Chapter 1

A deeper look at Python syntax

1.1 Where did we leave off?

In the previous Introduction to Python chapter, We started by installing Python and setting up VSCode, then practiced using the console (terminal) to execute `.py` files. Recall that runs code **sequentially**, one line at a time, and that output appears in the console.

We can output to the screen with `print()`, including how Python formats output by default and how to customize it. Next, we introduced variables and common datatypes such as strings, integers, floats, booleans, lists, and dictionaries.

Mathematical operations, including arithmetic operators were demonstrated. We played with in `str()` and the method to get input from the console, `input()`. Finally, we worked with conditionals and loops to control program flow, then introduced strings and lists as sequence types, and worked with them. With these tools, you should now be able to write basic Python programs that read input, store data, make decisions, repeat actions, and produce useful output.

Let's now build on this foundation with some more advanced concepts, such as functions, classes, and libraries. We will also cover error handling, and experiment with basic graphing ability using the Matplotlib library.

1.2 Functions

As our programs get larger, we begin to repeat the same sets of steps again and again. Rather than copying and pasting code (which is hard to maintain and easy to break), we can group instructions into *functions*. A *function* is a named block of code that performs a task. Once a function is defined, it can be *called* (used) as many times as needed.

You have already used functions before, even if you did not know it! For example, `print()`, `input()`, `len()`, and `type()` are built-in Python functions.

1.2.1 Defining and Calling a Function

To define a function in Python, we use the `def` keyword, followed by:

- a function name
- parentheses `()`
- a colon `:`
- an indented block of code

```
def say_hello():
    print("Hello!")
    print("Welcome to Python.")
```

This code *defines* the function, but it does not run the code inside it yet. To execute the function, we must *call* it by using its name followed by parentheses:

```
say_hello()
```

When Python reaches a function call, it temporarily jumps into the function, runs the indented code, and then returns to the line after the call. Just like conditionals and loops, **indentation matters**.

1.2.2 Parameters and Arguments

Most functions are more useful when they can work with different values. A *parameter* is a variable name listed inside the parentheses of a function definition. An *argument* is the actual value you pass into the function when you call it.

```
def greet(name):
    print("Hello", name)
```

Here, `name` is a parameter. We can call the function with different arguments:

```
greet("Alex")
greet("Chicago")
```

1.2.3 Return Values

Some functions *return* a value. Returning a value allows the function to produce a result that can be stored in a variable or used in an expression.

```
def add(a, b):
    return a + b
```

Now we can use the returned value:

```
x = add(3, 4)
print(x)
```

Output:

```
7
```

We can alter the function to expect a certain parameter type, such as integers, and return a specific type as well:

```
def add(a: int, b: int) -> int:
    return a + b
```

It is important to note that Python won't raise a TypeError if you pass in the wrong type, but this is a helpful way to document your code for other programmers, which you may often work with, and yourself.

We can force the parameters to be a certain type, and raise an error if the wrong type is passed in:

```
def add(a: int, b: int) -> int:
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("Both arguments must be integers")
    return a + b
```

Important: `print()` displays information to the console, but it does not return a useful value. `return` sends a value back to where the function was called. Functions that do not have a `return` statement return `None` by default, and are often called *void* functions.

1.2.4 Scope

A variable created inside a function only exists inside that function. This idea is called *scope*. If you define a variable inside a function, you cannot use it outside of the function unless you return it.

```python
def make_number():
    x = 10
    return x

y = make_number()
print(y)
```

The variable `x` exists only inside `make_number()`, but the value it returns is stored in `y` outside the function.

1.2.5 Summary

- Functions group code into reusable blocks
- Functions are defined with `def` and called using parentheses
- Parameters receive input values (arguments) when a function is called
- `return` sends a value back to the caller
- Variables inside a function have local scope

Exercise 1 Tracing a Function Call

Consider the following code:

```python
def double(x):
    return x * 2

a = 3
b = double(a)
print(b)
```

What is printed? What is the value of `a` after the program runs?

Working Space

Answer on Page 19

Exercise 2 Write a Function

Write a function called `is_even(n)` that returns `True` if `n` is even and `False` otherwise. Then show an example call to your function using `n = 7`.

Working Space

Answer on Page 19

1.3 Classes and More Basic Object-Oriented Programming

So far, we have worked mostly with individual variables, lists, and functions. As programs grow larger, it becomes useful to group related data and behavior together. *Object-Oriented Programming* (OOP) is a programming style that organizes code around *objects* rather than individual functions.

In Python, objects are created from *classes*. A *class* is a blueprint for creating objects that contain both data (variables) and behavior (functions).

