Python for Absolute Beginners

Tutorial Workbook

Al Sweigart (last name rhymes with “why dirt”)

al@inventwithpython.com

https://inventwithpython.com/pyconus2026

Mu, the Code Editor App

Mu is the free software that is like a word processor app for code. It includes the Python interpreter that runs your Python code. Download and install it from https://codewith.mu

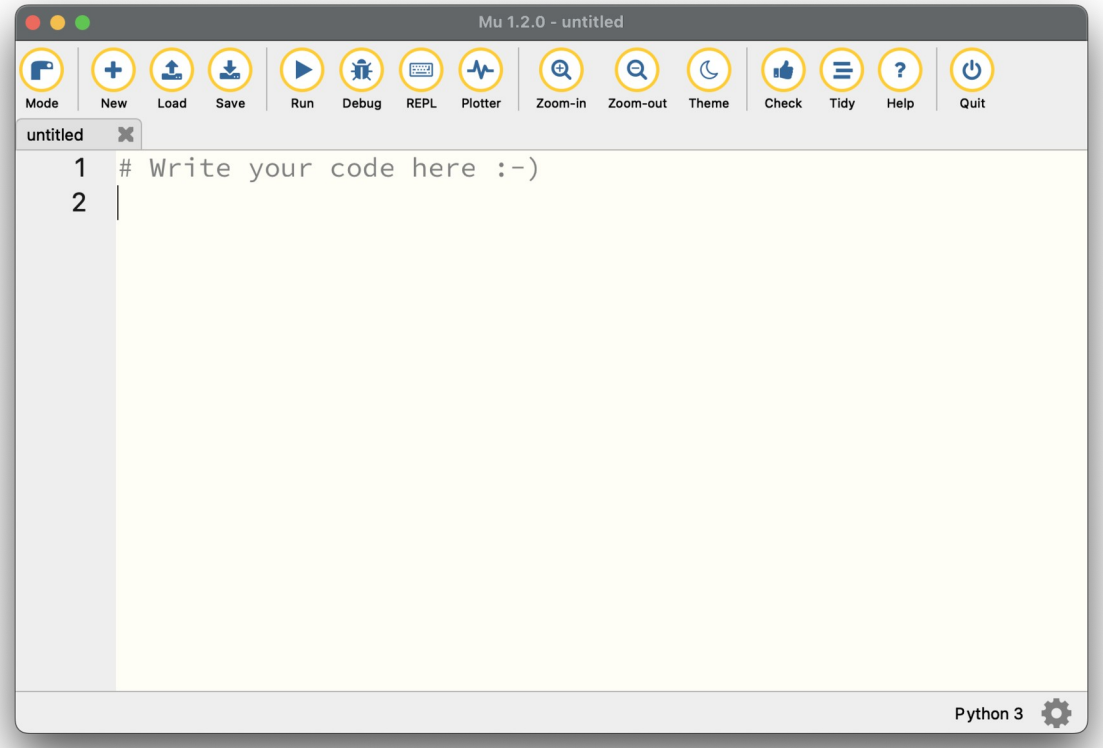
Mu runs on the Windows, macOS, and Linux operating systems.

On the Download page, click the Download button for your operating system, then open the downloaded installer file.

The installer file for Windows is *MuEditor-win64-1.2.0.msi*

The installer file for macOS is *MuEditor-OSX-1.2.0.dmg*

When you run the Mu code editor app, it looks like this:



The big text box is where you type your Python source code. This is no menu, just 15 buttons:

- **Mode** – You can ignore this; always keep it on “Python 3” mode
- **New, Load, Save** – Create, open, or save the source code file in the editor
- **Run, Debug** – Run the program in the editor, or run it under the debugger
- **REPL** – Shows/hides the Read-Evaluate-Print-Loop or interactive shell pane
- **Zoom-In, Zoom-Out** – Increase or decrease the font size
- **Theme** – Switch between light mode and dark mode color schemes for the editor
- **Check** – Check for any typos and syntax errors in your code
- **Tidy** – Automatically format your code with proper whitespace
- **Help** – Open the web page at https://codewith.mu/en/help/1.2
- **Quit** – Quit the Mu editor

Example Program: Hello, World!

When you run the program, it will look like this (**bold text** is entered by the user):

```
Hello, world!

What is your name?
>Albert
It is good to meet you, Albert
```

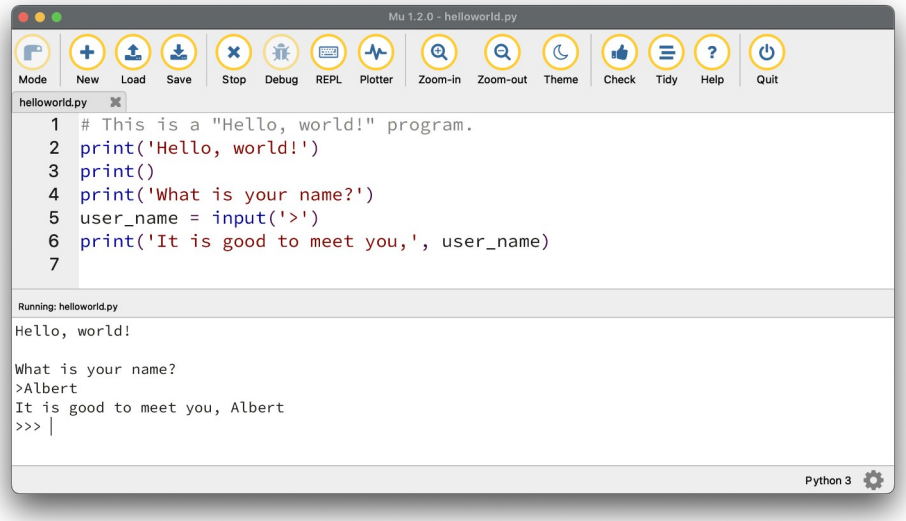
Open the Mu editor. Save the *untitled* tab as *hello.py*.

Erase the `# Write your code here :-)` text.

Type the following 6-line program in the *file editor*. (Do NOT type the line numbers, they're just in this workbook for your reference. You can also skip the gray text after the # hash tags.)

```
1 # This is a "Hello, world!" program.
2 print('Hello, world!')  # print() function call w/ a string arg
3 print()  # print() function call w/ zero arguments
4 print('What is your name?')
5 user_name = input('>')  # input() func call, assigns to variable
6 print('It is good to meet you,', user_name)  # two args
```

Click the **Run** button to run the program. When you see the > prompt, type your name and press Enter/Return. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done. (Most code editors don't have a Stop button and simply stop at the end of the program.)

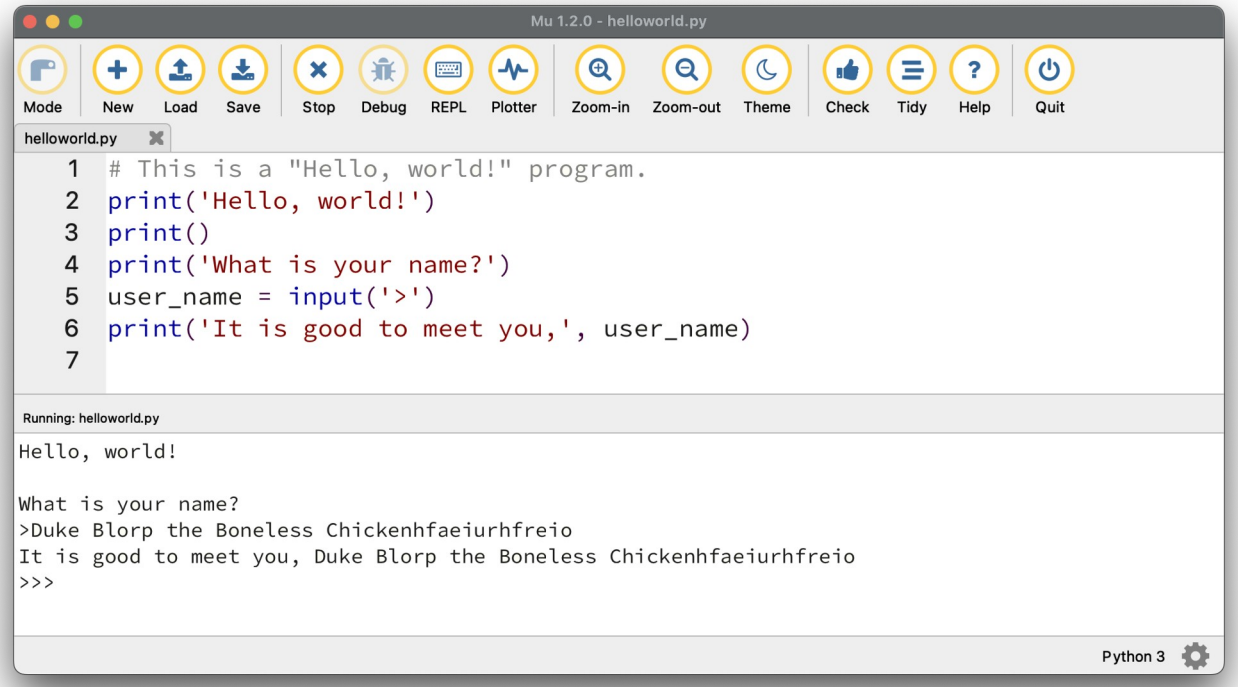


Run the “Hello, World!” Program Again With Silly Input

The program doesn’t understand the text we tell it to print (on the screen, not the printer) or the text input from the keyboard. You can enter any text! Click the Run button again and type a silly name and press Enter/Return:

