Introduction to programming for data science

STAT 201 Winter 2025

2025-01-07

Table of contents

Preface

This book serves as the course notes for STAT201 Winter 2025, and it is an evolving resource developed to support the learning objectives of the course. It builds upon the foundational work of the original iteration, authored and maintained by Professor Arvind Krishna. We are deeply grateful for Professor Krishna's contributions, as his work has provided a robust framework and valuable content upon which this version of the book is based.

As the course progresses during this quarter, the notes will be continually updated and refined to reflect the content taught in real time. The modifications aim to enhance the clarity, depth, and relevance of the material to better align with the current teaching objectives and methodologies.

This book is a living document, and we welcome feedback, suggestions, and contributions from students, instructors, and the broader academic community to help improve its quality and utility.

Thank you for being part of this journey, and we hope this resource serves as a helpful guide throughout the course.

Part I

Getting started: Coding environment

1 Setting up your environment with VS Code

```
<IPython.core.display.Image object>
```

1.1 Learning Objectives

By completing this lecture, you will be able to:

- Install and configure Visual Studio Code (VS Code) for Python programming.
- Leverage Jupyter Notebook within VS Code for your data science Python programming.
- Use Quarto to create HTML documents for your upcoming homework submissions.

1.2 Introduction to Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a free, open-source, and lightweight code editor developed by Microsoft. It's widely used for coding, debugging, and working with **various programming languages** and frameworks. Here's an overview of its key features and functionalities:

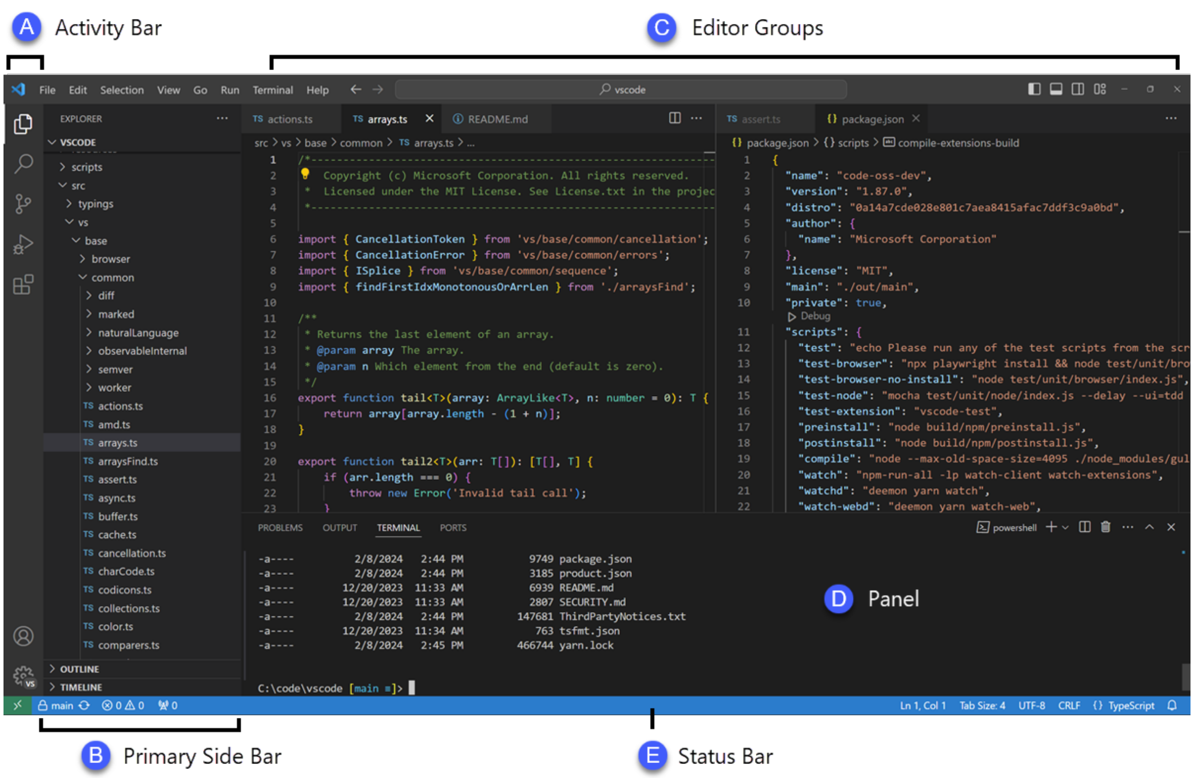
1.2.1 Core Features

- **Multi-language Support:** VS Code supports a wide range of programming languages out of the box, including Python, JavaScript, TypeScript, HTML, CSS, and more. Additional language support can be added via extensions.
- **Extensibility:** The editor has a rich ecosystem of extensions available through the Visual Studio Code Marketplace. These extensions add support for additional programming languages, themes, debuggers, and tools like Git integration.
- **IntelliSense:** Provides intelligent code completion, parameter info, quick info, and code navigation for many languages, enhancing productivity and reducing errors.
- **Integrated Terminal:** Allows you to run command-line tools directly from the editor, making it easy to execute scripts, install packages, and more without leaving the coding environment.

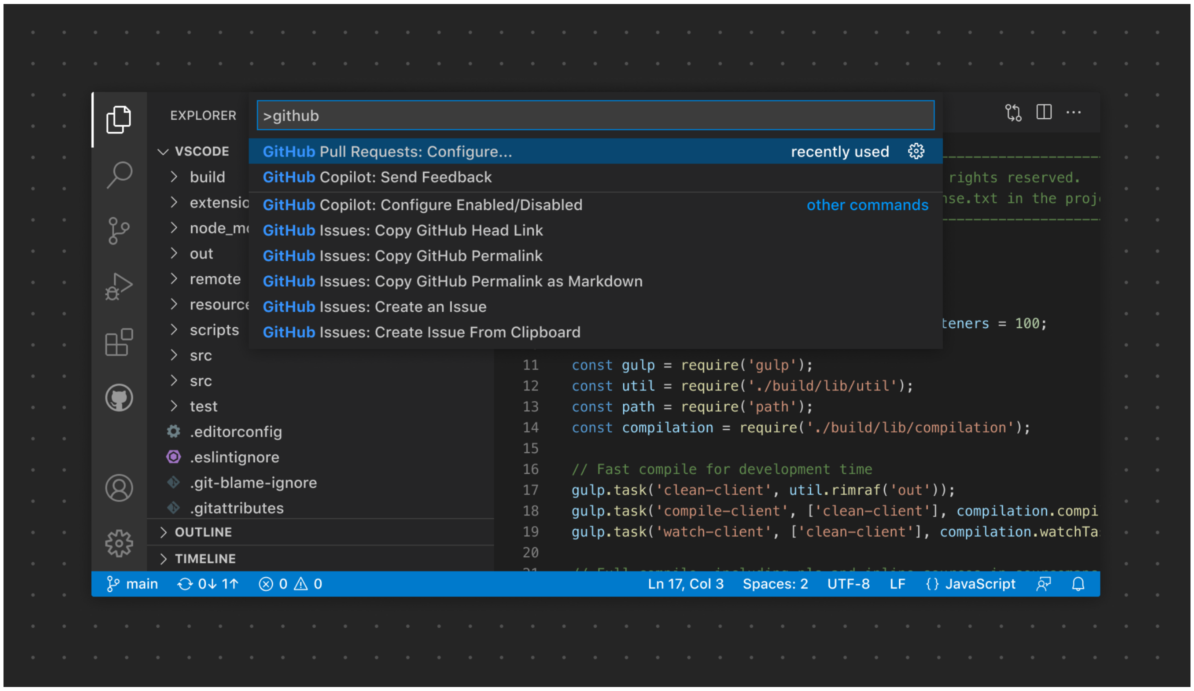
- **Version Control Integration:** Seamless integration with Git and other version control systems, allowing you to manage source code repositories, stage changes, commit, and view diffs within the editor.
- **Debugging:** Supports debugging with breakpoints, call stacks, and an interactive console for various languages and frameworks.

1.2.2 User Interface

- **Editor:** The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- **Primary Side Bar:** Contains different views like the Explorer to assist you while working on your project.
- **Activity Bar:** Located on the far left-hand side. Lets you switch between views and gives you additional context-specific indicators, like the number of outgoing changes when Git is enabled. You can change the position of the Activity Bar.
- **Panel:** An additional space for views below the editor region. By default, it contains output, debug information, errors and warnings, and an integrated terminal. The Panel can also be moved to the left or right for more vertical space.

