Chapter 1

Introduction to Python

In this chapter we will discuss the very basics of Python. No we are not talking about snake anatomy, but the programming language Python. Python is a high-level, interpreted programming language known for its readability and versatility. It is widely used in various fields such as web development, data analysis, artificial intelligence, scientific computing, and more. It can be used for data parsing, visualization, and even machine learning. Python's syntax is designed to be easy to read and write, making it an excellent choice for beginners and experienced programmers alike. Throughout this chapter, we encourage you to write out the provided lines of code yourself! This will reinforce skills like computer literacy, Python syntax and troubleshooting, and of course, typing speed.

1.1 Getting Started with Python

To get started with Python, you need to have it installed on your computer. You can download the latest version of Python from the official website: `https://www.python.org/downloads/`. Follow the installation instructions for your operating system.

Once you have python installed, you can write and run Python code using various methods:

- **Interactive Shell:** You can open a terminal or command prompt and type `python` or `python3` to start an interactive Python shell where you can type and execute Python commands line by line.
- **Script Files:** You can write your Python code in a text file with a `.py` extension and run it using the command `python filename.py` or `python3 filename.py` in the terminal.
- **Integrated Development Environments (IDEs):** There are several IDEs available for Python, such as PyCharm, VSCode, and Jupyter Notebooks, which provide a more user-friendly environment for writing and running Python code. We will assume you will use VSCode, as it is widely used among computer scientists and devlopers around the world, and has direct integration to GitHub.

We are going to install VSCode as our IDE for this course. You can download it from `https://code.visualstudio.com/`. After installing VSCode, you will also need to install the Python extension for VSCode, which can be found in the Extensions Marketplace within VSCode.

1.1.1 The Console and Running Python Programs

When working with Python, it is important to understand the difference between *writing code* and *executing code*. Writing code simply means typing instructions into a file using a text editor or IDE. Executing code means telling Python to read those instructions and perform them.

The *console*, also called the *terminal* or *command prompt*, is a text-based interface where you can run programs and view their output. Any text printed using the `print()` function will appear in the console.

When you run a Python program, Python reads your file from top to bottom and executes each line in order. This is an important idea that will come up often: **Python code is executed sequentially, one line at a time**.

1.1.2 Running a Python File

To run a Python file, you must use the console. In VSCode, you can open the integrated terminal by selecting `Terminal` → `New Terminal` from the menu. Make sure your terminal is open in the same folder as your Python file.

If your file is named `hello.py`, you can run it by typing:

```
python hello.py
```

or, on some systems:

```
python3 hello.py
```

After pressing Enter, Python will execute the file and display any output in the console. If there are errors in your code, Python will stop execution and display an error message instead. You may also see a green play button at the top of your window, you could also run your program with that!

At this point, it is helpful to remember:

- Writing code does nothing by itself
- Code only runs when you explicitly execute the file
- Output from `print()` appears in the console

1.2 Output `print`

Let's talk more about the `print()` statement. It can take a string, multiple variables, or a combination of strings and variables. We are going to introduce the syntax first, and then expand on the definitions later.

```
x = "This is a variable."
print("This is a string.", x)
```

This should output `This is a string. This is a variable`. You can also do multiple variables:

```
x = "Good"
y = "Morning,"
z = "Chicago"
print(x, y, z)
```

This will output `Good Morning, Chicago`. Alternatively, printing can have addition within it:

```
x = "Good"
y = "Morning,"
z = "Chicago"
print(x+y+z)
```

This will output `GoodMorning,Chicago`. Notice the difference!

If you want to combine numbers and strings,

```
x = "it is"
y=78
z="degrees outside"
print(x+y+z)
```

This will not work! Instead, we should use commas:

```
x = "it is"
y=78
z="degrees outside"
print(x, y, z)
```

This will correctly output "it is 78 degrees outside"

By default, a space is added in between each argument (where the commas would be).