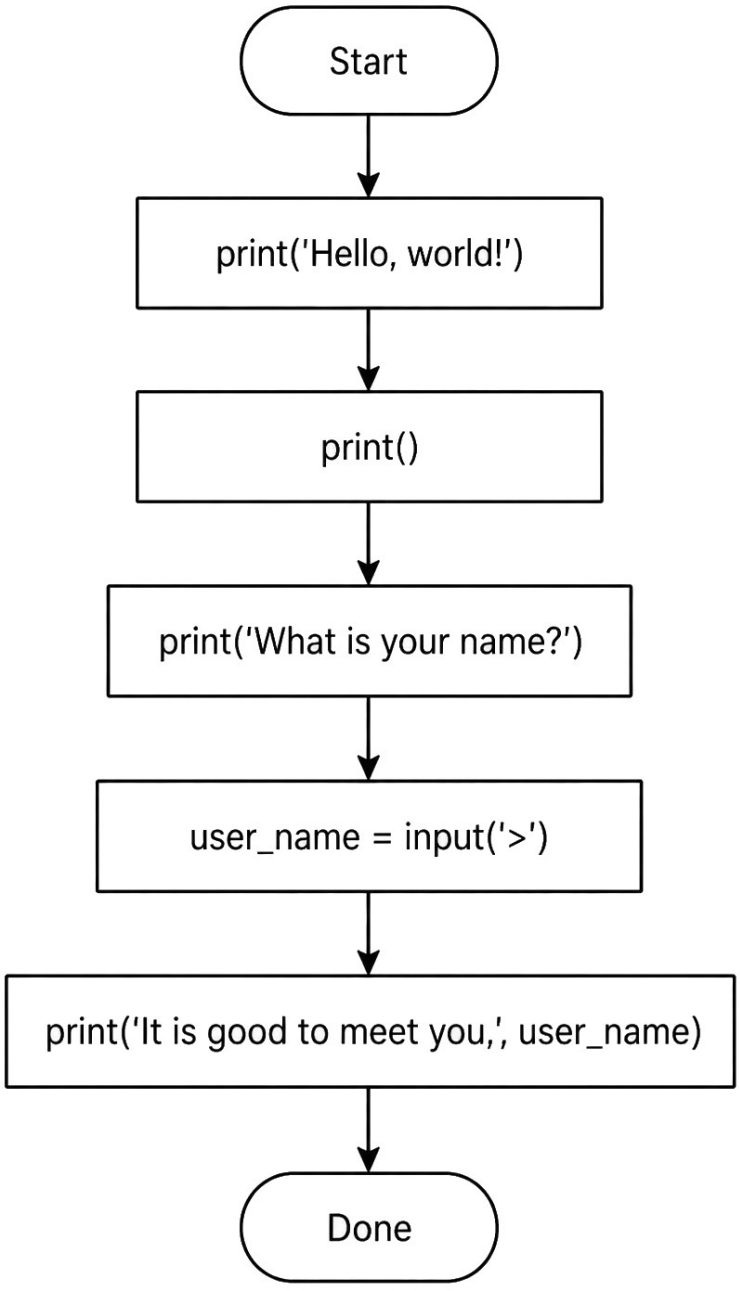
```
Hello, world!

What is your name?
>Duke Blorp the Boneless Chickenhfaeiurhfreio
It is good to meet you, Duke Blorp the Boneless Chickenhfaeiurhfreio
```

Click the **Stop** button when the program is done.



Flow Chart for the “Hello, World!” Example Program

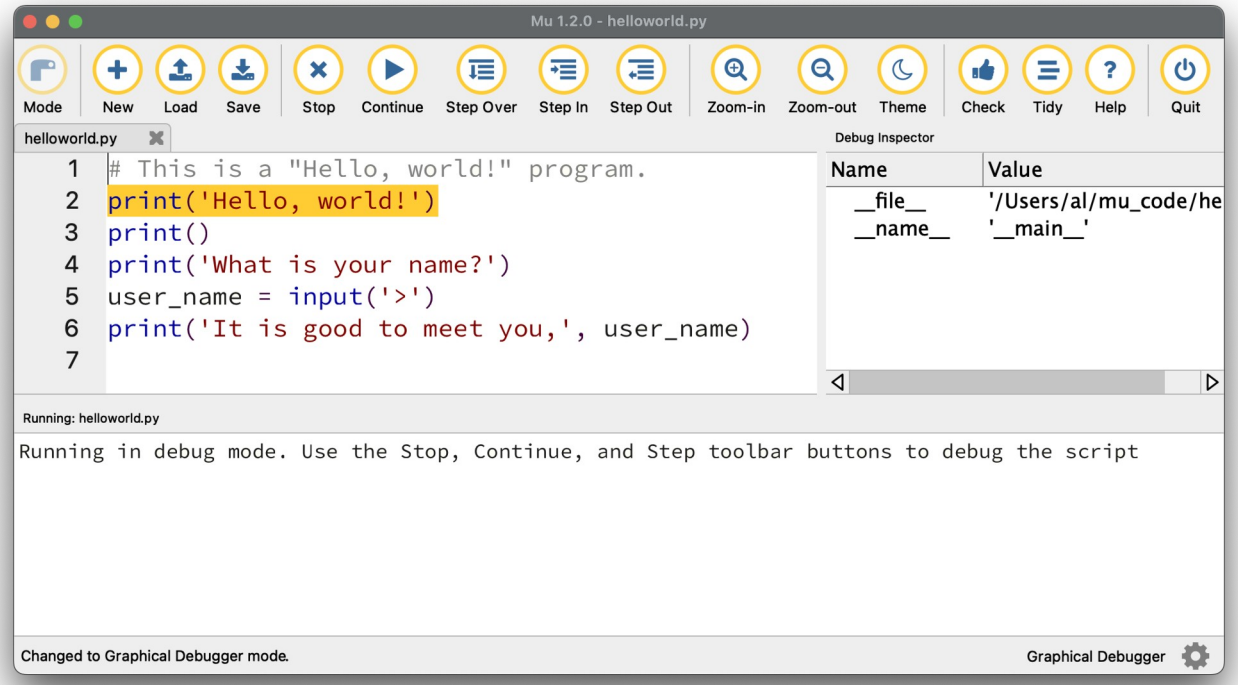
The line of code in the program that is currently being run is called the *execution*. The execution in this program starts at the top and moves down. Trace the path of the execution from Start to Done with your finger. You can imagine a flow chart for this program looking like this:



Run the “Hello, World!” Program Under the Debugger

The debugger runs your program one instruction at a time and shows you the data stored in variables (such as `user_name`).

1. Press the **Debug** button to start running the program “under the debugger”.



Notice the *Debug Inspector* pane on the right. (You might want to enlarge this by dragging the vertical bar to the left.) The Debug Inspector lists all the variables and the data they contain. (You can ignore the `__file__` and `__name__` variables which are a part of all programs.)

The `print('Hello, world!')` line is highlighted and the debugger is in a paused state.

The highlighted line of code will be run the next time you press the Step Over button.

(If you press the Step In, Step Out, or Continue buttons by accident, press the **Stop** button and then start over again by pressing the **Debug** button.)

2. Press the **Step Over** button. The `print('Hello, world!')` line of code runs, makes Hello, world! appear in the output pane, and the next line, `print()` is now highlighted. The debugger is paused again.

3. Press the **Step Over** button three more times. Then click in the output pane, type a name, and press Enter/Return.

4. Press the **Step Over** button to finish running the rest of the `user_name = input('>')` line. Notice that a new variable `user_name` has appeared in the Debug Inspector pane.

5. Press the **Continue** button to run the rest of the program at normal speed.

6. Press the **Stop** button to close the output pane.

“Calculator Stuff” in the REPL / Interactive Shell

Expressions are a type of instruction made up of values and operators. An expression always evaluates down to a single value. Press the **REPL** button, type the following, and write down the answer. Add a tally mark for each time addition, subtraction, multiplication, or division is done.

`In [x]: 2 + 2`
 	
`In [x]: 5 - 3`
 	
`In [x]: 3 - 5`
 	
`In [x]: 0+0+0`
 	
`In [x]: -3 + 2`
 	
`In [x]: 3 * 3`
 	
`In [x]: 3 x 3  # (Error! X isn’t an op.)`
 	
`In [x]: 2 + 3 * 4  # (Note order of ops)`
 	
`In [x]: (2 + 3) * 4`
 	
`In [x]: 12 / 4`
 	
`In [x]: 75 / 100`
 	
`In [x]: 14 / 4`
 	
`In [x]: (5 - 1) * ((7 + 1) / (3 – 1))`
	ADDITION:
	
	SUBTRACTION:
	
	MULTIPLICATION:
	
	DIVISION:

String Concatenation and String Replication in the REPL / Interactive Shell

Text values are called strings and begin and end with a single quote when written as code. You can also use strings in expressions with the concatenation operator (+) and replication operator (*) to create new strings. Press the **REPL** button, type the following, and write down the answer:

```
In [x]: 'Hello' + 'World'


In [x]: 'Hello' + ' ' + 'World'


In [x]: '' + 'Hello'


In [x]: 'A' + 'B' + 'C'


In [x]: 'AB' + 'C'


In [x]: '2' + '2'


In [x]: '2' + 2  # Causes an error!


In [x]: planet = 'Earth'
In [x]: 'Welcome to ' + planet


In [x]: planet


In [x]: 'Hello' * 5


In [x]: 5 * 'Hello'


In [x]: 'Hello' * 5.0  # Error!


In [x]: '*' * 1000000
```

Boolean Values and the Comparison Operators in the REPL / Interactive Shell

There are two Boolean values: `True` and `False`. There are six comparison operators: `== != < <= > >=`. Press the **REPL** button, type the following, and write down the answer:

```
In [x]: 2 == 2


In [x]: 2 != 2


In [x]: 9999 == 2


In [x]: 2 + 2 == 4


In [x]: 4 < 4


In [x]: 4 <= 4


In [x]: 'Hello' == 'HELLO'


In [x]: 2 == 2.0


In [x]: 2 == '2'


In [x]: cakes = 15
In [x]: cakes > 25


In [x]: cakes != 9999


In [x]: True == False
```

