition1 is True and condition2 is False
else:
    # Code block to execute if condition1 is False
```

```
Example:
age = 30
employment_status = "employed"

if age >= 18:
    if employment_status == "employed":
        print("You are an employed adult.")
    else:
        print("You are an unemployed adult.")
else:
    print("You are a minor.")
```

Here:

- If **age** is 18 or older and **employment_status** is "employed," it prints "You are an employed adult."
- If **age** is 18 or older but **employment_status** is not "employed," it prints "You are an unemployed adult."
- If **age** is less than 18, it prints "You are a minor."

Comparison Operators

In conditional statements, you often need to compare values. Python provides several comparison operators for this purpose:

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
Example of Comparison Operators:
age = 25
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

In this case, the **>=** operator checks if **age** is greater than or equal to 18.

Logical Operators

You can combine multiple conditions using logical operators: and, or, and not.

- **and**: Returns **True** if both conditions are **True**.
- **or**: Returns **True** if at least one condition is **True**.
- **not**: Reverses the Boolean value of the condition.

```
Example of Logical Operators:
age = 30
citizen = True

if age >= 18 and citizen:
    print("You are an eligible voter.")
else:
    print("You are not eligible to vote.")
```

Here, the program checks if both **age** is greater than or equal to 18 and if the person is a citizen. Only if both conditions are **True**, the message "You are an eligible voter." is printed.

Conclusion

In this chapter, we've covered the essentials of using conditional statements in Python with if, elif, and else. You can combine comparison operators, logical operators, and even nest conditions to create more complex decision-making logic. Understanding how to use these statements will allow you to create more interactive and dynamic programs.

Key Takeaways:

- Use **if** to execute a block of code if a condition is **True**.
- Use **else** to provide an alternative if the if condition is **False**.
- Use **elif** to check multiple conditions.

- Combine conditions with logical operators like **and**, **or**, and **not**.
- Nest **if** statements for more complex decision-making.

Now that you know how to make decisions in Python, you can start creating programs that react differently based on various conditions !

The pass statement

In Python, the **pass** statement is a placeholder that does nothing. It is used when a statement is required syntactically but you do not want to execute any code. The **pass** statement can be useful in conditional structures when you want to define the condition but leave the block empty for now or for future implementation.

```
Syntax of pass Statement
if condition:
    pass  # Placeholder, does nothing
else:
    # Code block to execute if condition is False
```

```
Example:
age = 16

if age >= 18:
    print("You are an adult.")
else:
    pass  # No action if the person is under 18
```

In this example:

- The **else** block is executed only if **age** is less than 18.
- However, the **pass** statement inside the **else** block means that nothing happens when the condition is met. The program will simply continue executing after the conditional statement.

Another Example with Future Code

The **pass** statement can also be used when you plan to implement code later but need to structure your program now.

```
age = 20
status = "employed"

if age >= 18:
    pass  # Placeholder for future implementation
elif status == "employed":
    print("You are employed.")
else:
    print("You are a minor.")
```

In this case, the **pass** statement in the first **if** block indicates that you plan to add additional logic or code in the future, but for now, nothing will happen when **age** is 18 or older.

Conclusion

The **pass** statement is a simple yet powerful tool in Python for handling empty blocks of code. It is especially useful when building out the structure of your program, allowing you to write placeholder code that can be filled in later.

While Loops in Python

Introduction

In programming, loops allow a block of code to be executed multiple times. This is useful when you need to repeat a task, such as processing items in a list, handling user input, or performing calculations repeatedly. In Python, one of the most common types of loops is the **while** loop. A **while** loop repeatedly executes a block of code as long as a specified condition remains **True**.

In this chapter, we will explore how to use the **while** loop in Python, understand its structure, and look at various examples of how it can be applied.

The syntax of a **while** loop

The basic syntax of a **while** loop is as follows:

```
while condition:
    # Code block to execute as long as the condition is True
```

- **condition**: A boolean expression that is evaluated before each iteration of the loop.
- The code inside the loop runs as long as the condition evaluates to **True**. When the condition becomes **False**, the loop stops.

```
Example:
count = 0

while count < 5:
    print("Count is:", count)
    count += 1
```

In this example:

- The loop will print the value of **count** while it is less than 5.
- After each iteration, the value of **count** is incremented by 1.
- Once **count** reaches 5, the condition **count < 5** becomes **False**, and the loop stops.

```
Output:
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
```

Infinite Loops

An infinite loop occurs when the condition of the **while** loop never becomes **False**. This can happen if the condition is always **True** or if there is no mechanism to update the condition.

```
For example:
while True:
    print("This will run forever!")
```

This loop will continuously print "This will run forever!" without ever stopping. **Infinite loops** are typically used in server programs, video games, or applications where the program needs to keep running and waiting for events or input from the user.

Stopping an Infinite Loop

You can break out of an infinite loop using the **break** statement. This allows you to exit the loop under certain conditions.

```
Example:
count = 0
```

```
while True:
    print("Count is:", count)
    count += 1
    if count >= 5:
        break  # Exit the loop when count reaches 5
```

Here, the loop runs until **count** reaches 5, at which point the **break** statement exits the loop.

```
Output:
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
```

The else Clause in a while Loop

In Python, a **while** loop can have an optional **else** block that is executed when the loop condition becomes **False**. The **else** block will not execute if the loop is terminated by a **break** statement.

```
Syntax:
while condition:
    # Code block to execute as long as the condition is True
else:
    # Code block to execute when the condition becomes False
```

```
Example:
count = 0

while count < 5:
    print("Count is:", count)
    count += 1
else:
    print("Loop has finished.")
```

Here:

- The loop will run while **count** is less than 5, printing the value of **count** during each iteration.
- When **count** becomes 5, the condition is no longer **True**, so the **else** block is executed, printing "Loop has finished."

```
Output:
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Loop has finished.
```

Controlling the flow with break, continue, and pass

Python provides a few control flow statements that can be used inside **while** loops to alter the normal behavior:

*The **break** Statement*

The **break** statement is used to exit the loop prematurely, regardless of the loop's condition.

```
count = 0

while count < 10:
    if count == 5:
        break  # Exit the loop when count is 5
    print(count)
    count += 1
```

Here, the loop will print the values of count from 0 to 4 and will exit once count reaches 5 because of the break statement.

```
Output:
0
1
2
3
4
```

The continue Statement

The **continue** statement is used to skip the current iteration and proceed to the next one. It can be useful if you want to skip specific conditions but continue the loop.

```
count = 0

while count < 5:
    count += 1
    if count == 3:
        continue  # Skip the rest of the loop when count is 3
    print(count)
```

In this example:

- The loop prints all values of **count** except when **count** is 3, as the **continue** statement skips the printing when the condition is met.

```
Output:
1
2
4
5
```

The pass Statement

The **pass** statement is a placeholder that does nothing. It is used when a statement is required syntactically but you do not want to execute any code. In a **while** loop, it can be used as a placeholder for future implementation.

```
count = 0

while count < 5:
    pass  # No action, but the loop will continue until count reaches 5
```

Nested **while** Loops

You can place one **while** loop inside another, called a **nested while loop**. This is useful for working with multi-dimensional data, such as a matrix or grid, or when you need to perform a set of actions multiple times within each iteration of another loop.

```
Example:
i = 0
while i < 3:
    j = 0
    while j < 2:
        print(f"i = {i}, j = {j}")
        j += 1
    i += 1
```

This example prints all combinations of i and j where i ranges from 0 to 2 and j ranges from 0 to 1.