We can change what seperates each variable by the `sep=` argument.

```
x = "it is"
y=78
z="degrees outside"
print(x, y, z, sep="..!..")
```

This will output `it is..!..78..!..degrees outside`

By default, Python includes a new line at the end of each print statement. We can alter this to anything else by using the `end=` argument:

```
print("Good Morning, Chicago.", end="!!")
```

Outputting `Good Morning, Chicago.!!` to the console, with no new line character inserted. Note that the next print statement will continue on the same line unless it contains a new line character.

```
print("Hello World!", end=" ")
print("I will print on the same line.")
```

We have used all this terminology like strings and variables a lot, let's dive a bit deeper into it!

1.3 Datatypes and Variables

There are many different datatypes in Python, and all of them can be passed to `print` function. Here's the main few:

Strings Strings are just a sequence of characters. Anything closed between quotes gets included in the string, such as "This is the Python Chapter for the Kontinua Sequence!!%& × ()-+=1234567890 &"

Integers Integers are any whole number, positive negative or 0. Theoretically, integer values can go on for infinity, but it is important to understand they take up space in computer memory.

Floats Floats, or floating point values, are numbers that include decimal places. Division can be done between integers and floats but may result in a different value.

Booleans Booleans only two values: `True` or `False`. The output of conditional statements (such as if's or while's) are booleans. They only answer Yes and No questions, and are important on deciding between two possible paths in code.

Sequence Types Sequences are consecutive lists or pairs of other data types can be represented in various froms: `list`, `set`, and `tuple`. Values are accessed by positions.

Mapping Types The main mapping type is a `dict`, short for dictionary. Mapping types store values as key–value pairs instead of positions. You access items using a key, not an index.

Let's create some of these in our code:

```
x = 5
y = 5.6
z = True
a = [x, y, z]
```

A *variable* is a container for information, usually containing one of the datatypes established above. In Python, variables are assigned using the assignment operator =, with the name of the variable on the left side of the operator and the value it is assigned to on the right side.

Python has specific naming rules for variables. The name

- Must start with a letter or underscore
- Cannot contain spaces
- Are case-sensitive

In our code above, `x` is a *variable* containing the number 5, an Integer. `y` contains the float 5.6, while `z` is a boolean with the value `True`. We then created a list containing the variables `x`, `y`, and `z`.

A unique feature of Python is that variables can change their datatypes even after assignment. We can see the datatype of an object using the `type()`

```
x = 10
print(x)
print(type(x))

x = "Hello!"
print(x)
print(type(x))
```

Output:

```
10
<class 'int'>
Hello!
<class 'str'>
```

Notice that, because of that feature, the datatype of a variable is not defined beforehand. This is a core difference between Python and other programming languages like Java or C++, which we will explore in the future.

We can also define our floats using scientific notation.

```
x = 1e3      # 1000.0
y = 2.5e-4   # 0.00025
```

Exercise 1 Identify the variable type

Working Space

Below are 5 variables defined in Python. Identify their type as `str`, `float`, `bool`, `list`, or `int` by filling in the table below.

```
a = 10
b = 3.5
c = "10"
d = True
e = [1, 2, 3]
```

Variable	Value	Data Type
a	10	
b	3.5	
c	”10”	
d	True	
e	[1, 2, 3]	

Answer on Page 25

Exercise 2 What does this do?

What is the output of the following code?

```
print(type(3.0) == type(3))
```

Working Space

Answer on Page 25

1.4 Operations

Operations can be done on both variables and numbers alike. Most commonly they will be used for mathematical operations or simplifying operations of equivalence.