Writing a Math Equation as Python Code

This is the math equation for converting Fahrenheit to Celsius:

$$°C = (°F - 32) \times \frac{5}{9}$$

As Python code with variables named `celsius` and `fahrenheit`, it would be:

```
celsius = ( fahrenheit - 32) *  (5 /  9)
```

This is the math equation for converting Fahrenheit to Celsius:

$$°F = °C \times \frac{9}{5} + 32$$

Fill in the blanks for the Python code:

```
fahrenheit = _________________ * (9 ____ 5) + ________
```

Test your code out by pressing the REPL button and entering the following code:

```
In [x]: fahrenheit = 72
In [x]: celsius = fahrenheit – 32) * (5 / 9)
In [x]: celsius
```

```
In [x]: celsius = 23
In [x]: fahrenheit = Your code goes here
In [x]: fahrenheit
```

Example Program: Fahrenheit / Celsius Temperature Conversion

When you run the program, it will look like this (**bold text** is entered by the user):

```
Enter the temperature unit, F or C
>F
Enter the temperature degrees in F
>72
The temperature in C is 22.22222222222222
```

Click the **New** button. Save the tab as *tempconvert.py*. Type the following 15-line program:

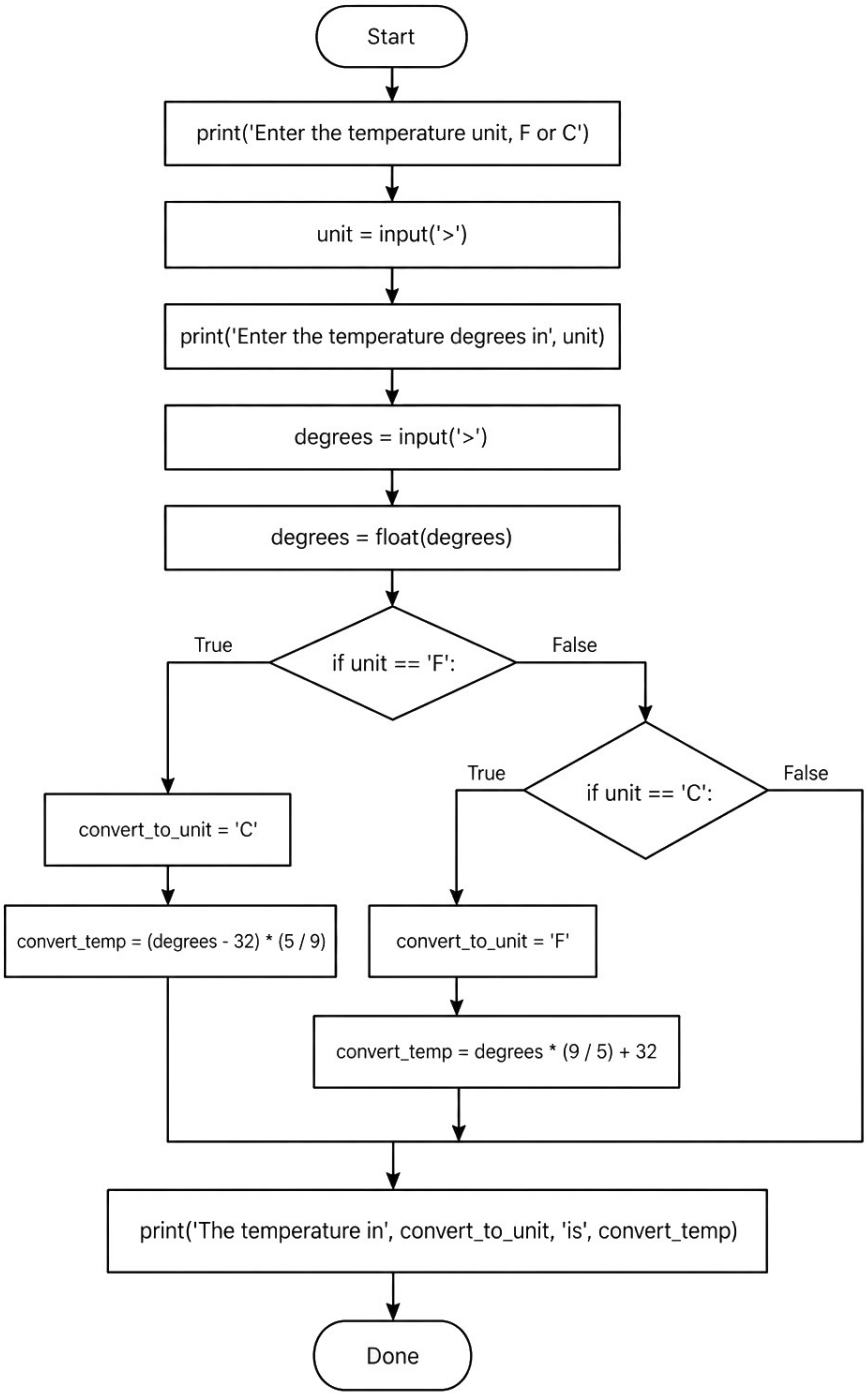
```
 1 # Convert between Celsius and Fahrenheit
 2 print('Enter the temperature unit, F or C')
 3 unit = input('>')
 4 print('Enter the temperature degrees in', unit)
 5 degrees = input('>')
 6 degrees = float(degrees)
 7
 8 if unit == 'F':
 9     convert_to_unit = 'C'
10     convert_temp = (degrees - 32) * (5 / 9)
11 if unit == 'C':
12     convert_to_unit = 'F'
13     convert_temp = degrees * (9 / 5) + 32
14
15 print('The temperature in', convert_to_unit, 'is', convert_temp)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Run the program again, but this time enter a lowercase `f` instead of uppercase `F`. Note that you get an error message. We'll look at this under the debugger next.

Flow Chart for the Temperature Conversion Program

The line of code in the program that is currently being run is called the *execution*. The execution in this program starts at the top and moves down. Trace the path of the execution from Start to Done with your finger. You can imagine a flow chart for this program looking like this:



Run the “Temperature Conversion” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice, then enter **F** into the output pane and press Enter/Return. Notice the `unit` variable in the Debug Inspector has the value `'F'`.

3. Click the **Step Into** button twice, then enter **72** into the output pane and press Enter/Return. Notice the `degrees` variable in the Debug Inspector has the value `'72'`.

4. Click the **Step Into** button to run the `degrees = float(degrees)` line. Notice that this changes the `degrees` variable in the Debug Inspector from the string value `'72'` to the floating point value `72.0`.

5. Click the **Step Into** button twice. Notice that the highlighted line is inside the `if unit == 'F':` block.

6. Click the **Step Into** button twice. The Debug Inspector now has `convert_to_unit` and `convert_temp` variables.

7. Click the **Step Into** button twice. The program is done and you can click the **Stop** button.

Let’s run the program under the debugger and see what happens when we enter lowercase f.

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice, then enter lowercase **f** into the output pane and press Enter/Return. Notice the `unit` variable in the Debug Inspector has the value `'f'`.

3. Click the **Step Into** button twice, then enter **72** into the output pane and press Enter/Return. Notice the `degrees` variable in the Debug Inspector has the value `'72'`.

4. Click the **Step Into** button twice. Notice that both code blocks under `if unit == 'F':` and `if unit == 'C':` are skipped. There are no `convert_to_unit` or `convert_temp` variables in the Debug Inspector!

5. Click the **Step Into** button to run the last line. Python stops the program with an error message because the instruction says to print variables that don’t exist.

The round() Function

Click the REPL button. Find out about the `round()` function by calling the `help()` function:

```
In [x]: help(round)
```

Enter the following code, and write down what they return:

```
In [x]: round(3.14)

In [x]: round(5.999)

In [x]: round(3.5)

In [x]: round(2.5)
```

(NOTE: The `round()` function uses “banker’s rounding”, so numbers that end with .5 round to the closest even number.)

You can pass a second argument to `round()` for how many decimal places to round to:

```
In [x]: 5 / 3

In [x]: round(5 / 3)

In [x]: round(5 / 3, 2)

In [x]: round(5 / 3, 4)
```

Add the round() Function to the Temperature Conversion Program

Add a call to round() into the temperature conversion program:

```
--snip--
rounded_temp = round(convert_temp, 2)
print('The temperature in', convert_to_unit, 'is', convert_temp)
```

Run the program. Enter F and 72 for the inputs. Notice that the program output hasn't changed! It still outputs 22.22222222222222 instead of 22.22.

We stored the rounded number in a variable named `rounded_temp`, but the last line of code still prints the `convert_temp` variable!

Change the last line of code to this:

```
print('The temperature in', convert_to_unit, 'is', rounded_temp)
```

Then run the program again. Notice that 72 degrees Fahrenheit converts to `22.22`.

Importing Modules and the random.randint() and sys.exit() Functions

For the Cho Han program next, we'll need to simulate rolling dice.

Some functions that come with Python exist in modules that need to be imported.

This `import` statement imports the `random` module: `import random`

After importing `random`, you can call `random.randint()` function. Click the REPL button and type the following. (Hint: You can press the up arrow key to get previous REPL instructions.)

```
In [x]: random.randint(1, 6)  # Causes an error!

