Receiver Notes and Hints

Contents

Portfolio

Process	Prior Knowledge	After Task
Using a micro:bit	❌	✓
Importing Modules		
`if` Statements		
`if-elif-else` Statements		
`while` Loops		
`for` Loops		
`continue` Keyword		
Functions and `return`-ing from functions		
User Input		

Structure Guide: Indentation

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
Python uses indentation to indicate a *block* of code.

> Indentation refers to the spaces at the beginning of a code line.

In this code, Python can see that the `print` statement *belongs to* the `if` statement and will only be run if the *condition* `5 > 2` is `True`.
The second `print` statement isn't part of the block, and so will run even if the `if` statement condition returns `False`.

```
if 5 > 2:
    print("Five is greater than two!")
print("This line will run even if the condition returns false!")
```

Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")
```

```
IndentationError: expected an indented block
```

Traditionally, 4 spaces are used as indentation, however you can use as many as you prefer to **as long as you are consistent**. You have to use the same number of spaces in the same block of code, otherwise Python will give you an error.

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

```
IndentationError: unexpected indent
```

python.microbit.org/v/3/

Template

Here is the code you should start with:

```
from microbit import *

while True:
    ...
```

1. Importing Modules

Start by importing all the modules / dependencies you will need to complete the task; you want to use the `radio` module.

Enable the radio on the micro:bit.

> Reference > Radio > On and Off

The radio channel needs to be set to the same as the micro:bit transmitting the message. For now set this to any number between 0 and 255.

> Reference > Radio > Groups

2. Receiving a Message

Next comes the control loop. The control loop is the `while True` loop that will run infinitely whilst the device is powered.

Remove the ellipsis placeholder. Similar to using three dots in English to omit content, you can use the ellipsis in Python as a placeholder for unwritten code.

When a button is pressed, attempt to `receive` a message. Assign this to button `B`.

> Reference > Buttons > Button was pressed
> Reference > Radio > Receive a message

If you don't receive a message with content, `continue` to the next iteration of the `while` loop.
The *logical operator* `not` can be used here to determine if the variable has no value: `if not variable_name:`

> The keyword `continue` can be used here to skip the remaining code in the loop and move on to the next iteration to try again.

3. Display the Message

If you have received a message, output it to the console and / or to the LED panel.

> Reference > Display > Scroll

Written in *pseudocode*, this section of the code will look like:

```
while true
    if button b was pressed
        receive message
        if message has no value
            continue to next iteration
        output message
```

4. Writing a function

A function is a "chunk" of code that you can use over and over again, rather than writing it out multiple times. Functions enable programmers to break down or *decompose* a problem into smaller chunks, each of which performs a particular task. The basic structure for a function in Python is as so:

```
def function_name(arg1, arg2):
    ...
```

> Reference > Functions

Upon testing you find that the `scroll` method doesn't give you enough time to record the message being output by the micro:bit. Replace this line with a function call to `display_message`, passing in the `message` as an argument.
If you tried to run the program now an *error* would occur as we have not yet provided a *definition* for this function!

Back towards the start of your program, define the function you have just called.
In this function, *iterate* through each `letter` in `message` using a `for` loop, and use the `display.show()` method to output this letter to the LED panel.

> Reference > Display > Show
> Reference > Loops > For Loops > Letters
> Remember here that `message` is a *string*!

Use the micro:bit's `sleep(ms)` function to create a delay between showing each character, using an appropriate delay as the argument.
You may also want to use the `display.clear()` function after each letter to ensure there is a clear distinction between repeating letters.

5. Maintainability and Comments

Comments in Python is the inclusion of short descriptions along with the code to increase its readability. A developer uses them to record their thought process while writing the code. It explains the basic logic behind why a particular line of code was written.

In Python, comments are written using a hash symbol, `#`.

For example, you could comment your code like so:

```
# Open file explorer window for user to select CSV Data
file = fd.askopenfilename(
    filetypes=[("CSV files", "*.csv")], title="Set input .csv file" )
# Read the selected CSV data into a 'Pandas' dataframe
data = pd.read_csv(file, header=0)