1.3.1 Defining a Class

A class is defined using the `class` keyword, followed by:

- the class name (by convention, written in `CamelCase`)
- a colon `:`
- an indented block of code

```
class Person:
    pass
```

This defines an empty class. It does not do anything yet, but it gives Python a new type called `Person`.

1.3.2 Creating Objects

An *object* is an instance of a class. Objects are created by calling the class name like a function.

```
p1 = Person()
p2 = Person()
```

Here, `p1` and `p2` are two separate objects, both created from the same class.

1.3.3 The `__init__` Method

Most classes need to store data. This is done using a special function called `__init__`. This function runs automatically when a new object is created.

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

The parameter `self` refers to the current object being created. Each object gets its own copy of the variables defined using `self`.

```
p = Person("Alex", 20)
```

Now the object `p` has two attributes:

- `p.name`
- `p.age`

1.3.4 Accessing Object Attributes

Attributes are accessed using dot notation.

```
print(p.name)
print(p.age)
```

Output:

```
Alex
20
```

Each object stores its own values. Creating another object does not overwrite existing ones.

1.3.5 Methods

Functions defined inside a class are called *methods*. Methods describe behavior that belongs to the object.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is", self.name)
```

Calling a method uses dot notation:

```python
p = Person("Alex", 20)
p.greet()
```

Output:

```
Hello, my name is Alex
```

1.3.6 Printing Objects

By default, printing an object produces an unreadable result:

```python
print(p)
```

Output (example):

```
<__main__.Person object at 0x7f9c1a3d>
```

This actually is the object's memory address in the computer's memory, but it may be a bit tedious want to print that and manually search ever byte. Instead, to control how an object is printed, we can define the special method `__str__`.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} ({self.age} years old)"
```

Now printing the object gives a meaningful result:

```python
p = Person("Alex", 20)
print(p)
```

Output:

```
Alex (20 years old)
```

1.3.7 Why Use Classes?

Classes allow us to:

- group related data and behavior together
- create many objects with the same structure
- write code that is easier to understand and maintain

1.3.8 Summary

- A class is a blueprint for creating objects
- Objects store data using attributes
- `__init__` initializes new objects

- Methods define behavior for objects
- `__str__` controls how objects are printed
- This was only a foundation of OOP; more advance concepts exist and are stronger in other programming languages such as Java and C++

Exercise 3 Tracing an Object

Consider the following code:

```
class Counter:
    def __init__(self, value):
        self.value = value

    def increment(self):
        self.value += 1

c = Counter(5)
c.increment()
c.increment()
print(c.value)
```

What is printed?

Working Space

Answer on Page 19

Exercise 4 Custom Printing

Write a class called `Book` that stores a title and an author. Add a `__str__` method so that printing a book displays:

```
Title by Author
```

Working Space

Answer on Page 19

1.4 Libraries

A *library* is a collection of pre-written code that provides additional functionality without requiring you to write everything from scratch. Python includes many built-in libraries that can be used to perform common tasks such as mathematical calculations, data handling, and visualization. You'll often find that code you are seeking can be found through an open-source library which already exists.

To use a library, it must first be imported.

1.4.1 Importing Libraries

The simplest way to import a library is using the `import` keyword.

```
import math
print(math.sqrt(16))
```

In this example, the `math` library provides access to mathematical functions such as square roots.

It is also possible to import specific values or functions from a library.

```
from math import pi
print(pi)
```

This allows direct access to `pi` without referencing the library name.

1.4.2 Summary

- Libraries extend Python's functionality
- The `import` keyword is used to load libraries
- Specific components can be imported directly when needed

1.5 Matplotlib

Matplotlib is the most widely used plotting library in Python. It can produce simple charts quickly (line plots, scatter plots, bar charts, histograms) and also supports publication-quality figures with precise control over labels, legends, and layout. We will use it widely throughout this course to visualize data, create simulations for proving formulas, and experiment with datasets.