In [x]: import random

In [x]: random.randint(1, 6)

In [x]: random.randint(1, 6)

In [x]: random.randint(1, 6)

In [x]: random.randint(1, 6)

In [x]: random.randint(1, 6)
```

The `random.randint()` function returns a random integer between (**and including**) the two arguments: calling `random.randint(1, 6)` randomly returns `1`, `2`, `3`, `4`, `5` or `6`. *Just like rolling a six-sided die!*

The Cho Han program will also import the `sys` module to give us access to the `sys.exit()` function, which ends the program as soon as the execution reaches it.

Calculating If a Number is Even or Odd

For the Cho Han program next, we'll need to figure out if a number is even or odd. Here's a technique that programmers use in all programming languages, not just Python.

The modulo operator % in Python is a "division remainder" operator that gives you the remainder.

For example, eleven divided by four is two, remainder three. 11 / 4 evaluates to 2.75. But 11 % 4 evaluates to 3, the remainder.

(There is an "integer division" operator // that gets the 2 part: 11 // 4 evaluates to 2.)

All even numbers modulo two will evaluate to *zero*. (They have no remainder.)

All odd numbers modulo two will evaluate to *one*.

Click the **REPL** button, enter the following, and write down the results:

```
In [x]: 2 % 2


In [x]: 4 % 2


In [x]: 10 % 2


In [x]: 3 % 2


In [x]: 15 % 2


In [x]: 10 % 2 == 0


In [x]: 10 % 2 == 1


In [x]: number = 7
In [x]: number_is_even = number % 2 == 0
In [x]: number_is_even


In [x]: number_is_odd = number % 2 == 1
In [x]: number_is_odd
```

Boolean Operators and Truth Tables

- The `and` operator works on two Boolean values. It evaluates to `True` if both are `True`, otherwise it evaluates to `False`.
- The `or` operator works on two Boolean values. It evaluates to `False` if both are `False`, otherwise it evaluates to `True`.
- The `not` operator works on one Boolean value. It evaluates the opposite Boolean value.

A *truth table* shows all possible evaluations for an expression with Boolean operators:

`and`	`or`	`not`
`True and True → ________`	`True or True → ________`	`not True → ________`
`True and False → ________`	`True or False → ________`	`not False → ________`
`False and True → ________`	`False or True → ________`	
`False and False → ________`	`False or False → ________`	

Click the REPL button, type the following, and write down the answer:

```
In [x]: True or False and False

In [x]: (True or False) and False

In [x]: not True

In [x]: not (True and True)

In [x]: (not True) and True

In [x]: not True or not True

In [x]: not not False
```

Example Program: Cho Han Dice Game

When you run the program, it will look like this (**bold text** is entered by the user):

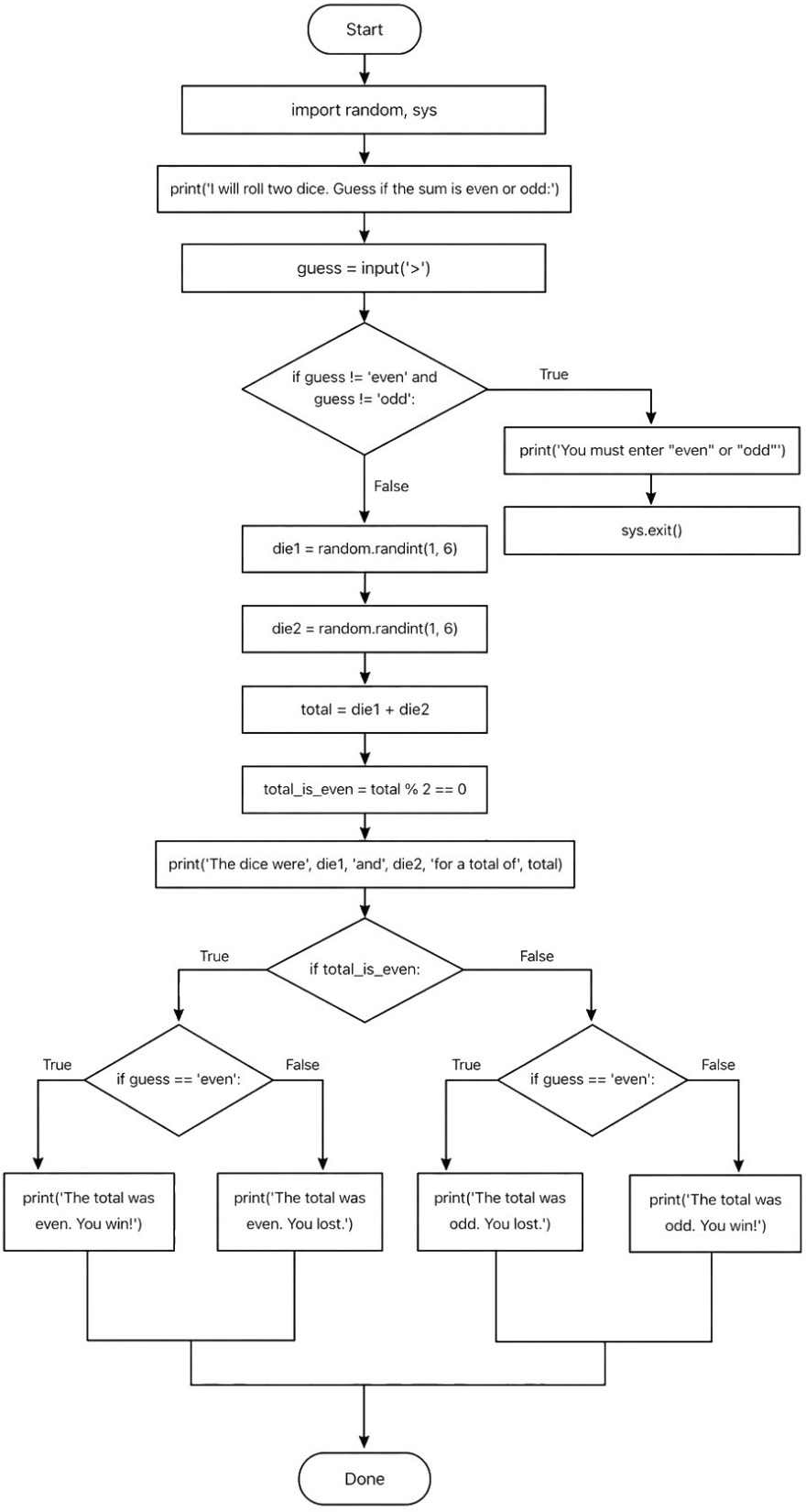
```
I will roll two dice. Guess if the sum is even or odd:
>even
The dice were 5 and 3 for a total of 8
The total was even. You win!
```

Click the **New** button. Save the *untitled* tab as *chohan.py*. Type the following 29-line program:

```
 1 import random, sys
 2
 3 # Let the user enter their guess:
 4 print('I will roll two dice. Guess if the sum is even or odd:')
 5 guess = input('>')
 6
 7 # Do input validation:
 8 if guess != 'even' and guess != 'odd':
 9     print('You must enter "even" or "odd"')
10     sys.exit()
11
12 # Get two random numbers for the dice rolls:
13 die1 = random.randint(1, 6)
14 die2 = random.randint(1, 6)
15 total = die1 + die2
16 total_is_even = total % 2 == 0
17
18 # Display the results:
19 print('The dice were', die1, 'and', die2, 'for a total of', total)
20 if total_is_even:
21     if guess == 'even':
22         print('The total was even. You win!')
23     else:
24         print('The total was even. You lost.')
25 else:
26     if guess == 'even':
27         print('The total was odd. You lost.')
28     else:
29         print('The total was odd. You win!')
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the Cho Han Dice Game Program



Run the “Cho Han Dice” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button. The `random` and `sys` modules appear in the Debug Inspector after they’ve been imported.

3. Click the **Step Into** button twice and entered either “even” or “odd”. The `guess` variable appears in the Debug Inspector.

4. Click the **Step Into** button. The execution leaps over the sys.exit() call as long as you entered either “even” or “odd”. If you entered something else, the program prints a message and then exits.

5. Click the **Step Into** button. The `die1` variable appears in the Debug Inspector. You can see what random number the `random.randint(1, 6)` call returned.

6. Click the **Step Into** button. The `die2` variable appears in the Debug Inspector. You can see what random number the `random.randint(1, 6)` call returned.

7. Click the **Step Into** button. The `total` variable appears in the Debug Inspector.

8. Click the **Step Into** button. The `total_is_even` variable appears in the Debug Inspector and is either set to the Boolean `True` or `False` value.

9. Keep clicking the **Step Into** button and see which lines of code are executed and which are skipped, based on the conditions of the `if` statements.

10. Click the **Stop** button when the program ends.

while Loops and the Example Program: "Your Name" Joke

When you run the program, it will look like this (**bold text** is entered by the user):

