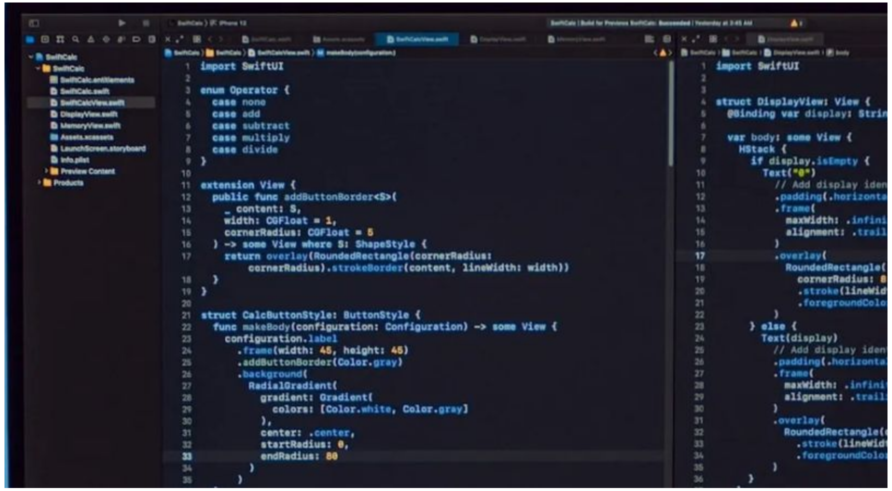
Advanced Python Functions

From the **map()** function, which applies a specified function to each element of an iterable and returns a new iterable with the modified elements, to the **enumerate()** function which adds a counter to an iterable and returns an iterable of tuples, these functions can help you write cleaner and more concise code.
Let’s take a closer look at these advanced Python functions and see how to use them in your code.



```swift
import SwiftUI

enum Operator {
  case none
  case add
  case subtract
  case multiply
  case divide
}

extension View {
  public func addButtonBorder<S>(
    _ content: S,
    width: CGFloat = 1,
    cornerRadius: CGFloat = 5
  ) -> some View where S: ShapeStyle {
    return overlay(RoundedRectangle(cornerRadius:
        cornerRadius).strokeBorder(content, lineWidth: width))
  }
}

struct CalcButtonStyle: ButtonStyle {
  func makeBody(configuration: Configuration) -> some View {
    configuration.label
      .frame(width: 45, height: 45)
      .addButtonBorder(Color.gray)
      .background(
        RadialGradient(
          gradient: Gradient(
            colors: [Color.white, Color.gray]
          ),
          center: .center,
          startRadius: 0,
          endRadius: 80
        )
      )
```

```swift
import SwiftUI


struct DisplayView: View {
  @Binding var display: Strin

  var body: some View {
    HStack {
      if display.isEmpty {
        Text("0")
          // Add display iden
          .padding(.horizonta
          .frame(
            maxWidth: .infini
            alignment: .trail
          )
          .overlay(
            RoundedRectangle(
              cornerRadius: 8
              .stroke(lineWid
              .foregroundColo
          )
      } else {
        Text(display)
          // Add display iden
          .padding(.horizonta
          .frame(
            maxWidth: .infini
            alignment: .trail
          )
          .overlay(
            RoundedRectangle(c
              .stroke(lineWid
              .foregroundColo
          )
      }
```

map()

The map() function applies a specified function to each element of an iterable (such as a list, tuple, or string) and returns a new iterable with the modified elements.

For example, you can use map() to apply the len() function to a list of strings and get a list of the lengths of each string:

```python
def get_lengths(words):
    return map(len, words)

words = ['cat', 'window', 'defenestrate']
lengths = get_lengths(words)
print(lengths)  # [3, 6, 12]
```

You can also use map() with multiple iterables by providing multiple function arguments. In this case, the function should accept as many arguments as there are iterables. The map() function will then apply the function to the elements of the iterables in parallel, with the first element of each iterable being passed as arguments to the function, the second element of each iterable being passed as arguments to the function, and so on.

For example, you can use map() to apply a function to two lists element-wise:

```
def multiply(x, y):
    return x * y


a = [1, 2, 3]
b = [10, 20, 30]
result = map(multiply, a, b)
print(result)  # [10, 40, 90]
```

reduce()

The reduce() function is a part of the functools module in Python and allows you to apply a function to a sequence of elements in order to reduce them to a single value.

For example, you can use reduce() to multiply all the elements of a list:

```
from functools import reduce

def multiply(x, y):
    return x * y

a = [1, 2, 3, 4]
result = reduce(multiply, a)
print(result)  # 24
```

You can also specify an initial value as the third argument to reduce(). In this case, the initial value will be used as the first argument to the function and the first element of the iterable will be used as the second argument.

For example, you can use reduce() to calculate the sum of a list of numbers:

```
from functools import reduce

def add(x, y):
    return x + y

a = [1, 2, 3, 4]
result = reduce(add, a, 0)
print(result)  # 10
```

filter()

The filter() function filters an iterable by removing elements that do not satisfy a specified condition. It returns a new iterable with only the elements that satisfy the condition.