# Calculate the rolling average of the data.
# This smoothens the data to account for noise; signal processing.
ROLLER = 4
rolling = data.rolling(ROLLER).mean()
rolling = rolling.dropna().reset_index()
```

Notice how not every line has a comment. Avoid self-explanatory comments such as:

```
ROLLER = 4 # Sets `ROLLER` to 4
```

Add some *descriptive* comments to your code.

> This would help you or another developer understand why the code was written in a certain way if you had to come back to this project in a number of years time when the original processes had been forgotten.

6. Decoding the message!

Use the 'Documentation Hunt' worksheet to follow the program flow and determine the output of the code provided. This should be done on the paper provided, not written into this script.

The output of this sheet will give you the `group` to use in the `radio.config()` command as well as another value which will be useful...

Encrypted Message: _ _ _ _ _ _ _ _ _ _ _ _ !
Decrypted Message: _ _ _ _ _ _ _ _ _ _ _ _ !

```
import all from microbit
import radio module

enable the radio
set the radio group

define 'display_message' function with argument 'message'
    for each letter in the message
        display the letter
        pause
        clear the screen
        pause

while true (control loop)
    if button b was pressed
    receive a message
    if message is empty
        continue to next loop iteration

    display the message with 'display_message' fuction
```

Extension - Manual Decipher vs. Software Decipher

> This extension assumes some level of confidence in Python. It can also be completed on paper if preferred.

What does `"ifmmp xpsme!"` mean?
The message is encrypted using a Caesar Shift!

Whilst it is certainly possible, it will become very tedious if you had to keep decrypting these messages yourself. What if the messages you needed to decrypt were longer, or you had multiple to decrypt? You could use an *automation script* to do this for you!

A Caesar Shift Decoder can be written here as a function.

For a Caesar Shift we have two arguments to pass in:

- The Encrypted Message
- The Magnitude of the Shift

> `"a"` --> `"b"` would have a shift magnitude of `1`.

In psuedo-code this function would look like:

```
define 'decode' with parameters 'encrypted' and 'shift'
    initialise empty string called 'decoded'
    initialise string 'ALPHABET'
    initialise string 'NUMBERS'
    for character in encrypted message
        if character is in the ALPHABET
            find position of the character in 'ALPHABET'
            add the letter at index 'value - shift' to 'decoded'
        else if character is in the 'number' string
            find position of the character in 'NUMBERS'
            add the letter at index 'value - shift' to 'decoded'
        else
            # assume value is punctuation, doesn't need to be shifted.
            add the character to 'decoded'
    return 'decoded' to main program
```

```
decoded = ""
ALPHABET = "abcdefghijklmnopqrstuvwxyz"
NUMBERS = "0123456789"
```

> Constants are unchanging variables. They are written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.
> You may need to see the notes on the `.find()` method towards the back of this document.
> Alternatively, you could find *documentation* explaining the usage of this method online!

You should then call the `decode` function from the control loop as an argument of `display_message`. Add this to the area where you receive the message when a button is pressed.

> Functions can be arguments to other *functions* and *procedures*.
>
> ```
> function_one( function_two(f2_arg_one, f2_arg_two) )
> ```

Add functionality to set the magnitude of the `shift` using the micro:bit. This could be done with button presses which increment a *counter variable*. Assign this to button `A`. Don't allow this value to be greater than 9.

```
initialise shift to 0
while true (control loop)
    ...
    on button press
        add 1 to shift
        if shift bigger than 9
            reset shift to 0
```

Encrypted Message: _

Decrypted Message: _

String `.find()` Method

The `.find()` method finds the first occurrence of the specified value.
The `.find()` method returns `-1` if the value is not found.

```
company = "Collins Aerospace"
print( company.find("A") ) ## prints "8"
```

Indexing Into a List

Use square brackets after the list name to index into a list. Python uses zero-indexing for lists.

```
ALPHABET = "abcdefghijklmnopqrstuvwxyz"
print(ALPHABET[0]) # prints "a"
print(ALPHABET[1]) # prints "b"
print(ALPHABET[-1]) # prints "z"
```