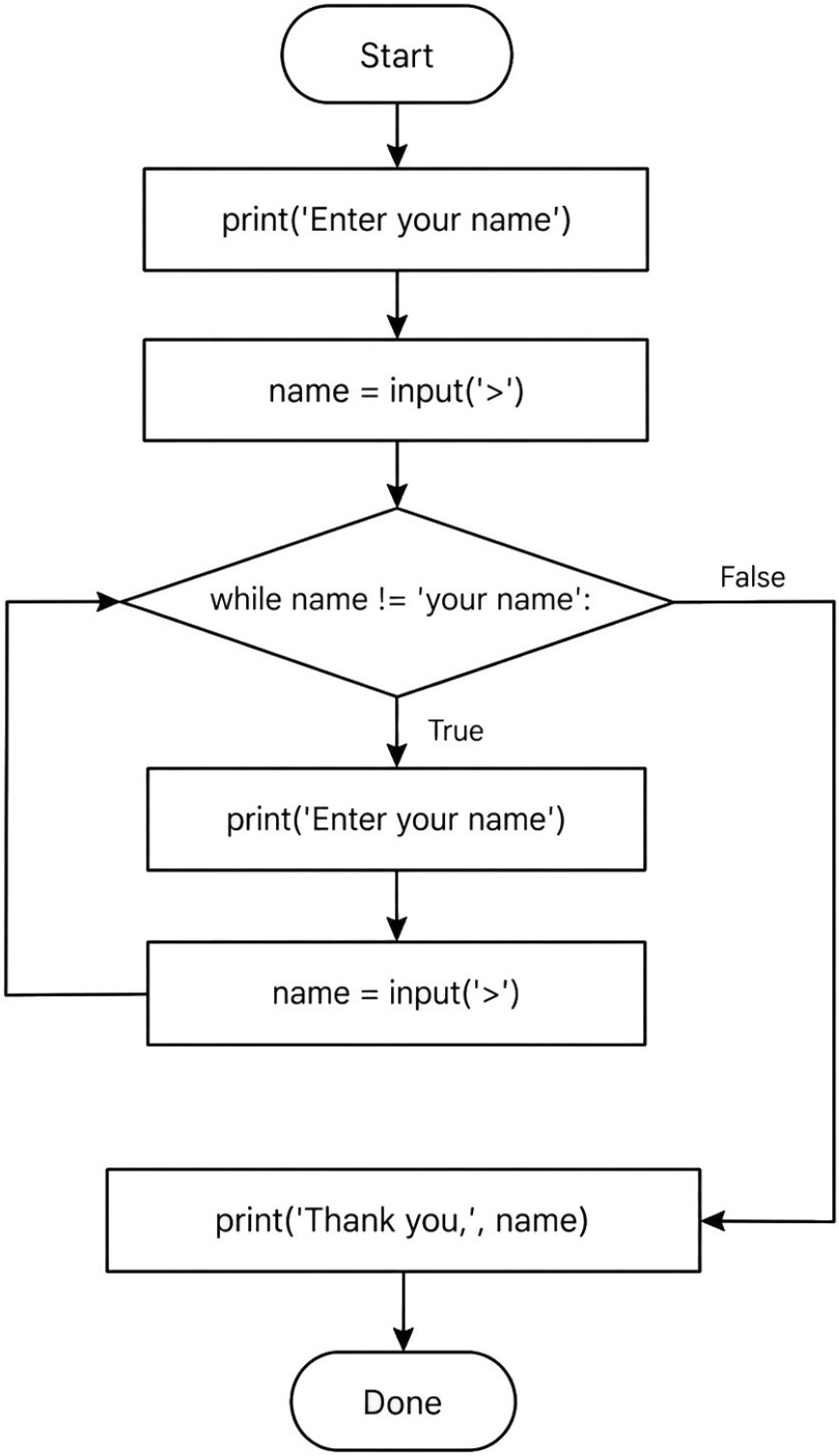
```
Enter your name
>Al
Enter your name
>Albert
Enter your name
>$(*!F*%@H!!!
Enter your name
>your name
Thank you, your name
```

Click the **New** button. Save the *untitled* tab as *yourname.py*. Type the following 10-line program:

```
 1 # Ask for your name:
 2 print('Enter your name')
 3 name = input('>')
 4
 5 while name != 'your name':
 6     # Ask for your name again:
 7     print('Enter your name')
 8     name = input('>')
 9
10 print('Thank you,', name)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the “Your Name” Program



Run the “Your Name” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice and enter your actual name. The `name` variable in the Debug Inspector appears.

3. Click the **Step Into** button three times and watch the execution enter the `while` loop. Enter your actual name again. Notice that the execution went back up to the `while` statement.

4. Click the **Step Into** button three times and then enter literally “your name”.

5. Click the **Step Into** button and watch as the execution jumps down to the first line after the `while` loop.

6. Click the **Step Into** button to run the last line.

7. Click **Stop**.

Infinite Loops, break Statements, and the “Your Name 2” Program

This program is functionally identical to the previous program.

When you run the program, it will look like this (**bold text** is entered by the user):

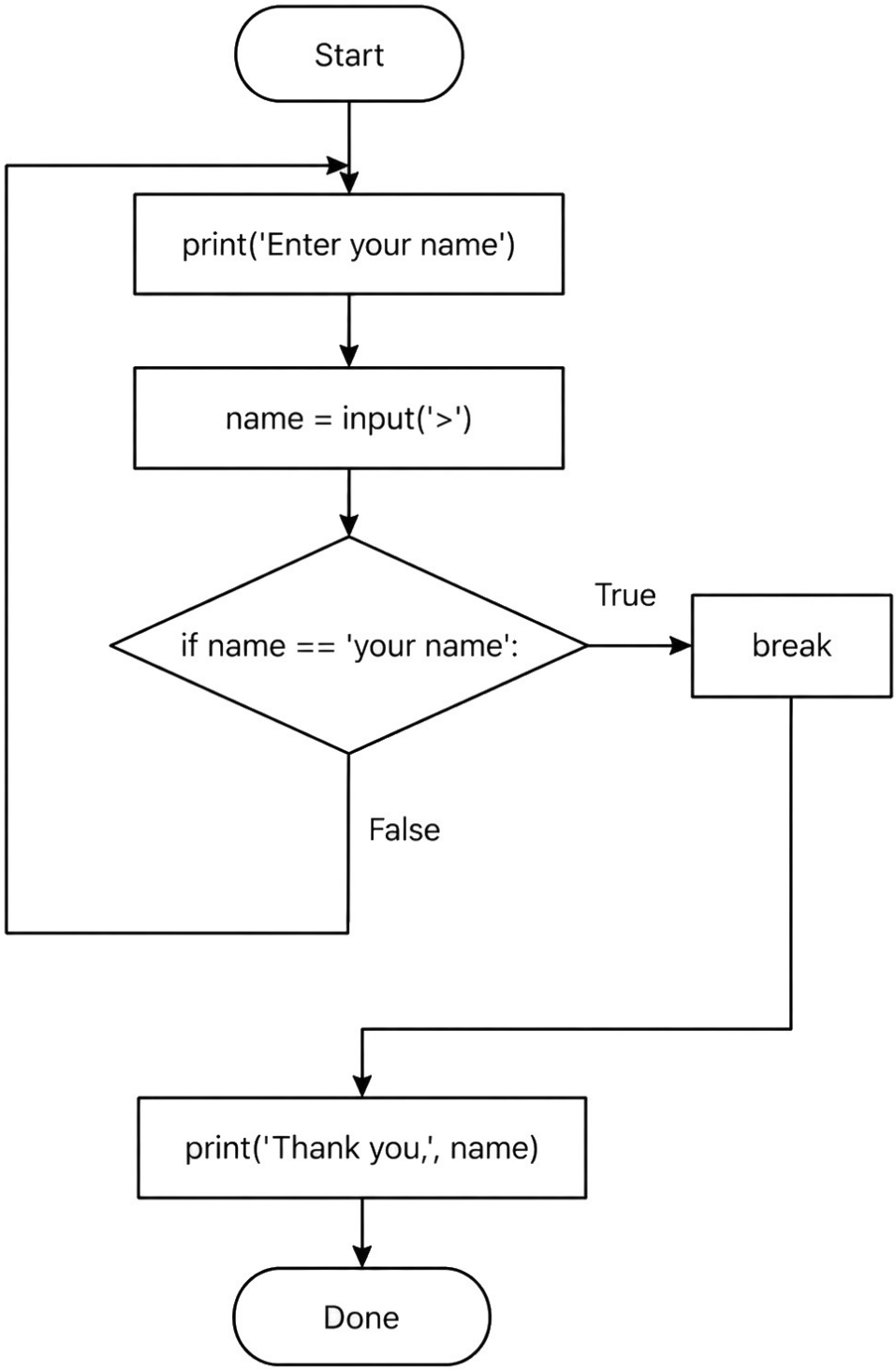
```
Enter your name
>Al
Enter your name
>Albert
Enter your name
>$(*!F*%@H!!!
Enter your name
>your name
Thank you, your name
```

Click the **New** button. Save the *untitled* tab as *yourname2.py*. Type the following 7-line program:

```
1 while True:
2     print('Enter your name')
3     name = input('>')
4     if name == 'your name':
5         break
6
7 print('Thank you,', name)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the “Your Name 2” Program (with a break Statement)



Run the “Your Name 2” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice and enter your actual name. The `name` variable in the Debug Inspector appears.

3. Click the **Step Into** button twice and watch the execution re-enter the `while` loop. Enter your actual name again.

4. Click the **Step Into** button twice and then enter literally “your name”.

5. Click the **Step Into** button twice and watch as the execution enters the `if` statement’s block, reaches the `break` statement, and then jumps down to the first line after the `while` loop.

6. Click the **Step Into** button to run the last line.

7. Click **Stop**.

Example Program: Clever Carl

A story goes that when mathematician Carl Friedrich Gauss was a boy, his teacher wanted to nap so to keep the class busy he told them to add all the numbers from 1 to 100: 1 + 2 + 3 + 4 + 5 + … + 100. But young Carl figured out a method to let him answer it in seconds. We don't need to be clever; we can write a program to do the math for us.

(The story about the teacher probably isn't true, but it's a good story!)