For example, you can use filter() to remove all the even numbers from a list:

```
def is_odd(x):
   return x % 2 == 1


a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = filter(is_odd, a)
print(odd_numbers)  # [1, 3, 5, 7,
9]
```

You can also use filter() with a lambda function as the first argument:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = filter(lambda x: x %
2 == 1, a)
print(odd_numbers)  # [1, 3, 5, 7,
9]
```

zip()

The zip() function combines multiple iterables into a single iterable of tuples. The zip() function stops

when the shortest iterable is exhausted.

For example, you can use zip() to combine two lists into a list of tuples:

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
zipped = zip(a, b)
print(zipped)  # [(1, 'a'), (2,
'b'), (3, 'c')]
```

You can also use zip() with more than two iterables:

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
c = [True, False, True]
zipped = zip(a, b, c)
print(zipped)  # [(1, 'a', True),
(2, 'b', False), (3, 'c', True)]
```

You can unpack the tuples in a zip() object by using the * operator in a function call or a list comprehension:

```python
def print_tuples(a, b, c):
    print(f'a: {a}, b: {b}, c: {c}')

a = [1, 2, 3]
b = ['a', 'b', 'c']
c = [True, False, True]
zipped = zip(a, b, c)

# Unpack the tuples in a function call
for t in zipped:
    print_tuples(*t)

# Unpack the tuples in a list comprehension
unzipped = [print_tuples(*t) for t in zipped]
```

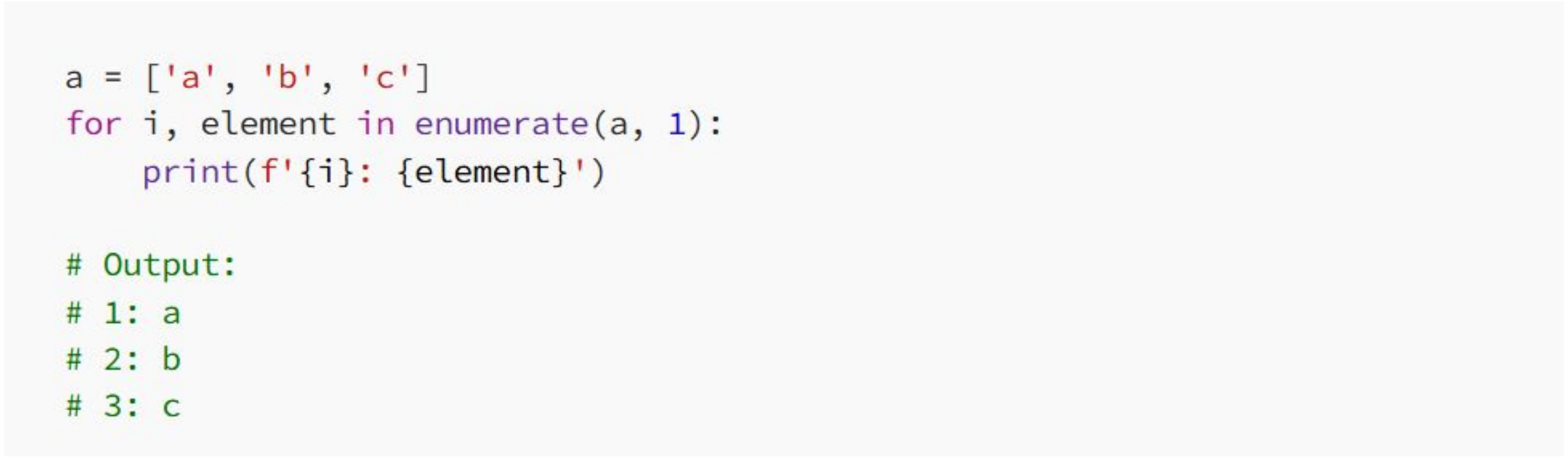
enumerate()

The enumerate() function adds a counter to an iterable and returns an iterable of tuples, where each tuple consists of the counter and the original element.

For example, you can use enumerate() to loop over a list and print the index and value of each element:

```
a = ['a', 'b', 'c']
for i, element in enumerate(a):
    print(f'{i}: {element}')

# Output:
# 0: a
# 1: b
# 2: c
```

You can also specify a start value for the counter as the second argument to enumerate():



```
a = ['a', 'b', 'c']
for i, element in enumerate(a, 1):
    print(f'{i}: {element}')

# Output:
# 1: a
# 2: b
# 3: c
```

we covered some advanced Python functions that can be useful in your code. We looked at the map(), reduce(), filter(), zip(), and enumerate() functions, and provided examples of how to use them. These functions can help you write more efficient and concise code, and are worth exploring further.

Modules and Libraries in Python

1. What is a Module?
A module in Python is a file that contains Python code (functions, classes, or variables) that you can reuse in your programs.

Any Python file with the .py extension is a module.
You can import a module into another program to access its functions or classes.

2. What is a Library?
A library is a collection of modules or packages that provide pre-written functionality to simplify coding tasks.

Example: Libraries like NumPy for arrays, Pandas for data analysis, and Matplotlib for plotting.

3. Importing Libraries and Modules
You can import libraries or modules using the import statement.

Basic Import Syntax:

```python
import math  # Importing the built-in math module


# Using the math module
print(math.sqrt(16))  # Output: 4.0
print(math.pi)        # Output: 3.141592653589793
```

Importing Specific Functions or Classes