```
Output:
i = 0, j = 0
i = 0, j = 1
i = 1, j = 0
i = 1, j = 1
i = 2, j = 0
i = 2, j = 1
```

Conclusion

The **while** loop is a powerful tool in Python that allows you to repeat actions as long as a condition is **True**. By using **break**, **continue**, and **pass**, you can control the flow of your loops, making your code more flexible and efficient. Whether you are processing data, waiting for user input, or managing repetitive tasks, understanding and mastering **while** loops will help you write better and more dynamic Python programs.

Key Takeaways:

- The **while** loop repeats a block of code as long as a condition is **True**.
- Use the **break** statement to exit the loop early.
- Use the **continue** statement to skip the current iteration.
- Use the **pass** statement as a placeholder when no action is required.
- You can nest **while** loops to handle more complex tasks.

Now that you understand how to work with **while** loops, you can use them to write programs that handle repetitive tasks efficiently !

The for Loop in Python

Introduction

In programming, loops are used to repeat a block of code multiple times. A **for** loop is one of the most commonly used loops in Python. Unlike the **while** loop, which repeats code while a condition is true, a **for** loop iterates over a sequence (such as a list, tuple, dictionary, string, or range) and executes the code block for each item in that sequence.

In this chapter, we will explore how to use the **for** loop in Python, understand its structure, and look at different examples to illustrate how you can use **for** loops effectively.

The Syntax of a for Loop

The basic syntax of a **for** loop in Python is:

```
for item in sequence:
    # Code block to execute for each item in the sequence
```

- **item**: This represents each element in the sequence.
- **sequence**: This is the collection (list, tuple, string, etc.) over which the **for** loop iterates.
- The code inside the loop executes once for each item in the sequence.

```
Example:
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

In this example:

- The **for** loop iterates over each element in the **fruits** list.
- For each element (or **fruit**), the program prints its value.

```
Output:
apple
banana
cherry
```

The range() Function

In Python, the **range()** function is commonly used with **for** loops to iterate over a sequence of numbers. The **range()** function generates a sequence of numbers, which is useful when you want to repeat an action a specific number of times.

```
Syntax of range()
range(start, stop, step)
```

- **start**: The value of the first number in the sequence (inclusive). The default is 0.
- **stop**: The value at which the sequence stops (exclusive).
- **step**: The increment between each number in the sequence. The default is 1.

```
Example with range():
for i in range(5):
    print(i)
```

This loop will iterate from 0 to 4, printing each number in the sequence.

```
Output:
0
1
2
```

```
3
4
```

```
Example with range() and custom start/step values:
for i in range(2, 10, 2):
    print(i)
```

In this example:

- The loop starts at 2, increments by 2, and stops before 10.
- It prints the even numbers between 2 and 8.

```
Output:
2
4
6
8
```

Iterating Over Strings

You can also use a **for** loop to iterate over the characters in a string. Each character in the string is treated as an individual item.

```
Example:
word = "hello"

for letter in word:
    print(letter)
```

In this example:

- The loop iterates over each character in the string **word** and prints each letter.

```
Output:
h
e
l
l
o
```

Iterating Over Dictionaries

In Python, a dictionary is a collection of key-value pairs. You can iterate over a dictionary in various ways: through keys, values, or both.

```
Example: Iterating Over Dictionary Keys
person = {"name": "Alice", "age": 30, "city": "New York"}

for key in person:
    print(key)
```

This loop prints the keys of the dictionary **person**.

```
Output:
name
age
city
```

```
Example: Iterating Over Dictionary Values
for value in person.values():
    print(value)
```

This loop prints the values of the dictionary person.

```
Output:
Alice
```

```
30
New York
```

```
Example: Iterating Over Both Keys and Values
for key, value in person.items():
    print(f"{key}: {value}")
```

This loop prints both the keys and values of the dictionary.

```
Output:
name: Alice
age: 30
city: New York
```

Nested for Loops

You can use **for** loops inside other **for** loops, known as **nested for loops**. This is useful when working with multi-dimensional data structures like lists of lists (2D arrays) or matrices.

```
Syntax of Nested for Loops:
for item1 in sequence1:
    for item2 in sequence2:
        # Code block to execute for each combination of item1 and item2
```

```
Example: Nested for Loop to Print a 2D Matrix
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for element in row:
        print(element, end=" ")
    print()  # Newline after each row
```

In this example:

- The outer loop iterates over each row in the matrix.
- The inner loop iterates over each element within the row.

```
Output:
1 2 3
4 5 6
7 8 9
```

The **else** Clause in a **for** Loop

Just like in an **if** statement, you can use an **else** clause with a **for** loop. The **else** block will execute when the loop completes its normal iteration, i.e., when the loop is not terminated by a **break** statement.

```
Example:
for i in range(5):
    print(i)
else:
    print("Loop has finished.")
```

In this case:

- The else block will execute after the loop has completed all iterations.

```
Output:
0
1
2
```

```
3
4
Loop has finished.
```

```
Example with break and else:
for i in range(5):
    if i == 3:
        print("Found 3, exiting loop!")
        break
else:
    print("Loop completed without breaking.")
```

In this case:

- The loop will stop when **i** equals 3 due to the **break** statement, so the **else** block will not execute.

```
Output:
Found 3, exiting loop!
```

The continue Statement

The continue statement is used to skip the rest of the code in the current iteration and jump to the next iteration of the loop. It is often used when you want to ignore certain elements in a sequence based on a condition.

```
Example with continue:
for i in range(5):
    if i == 3:
        continue  # Skip the iteration when i is 3
    print(i)
```

In this example:

- The loop skips the iteration when **i** equals 3, so **3** is not printed.

```
Output:
0
1
2
4
```

Conclusion

The **for** loop is an essential tool in Python, allowing you to iterate over sequences and perform repetitive tasks efficiently. By combining **for** loops with functions like **range()**, iterating over strings, lists, and dictionaries, you can write flexible and dynamic code. Additionally, features like nested **for** loops, the **continue** statement, and the **else** clause further enhance the power and flexibility of **for** loops.

Key Takeaways:

- Use the **for** loop to iterate over sequences (lists, strings, dictionaries, etc.).
- The **range()** function helps you generate sequences of numbers for use in loops.
- You can nest **for** loops to handle more complex data structures.
- The **else** clause with **for** loops runs when the loop completes without a break.
- Use the **continue** statement to skip specific iterations in the loop.

With this knowledge, you are now equipped to leverage **for** loops to write efficient and effective Python code!

Functions in Python

Introduction

In programming, functions are used to group a set of statements or instructions that perform a specific task. Functions allow you to break down complex problems into smaller, manageable parts, make your code reusable, and improve readability. Python makes it easy to define and use functions, making them one of the most essential and powerful tools in your coding toolkit.

This chapter will explain how to define, call, and work with functions in Python. We will cover the basic syntax, parameters, return values, explore some more advanced concepts like variable scope, and introduce recursive functions.

Defining a Function

In Python, you define a function using the **def** keyword, followed by the function name and a set of parentheses that may contain parameters.

```
Syntax:
def function_name(parameters):
    # Code block that performs the task
    # Optional return statement
```

- **def**: This keyword tells Python you are defining a function.
- **function_name**: This is the name you give to your function, and it follows the same rules as variable names.
- **parameters**: These are the inputs that the function can accept. Parameters are optional, and you can define a function without any.
- **Code block**: This contains the logic that the function will execute.
- **return**: The **return** statement is optional, and it sends the output from the function back to the caller.

```
Example: A Simple Function
def greet():
    print("Hello, world!")
```

In this example:

- The function greet takes no parameters and simply prints "Hello, world!" when called.

Calling the Function:

To execute the code inside the function, you "call" it by using its name followed by parentheses.

```
greet()  # Output: Hello, world!
```

```
Output:
Hello, world!
```

Functions with Parameters

You can define a function that accepts parameters (inputs) to make it more dynamic. Parameters allow you to pass data into the function, making it more flexible and reusable.

```
Syntax of Function with Parameters:
def function_name(parameter1, parameter2):
    # Code block using the parameters
```

```
Example: Function with Parameters
def greet_person(name):
    print(f"Hello, {name}!")
```

In this example:

- The function **greet_person** takes one parameter **name** and prints a personalized greeting.

```
Calling the Function with Arguments:
greet_person("Alice")  # Output: Hello, Alice!
greet_person("Bob")    # Output: Hello, Bob!
```

```
Output:
Hello, Alice!
Hello, Bob!
```

Returning Values from Functions

Functions can return values using the return keyword. This allows you to capture the result of the function and use it elsewhere in your program.

```
Syntax of Return:
def function_name(parameters):
    # Some code
    return result
```

- The **return** statement sends the result back to the caller, and it can be stored in a variable or used directly.

```
Example: Function with Return Value
def add(a, b):
    return a + b
```