When you run the program, it will look like this:

```
0 + 1 is 1
1 + 2 is 3
3 + 3 is 6
6 + 4 is 10
10 + 5 is 15
--snip--
4851 + 99 is 4950
4950 + 100 is 5050
5050
```

Click the **New** button. Save the *untitled* tab as *clevercarl.py*. Type the following 5-line program (but do not type the line numbers) in the *file editor*:

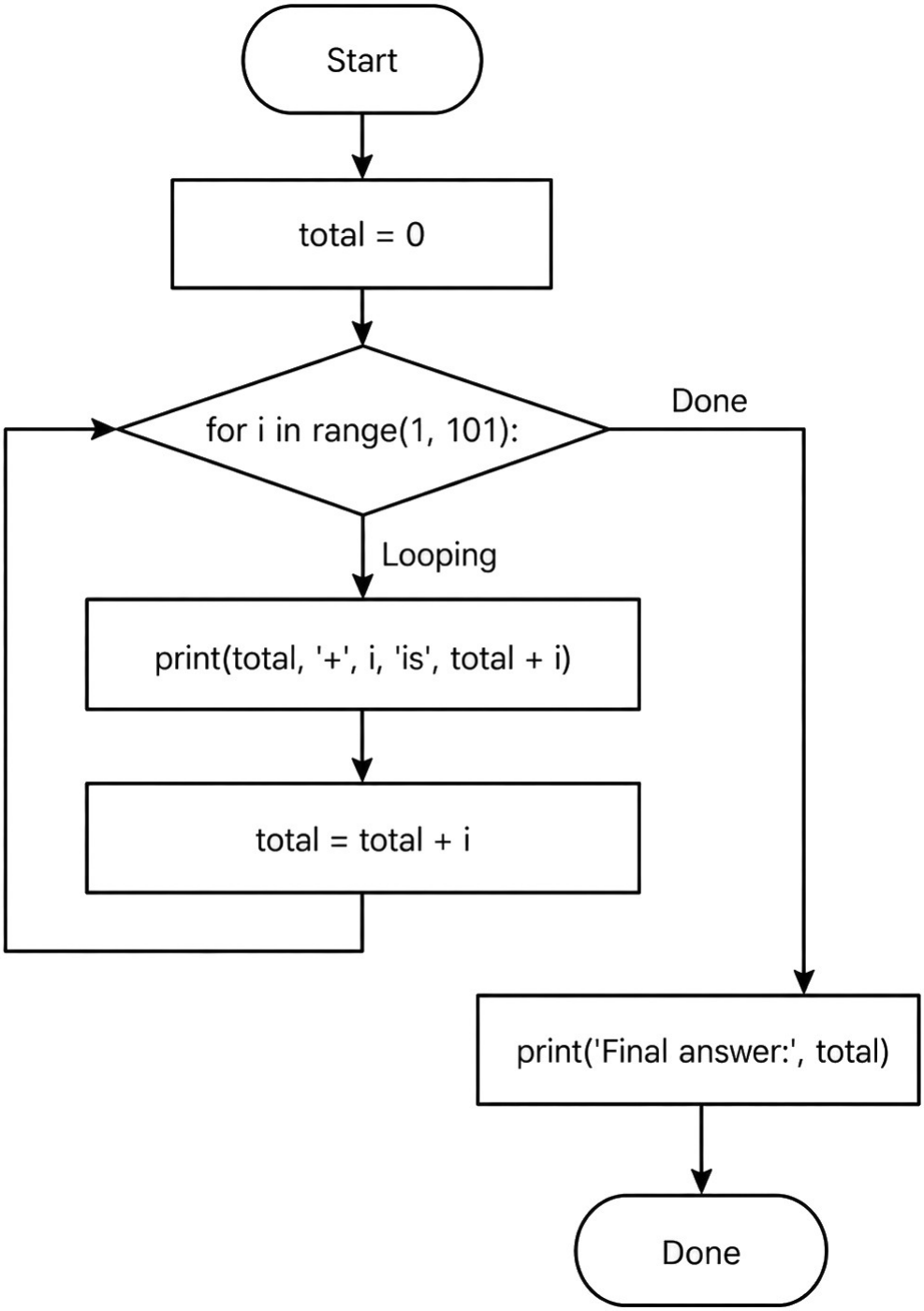
```
1 total = 0
2 for i in range(1, 101):
3     print(total, '+', i, 'is', total + i)
4     total = total + i
5 print('Final answer:', total)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

An Equivalent Clever Carl Program Without the for Loop

```
total = 0
i = 1
print(total, '+', i, 'is', total + i)
total = total + i
i = 2
print(total, '+', i, 'is', total + i)
total = total + i
i = 3
print(total, '+', i, 'is', total + i)
total = total + i
i = 4
print(total, '+', i, 'is', total + i)
total = total + i
i = 5
print(total, '+', i, 'is', total + i)
total = total + i
i = 6
print(total, '+', i, 'is', total + i)
total = total + i
i = 7
print(total, '+', i, 'is', total + i)
total = total + I
i = 8
print(total, '+', i, 'is', total + i)
total = total + I
i = 9
print(total, '+', i, 'is', total + i)
total = total + I
i = 10
print(total, '+', i, 'is', total + i)
total = total + I
i = 11
print(total, '+', i, 'is', total + i)
total = total + i

--snip--

i = 99
print(total, '+', i, 'is', total + i)
total = total + i
i = 100
print(total, '+', i, 'is', total + i)
total = total + i
print('Final answer:', total)
```

Flow Chart for the “Clever Carl” Program



Break Points and Running the “Clever Carl” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice. The `i` variable in the Debug Inspector appears set to 1.

3. Click the **Step Into** button three times. The `i` variable now has the value 2.

4. Keep clicking the **Step Into** button several times and watch the execution loop around.

5. Click the 4 line number on the left side and notice a red dot appears. This is a *break point*.

6. Click the **Continue** button. Notice that the program runs at full speed until it encounters a break point, where it then pauses.

7. Keep click the **Continue** button several times.

8. Click the 4 line number on the left side to make the red dot disappear. This removes the break point.

9. Click the **Continue** button to run the last line.

10. Click **Stop**.

The range() Function

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

Write the arguments to the range() function that would produce these outputs:

```
0 1 2 3 4 5 6 7 8 9

for i in range(____10_____):
    print(i, end=' ')
```

```
0 2 4 6 8

for i in range(_____________):
    print(i, end=' ')
```

```
4 5 6 7 8 9 10 11 12

for i in range(__4, 13___):
    print(i, end=' ')
```

```
0 2 4 6 8 10 12

for i in range(_____________):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8

for i in range(_____________):
    print(i, end=' ')
```

```
0 5 10 15 20 25 30 35

for i in range(_____________):
    print(i, end=' ')
```

```
2 3 4 5 6 7

for i in range(_____________):
    print(i, end=' ')
```

```
5 10 15 20 25 30 35

for i in range(_____________):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6

for i in range(_____________):
    print(i, end=' ')
```

```
5 4 3 2 1

for i in range(_____________):
    print(i, end=' ')
```

```
0

for i in range(_____________):
    print(i, end=' ')
```

```
5 4 3 2 1 0

for i in range(_____________):
    print(i, end=' ')
```

```
100 101 102 103 104 105

for i in range(_____________):
    print(i, end=' ')
```

```
10 8 6 4 2 0

for i in range(_____________):
    print(i, end=' ')
```

Example Program: Guess the Number

Let's create a "guess the number" game that gives you hints whether your guess is too low or too high. This program combines several of the previous concepts we've learned. When you run the program, it will look like this:

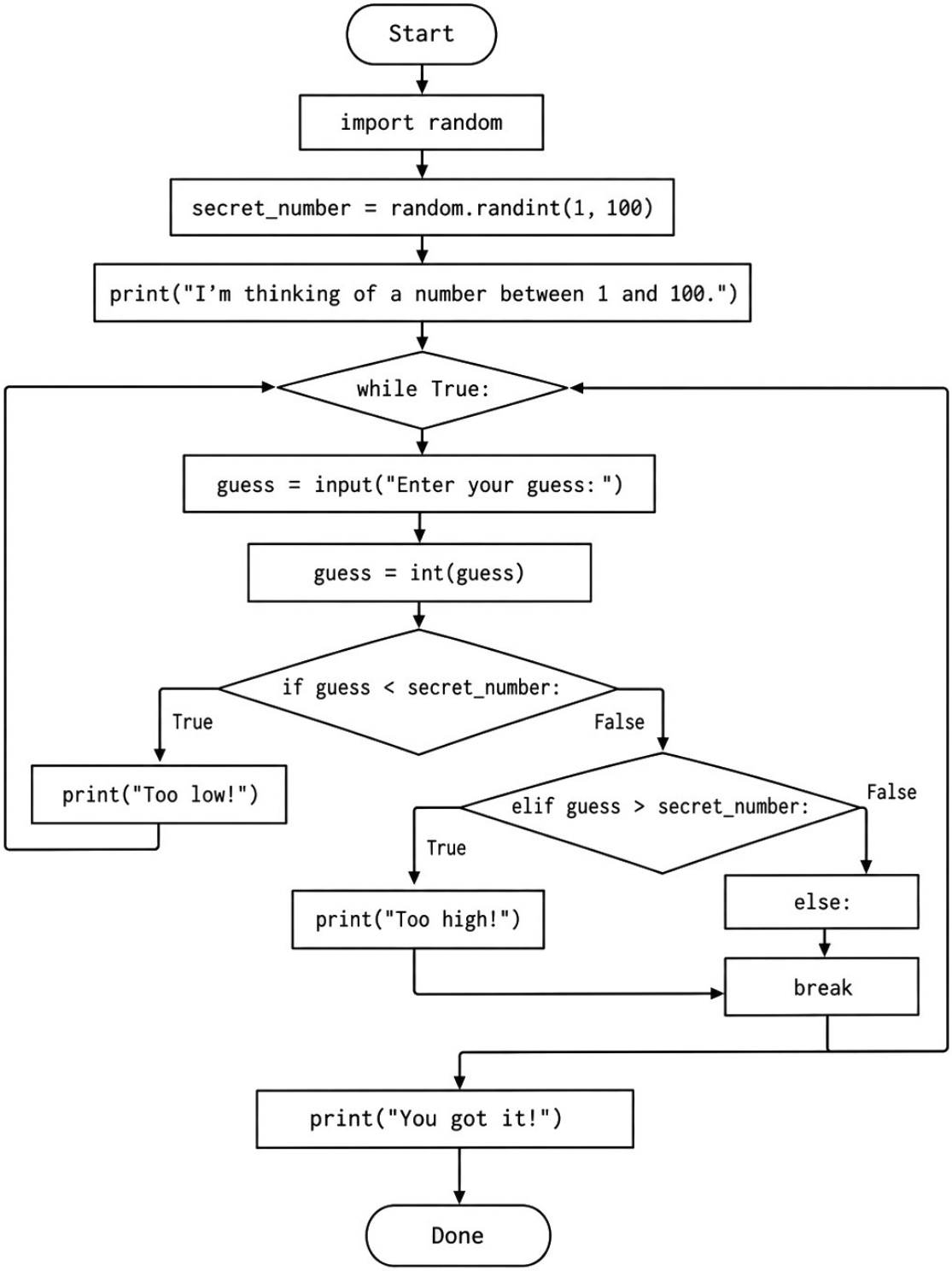
```
I'm thinking of a number between 1 and 100.
Enter your guess: 50
Too low!
Enter your guess: 75
Too high!
Enter your guess: 60
Too high!
Enter your guess: 55
Too low!
Enter your guess: 57
Too low!
Enter your guess: 58
You got it!
```

Click the **New** button. Save the *untitled* tab as *guess.py*. Type the following 17-line program:

```
 1 import random
 2
 3 secret_number = random.randint(1, 100)
 4 print("I'm thinking of a number between 1 and 100.")
 5
 6 # Keep looping until the guess is correct
 7 while True:
 8     guess = input("Enter your guess: ")
 9     guess = int(guess)
10
11     if guess < secret_number:
12         print("Too low!")
13     elif guess > secret_number:
14         print("Too high!")
15     else:
16         break
17 print("You got it!")
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the “Guess the Number” Program



Run the “Guess the Number” Program Under the Debugger

We’ll be able to cheat if we run the program under the debugger because we’ll be able to see the random number in the `secret_number` variable.