1.4.1 Arithmetic Operations

Math operations in Python are very similar to standard math operations. We need them for any type of data operation or math parsing.

```
a = 100
b = 3
c = 100.0
d = -100
print(a + b) # addition
print(a - b) # subtraction
print(a * b) # multiplication
print(a / b) # division (results in a float)
print(a % b) # modulus (remainder) operator - useful for division checks
print(a ** b) # exponentiation
print(a // b) # floor division
print(c // b) # floor division as a float
print(d // b) # negative flooring goes further DOWN
```

Output:

```
103
97
300
33.333333333333336
```

```
    1
    1000000
    33
    33.0
    34
```

Note that division by `0` is not a valid operation, just as in algebra. This will cause an `ZeroDivisionError`.

1.4.2 Augmented Assignment Operators

You may want to do the following operation

```
x = 0
while True:
    print(x)
    x = x + 1
```

This would print the counting numbers, increasing by 1, forever. The `x = x + 1` rewrites the value of `x` equal to the current value of `x`, and adds one to it. There is a short hand for this kind of operation, called an *augmented assignment operator*, which combines assignment operators and operations. These are commonly used in loops and accumulation counters:

```
x = 10
x += 5   # same as x = x + 5
x -= 3   # same as x = x - 3
x *= 2   # same as x = 2 * x
x /= 4   # same as x = x / 4
```

Following order of operations, `x = 6` after all lines are executed.

1.4.3 Summary

- Math operations are done with `+`, `-`, `*`, and `/`
- Augmented Operations `+=`, `-=`, `*=`, `/=` are common for loops and counting operations.

Exercise 3 Tracing Chart

Let's create a tracing chart. What happens after each line of code is run?

```
n = 5
m = "3"
n = n + 1
m = m + "1"
```

Line Executed	n (value, type)	m (value, type)
After line 2		
After line 3		
After line 4		

Working Space

Answer on Page 25

1.5 Input and Output

So far, we have only displayed information using the `print()` function. While output is useful, most programs also need a way to receive information from the user. In Python, this is done using the built-in `input()` function.

1.5.1 Getting User Input

The `input()` function pauses the program and waits for the user to type something into the console. Whatever the user types is then returned and can be stored in a variable.

```
name = input("Enter your name: ")
print("Hello,", name)
```

In this example:

- The text inside `input()` is displayed as a prompt in the console
- The program waits for the user to type a response and press Enter

- The entered text is stored in the variable `name`
- The `print()` function then outputs a greeting using that value

1.5.2 Input Always Returns a String

A very important rule in Python is that `input()` **always returns a string**, even if the user types a number. This means that mathematical operations cannot be performed on input values unless they are converted to a numeric type.

Consider the following incorrect example:

```
age = input("Enter your age: ")
print(age + 1)
```

This code will result in an error, because Python cannot add a number to a string.

To perform calculations, the input must be *cast* to a different datatype.

1.5.3 Casting Input Values

Casting is the process of converting one datatype into another. Python provides built-in functions such as `int()`, `float()`, and `str()` for this purpose.

```
age = int(input("Enter your age: "))
print(age + 1)
```

Here:

- `input()` returns a string
- `int()` converts that string into an integer
- The program can now perform arithmetic operations

1.5.4 Common Input Errors

If the user enters something that cannot be converted to the requested datatype, Python raises a runtime error called a `ValueError`. For example:

```
age = int(input("Enter your age: "))
```

If the user types `hello` instead of a number, Python will produce an error similar to the following:

```
ValueError: invalid literal for int() with base 10
```

This error occurs because the string `"hello"` cannot be converted into an integer. Handling these types of errors will be discussed later when we introduce `try-except` statements.

1.5.5 Summary

- `print()` is used to display output to the console
- `input()` is used to receive text input from the user
- `input()` always returns a string
- Casting is required to convert input into numeric types

Exercise 4 User Input Types

```
x = input("Enter a number: ")
print(x + 5)
```

What will happen when this code is run and the user enters `10`? If there is an issue, explain what it is, why it occurs, and how to fix it.