```python
from math import sqrt, pi  # Importing specific functions and variables


print(sqrt(25))  # Output: 5.0
print(pi)        # Output: 3.141592653589793
```

```python
import math as m  # Giving an alias to the math module


print(m.sqrt(36))  # Output: 6.0
print(m.pi)        # Output: 3.141592653589793
```

We will study little from this site now…

https://www.learnpython.dev/03-intermediate-python/50-libraries-modules/30-modules-and-imports/

4. Installing External Libraries Using `pip`

What is `pip`?

- `pip` is Python's package manager used to install external libraries that are not built into Python.
- Libraries like NumPy, Pandas, and Matplotlib are installed using `pip`.

Installing a Library

pip install library_name

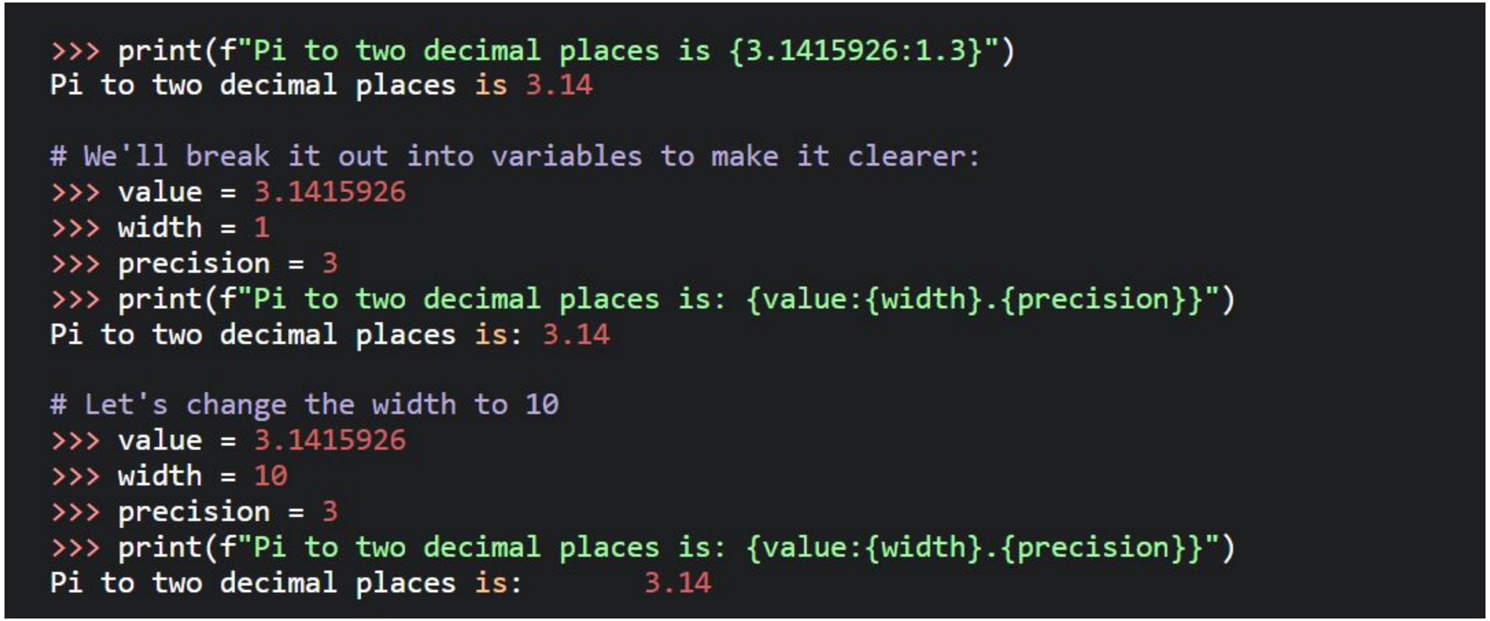
For example:

pip install numpy
pip install pandas

Advanced Strings in Python

Decimal Formatting:

Formatting decimal or floating point numbers with f-strings is easy - you can pass in both a field width and a precision. The format is {value:width.precision}. Let's format pi (3.1415926) to two decimal places - we'll set the width to 1 because we don't need padding, and the precision to 3, giving us the one number to the left of the decimal and the two numbers to the right

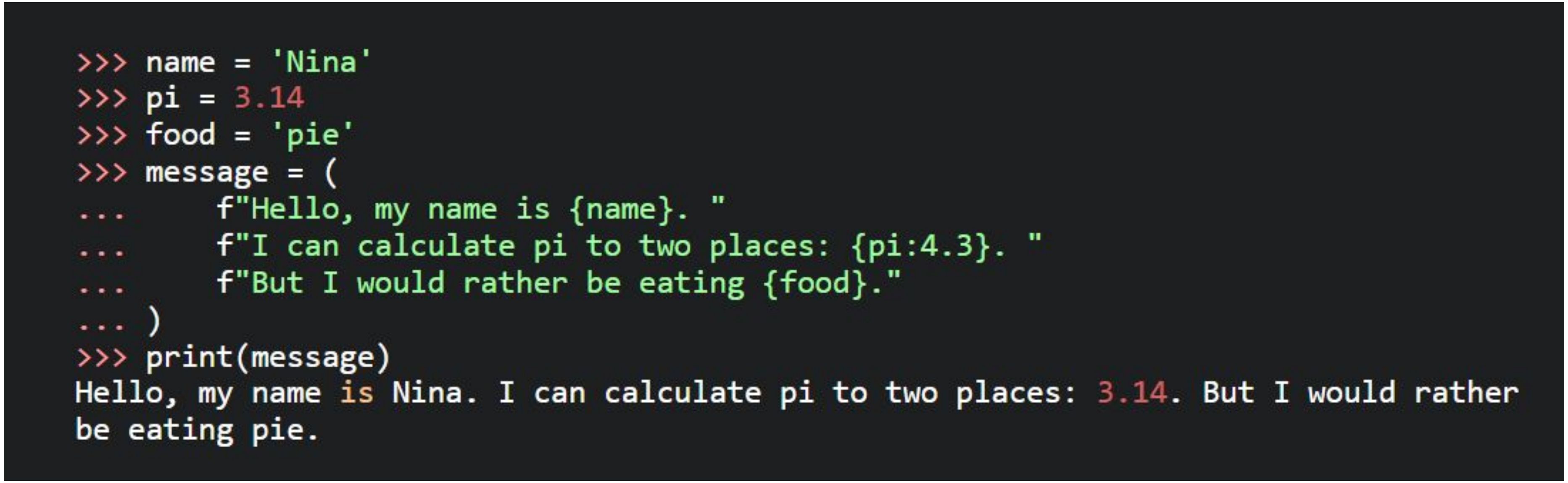