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button twice. The `secret_number` variable in the Debug Inspector appears with the random number.

3. Click the **Step Into** button twice and enter the same number as in `secret_number`.

4. Notice that the `guess` variable contains a string of the number you entered. When you click the **Step Into** button to run `guess = int(guess)` line, the guess variable contains an integer form of its previous string value.

5. Click the **Step Into** button twice. The break statement in the else block runs, moving the execution out of the loop and to the print('You got it!') line.

6. Click the **Step Into** button to run the last line.

7. Click **Stop**.

Now run the program under the debugger again, this time purposefully enter a too high and a too low number. When testing software, we need to thoroughly examine every possible input the user could give us.

Example Program: Clock Hands 1

Loop inside a loop demo: https://inventwithpython.com/nested-loops.html

Let's create a demo program that shows how nested loops work. When you run the program, it will look like this:

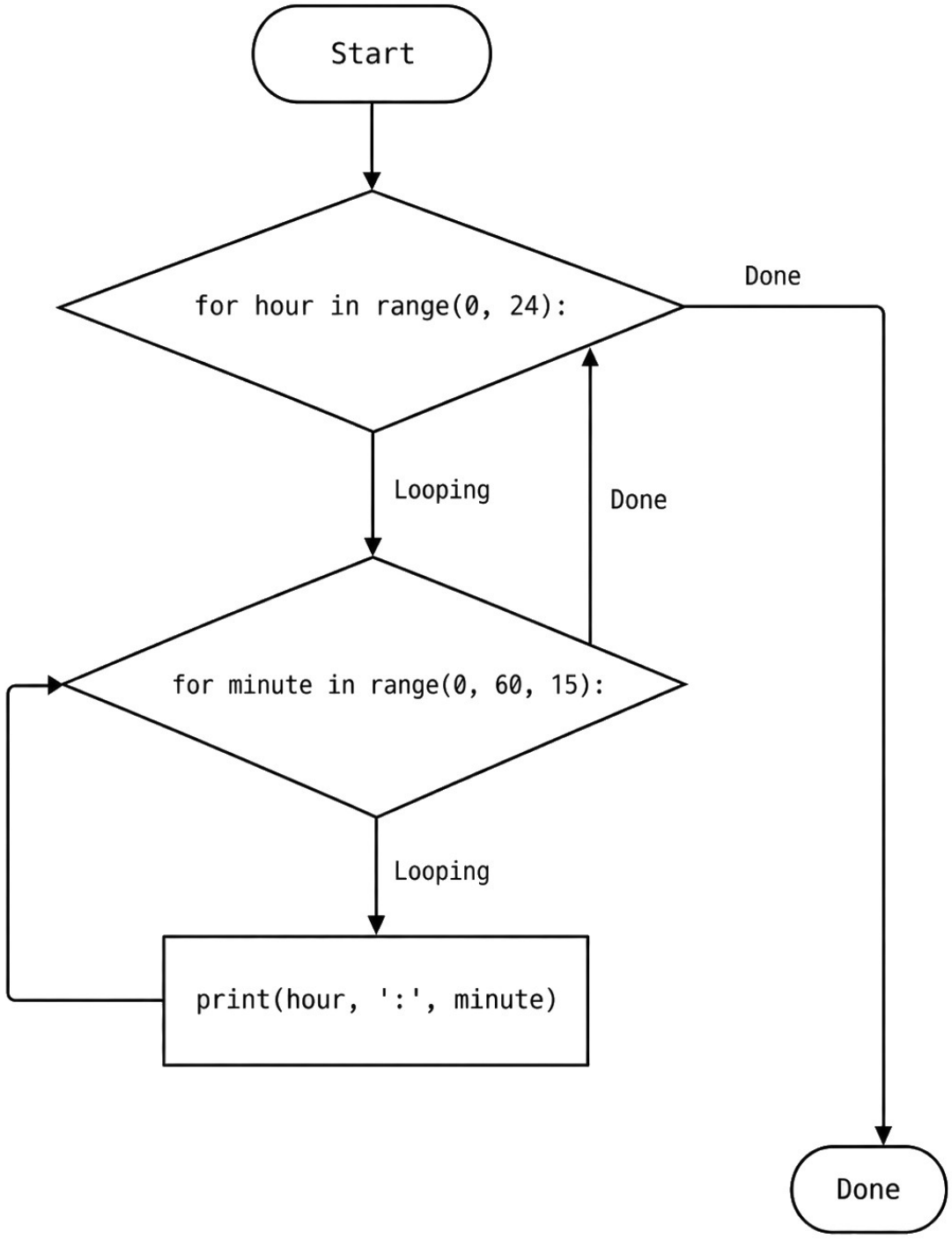
```
0 : 0
0 : 15
0 : 30
0 : 45
1 : 0
1 : 15
1 : 30
--snip--
22 : 30
22 : 45
23 : 0
23 : 15
23 : 30
23 : 45
```

Click the **New** button. Save the *untitled* tab as *clockhands1.py*. Type the following 3-line program:

```
1 for hour in range(0, 24):
2     for minute in range(0, 60, 15):
3         print(hour, ':', minute)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the “Clock Hands 1” Program



Break Points and Running the “Clock Hands 1” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button 13 or more times. Watch how the `minute` and `hour` variables in the Debug Inspector change and the path of the execution.

5. Click the 1 line number on the left side and notice a red dot appears. This is a *break point*.

6. Click the **Continue** button. Notice that the program runs at full speed for the entire inner loop, until it encounters the break point on the outer loop where it then pauses.

7. Keep click the **Continue** button several times.

8. Click the 1 line number on the left side to make the red dot disappear. This removes the break point.

9. Click the **Continue** button to run the last line.

10. Click **Stop**.

Example Program: Clock Hands 2

Let's create another demo program that shows how nested loops work, this time with three loops. When you run the program, it will look like this:

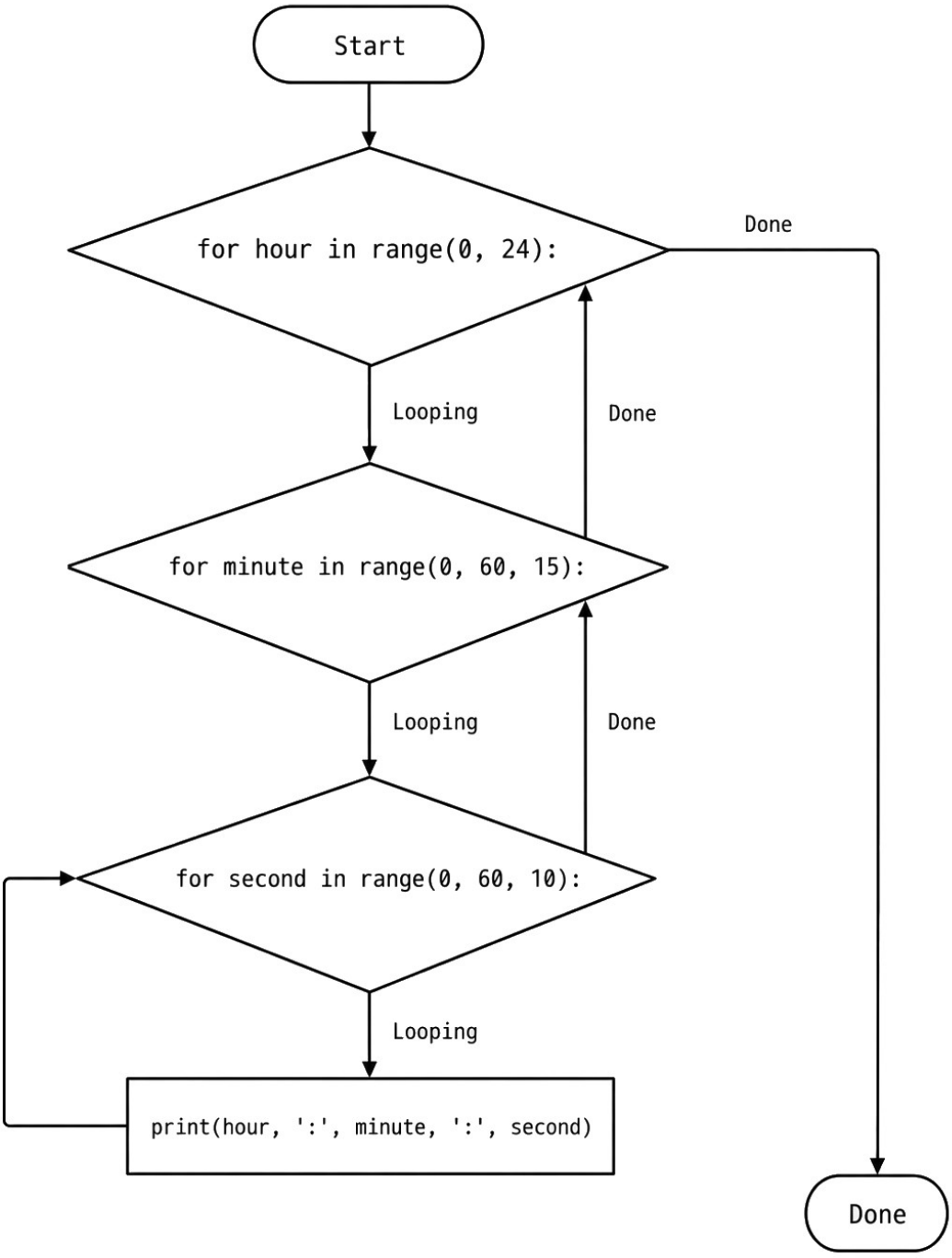
```
0 : 0 : 0
0 : 0 : 10
0 : 0 : 20
0 : 0 : 30
0 : 0 : 40
0 : 0 : 50
0 : 15 : 0
0 : 15 : 10
--snip--
23 : 45 : 30
23 : 45 : 40
23 : 45 : 50
```

Click the **New** button. Save the *untitled* tab as *clockhands2.py*. Type the following 4-line program:

```
1 for hour in range(0, 24):
2     for minute in range(0, 60, 15):
3         for second in range(0, 60, 10):
4             print(hour, ':', minute, ':', second)
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Flow Chart for the “Clock Hands 2” Program



Run the “Clock Hands 2” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button 20 or more times. Watch how the `second`, `minute`, and `hour` variables in the Debug Inspector change and the path of the execution.

3. Click the 2 line number on the left side and notice a red dot appears. This is a *break point*.

4. Click the **Continue** button. Notice that the program runs at full speed for the entire inner loop, until it encounters the break point on the minute outer loop where it then pauses.

5. Keep click the **Continue** button several times.

6. Click the 2 line number on the left side to make the red dot disappear. This removes the break point.

7. Click the **Continue** button. Notice that the program runs at full speed for BOTH inner loops, until it encounters the break point on the hour outer loop where it then pauses.

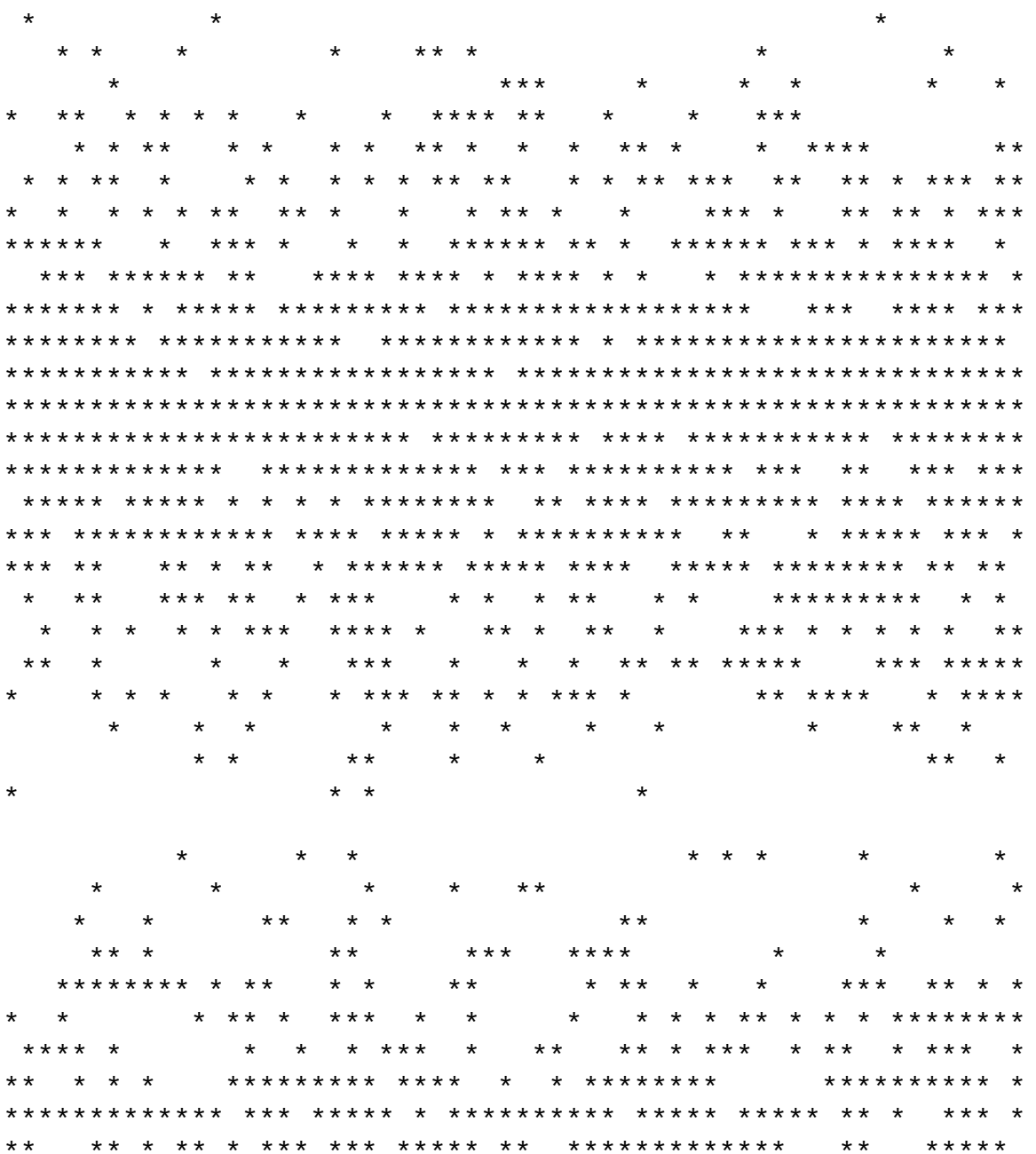
8. Click the **Continue** button a few more times.

9. Click the 1 line number on the left side to make the red dot disappear. This removes the break point.

10. Click **Stop**.

Example Program: Starfield Scroll Art

Let’s write a program that prints rows of stars (as asterisks) that increase in density over time, decrease in density, and then repeats. When you run the program, it will look like this:



Click the **New** button. Save the *untitled* tab as *starfield.py*. Type the following 33-line program:

```
 1 import random, time
 2
 3 WIDTH = 80
 4 DENSITY_CHANGE = 0.02
 5 DELAY = 0.1
 6 STAR_CHAR = '*'
 7 BLANK_CHAR = ' '
 8
 9
10 def print_row_of_stars(density):
11     for i in range(WIDTH):
12         if random.random() < density:
13             print(STAR_CHAR, end='')
14         else:
15             print(BLANK_CHAR, end='')
16     print()  # Print newline at the end of the row
17
18
19 current_density = 0.0
20 while True:
21     # Keep increasing the density until it reaches 1.0:
22     while current_density < 1.0:
23         print_row_of_stars(current_density)
24         time.sleep(DELAY)
25         # Increase the star density:
26         current_density = current_density + DENSITY_CHANGE
27
28     # Keep decreasing the density until it reaches 0.0:
29     while current_density > 0.0:
30         print_row_of_stars(current_density)
31         time.sleep(DELAY)
32         # Decrease the star density:
33         current_density = current_density - DENSITY_CHANGE
```

Click the **Run** button to run the program. While running the program, Mu changes the Run button to the Stop button. Click the **Stop** button when the program is done.

Try changing the `WIDTH`, `DENSITY_CHANGE`, `DELAY`, `STAR_CHAR`, and `BLANK_CHAR` constants to different values and re-running the program.

Run the “Starfield Scroll Art” Program Under the Debugger

1. Click the **Debug** button to start running the program “under the debugger”.

2. Click the **Step Into** button 20 or more times. Watch how execution jumps into the `print_row_of_stars()` function when it is called, and then returns to the line after the function call..

3. Click the 23 and 30 line numbers on the left side and notice a red dot appears. These are *break points*.

4. Click the **Continue** button. Notice that the program runs at full speed until it encounters the break point on the minute outer loop where it then pauses.

5. Keep click the **Continue** button several times.

6. Click the 23 and 30 line numbers on the left side to make the red dots disappear. This removes the break points.

8. Click the **Continue** button to let the program continue running.

9. Click **Stop** when you’ve seen enough.

More examples of scroll art are available at https://scrollart.org

Next Steps for After This Tutorial

You can continue to make several programs like these, but the next step is to learn about *data structures*. The first data structures you can learn about are Python *lists* and *dictionaries*.

You'll probably have "blank editor syndrome" where you stare at the blank code editor and just don't know how to start making your own programs. *This is fine and expected.* Find additional tutorials and resources to continue gaining experience.

- *Automate the Boring Stuff with Python* is a free book for complete beginners that explores these topics at https://automatetheboringstuff.com/
- I have several other free Python books at https://inventwithpython.com
- This blog post details which of my free books might be right for your interests: https://inventwithpython.com/blog/which-al-sweigart-python-books-should-i-start-reading.html
- Python Programming Exercises, Gently Explained has exercises for beginners at https://inventwithpython.com/pythongently/

You can find this workbook and links to the slides at https://inventwithpython.com/pyconus2026

While online video courses might be helpful (such as https://www.freecodecamp.org/) I advise people against one-off YouTube videos. They're often too short to provide any real value.

Feel free to contact me at al@inventwithpython.com

Good luck on your programming journey!

Gotta Catch ‘Em All – Python Error Messages

Write down error messages you encounter, along with the code you wrote that caused them! Cross these out as you encounter them:

NameError	SyntaxError	TypeError	ValueError
IndexError	ModuleNotFoundError	KeyError	IndexError
OSError	ZeroDivisionError		

Full Error Message:	Code That Caused It:
`NameError: name 'my_naem' is not defined.`	`print(my_naem)`