Working Space

Answer on Page 25

1.6 Conditionals, Loops, and Match-Case

Most programs need to make decisions and repeat actions. Programmers do not want to write the same code multiple times, as this introduces *redundancy*. In Python, this is done using *conditionals* and *loops*. Conditionals allow the program to choose between different paths of execution, while loops allow code to be run multiple times.

1.6.1 Conditional Statements

Conditional statements execute code only if a given condition is true. Python uses the keywords `if`, `elif`, and `else` to define these conditions.

```
x = 10

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

In this example:

- The expression after `if` is evaluated as a boolean
- If the condition is `True`, the indented block runs
- If it is `False`, Python checks the next condition. The `elif` block is the only way to check another branch after one is returned `False`. The `else` branch is a last resort, and only runs if all previous statements are false
- Only one branch of the conditional will execute

It is important to note that **indentation is required** in Python. All statements belonging to a conditional block must be indented at the same level.

1.6.2 Boolean Expressions

Conditions are built using comparison and logical operators. These expressions always evaluate to either `True` or `False`.

```
a = 5
```

```
b = 10

print(a < b)   # less than
print(a == b)  # equal to
print(a != b)  # not equal to
```

Logical operators can be used to combine conditions. Condtions can be *strung together* using `and`, `or`, and `not`

```
x = 7

if x > 0 and x < 10:
    print("x is between 1 and 9")
if x > 0 or x < -3:
    print("x is greater than 0 or less than -3")

y = False
print(not y)
```

1.6.3 Loops

Loops allow code to be repeated multiple times. Python provides two main types of loops: `for` loops and `while` loops.

To define a loop, Python uses the keywords `for` and `while`, followed by a condition or sequence to iterate over.

- the loop header, with correctly defined endpoints
- a colon `:`
- an indented block of code to iterate over

1.6.4 For Loops

A `for` loop is used to iterate over a sequence, such as a list or a range of numbers.

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

This code will print the numbers 0 through 4. The `range()` function generates a sequence of integers starting at 0 and stopping before the given value.

```
numbers = [10, 20, 30]

for n in numbers:
    print(n)
```

1.6.5 While Loops

A `while` loop repeats as long as a condition remains true.

```
x = 5

while x > 0:
    print(x)
    x -= 1
```

In this example:

- The condition is checked before each iteration
- The loop stops once the condition becomes `False`

Care must be taken to ensure that the condition eventually becomes false. Otherwise, the program will enter an infinite loop, one which never stops and the computer will run forever, risking hardware component issues.

1.6.6 Breaking and Continuing Loops

Python provides keywords to control loop execution.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

The `break` statement immediately exits the loop.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

The `continue` statement skips the current iteration and moves to the next one.

1.6.7 Match-Case Statements

Python also provides the `match-case` statement, which allows a value to be compared against several possible patterns. This is similar to a switch statement in other programming languages.

```
command = input("Enter a command: ")

match command:
    case "start":
        print("Starting program")
    case "stop":
        print("Stopping program")
    case "pause":
        print("Pausing program")
    case _:
        print("Unknown command")
```

The underscore (_) acts as a default case if no other patterns match.

1.6.8 Summary

- Conditionals allow programs to make decisions
- Boolean expressions evaluate to `True` or `False`
- Loops allow code to be repeated efficiently
- `for` loops iterate over sequences
- `while` loops repeat based on a condition
- `match-case` provides a structured way to handle multiple cases

Exercise 5 Predict the Output

Working Space

Fill in the table with the outputs for the given inputs.

```
x = int(input("Enter a number: "))

if x > 10:
    print("Large")
elif x == 10:
    print("Ten")
else:
    print("Small")
```

Input	Output
5	
10	
15	

Answer on Page 26

Exercise 6 Fill in the Blanks: Loop Edition

Working Space

Fill in the following code to print all even numbers from 0 to 20 (inclusive).