1.5.1 Installing and importing

If you are working locally, you can install Matplotlib with:

```
pip install matplotlib
```

In most scripts, you will import the plotting interface `pyplot`:

```
import matplotlib.pyplot as plt
```

1.5.2 A simple line graph

A line plot is ideal for showing how a value changes over time or across an ordered variable. Let's create `line.py`

```
import matplotlib.pyplot as plt # the matplotlib library, which we will call
↪  using plt

# two lists containing values
x = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

# a 2d plot, used for line graphs
plt.plot(x, y)
# add a title to our plot
plt.title("A Simple Line Plot")
# add axis labels
plt.xlabel("x")
plt.ylabel("y = x^2")
plt.show() # pops up a new window
```

Running `python3 line.py` will open a new *interactive window* in your computer, with the following image:

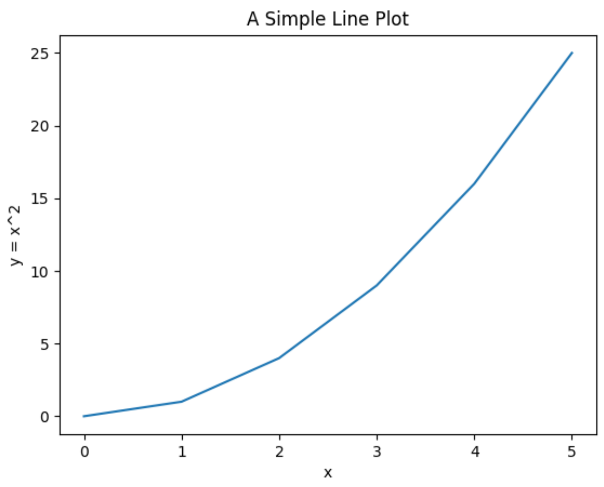

Figure 1.1: The output of line.py.

1.5.3 Labels, legends, and grids

A plot is more useful when it clearly communicates what each element represents. Let's add on to `line.py` by creating `lineTwoOutputs.py` and adding labels, a legend, and another set of data.

```python
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y1 = [0, 1, 4, 9, 16, 25]
y2 = [0, 1, 8, 27, 64, 125]

plt.plot(x, y1, label="x^2")
plt.plot(x, y2, label="x^3")

plt.title("Two Functions on One
↪  Axes")
plt.xlabel("x")
plt.ylabel("value")
plt.grid(True)
plt.legend()
plt.show()
```

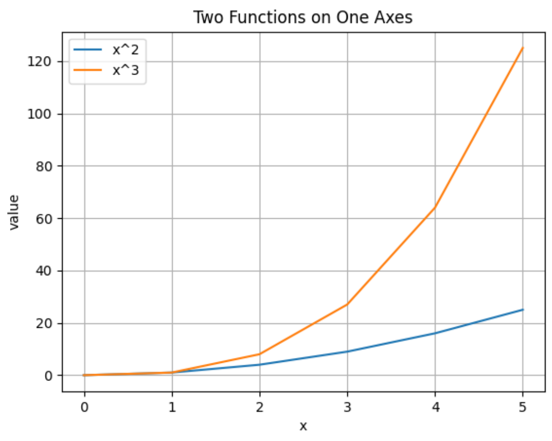

Figure 1.2: $y = x^2$ and $y = x^3$ on the same graphed.

1.5.4 Visualizing Relationships with scatter plots

Scatter plots are excellent for showing how two variables relate (for example, height and weight).

This very basic scatter plot shows the relationship between studying time and score.

```
import matplotlib.pyplot as plt

hours = [1, 2, 3, 4, 5, 6]
scores = [55, 60, 66, 72, 78,
↪  85]

plt.scatter(hours, scores)
plt.title("Study Time vs.
↪  Score")
plt.xlabel("hours studied")
plt.ylabel("score")
plt.show()
```

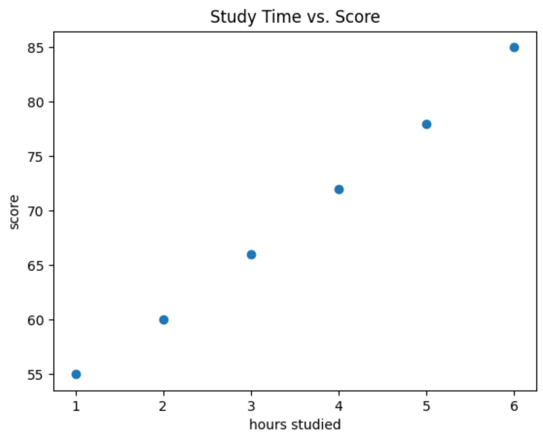

Figure 1.3: A scatterplot generated by matplotlib from the given data.