```
>>> print(f"Pi to two decimal places is {3.1415926:1.3}")
Pi to two decimal places is 3.14

# We'll break it out into variables to make it clearer:
>>> value = 3.1415926
>>> width = 1
>>> precision = 3
>>> print(f"Pi to two decimal places is: {value:{width}.{precision}}")
Pi to two decimal places is: 3.14

# Let's change the width to 10
>>> value = 3.1415926
>>> width = 10
>>> precision = 3
>>> print(f"Pi to two decimal places is: {value:{width}.{precision}}")
Pi to two decimal places is:       3.14
```

Note how the second one is padded with extra spaces - the number is four characters long (including the period), so the formatter added six extra spaces to equal the total width of 10.

Multiline Strings

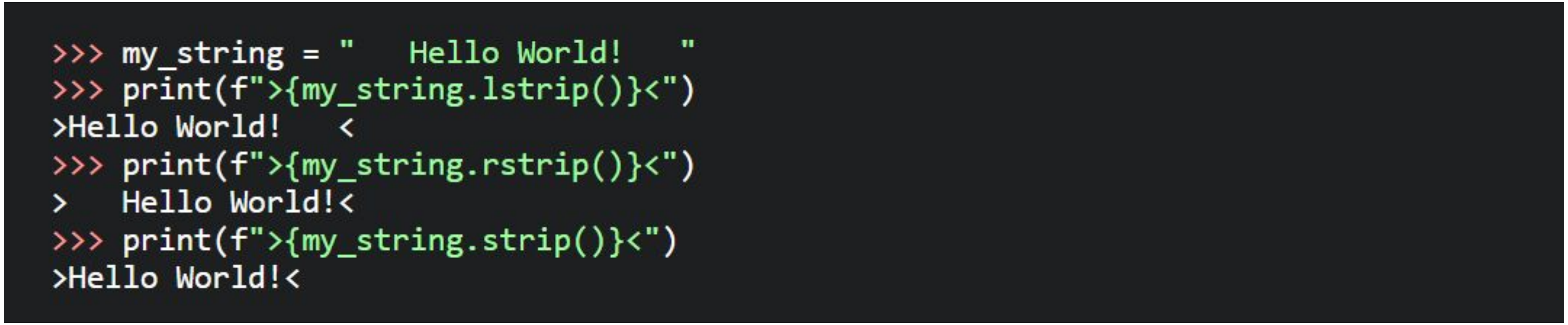
Sometimes it’s easier to break up large statements into multiple lines. Just prepend every line with f:



```
>>> name = 'Nina'
>>> pi = 3.14
>>> food = 'pie'
>>> message = (
...     f"Hello, my name is {name}. "
...     f"I can calculate pi to two places: {pi:4.3}. "
...     f"But I would rather be eating {food}."
... )
>>> print(message)
Hello, my name is Nina. I can calculate pi to two places: 3.14. But I would rather
be eating pie.
```

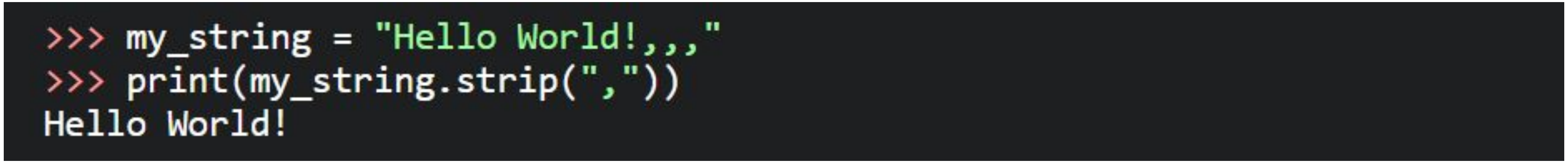
Trimming a string

Python strings have some very useful functions for trimming whitespace. strip() returns a new string after removing any leading and trailing whitespace. rstrip() and does the same but only removes trailing whitespace, and lstrip() only trims leading whitespace. We'll print our string inside >< characters to make it clear:



```
>>> my_string = "   Hello World!   "
>>> print(f">{my_string.lstrip()}<")
>Hello World!   <
>>> print(f">{my_string.rstrip()}<")
>   Hello World!<
>>> print(f">{my_string.strip()}<")
>Hello World!<
```

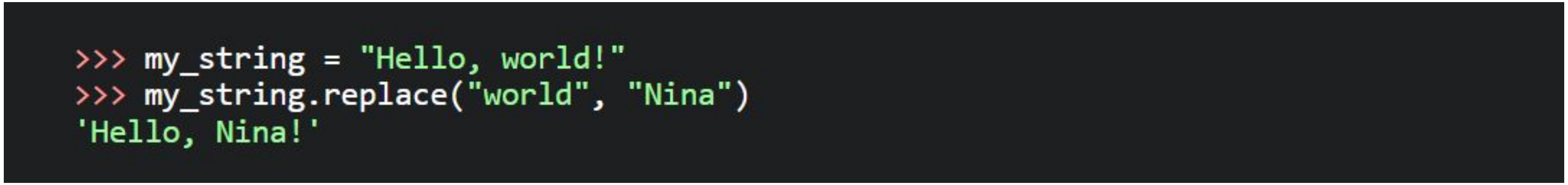
Note the different spaces inside of the brackets. These functions also accept an optional argument of characters to remove. Let's remove all leading or trailing commas:



```
>>> my_string = "Hello World!,,,"
>>> print(my_string.strip(","))
Hello World!
```

Replacing Characters

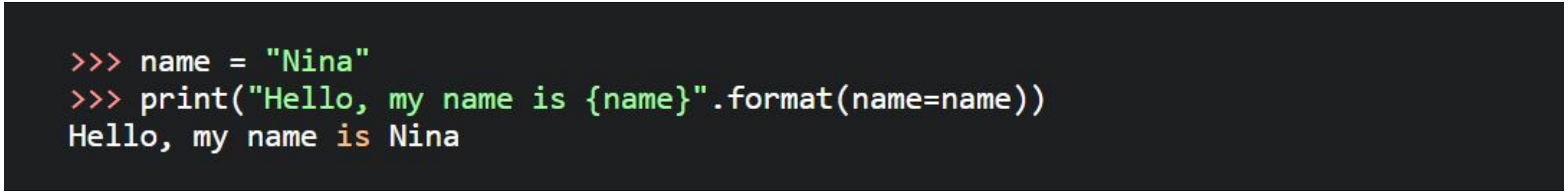
Strings have a useful function for replacing characters - just call replace() on any string and pass in what you want replace, and what you want to replace it with:



```
>>> my_string = "Hello, world!"
>>> my_string.replace("world", "Nina")
'Hello, Nina!'
```

`str.format()` and `%` formatting

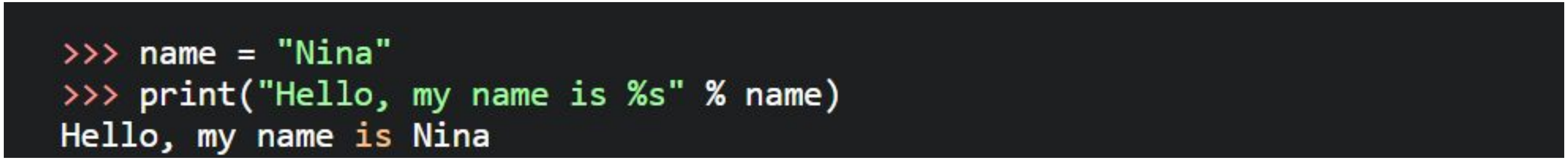
Python has two older methods for string formatting that you'll probably come across at some point. **str.format()** is the more verbose older cousin to f-strings - variables appear in brackets in the string but must be passed in to the **format()** call. For example:



```
>>> name = "Nina"
>>> print("Hello, my name is {name}".format(name=name))
Hello, my name is Nina
```

Note that the variable name inside the string is local to the string - it must be assigned to an outside variable inside the format() call, hence .format(name=name).

%-formatting is a much older method of string interpolating and isn't used much anymore. It's very similar to the methods used in C/C++. Here, we'll use %s as our placeholder for a string, and pass the name variable in to the formatter by placing it after the % symbol.