This function takes two parameters, **a** and **b**, adds them together, and returns the sum.

```
Calling the Function with Return:
result = add(3, 5)
print(result)  # Output: 8
```

```
Output:
8
```

```
Example: Using the Return Value Directly
print(add(10, 20))  # Output: 30
```

```
Output:
30
```

Default Parameters

In Python, you can provide default values for parameters in case the caller does not supply them. Default parameters are assigned values during function definition.

```
Syntax with Default Parameters:
def function_name(parameter1=value1, parameter2=value2):
    # Code block
```

- If no argument is provided for a parameter, its default value is used.

```
Example: Function with Default Parameters
def greet_person(name="Guest"):
    print(f"Hello, {name}!")
```

In this example:

- If no name is provided, the default value **"Guest"** will be used.

```
Calling the Function:
greet_person()        # Output: Hello, Guest!
greet_person("Alice") # Output: Hello, Alice!
```

```
Output:
Hello, Guest!
Hello, Alice!
```

Keyword Arguments

In Python, you can specify the argument names explicitly when calling a function. This is useful when a function has many parameters, and you want to pass values in a specific order or provide clarity.

```
Syntax for Keyword Arguments:
function_name(parameter1=value1, parameter2=value2)
```

```
Example: Using Keyword Arguments
def greet_person(name, age):
    print(f"Hello, {name}. You are {age} years old.")
```

```
Calling the Function with Keyword Arguments:
greet_person(name="Alice", age=25)  # Output: Hello, Alice. You are 25 years old.
```

This approach makes the code more readable, especially when you have many parameters.

```
Output:
Hello, Alice. You are 25 years old.
```

Variable Scope in Functions

In Python, the **scope** refers to the region of the program where a variable is accessible. When you define a variable inside a function, it is local to that function and cannot be accessed outside of it. Variables defined outside functions are global and can be accessed anywhere.

```
Example: Local Variables
def my_function():
    local_variable = 10
    print(local_variable)

my_function()  # Output: 10
print(local_variable)  # Error: NameError: name 'local_variable' is not defined
```

In this case:

- **local_variable** is defined inside **my_function** and cannot be accessed outside it.

```
Example: Global Variables
global_variable = 20

def my_function():
    print(global_variable)

my_function()  # Output: 20
```
In this case:

- **global_variable** is defined outside the function and is accessible inside the function.

Variable-Length Arguments

In some cases, you might not know beforehand how many arguments will be passed to the function. Python provides a way to pass a variable number of arguments using ***args** (for non-keyword arguments) and ****kwargs** (for keyword arguments).

```
Using *args for Variable-Length Non-Keyword Arguments:
def my_function(*args):
    for arg in args:
        print(arg)
```

```
Calling the Function:
my_function(1, 2, 3, 4)  # Output: 1 2 3 4
```
Here, ***args** allows the function to accept any number of arguments.

```
Using **kwargs for Variable-Length Keyword Arguments:
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

```
Calling the Function:
my_function(name="Alice", age=25)  # Output: name: Alice, age: 25
```
In this example, ****kwargs** collects keyword arguments into a dictionary.

Recursive Functions

A **recursive function** is a function that calls itself in order to solve a problem. Recursion is useful when a problem can be broken down into smaller sub-problems of the same type. However, it is important to define a **base case** that stops the recursion to prevent infinite loops.

```
Syntax of a Recursive Function:
def function_name(parameters):
    # Base case: stop the recursion
    if condition:
        return result
    else:
        # Recursive call
        return function_name(modified_parameters)
```

Example: Factorial Function (Recursive)

The **factorial** of a number n is the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120. The factorial of n can be defined recursively as:

```
Recursive Function to Calculate Factorial:
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
Calling the Function:
result = factorial(5)
print(result)  # Output: 120
```

```
Output:
120
```

In this example:

- The function **factorial** calls itself with a smaller value of **n** until it reaches the base case **n == 0** or **n == 1**.

Important Points About Recursion:

- **Base Case**: Every recursive function must have a base case that terminates the recursion.
- **Recursive Case**: This is where the function calls itself, gradually breaking down the problem.
- **Stack Overflow**: Recursion uses the call stack, and excessive recursion without a base case or with too many recursive calls can cause a "stack overflow."

Conclusion

Functions are a powerful feature in Python that allow you to organize your code, make it reusable, and improve readability. By defining functions with parameters, return values, default values, and variable-length arguments, you can handle complex tasks efficiently. Understanding the scope of variables and how to work with both local and global variables will help you write more effective and modular code.

Additionally, **recursive functions** are a unique way of solving problems that can be broken down into smaller, similar sub-problems. They are a valuable tool for certain types of problems, such as mathematical computations or tree-like structures.

Key Takeaways:

- Use the **def** keyword to define functions in Python.
- Functions can accept parameters and return values.
- Default parameters and keyword arguments make functions more flexible.
- Local variables are only accessible within the function, while global variables are accessible everywhere.
- ***args** and ****kwargs** allow functions to accept a variable number of arguments.
- Recursive functions allow you to solve problems by breaking them down into smaller sub-problems.

Now that you have a solid understanding of functions and recursion, you can start using them to break your programs into smaller, manageable pieces !

Lambda Functions

Introduction

In Python, **lambda functions** are a type of anonymous function—functions that are defined without a name. They are concise and often used for short-term or temporary tasks. Lambda functions are particularly useful when you need a function for a brief operation and don't want to formally define it using the regular **def** syntax.

This chapter will introduce you to lambda functions, explain their syntax, and provide examples of how to use them in Python programming.

Syntax of Lambda Functions

A **lambda function** is defined using the **lambda** keyword, followed by one or more parameters, a colon, and the expression to be evaluated and returned.

```
Syntax:
lambda arguments: expression
```

- **lambda**: The keyword that tells Python you are defining a lambda function.
- **arguments**: The parameters that the function will accept (can be multiple).
- **expression**: A single expression that will be evaluated and returned. This is the result of the lambda function.

```
Example: Basic Lambda Function
# A simple lambda function that adds 2 to a number
add_two = lambda x: x + 2
print(add_two(3))  # Output: 5
```

In this example:

- **lambda x: x + 2** defines an anonymous function that adds 2 to the argument **x**.
- **add_two(3)** calls the function with **x = 3**, and the result is **5**.

```
Output:
5
```

Lambda Functions with Multiple Arguments

Lambda functions can take multiple arguments. These arguments are separated by commas within the parentheses.

```
Example: Lambda Function with Two Arguments
multiply = lambda x, y: x * y
print(multiply(4, 5))  # Output: 20
```

In this example:

- **lambda x, y: x * y** defines an anonymous function that multiplies two numbers **x** and **y**.
- **multiply(4, 5)** calls the function with **x = 4** and **y = 5**, and the result is **20**.

```
Output:
20
```

Using Lambda Functions with Built-in Functions

Lambda functions are commonly used with Python's built-in functions like **map()**, **filter()**, and **sorted()**. These functions allow you to apply the lambda function to a sequence of data or sort data efficiently.

Example: Using map() with Lambda

map() applies a given function to all items in an input list (or any iterable).

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

In this example:

- **map(lambda x: x ** 2, numbers)** applies the lambda function that squares each element in the **numbers** list.

```
Output:
[1, 4, 9, 16, 25]
```

Example: Using filter() with Lambda

filter() filters a sequence of items based on a condition defined by a lambda function.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

In this example:

- **filter(lambda x: x % 2 == 0, numbers)** filters the **numbers** list, keeping only even numbers.

```
Output:
[2, 4, 6]
```

Example: Using sorted() with Lambda

sorted() can be used to sort elements based on a custom key, which can be provided using a lambda function.

```
data = [("apple", 3), ("banana", 1), ("cherry", 2)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [('banana', 1), ('cherry', 2), ('apple', 3)]
```

In this example:

- **sorted(data, key=lambda x: x[1])** sorts the list of tuples based on the second element in each tuple.

```
Output:
[('banana', 1), ('cherry', 2), ('apple', 3)]
```

Lambda Functions for Simple Operations

Lambda functions are especially useful when performing simple operations where a full function definition would be overkill.