1.5.5 Histograms and Distributions

A histogram shows how values are distributed and how common different ranges are.

```
import matplotlib.pyplot as plt
# a set of data, for example maybe
↪ these are all test scores
data = [4, 5, 5, 6, 7, 7, 7, 8, 8,
↪ 9, 10, 10, 10, 11, 12]

# creates a histogram plot
plt.hist(data, bins=5)
plt.title("Histogram of Values")
plt.xlabel("value")
plt.ylabel("frequency")
plt.show()
```

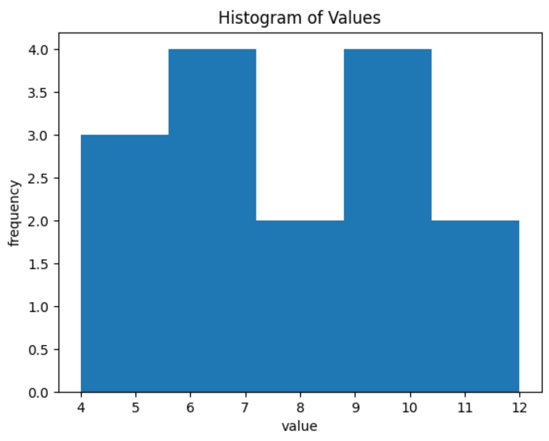

Figure 1.4: A histogram of a set of data.

1.5.6 The object-oriented approach (recommended)

Matplotlib has two main styles:

- **State-based** (using `plt.plot`, `plt.title`, ...): quick and convenient.
- **Object-oriented** (creating a `Figure` and `Axes`): more explicit and easier to scale.

For multi-plot layouts or different analysis of datasets, prefer the object-oriented approach:

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y1 = [0, 1, 4, 9, 16, 25]
y2 = [0, 1, 8, 27, 64, 125]

# two figures, side by side. axes becomes an indexable list
fig, axes = plt.subplots(nrows=1, ncols=2)

axes[0].plot(x, y1)
axes[0].set_title("x^2")
axes[0].set_xlabel("x")
axes[0].set_ylabel("value")

axes[1].plot(x, y2)
axes[1].set_title("x^3")
axes[1].set_xlabel("x")
axes[1].set_ylabel("value")
```

```
fig.suptitle("Two Subplots")
fig.tight_layout()
plt.show()
```

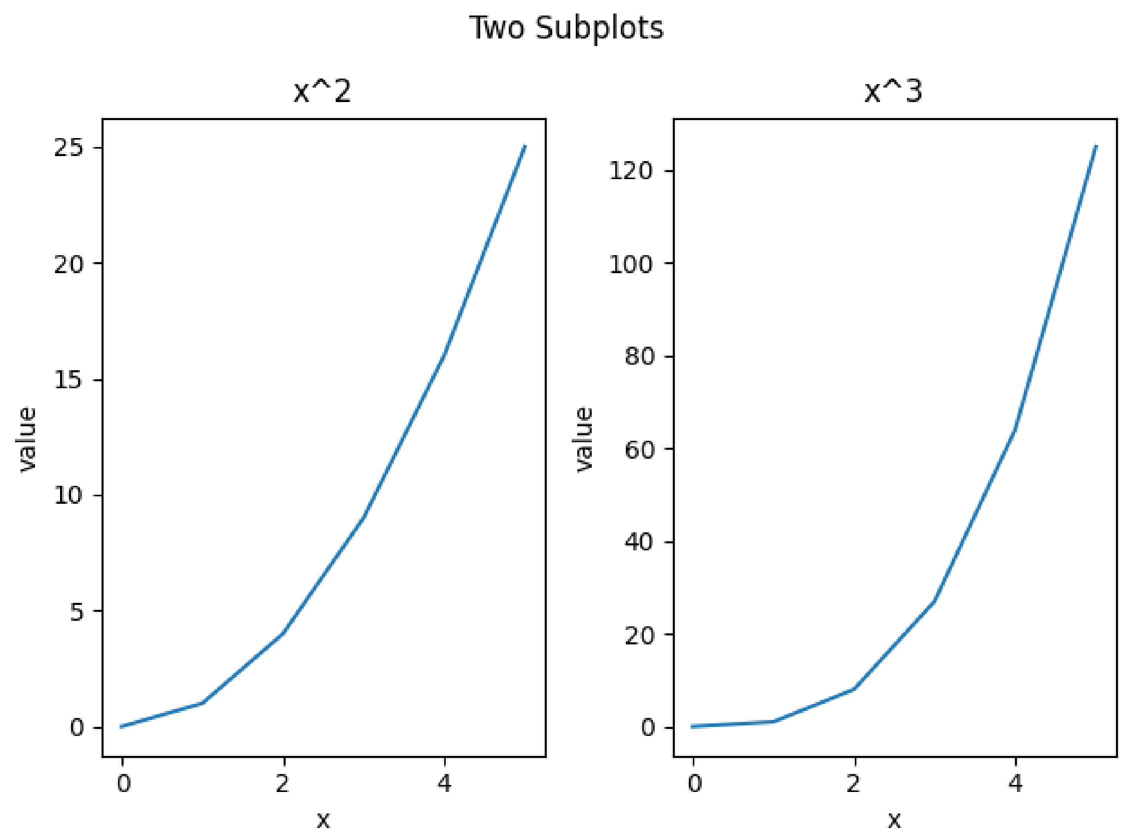

Figure 1.5: The two plots, generated side by side.

1.6 Errors

Even the most experienced programmers can make mistakes. If the code seems to run into issues, your code may crash or output an *error*. Python reports errors by raising a halt in the console output, which include a message explaining the problem and the exact line where it occurred. By learning how to read and interpret these messages, you'll be able to *debug*, or fix issues, in your programs more efficiently and write more reliable code.

You will encounter two main types of errors in your code: **Syntax Errors** and **Runtime Errors/Exceptions**.

- Syntax errors occur when your the formatting, or *syntax*. These errors can usually be found before your code is compiled.

- Runtime Errors, or *Exceptions*, occur when operations in your code cause an invalid calculation, or attempt to do an invalid action. These can range anywhere from basic misnamed variables to imported libraries crashing from incorrect data.

1.6.1 Syntax Errors

What do you notice is immediately wrong with this code?