```
for i in __________:
    print(__________)
```

Answer on Page 26

1.7 Lists and Strings

We talked about creating strings, any set of characters between quotes. These can either be single quotes `' '` or double quotes `"    "`.

Multiline strings must be surrounded by three quotations on each side of the text sequence.

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
```

Strings are an example of a *sequence type*, meaning they store values in a specific order and allow individual elements to be accessed using an index.

1.7.1 Indexing Strings

Each character in a string has a position, called an *index*. Indexing in Python begins at 0, so the first character is at index 0. Think of it like asking the question for each letter: "How far from the start is this element?" The first element is zero positions away from the start, so its index is 0. The last element is always at index $n-1$, where n is the length of string. In memory, all of these characters are stored consecutively.

```
word = "Python"

print(word[0])
print(word[1])
print(word[5])
```

Output:

```
P
y
n
```

Attempting to access an index that does not exist will result in a runtime error.

1.7.2 Length of a String

The number of characters in a string can be found using the built-in `len()` function.

```
word = "Python"
print(len(word))
```

Output:

```
6
```

1.7.3 Iterating Over Strings

Strings can be iterated over one character at a time using a `for` loop.

```
for char in "Hello":
    print(char)
```

This loop runs once for each character in the string.

1.7.4 String Methods

Strings come with built-in functions, called *methods*, that perform common operations. These methods are called using dot notation.

```
text = "  Python Programming  "

print(text.upper())
print(text.lower())
print(text.strip())
```

Common string methods include:

- `upper()` converts all characters to uppercase
- `lower()` converts all characters to lowercase
- `strip()` removes leading and trailing whitespace

1.7.5 Lists

A list is another sequence type used to store multiple values in a single variable. Lists are created using square brackets, with elements separated by commas.

```
numbers = [1, 2, 3, 4]
```

Lists can store values of different datatypes, including strings, numbers, booleans, and even other lists.

```
data = [10, 3.14, True, "Python"]
```

1.7.6 Indexing Lists

Like strings, lists are indexed starting at 0.

```
numbers = [10, 20, 30]

print(numbers[0])
print(numbers[2])
```

Output:

```
10
30
```

For nested lists, you search an indexe using a bracket for every nested list until you reach the element you are aiming to reach. Later, we will talk about `.json` files, which are files essentially comprised of strung-nested lists.

```
nested_numbers = [0, [1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(numbers[0])
print(numbers[1][2])
print(numbers[2][1])
```

Output:

```
0
3
5
```

Slicing Strings

You can also use *slicing* in Python to get all elements between two indices. This syntax is Python-specific, although there are equivalences in other programming languages. This syntax works for *both* lists and strings.

For any sequence, we can slice it or operate on it with the following index notation:

```
sequence[start : stop : step]
```

where `start` is the index to begin at (inclusive), `end` is the index to end at (exclusive), and `step`, an optional argument for how many items to skip.

```
text = "abcdef"

text[1:4]     # 'bcd'
text[:3]      # 'abc'
text[3:]      # 'def'
text[::2]     # 'ace'
text[::-1]    # 'fedcba'  (reverse string)
```

The above example slices a string in a few different ways. Let's look at list slicing:

```
nums = [0, 1, 2, 3, 4, 5]

nums[2:5]     # [2, 3, 4]
nums[:4]      # [0, 1, 2, 3]
nums[4:]      # [4, 5]
nums[::2]     # [0, 2, 4]
nums[::-1]    # [5, 4, 3, 2, 1, 0]
```

Negative indices start from the end, and work the way up the sequence:

```
text = "python"

text[-3:]     # 'hon'
text[:-2]     # 'pyth'
```

Note that setting a variable to a sliced string creates a new string object, while leaving the original unchanged.

Exercise 7 Mystery Sequence

What will the code `seq[:]` output, assuming `seq = [a, f, j, k]` is a valid list?

Working Space

Answer on Page 26

Exercise 8 Computer... Indexing!