```
Example: Adding Two Numbers
add = lambda a, b: a + b
print(add(10, 20))  # Output: 30
```

```
Example: Checking Even or Odd
is_even = lambda x: x % 2 == 0
print(is_even(4))  # Output: True
print(is_even(7))  # Output: False
```

Advantages of Lambda Functions

Lambda functions offer several advantages:

- **Concise**: Lambda functions provide a more compact way to define simple functions.

- **Functional Style**: They are often used in functional programming techniques where functions are passed as arguments to other functions like **map()**, **filter()**, and **reduce()**.
- **Anonymous**: Since lambda functions are anonymous (i.e., they do not require a name), they are useful for short tasks where you don't want to define a full function.

Limitations of Lambda Functions

Despite their advantages, lambda functions have some limitations:

- **Single Expression**: A lambda function can only contain a single expression. It cannot contain multiple statements or perform multiple operations.
- **Readability**: While concise, lambda functions can sometimes reduce the readability of the code, especially for more complex operations.

When to Use Lambda Functions

Lambda functions are ideal for simple, short-term tasks where you need a function quickly without the overhead of defining a full function using **def**. Common use cases include:

- **Sorting**: Sorting data based on a custom key using **lambda** in **sorted().**
- **Filtering**: Filtering data using **lambda** with **filter().**
- **Mapping**: Transforming a collection of data using **lambda** with **map().**
- **Function Arguments**: Passing a simple function as an argument to other functions.

Conclusion

Lambda functions are a powerful feature in Python that allows you to define small, anonymous functions for short-term use. They are widely used in functional programming techniques and provide a clean, compact way to write simple functions.

Key Takeaways:

- A lambda function is a small, anonymous function defined using the lambda keyword.
- It can accept multiple arguments but can only contain a single expression.
- Lambda functions are commonly used with Python's built-in functions like **map(), filter(),** and **sorted().**
- They are concise and useful for simple, one-off operations but should be used judiciously to maintain code readability.

Now that you have a basic understanding of lambda functions, you can use them to simplify your code, especially for short-term tasks!

Arrays in Python

Arrays, also known as lists in Python, allow you to store multiple items in a single variable. They are useful for handling multiple objects or values in an ordered and repetitive manner. Unlike some other languages, Python doesn't have a specific native "array" type, but lists serve this purpose with great flexibility.

Creating a List

To create a list, use square brackets **[]** and separate the items with commas:

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# List of strings
words = ["Python", "is", "fun"]
```

Lists in Python can store elements of various types, even complex objects:

```
from pyb import LED
leds = [LED(1), LED(2), LED(3)]  # List of LED objects
```

Loops with Lists

Lists allow easy iteration over their elements using a for loop, simplifying repetitive tasks.

Practical Example: Controlling LEDs with a List

In this example, we use a MicroPython board to turn LEDs on and off. The LED objects are stored in a list, and a **for** loop allows controlling each LED easily:

```
from pyb import LED
import time

# Create a list of LEDs
leds = [LED(1), LED(2), LED(3)]

# Turn on each LED one by one for half a second
for led in leds:
    led.on()
    time.sleep_ms(500)
    led.off()
```

Manipulating Lists

Below are some of the most frequently used methods to manipulate lists:

append()

Adds an element to the end of the list.

```
leds.append(LED(4))  # Add an extra LED to the list
```

insert(pos, elem)

Inserts an element at the specified position.

```
leds.insert(1, LED(5))  # Inserts LED(5) in the second position
```

pop(pos)

Removes and returns the element at the specified position.

- **Positive index:** Refers to a position from the beginning of the list (index starts at 0).
  ```
  leds.pop(0)  # Removes the first element of the list
  ```
- **Negative index:** Refers to a position from the end of the list.
  ```
  leds.pop(-1) # Removes the last element of the list
  ```
- **Out of bounds:** If pos is beyond the valid indices of the list, Python raises an **IndexError**.

Note: The pop(pos) method removes and returns the item at the given index. If no index is provided, it removes the last item by default.

Here's an example where we remove the second LED from the list and control it separately:

```
# Remove and control the second LED
led_removed = leds.pop(1)  # Removes and returns LED 2
led_removed.on()
time.sleep_ms(1000)
led_removed.off()
```

In this example, **led_removed** contains the object corresponding to **LED(2)**, which can be manipulated separately.

remove(elem)

Removes the first occurrence of the specified element.

```
leds.remove(LED(2))  # Remove the LED corresponding to LED 2 from the list
```

clear():

Removes all elements from the list.

```
leds.clear()  # Empties the list
```

count(elem):

Counts how many times an element appears in the list.

```
print(leds.count(LED(1)))  # Counts occurrences of LED(1)
```

index(elem)

Returns the index of the first occurrence of the element.

```
position = leds.index(LED(2))  # Finds the position of LED(2)
```

extend(iterable)

Adds multiple elements to the end of the list.

```
leds.extend([LED(5), LED(6)])  # Adds multiple LEDs
```

reverse()

Reverses the order of elements in the list.

```
leds.reverse()  # Reverses the order of the list
```

sort()

Sorts the list in ascending order.

```
numbers.sort()  # Sorts numbers in ascending order
```

Note: The sort() method can take a **key** parameter for custom sorting and a **reverse=True** argument to sort in descending order.

Practical Example: Manipulating a List of LEDs

Test if an element if present in the list

```
if LED(1) in leds:
    print("LED 1 is in the list.")
```

Here's a detailed example showing how to use various list operations while controlling LEDs on a Momentum board:

```
from pyb import LED
import time

# Create a list of LEDs
leds = [LED(1), LED(2), LED(3)]

# Remove and control the first LED
first_led = leds.pop(0)  # Removes the first LED
first_led.on()
time.sleep_ms(500)
first_led.off()

# Remove and control the last LED
last_led = leds.pop(-1)  # Removes the last LED
last_led.on()
time.sleep_ms(500)
last_led.off()

# Add multiple LEDs and reverse their order
leds.extend([LED(3), LED(1)])
leds.reverse()

# Blink each LED in the reversed list
for led in leds:
    led.on()
    time.sleep_ms(300)
    led.off()

# Clear the list of all LEDs
leds.clear()
```

Conclusion

Lists are a powerful tool in Python, enabling efficient management of groups of objects or data. Understanding how to use list methods like **append()**, **insert()**, **pop()**, and **sort()** allows you to create flexible and dynamic code. These techniques are particularly useful for hardware projects.

Classes and Objects in Python

In Python, a class is a blueprint or template used to create objects. **Objects** are instances of a class, representing concrete entities with **attributes** (properties) and **methods** (actions). Classes and objects are essential for organizing code and managing complexity, particularly in MicroPython projects, such as controlling hardware or structuring data.

Methods

A method is a function defined inside a class that performs an action related to the class. Methods operate on the object's data and provide functionality.

```
class LED:
    def __init__(self, pin):
        from machine import Pin
        self.led = Pin(pin, Pin.OUT)
        self.state = False

    def on(self):
        self.led.value(0)
        self.state = True
        print("LED is ON")

    def off(self):
        self.led.value(1)
        self.state = False
        print("LED is OFF")

    def toggle(self):
        self.state = not self.state
        self.led.value(0 if self.state else 1)
        print(f"LED toggled to {'ON' if self.state else 'OFF'}")
```

In this example, **on**, **off**, and **toggle** are methods that allow the object **LED** to perform specific actions.

self

The keyword **self** represents the current instance of the class. It is used to access the attributes and methods of the object. All instance methods must include **self** as their first parameter.

class Example:

```
    def __init__(self, value):
        self.value = value  # Attribute tied to the current object

    def show_value(self):
        print(f"The value is {self.value}")  # Using self
```

The Special Method __str__()

The **__str__()** method is called when you try to display an object with **print()** or convert it to a string. It allows you to define a human-readable representation of the object.

```
class LED:
    def __init__(self, pin):
        self.pin = pin
        self.state = False

    def __str__(self):
        return f"LED on pin {self.pin}, state: {'ON' if self.state else 'OFF'}"
```

Usage:

```
led = LED("C0")
print(led)  # Output: LED on pin C0, state: OFF
led.state = True
print(led)  # Output: LED on pin C0, state: ON
```

Deleting a Method with del

In Python, you can dynamically delete a method or modify the behavior of objects. However, deleting methods is rarely needed and is generally not a common practice in MicroPython due to resource constraints.

```
class DynamicClass:
    def method1(self):
        print("Method1 exists.")

    def method2(self):
        print("Method2 exists.")

obj = DynamicClass()
obj.method1()    # Output: Method1 exists.
del obj.method1  # Deletes method1
# obj.method1()  # Raises AttributeError: method1 no longer exists
```

Deleting an Object with del

In MicroPython, you can use **del** to delete a reference to an object and free up memory. However, unlike standard Python, MicroPython does not implement the special **__del__()** method for performing actions when an object is deleted. Instead, memory management is handled by the garbage collector. If you need to clean up resources, you should implement explicit cleanup methods.