```
    x = "Welcome Home"
    print(type(x)
```

If we run it, we get the output:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/usr/lib/python3.12/py_compile.py", line 150, in compile
    raise py_exc
py_compile.PyCompileError:   File "./prog.py", line 2
    print(type(x)
         ^
SyntaxError: '(' was never closed
```

We notice that every opening parentheses, bracket, or brace must have a closing supplement. Without out, we run into Syntax Errors, letting us know our format is off.

Another type of Syntax Error can be found in incorrect indentation. Take for example the following code:

```
if True:
print("Hello!")
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/usr/lib/python3.12/py_compile.py", line 150, in compile
    raise py_exc
py_compile.PyCompileError: Sorry: IndentationError: expected an indented block
↪   after 'if' statement on line 1 (prog.py, line 2)
```

Here, there is no indentation (usually obtained by pressing the `Tab` key on your keyboard) for lines under the conditional statement. This causes a Syntax Error to be raised.

1.6.2 Exceptions

Let's say I try to run the following code:

```
    print(x)
    x = "hello"
```

You may see the following output:

```
Traceback (most recent call last):
    File "./program.py", line 1, in <module>
NameError: name 'x' is not defined
```

You have encounted a *NameError*, because `x` was not assigned before it was attempted to be printed. This brings up an important note on Python code: **code is executed line-by-line, sequentially**. So even if you define `x` after you try and print it, Python will not understand what you are trying to print. Let's look at another example:

1.6.3 Try-Except

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Here, we attempt to divide 10 by some input number `x`. There are two `Exceptions` that could cause this to fail:

- The user inputs a string a characters, or anything that isn't an integer
- The user inputs 0, an invalid divisor.

We use a `Try-Except` block to check for different errors generated, similar to an if statement. The try block surrounds a block of code that may cause faulty runtime output or errors.

Exercise 5 Fix the Code

Working Space

```
items = ["pen", "book", "eraser",
↪  "ruler"]

index = input("Enter an index (0 to
↪  3): ")
print("You chose:", items[index])

items.remove("pencil")
print("Updated list:", items)
```

Name two errors that could occur when this code is run. For each error, explain why it occurs and how to fix it.

Answer on Page 20

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 4)

The function returns `3 * 2`, which is `6`, so the program prints:

```
6
```

The value of `a` is still `3` because the function does not change `a`; it only uses its value to compute a result.

Answer to Exercise 2 (on page 5)

```
def is_even(n):
    return n % 2 == 0

print(is_even(7))
```

This prints `False` because 7 is not divisible by 2.

Answer to Exercise 3 (on page 9)

The initial value is 5. The `increment()` method is called twice, so the value becomes 7. The program prints:

```
7
```

Answer to Exercise 4 (on page 9)

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"
```

Answer to Exercise 5 (on page 18)

1. **TypeError**: The `input()` function returns a string, so when the user inputs an index, it is treated as a string. Attempting to use this string as an index for the list will raise a `TypeError`. To fix this, we can cast the input to an integer using `int()` and handle the potential `ValueError` if the input is not a valid integer.
2. Pencil is not in the list, so attempting to remove it will raise a `ValueError`. To fix this, we can use a try-except block to catch the `ValueError` and print a message indicating that the item was not found in the list.

Index