```
>>> name = "Nina"
>>> print("Hello, my name is %s" % name)
Hello, my name is Nina
```

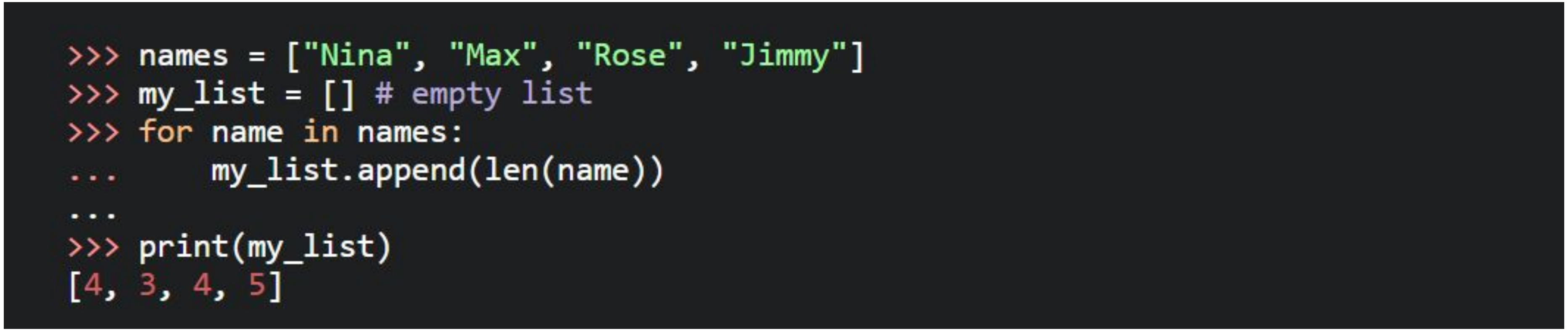
****

Advanced Looping with List Comprehensions

List Comprehensions

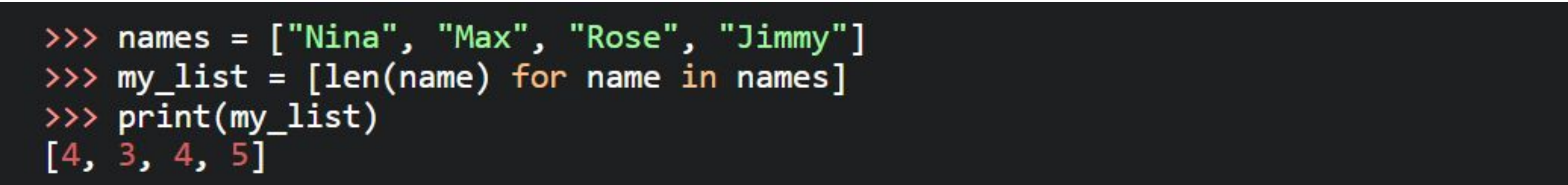
List comprehensions are a unique way to create lists in Python. A list comprehension consists of brackets containing an expression followed by a **for** clause, then zero or more for or **if** clauses. The expressions can be any kind of Python object. List comprehensions will commonly take the form of **[<value> for <vars> in <iter>]**.

A simple case: Say we want to turn a list of strings into a list of string lengths. We could do this with a for loop:



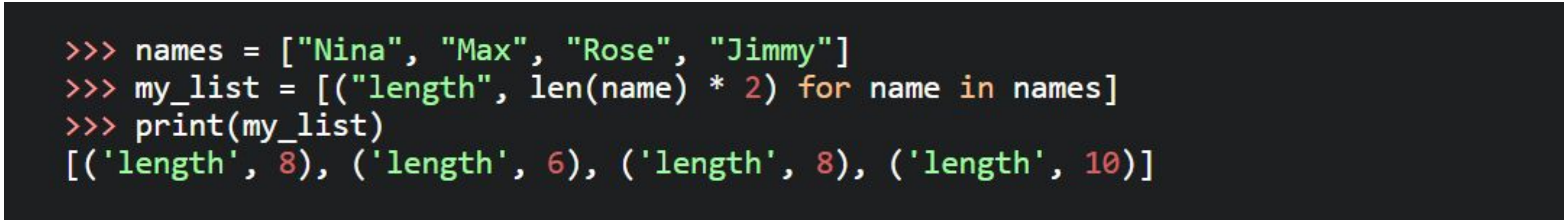
```
>>> names = ["Nina", "Max", "Rose", "Jimmy"]
>>> my_list = [] # empty list
>>> for name in names:
...     my_list.append(len(name))
...
>>> print(my_list)
[4, 3, 4, 5]
```

We can do this much easier with a list comprehension:



```
>>> names = ["Nina", "Max", "Rose", "Jimmy"]
>>> my_list = [len(name) for name in names]
>>> print(my_list)
[4, 3, 4, 5]
```

We can also use comprehensions to perform operations, and the lists we assemble can be composed of any type of Python object. For example:

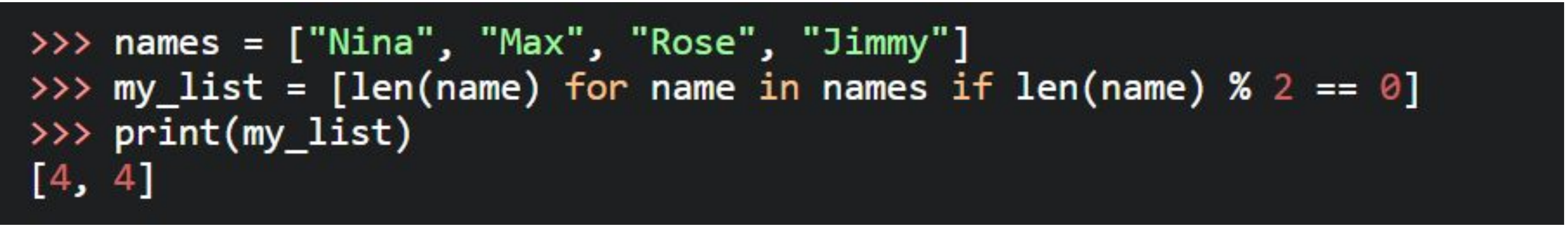


```
>>> names = ["Nina", "Max", "Rose", "Jimmy"]
>>> my_list = [("length", len(name) * 2) for name in names]
>>> print(my_list)
[('length', 8), ('length', 6), ('length', 8), ('length', 10)]
```

In the above example, we assemble a list of tuples - each tuple contains the element “length” as well as each number from the len() function multiplied by two.

Conditionals

You can also use conditionals (if statements) in your list comprehensions. For example, to quickly make a list of only the even lengths, you could do:



```
>>> names = ["Nina", "Max", "Rose", "Jimmy"]
>>> my_list = [len(name) for name in names if len(name) % 2 == 0]
>>> print(my_list)
[4, 4]
```

Here, we check divide every string length by 2, and check to see if the remainder is 0 (using the modulo operator).

String Joining with a List Comprehension

Back in our exercise on converting between types, we introduced the string.join() function. You can call this function on any string, pass it a list, and it will spit out a string with every element from the list “joined” by the string. For example, to get a comma-delimited list of numbers, you might be tempted to do:

```
>>> my_string = ",".join([0, 1, 2, 3, 4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

Unfortunately, you can’t join a list of numbers without first converting them to strings. But you can do this easily with a list comprehension:

```
>>> my_list = [0, 1, 2, 3, 4]
>>> my_string = ",".join([str(num) for num in my_list])
>>> print(my_string)
0,1,2,3,4
```

`sum`, `min`, `max`

Some mathematical functions, such as sum, min, and max, accept lists of numbers to operate on. For example, to get the sum of numbers between zero and five, you could do:

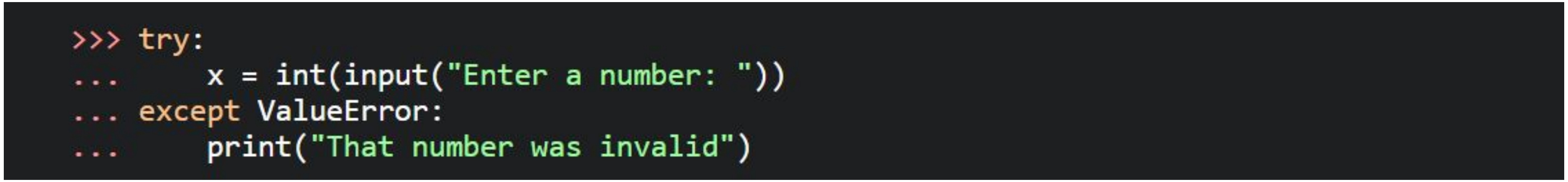
```
my_sum = sum([0, 1, 2, 3, 4])
print(my_sum)
```

But remember, anywhere you can use a list, you can use a list comprehension.
Say you want to get sum, minimum, and maximum of every number between 0 and 100 that is evenly divisible by 3? No sense typing out a whole list in advance, just use a comprehension:

```
>>> my_sum = sum([num for num in range(0, 100) if num % 3 == 0])
>>> print(my_sum)
1683
>>> my_min = min([num for num in range(0, 100) if num % 3 == 0])
>>> print(my_min)
0
>>> my_max = max([num for num in range(0, 100) if num % 3 == 0])
>>> print(my_max)
99
```

Exception Handling:

Many languages have the concept of the “Try-Catch” block. Python uses four keywords: try, except, else, and finally. Code that can possibly throw an exception goes in the try block. except gets the code that runs if an exception is raised. else is an optional block that runs if no exception was raised in the try block, and finally is an optional block of code that will run last, regardless of if an exception was raised. We’ll focus on try and except for this chapter.

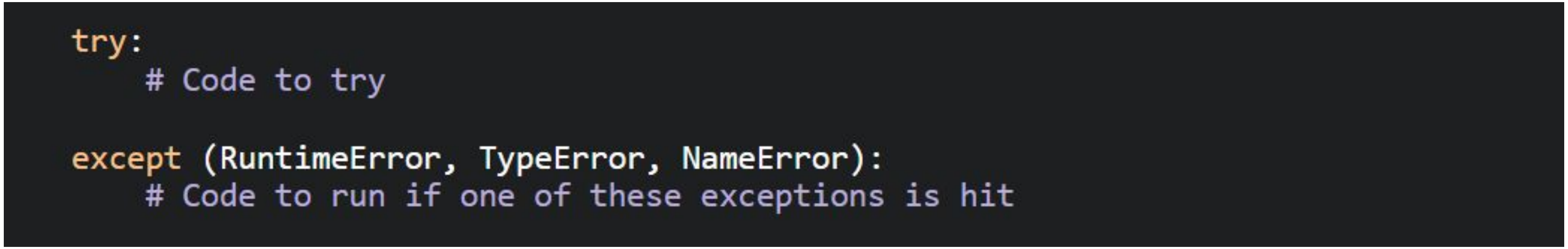


```
>>> try:
...     x = int(input("Enter a number: "))
... except ValueError:
...     print("That number was invalid")
```

First, the try clause is executed. If no exception occurs, the except clause is skipped and execution of the try statement is finished. If an exception occurs in the try clause, the rest of the clause is skipped. If the exception’s type matches the exception named after the except keyword, then the except clause is executed. If the exception doesn’t match, then the exception is unhandled and execution stops.