Example without __del__ in MicroPython

```
class LED:
    def __init__(self, pin):
        from machine import Pin
        self.pin = pin
        self.led = Pin(pin, Pin.OUT)
        self.state = False
        print(f"LED on pin {self.pin} initialized.")

    def on(self):
        self.led.value(0)
        self.state = True
        print(f"LED on pin {self.pin} is ON.")

    def off(self):
        self.led.value(1)
        self.state = False
        print(f"LED on pin {self.pin} is OFF.")

    def cleanup(self):
        """Explicit cleanup method for releasing resources."""
        self.off()
        print(f"LED on pin {self.pin} cleaned up.")

    def __str__(self):
        return f"LED(pin={self.pin}, state={'ON' if self.state else 'OFF'})"
```

Usage:

```
led = LED("C0")  # Initializes the LED object on pin C0 (RED)
print(led)       # Displays the object state
led.on()         # Turns the LED on
led.cleanup()    # Explicitly cleans up the object
del led          # Deletes the reference; no destructor is called
```

Key Takeaways for MicroPython

The **__del__()** method is not implemented. Use explicit cleanup methods, such as **cleanup()**, to release resources before deleting an object. Concepts like methods, **self**, and **__str__()** remain important for structuring your code and improving readability in resource-constrained environments.

Understanding Inheritance in Python

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a **child class**) to inherit attributes and methods from another class (called a **parent class**). This promotes code reuse and allows you to extend or modify the functionality of existing classes.

What is Inheritance?

In Python, inheritance allows a class to use the properties and behaviors of another class while introducing its own unique features.

The parent class is also known as the **base class**, and the child class is called the **derived class**.

Syntax of Inheritance

To create a child class, you include the parent class in parentheses after the name of the child class:

```
class Parent:
    # Parent class with common attributes and methods
    pass

class Child(Parent):
    # Child class that inherits from Parent
    pass
```

The child class automatically inherits all the methods and attributes of the parent class, and you can also add new ones or override the existing ones.

Basic Inheritance

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        print(f"{self.name} barks!")
```

Usage:

```
generic_animal = Animal("Generic Animal")
generic_animal.speak()  # Output: Generic Animal makes a sound.

dog = Dog("Buddy")
dog.speak()  # Output: Buddy barks!
```

Here:

- The Dog class inherits the __init__ method from Animal.
- The Dog class overrides the speak method to provide specific behavior for dogs.

Using super() to Extend Parent Behavior

The super() function allows the child class to call a method from the parent class. This is useful when you want to retain the parent class's behavior while adding new functionality.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

class Cat(Animal):  # Cat inherits from Animal
```

```
    def __init__(self, name, color):
        super().__init__(name)  # Call the parent class's __init__
        self.color = color

    def speak(self):
        super().speak()  # Call the parent class's speak method
        print(f"{self.name} meows!")
```

Usage:

```
cat = Cat("Whiskers", "black")
print(cat.color)  # Output: black
cat.speak()
# Output:
# Whiskers makes a sound.
# Whiskers meows!
```

Inheritance in MicroPython

Inheritance is also useful in MicroPython to create modular and reusable code for hardware devices. Here's an example:

```
class LED:
    def __init__(self, pin):
        from machine import Pin
        self.pin = Pin(pin, Pin.OUT)
        self.state = False

    def on(self):
        self.pin.value(0)
        self.state = True

    def off(self):
        self.pin.value(1)
        self.state = False

class BlinkingLED(LED):  # Inherits from LED
    def __init__(self, pin, duration):
        super().__init__(pin)
        self.duration = duration

    def blink(self):
        import time
        print(f"Blinking LED on pin {self.pin} for {self.duration} seconds.")
        self.on()
        time.sleep(self.duration)
        self.off()
```

Usage:

```
led = BlinkingLED("C0", 3)  # Blinking LED on GPIO pin C0 with a duration of 3 second
led.blink()
```

Key Points About Inheritance

1. **Reusability:** Inheritance lets you reuse code from a parent class in the child class, reducing duplication.
2. **Extensibility:** You can add new functionality or override existing methods in the child class.
3. **super():** The super() function allows you to call the parent class's methods explicitly.
4. **Multiple Inheritance:** Python supports multiple inheritance (a class can inherit from more than one parent), but it should be used carefully to avoid complexity.

Summary

Inheritance is a powerful tool that lets you create hierarchies of classes. It simplifies code by promoting reuse and reducing duplication. In MicroPython, it can help organize code for hardware components like sensors and actuators, making your programs cleaner and easier to maintain. By mastering inheritance, you unlock new possibilities for structuring efficient, scalable, and reusable code.

Understanding Iterators in Python

An **iterator** is a fundamental concept in Python that allows you to traverse through a sequence of data, such as a list, tuple, or string, one element at a time. Iterators provide a way to access the elements of a collection without exposing the underlying structure.

What is an Iterator?

An iterator is an object that implements two special methods:

1. **__iter__()**: Returns the iterator object itself.
2. **__next__()**: Returns the next item in the sequence. If no items are left, it raises a **StopIteration** exception.

Built-in Iterators

Python provides built-in iterators for collections like lists, tuples, and strings.

```
# Example: Using an iterator with a list
my_list = [1, 2, 3, 4]

# Get an iterator object
iterator = iter(my_list)

# Use the next() function to retrieve elements one at a time
print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
print(next(iterator))  # Output: 3
print(next(iterator))  # Output: 4

# If we call next() again, it will raise StopIteration
# print(next(iterator))  # Raises StopIteration
```

Creating a Custom Iterator

You can create your own iterator by defining a class that implements **__iter__()** and **__next__()**.

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self  # The iterator object returns itself

    def __next__(self):
        if self.current > self.end:
            raise StopIteration  # Stop iteration when the end is reached
        self.current += 1
        return self.current - 1
```

Usage:

```
counter = Counter(1, 5)
for num in counter:
    print(num)
# Output:
# 1
# 2
# 3
# 4
# 5
```

Infinite Iterators

Iterators can also represent infinite sequences. Be cautious when using them, as they do not stop automatically.

```
class InfiniteCounter:
    def __init__(self, start=0):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        self.current += 1
        return self.current
```

Usage:

```
infinite_counter = InfiniteCounter()
for num in infinite_counter:
    print(num)
    if num == 5:  # Break manually to prevent an infinite loop
        break
# Output:
# 1
# 2
# 3
# 4
# 5
```

Iterators in MicroPython

Iterators are especially useful in MicroPython for handling hardware components efficiently. For example, you can create an iterator to read sensor data multiple times.

```
class SensorReader:
    def __init__(self, sensor, readings):
        self.sensor = sensor
        self.readings = readings
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.readings:
            raise StopIteration
        self.count += 1
        return self.sensor.read()  # Example: Read sensor data

# Mock sensor example
class MockSensor:
    def read(self):
        import random
        return random.randint(0, 100)

sensor = MockSensor()
reader = SensorReader(sensor, 5)

for data in reader:
    print(f"Sensor reading: {data}")
# Output: 5 random sensor readings
```

Key Points About Iterators

1. **Lazy Evaluation:** Iterators compute values one at a time, making them memory efficient for large datasets.
2. **Custom Iterators:** You can create your own iterators to define custom traversal logic.
3. **for Loop Compatibility:** Iterators integrate seamlessly with Python's **for** loop.
4. **StopIteration:** Always raise **StopIteration** when there are no more items to iterate.

Summary

Iterators are a versatile tool for accessing and processing data in Python. They provide a memory-efficient way to work with sequences, especially in resource-constrained environments like MicroPython. By mastering iterators, you can write cleaner and more efficient code for both everyday programming tasks and hardware interaction.

Understanding Polymorphism in Python

Polymorphism is a key principle in object-oriented programming that allows a single interface to interact with objects of different types. This means methods, functions, or operations can behave differently depending on the object they are acting upon, making your code more flexible and reusable.

What is Polymorphism?

The term **polymorphism** means "many forms." In Python, polymorphism enables the same method name to work in different ways depending on the object that uses it. Polymorphism is most commonly achieved through:

1. **Method Overriding**: Subclasses provide their own implementation of methods defined in a parent class.
2. **Duck Typing**: Objects are used based on their behavior (methods and properties), not their type.

Polymorphism with Method Overriding in Inheritance

In this example, a parent class defines a general method, and subclasses override it to provide specific implementations.