You are given the following code:

```
word = "computer"
```

What is the output of each of the following expressions?

```
print(word[0])
print(word[2])
print(word[-1])
print(word[-3])
print(word[3:6])
```

Working Space

Answer on Page 26

1.7.7 Modifying Lists

Unlike strings, lists are *mutable*, meaning their contents can be changed after creation.

```
numbers = [1, 2, 3]
numbers[1] = 10
print(numbers)
```

Output:

```
[1, 10, 3]
```

1.7.8 List Length and Iteration

The number of elements in a list can be found using the `len()` function.

```
numbers = [1, 2, 3]
print(len(numbers))
```

Lists can be iterated over using a `for` loop.

```
numbers = [1, 2, 3]

for n in numbers:
    print(n)
```

1.7.9 Common List Methods

Lists include built-in methods that allow elements to be added or removed.

```
numbers = [1, 2, 3]
numbers.append(4)
numbers.remove(2)
print(numbers)
```

Common list methods include:

- `append()` - adds an element to the end of the list
- `remove()` - removes the first occurrence of a value
- `pop()` – removes and returns an element by index

1.7.10 Strings vs Lists

Although strings and lists are both sequence types, there is an important difference between them.

- Strings are immutable and cannot be modified after creation
- Lists are mutable and can be changed

1.7.11 Summary

- Strings and lists are sequence types
- Indexing starts at 0
- The `len()` function returns the size of a sequence
- Strings are immutable
- Lists are mutable and support modification

Exercise 9 What's in your backpack?

Working Space

Let's say the following code is run.

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

items.append("ruler")
items.pop()
items.remove("book")
items.append("marker")
```

Fill in the following chart with the contents of the `items` list after each operation.

Line Execution	`items` content
After Line 1	`items = ["pen", "book", "eraser"]`
After Line 3	
After Line 4	
After Line 5	
After Line 6	

Answer on Page 27

1.8 Is there more?

Of course, there is always something to learn with Python. We are going to do a deeper dive into functions, classes, and a few useful libraries in the next workbook. For now, this should be enough to get you started with Python. We will revisit many of these concepts in later chapters as we build more complex programs.

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 6)

Variable	Value	Data Type
a	10	`int`
b	3.5	`float`
c	”10”	`str`
d	True	`bool`
e	[1, 2, 3]	`list`

Answer to Exercise 2 (on page 7)

```
False
```

Answer to Exercise 3 (on page 9)

Line Executed	n (value, type)	m (value, type)
After line 2	(5, int)	(”3”, str)
After line 3	(6, int)	(”3”, str)
After line 4	(6, int)	(”31”, str)

Answer to Exercise 4 (on page 11)

The code will raise a `TypeError` because `x` is a string and cannot be added to the integer `5`. The error message will be similar to:

```
TypeError: can only concatenate str (not "int") to str
```

To fix this, we need to cast `x` to an integer using `int()`:

```
x = int(input("Enter a number: "))
print(x + 5)
```

Answer to Exercise 5 (on page 16)

Input	Output
5	Small
10	Ten
15	Large

Answer to Exercise 6 (on page 16)

```
for i in range(0, 21, 2):
    print(i)
```

Answer to Exercise 7 (on page 21)

The start index is omitted and the end index is omitted, which means they are 0 and `len(seq)`$= 4$ respectively. Since the bounds are 0 and $4 - 1 = 3$, every index is included, in the given order. Therefore, `seq[:]` returns a copy of the given string, `[a, f, j, k]`

Answer to Exercise 8 (on page 21)

```
c
m
r
t
put
```

Answer to Exercise 9 (on page 23)

Line Execution	`items` content
After Line 1	`items = ["pen", "book", "eraser"]`
After Line 3	`items = ["pen", "book", "eraser", "ruler"]`
After Line 4	`items = ["pen", "book", "eraser"]`
After Line 5	`items = ["pen", "eraser"]`
After Line 6	`items = ["pen", "eraser", "marker"]`

Index