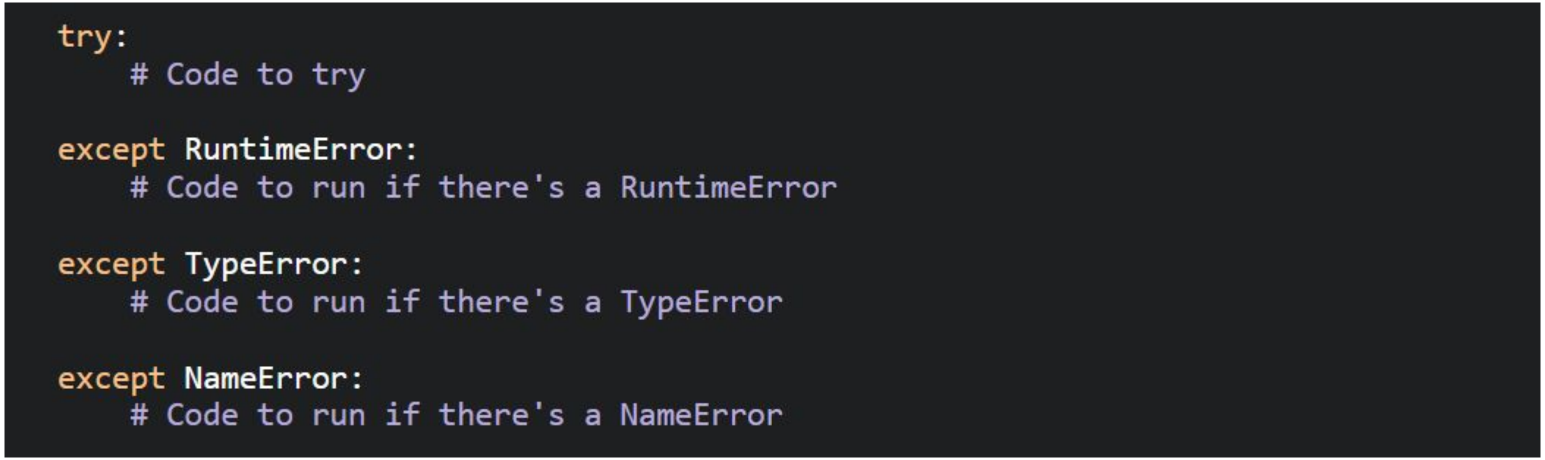
The except Clause

An except clause may have multiple exceptions, given as a parenthesized tuple:



```
try:
    # Code to try

except (RuntimeError, TypeError, NameError):
    # Code to run if one of these exceptions is hit
```

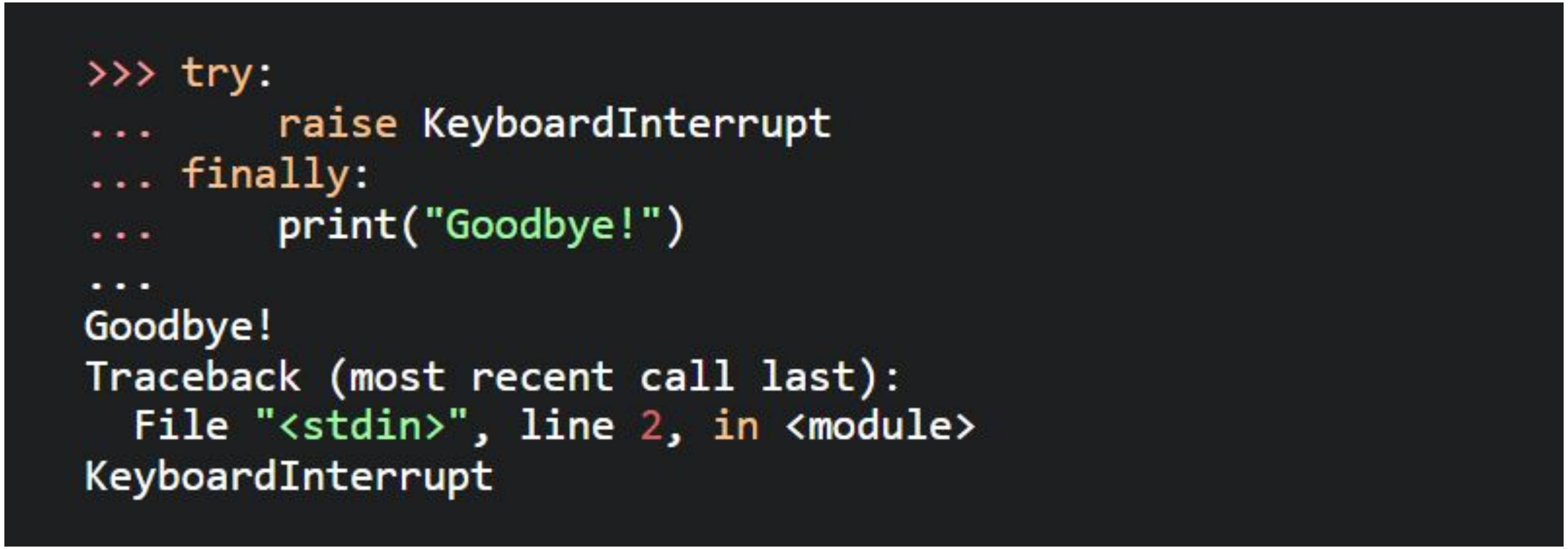
A try statement can also have more than one except clause:



```
try:
    # Code to try

except RuntimeError:
    # Code to run if there's a RuntimeError

except TypeError:
    # Code to run if there's a TypeError

except NameError:
    # Code to run if there's a NameError
```

Finally

Finally, we have finally. finally is an optional block that runs after try, except, and else, regardless of if an exception is thrown or not. This is good for doing any cleanup that you want to happen, whether or not an exception is thrown.



```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print("Goodbye!")
...
Goodbye!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

As you can see, the **Goodbye!** gets printed just before the unhandled **KeyboardInterrupt** gets propagated up and triggers the traceback.