```
class Vehicle:
    def description(self):
        return "This is a generic vehicle."

class Car(Vehicle):
    def description(self):
        return "This is a car. It has 4 wheels."

class Bike(Vehicle):
    def description(self):
        return "This is a bike. It has 2 wheels."
```

Usage:

```
vehicles = [Car(), Bike(), Vehicle()]
for vehicle in vehicles:
    print(vehicle.description())
# Output:
# This is a car. It has 4 wheels.
# This is a bike. It has 2 wheels.
# This is a generic vehicle.
```

Here:

- The description method in the Vehicle class is overridden in Car and Bike.
- Polymorphism allows the same method name to exhibit different behavior based on the object.

Polymorphism in Functions

A function can accept objects of different classes that share a common method.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
```

```
        return self.width * self.height

# Function leveraging polymorphism
def display_area(shape):
    print(f"The area is: {shape.area()}")
```

Usage:

```
shapes = [Circle(5), Rectangle(4, 6)]
for shape in shapes:
    display_area(shape)
# Output:
# The area is: 78.5
# The area is: 24
```

Polymorphism in MicroPython with Inheritance

In MicroPython, polymorphism is especially useful for handling hardware components that share similar behavior but have unique implementations.

```
class Device:
    def on(self):
        raise NotImplementedError("Subclasses must implement the 'on' method.")

    def off(self):
        raise NotImplementedError("Subclasses must implement the 'off' method.")

class LED(Device):
    def __init__(self, pin):
        from machine import Pin
        self.led = Pin(pin, Pin.OUT)

    def on(self):
        self.led.value(0)
        print("LED is ON.")

    def off(self):
        self.led.value(1)
        print("LED is OFF.")

class Buzzer(Device):
    def __init__(self, pin):
        from machine import Pin
        self.buzzer = Pin(pin, Pin.OUT)

    def on(self):
        self.buzzer.value(1)
        print("Buzzer is ON.")

    def off(self):
        self.buzzer.value(0)
        print("Buzzer is OFF.")

# Function leveraging polymorphism
def control_device(device, action):
    if action == "on":
        device.on()
    elif action == "off":
        device.off()
```

Usage:

```
devices = [LED(2), Buzzer(3)]
for device in devices:
    control_device(device, "on")
    control_device(device, "off")
# Output:
# LED is ON.
# LED is OFF.
# Buzzer is ON.
# Buzzer is OFF.
```

Here:

- The Device parent class defines a common interface with on and off methods.
- LED and Buzzer provide their own implementations of these methods.
- Polymorphism allows the control_device function to operate on any Device-type object.

Duck Typing in Python

Duck typing in Python emphasizes behavior over type. If an object implements the required method, it can be used regardless of its class.

```
class Bird:
    def fly(self):
        print("The bird is flying.")

class Airplane:
    def fly(self):
        print("The airplane is flying.")

# Function leveraging duck typing
def perform_flight(entity):
    entity.fly()
```

Usage:

```
entities = [Bird(), Airplane()]
for entity in entities:
    perform_flight(entity)
# Output:
# The bird is flying.
# The airplane is flying.
```

Key Benefits of Polymorphism

1. **Flexibility**: Functions and methods can work seamlessly with objects of different classes.
2. **Code Reusability**: Shared interfaces allow generic functions and methods to be reused.
3. **Extensibility**: Adding new object types with shared behavior doesn't require modifying existing code.

Summary

Polymorphism is a versatile feature of object-oriented programming that enhances flexibility, reusability, and scalability. By using a common interface across different classes, you can design cleaner, more maintainable code. In Python and MicroPython, polymorphism is especially useful for handling hardware components or modeling systems with shared but customizable behaviors.

Modules in Python

Modules are one of the most powerful features in Python. They allow you to organize your code into smaller, reusable components. A module is simply a file containing Python definitions and statements, such as functions, variables, or classes. By using modules, you can write cleaner and more maintainable code.

Why Use Modules?

1. **Code Organization**: Splitting your program into multiple files makes it easier to read and manage.
2. **Reusability**: Modules allow you to reuse code across multiple projects or programs.
3. **Namespace Management**: Modules provide separate namespaces to avoid naming conflicts.
4. **Access to Built-in Modules**: Python comes with a rich standard library of pre-built modules to extend your code's functionality.

Using a Module

You can import a module into your program using the **import** statement. Once imported, you can access its components with the syntax **module_name.component**.

Example 1: Importing a Built-in Module

```
import math

# Using the math module
radius = 5
area = math.pi * radius ** 2
print(f"The area of the circle is {area}")
# Output: The area of the circle is 78.53981633974483
```

Here: The math module provides mathematical functions like pi and sqrt.

Example 2: Importing Specific Components

Instead of importing the entire module, you can import only what you need.

```
from math import sqrt

print(sqrt(16))  # Output: 4.0
```

Creating Your Own Module

You can create a custom module by saving your Python code in a .py file.

Example: Custom Module

Create a file named **mymodule.py**:

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159
```

Now, import and use it in another Python file:

```
import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.PI)              # Output: 3.14159
```

MicroPython Modules

In MicroPython, you can use modules for working with hardware and other functionalities. MicroPython also supports custom modules for embedded systems.

Example: Using a MicroPython Built-in Module

```
from machine import Pin
```

```
led = Pin("C0", Pin.OUT)  # Initialize a pin for LED
led.value(1)              # Turn the LED on
```

Here: The machine module provides access to hardware components.

Custom MicroPython Module

Create a file named led_control.py:

```
from machine import Pin

class LED:
    def __init__(self, pin):
        self.led = Pin(pin, Pin.OUT)

    def on(self):
        self.led.value(0)
        print("LED is ON.")

    def off(self):
        self.led.value(1)
        print("LED is OFF.")
```

Use it in another script:

```
import led_control

my_led = led_control.LED("C0")
my_led.on()   # LED is ON.
my_led.off()  # LED is OFF.
```

Managing Module Imports

Python provides various ways to control module imports:

Renaming Modules: Use the as keyword to give an imported module a different name.

```
import math as m

print(m.sqrt(9))  # Output: 3.0
```

Importing All Components: Use * to import all components of a module.

```
from math import *

print(sin(0))  # Output: 0.0
```

Note: This is generally discouraged as it can lead to naming conflicts.

Checking Available Modules: Use the help('modules') command in the Python REPL to list available modules.

Key Benefits of Using Modules

1. **Reduces Redundancy**: Write code once and reuse it multiple times.
2. **Improves Maintenance**: Debugging and updating modular code is easier.
3. **Facilitates Collaboration**: Teams can work on different modules simultaneously.

Summary

Modules are essential for building scalable and maintainable Python programs. They allow you to organize code, reuse components, and leverage Python's extensive standard library. In MicroPython, modules play a critical role in accessing hardware and implementing custom features for embedded systems. By mastering modules, you can create efficient, clean, and reusable Python code for any project!

Using the math Module

The math module in MicroPython provides a set of mathematical functions that are useful for a wide range of applications, from simple arithmetic to more advanced scientific calculations. Since MicroPython is optimized for resource-constrained environments, the math module is lightweight and focused on essential mathematical operations.

Why Use the math Module?

The math module helps you:

1. Perform complex mathematical calculations easily.
2. Access common constants like π (pi) and e (Euler's number).
3. Use advanced functions like trigonometry, logarithms, and square roots.

To use the math module, you need to import it:

```
import math
```

Key Features of the math Module

Mathematical Constants

The module provides two commonly used constants:

- **math.pi:** The value of π (3.14159...).
- **math.e:** Euler's number (2.71828...).

Example:

```
import math

print("Value of pi:", math.pi)  # Output: 3.141592653589793
print("Value of e:", math.e)    # Output: 2.718281828459045
```

Basic Functions

- **math.sqrt(x):** Returns the square root of **x**.
- **math.pow(x, y):** Computes **x** raised to the power of **y**.
- **math.ceil(x):** Rounds **x** up to the nearest integer.
- **math.floor(x):** Rounds **x** down to the nearest integer.
- **math.fabs(x):** Returns the absolute value of **x**.

Example:

```
import math

print("Square root of 16:", math.sqrt(16))        # Output: 4.0
print("3 raised to the power 4:", math.pow(3, 4)) # Output: 81.0
print("Ceil of 3.7:", math.ceil(3.7))             # Output: 4
print("Floor of 3.7:", math.floor(3.7))           # Output: 3
print("Absolute value of -7:", math.fabs(-7))     # Output: 7.0
```

Trigonometric Functions

The math module includes trigonometric functions for angles in radians:

- **math.sin(x):** Computes the sine of **x**.
- **math.cos(x):** Computes the cosine of **x**.
- **math.tan(x):** Computes the tangent of **x**.
- **math.radians(deg):** Converts degrees to radians.
- **math.degrees(rad):** Converts radians to degrees.

Example:

```
import math

angle_deg = 45
angle_rad = math.radians(angle_deg)

print("Sine of 45°:", math.sin(angle_rad))    # Output: 0.7071067811865475
print("Cosine of 45°:", math.cos(angle_rad))  # Output: 0.7071067811865476
```

```
print("Tangent of 45°:", math.tan(angle_rad)) # Output: 1.0
```

Logarithmic and Exponential Functions

- **math.log(x):** Returns the natural logarithm (base e) of **x**.
- **math.log10(x):** Returns the logarithm of **x** to base 10.
- **math.exp(x):** Returns e raised to the power of **x**.

```
import math

print("Natural log of 10:", math.log(10))         # Output: 2.302585092994046
print("Log base 10 of 1000:", math.log10(1000))   # Output: 3.0
print("e^3:", math.exp(3))                        # Output: 20.085536923187668
```

Other Functions

- **math.modf(x)**: Splits x into its fractional and integer parts.
- **math.hypot(x, y)**: Computes the Euclidean distance from the origin to the point (x, y) (i.e., sqrt($x^2 + y^2$)).

```
import math

def hypot(x, y):
    return math.sqrt(x**2 + y**2)

print("Fractional and integer parts of 5.75:", math.modf(5.75))  # Output: (0.75, 5.0)
print("Hypotenuse of a triangle with sides 3 and 4:", hypot(3, 4))  # Output: 5.0
```

Application Example in MicroPython

Example: Calculating LED Brightness with PWM

You can use the **math** module to create a sine wave for controlling the brightness of an LED.

```
from pyb import Pin, Timer
import math
import time

# Configure the pin for PWM (LED red)
pin = Pin('C0', Pin.OUT)

# Initialize PWM using a timer
timer = Timer(1, freq=1000)  # Timer 1 at 1 kHz
pwm = timer.channel(1, Timer.PWM, pin=pin)

# Create a sine wave for brightness control
while True:
    for i in range(0, 360, 10):  # Degrees from 0 to 360
        # Calculate brightness as a sine wave scaled to 0-100%
        brightness = (math.sin(math.radians(i)) + 1) * 50  # Scale to 0-100
        pwm.pulse_width_percent(brightness)  # Set PWM duty cycle
        time.sleep(0.05)  # Wait for 50 ms
```

Limitations in MicroPython

While the math module in MicroPython provides many useful functions, it might not include advanced functions available in standard Python. Always consult the MicroPython documentation to check supported features.

Summary

The math module is a crucial tool for performing mathematical calculations in MicroPython. It offers:

1. Basic arithmetic functions.
2. Trigonometric calculations.
3. Logarithmic and exponential operations.
4. Constants like **π** and **e**.

Understanding how to use the math module can simplify your tasks and help you build efficient applications, especially in resource-constrained environments like MicroPython.

Handling Exceptions

In embedded application development, proper error handling is crucial for ensuring system reliability. MicroPython, like Python, uses exceptions to signal and handle errors. This chapter will explore how to use exceptions on a Pyboard to make your programs more robust.

What is an Exception?

An exception is an interruption in the normal flow of a program caused by an error. For example:

- A file not found.
- Division by zero.
- An attempt to access an unconnected peripheral.

MicroPython raises exceptions to signal these issues. You can catch and handle them to prevent your program from crashing.

Basic Exception Handling Structure

Exception handling in MicroPython follows this structure:

```
try:
    # Code that might raise an exception
    x = 10 / 0
except ZeroDivisionError:
    # Code executed if a "ZeroDivisionError" is raised
    print("Error: Division by zero is not allowed.")
finally:
    # Code that always executes, regardless of an exception
    print("Processing complete.")
```

Explanation of the blocks:

- try: Contains the code to monitor for exceptions.
- except: Contains the code executed if a specific error occurs.
- finally (optional): Contains the code executed no matter what, useful for releasing resources.

Common Exceptions in MicroPython on a Pyboard

Here are some common errors and how to handle them:

File Access Error

If you attempt to open a file that doesn't exist, an **OSError** will be raised.

```
try:
    with open("nonexistent_file.txt", "r") as file:
        content = file.read()
except OSError:
    print("Error: The file is not accessible or does not exist.")
```

Device Error

If you attempt to read from an unconnected sensor, an exception might be raised.

```
from pyb import ADC

try:
    adc = ADC(1)  # ADC channel 1
    value = adc.read()
    print("ADC Value:", value)
except ValueError:
    print("Error: The specified ADC channel is invalid.")
```

Memory Error

On embedded systems, excessive memory allocation may raise a **MemoryError**.

```
try:
    data = [0] * (10**6)  # Attempt to allocate a large array
except MemoryError:
    print("Error: Not enough memory to complete this operation.")
```

Raising Custom Exceptions

You can raise your own exceptions using **raise**.

```
def divide(a, b):
    if b == 0:
        raise ValueError("Error: Division by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)
```

Best Practices

1. Catch only specific exceptions
   Avoid catching all exceptions with a generic **except:** block, except for temporary debugging.
2. Release resources in **finally**
   Use this block to close files or disable devices.
3. Provide clear error messages
   Clear messages help identify the type and source of an error during debugging.
4. Test your program with error scenarios
   Simulate errors to verify that your exception handling works as intended.

Complete Example

Here is a complete example combining multiple concepts for a Pyboard program:

```
from pyb import LED

def turn_on_led(number):
    if number not in range(1, 4):
        raise ValueError("Invalid LED number. Choose between 1 and 3.")
    led = LED(number)
    led.on()

try:
    turn_on_led(4)  # Invalid number
except ValueError as e:
    print("Error detected:", e)
finally:
    print("Program finished.")
```

Conclusion

Exception handling is a powerful tool for making your MicroPython applications more reliable, even on constrained platforms like the Pyboard. By following best practices and anticipating potential errors, you can prevent unexpected failures and improve the overall quality of your projects.

Using input()

In MicroPython, the **input()** function allows programs to receive textual input from a user. This function is particularly useful for interactive scripts running in a terminal. While it is not typically used in standalone embedded systems, it is a valuable tool when your MicroPython board is connected to a computer via a serial terminal. In this chapter, we will explore how to use **input()** in MicroPython, its limitations, and some practical examples.

What is the input() Function?

The **input()** function in MicroPython is used to capture user input as a string from the serial terminal. When called, it pauses the program and waits for the user to type something and press Enter. The entered text is then returned as a string.

Syntax:

```
user_input = input("Prompt message: ")
```

"Prompt message": A string displayed to the user to indicate what kind of input is expected.

user_input: The variable that stores the entered text.

Basic Usage of input()

Here is a simple example of using **input()** to get user input:

```
# Asking for the user's name
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

Example Output:

```
Enter your name: Alice
Hello, Alice!
```

In this example, the program waits for the user to type their name and press Enter. The entered name is then printed back to the user.

Converting User Input

The **input()** function always returns a string. To use the input as a number or in calculations, you must convert it to the appropriate type (e.g., **int** or **float**).

Example: Converting Input to an Integer

```
# Asking for a number and performing calculations
number = int(input("Enter a number: "))
print(f"The square of {number} is {number ** 2}")
```

Example Output:

```
Enter a number: 4
The square of 4 is 16
```

Handling Non-Numeric Input

If the user enters a non-numeric value when converting input, a **ValueError** will be raised. You can handle this with a **try-except** block:

```
try:
    number = int(input("Enter a number: "))
    print(f"The square of {number} is {number ** 2}")
except ValueError:
    print("Error: Please enter a valid number.")
```

Using input() for Menu Systems

You can use **input()** to create simple text-based menu systems. This is useful for debugging or providing interactive controls.

Example: A Simple Menu

```
while True:
    print("\nMenu:")
    print("1. Say Hello")
    print("2. Add Two Numbers")
    print("3. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        print("Hello!")
    elif choice == "2":
        a = int(input("Enter the first number: "))
        b = int(input("Enter the second number: "))
        print(f"The sum of {a} and {b} is {a + b}")
    elif choice == "3":
        print("Exiting the program. Goodbye!")
        break
    else:
        print("Invalid option. Please try again.")
```

Example Output:

```
Menu:
1. Say Hello
2. Add Two Numbers
3. Exit
Choose an option: 1
Hello!

Menu:
1. Say Hello
2. Add Two Numbers
3. Exit
Choose an option: 3
Exiting the program. Goodbye!
```

Limitations of input() in MicroPython

1. **Terminal Dependency:** The **input()** function requires the board to be connected to a serial terminal. It cannot be used in standalone applications without a terminal.
2. **Blocking Nature:** The program pauses execution until the user provides input. This can make it unsuitable for real-time applications.
3. **No Timeout:** The **input()** function does not support a timeout. If no input is provided, the program will wait indefinitely.

Alternatives to input()

For standalone systems without a terminal, consider using other methods for user interaction, such as:

- Physical buttons connected to GPIO pins.
- Serial communication via **UART**.
- External peripherals like keypads.

Practical Example: Combining input() and GPIO Control

The following example demonstrates how to use **input()** to control an LED connected to Momentum board.

```
from pyb import LED

# LED Control Program
led = LED(1)  # Use LED 1 on the Momentum board

while True:
    print("\nLED Control:")
    print("1. Turn ON the LED")
    print("2. Turn OFF the LED")
    print("3. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        led.on()
        print("LED is ON.")
    elif choice == "2":
        led.off()
        print("LED is OFF.")
    elif choice == "3":
        print("Exiting the program. Goodbye!")
        break
    else:
        print("Invalid choice. Please try again.")
```

Best Practices for Using input()

1. **Validate User Input:** Always check that the user has entered valid data to avoid errors.
2. **Provide Clear Prompts:** Use descriptive messages to guide the user on what input is expected.
3. **Handle Exceptions:** Use **try-except** blocks to manage errors gracefully, such as invalid conversions.
4. **Avoid Overuse in Real-Time Systems:** Since **input()** blocks the program, it may not be suitable for time-sensitive tasks.

Conclusion

The **input()** function is a powerful tool in MicroPython for receiving user input when working with a serial terminal. It is ideal for debugging, creating interactive scripts, or simple menu-driven applications. However, its limitations make it less suitable for standalone or real-time embedded systems. By combining **input()** with other MicroPython features, you can create versatile and interactive applications.

File Manipulation in MicroPython

File manipulation is a core functionality in MicroPython, enabling you to store, retrieve, and modify data on a storage medium like an SD card or the internal filesystem. This chapter provides a comprehensive guide to handling files in MicroPython, covering basic operations, common pitfalls, and best practices.

The Filesystem in MicroPython

MicroPython includes a minimal filesystem that allows you to work with files similarly to Python on desktop systems. Common storage options include:

- **Internal flash memory**: Used for storing scripts and configuration files.
- **SD card**: For additional or removable storage.

On a Momentum board, the filesystem is mounted automatically, and you can access it via a terminal or programming.

Opening and Closing Files

Files in MicroPython are opened using the open() function. You specify the file path and mode (e.g., read, write).

Syntax:

```
file = open("filename.txt", mode)
```

Common Modes:

- **'r'**: Read-only (default).
- **'w'**: Write (overwrites the file if it exists).
- **'a'**: Append (adds to the file without overwriting).
- **'r+'**: Read and write.
- **'b'**: Binary mode (e.g., 'rb' for reading binary data).

Example: Opening and Closing a File

```
file = open("example.txt", "w")
file.write("Hello, MicroPython!")   # Write to the file
file.close()                        # Always close the file
```

Best Practice: Use a **with** statement to handle files, as it automatically closes the file when done.

```
with open("example.txt", "w") as file:
    file.write("Hello, MicroPython!")
```

Writing to a File

You can write data to a file using the **write()** or **writelines()** methods.

Example: Writing Text to a File

```
with open("data.txt", "w") as file:
    file.write("Line 1\n")
    file.write("Line 2\n")
```

Example: Writing Multiple Lines

```
lines = ["Line A\n", "Line B\n", "Line C\n"]
with open("data.txt", "w") as file:
    file.writelines(lines)
```

Reading from a File

You can read data from a file using **read()**, **readline()**, or **readlines()**.

Example: Reading Entire Content

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

Example: Reading Line by Line

```
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())  # Remove newline characters
```

Example: Reading a Single Line

```
with open("data.txt", "r") as file:
    first_line = file.readline()
    print(first_line)
```

Appending to a File

Use the 'a' mode to append content to an existing file.

Example: Appending Data

```
with open("data.txt", "a") as file:
    file.write("Appended Line\n")
```

Checking if a File Exists

Before accessing a file, it's good practice to check if it exists to avoid errors. Use the os module.

Example: Checking File Existence

```
import os

try:
    os.stat("data.txt")  # Raises an OSError if the file does not exist
    print("File exists!")
except OSError:
    print("File does not exist.")
```

Deleting or Renaming Files

The os module provides methods to manage files.

Example: Deleting a File

```
import os

os.remove("data.txt")  # Deletes the file
```

Example: Renaming a File

```
import os

os.rename("old_name.txt", "new_name.txt")  # Renames the file
```

Working with Directories

The os module also allows directory manipulation.

Example: Listing Files in a Directory

```
import os

files = os.listdir()  # Lists all files and directories
print(files)
```

Example: Creating and Removing Directories

```
os.mkdir("new_folder")  # Creates a new directory
os.rmdir("new_folder")  # Removes an empty directory
```

Binary File Operations

To work with non-text data (e.g., images or sensor logs), open files in binary mode.

Example: Writing and Reading Binary Data

```
# Writing binary data
with open("binary.dat", "wb") as file:
    file.write(b'\x01\x02\x03\x04')  # Write bytes

# Reading binary data
with open("binary.dat", "rb") as file:
    data = file.read()
    print(data)  # Output: b'\x01\x02\x03\x04'
```

Handling File Errors

File operations can fail due to reasons like missing files or insufficient storage. Use try-except blocks to handle such errors gracefully.

Example: Handling Errors

```
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except OSError as e:
    print(f"File error: {e}")
```

Practical Example: Logging Data to a File

Here's a practical example where data from a sensor is logged to a file.

```
from pyb import ADC, delay

# Configure ADC
adc = ADC("A1")

# Log data to a file
with open("sensor_log.txt", "w") as file:
    for _ in range(10):             # Collect 10 samples
        value = adc.read()          # Read ADC value
        file.write(f"{value}\n")    # Write value to the file
        delay(1000)                 # Wait 1 second
print("Data logging complete.")
```

Best Practices for File Manipulation

1. **Use with Statements**: Ensures proper file closure.
2. **Validate File Operations**: Check for file existence or write permissions.
3. **Minimize File Access**: Reduce unnecessary reads/writes to prolong flash memory life.
4. **Handle Errors Gracefully**: Use exception handling for robust applications.
5. **Clean Up**: Delete or move temporary files after use.

Conclusion

File manipulation in MicroPython allows you to manage data efficiently on embedded systems. From basic read/write operations to advanced logging and binary handling, these capabilities are essential for many applications. By following best practices, you can create robust programs that handle files reliably, even in resource-constrained environments.