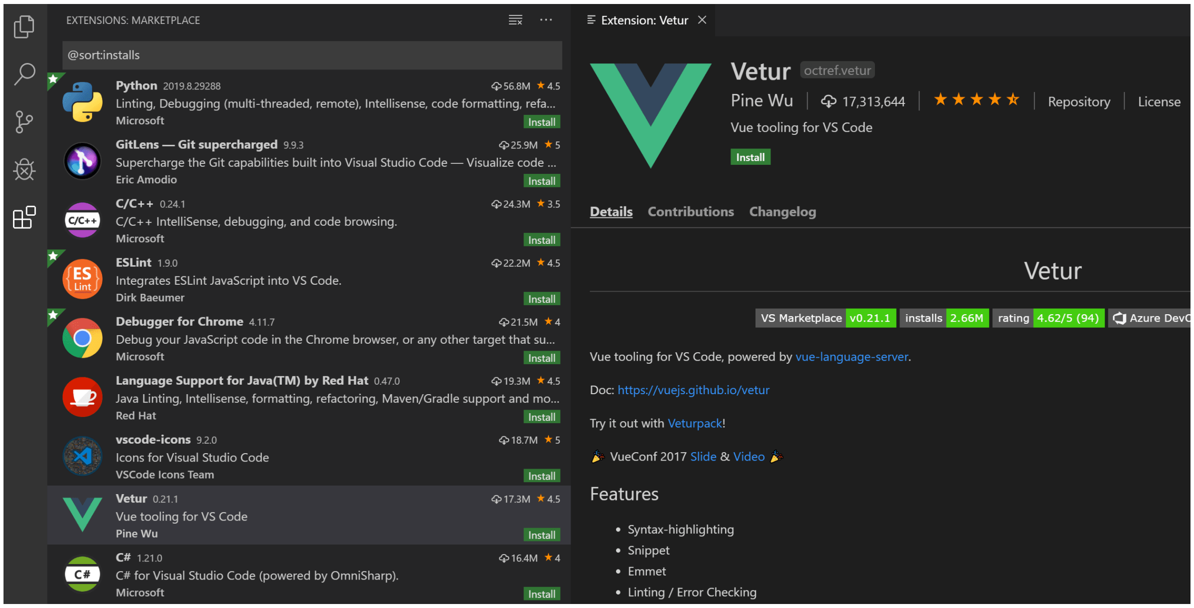


- **Command Palette:** Accessed with `Ctrl+Shift+P` (or `Cmd+Shift+P` on macOS), it provides a quick way to execute commands, switch themes, change settings, and more.



1.2.3 Extensions

- **Language Extensions:** Add support for additional languages such as Rust, Go, C++, and more.
- **Linters and Formatters:** Extensions like ESLint, Prettier, and Pylint help with code quality and formatting.
- **Development Tools:** Extensions for Docker, Kubernetes, database management, and more.
- **Productivity Tools:** Extensions for snippets, file explorers, and workflow enhancements.



1.2.4 Use Cases

- **Web Development:** VS Code is popular among web developers for its robust support for HTML, CSS, JavaScript, and front-end frameworks like React, Angular, and Vue.
- **Python Development:** With the Python extension, it provides features like IntelliSense, debugging, linting, and Jupyter Notebook support.
- **Data Science:** Supports Jupyter notebooks, allowing data scientists to write and run Python code interactively.
- **DevOps and Scripting:** Useful for writing and debugging scripts in languages like PowerShell, Bash, and YAML for CI/CD pipelines.

1.2.5 Cross-Platform

- Available on Windows, macOS, and Linux, making it accessible to developers across different operating systems.

Overall, VS Code is a versatile and powerful tool for a wide range of development activities, from simple scripting to complex software projects.

1.3 Installing Visual Studio Code

- Step 1: **Download VS Code:**

- Go to the official VS Code website and download the installer for your operating system.

- Step 2: **Install VS Code:**
  - Run the installer and follow the prompts to complete the installation.
- Step 3: **Launch VS Code:**
  - Open VS Code after installation to ensure it's working correctly.

1.4 Setting Up Python Development Environment in VS Code using python `venv`

Unlike `Spyder` and `PyCharm`, which are specifically designed for Python development, VS Code is a versatile code editor with multi-language support. As a result, setting up the Python environment requires some additional configuration.

This step-by-step guide will walk you through setting up your Python environment in Visual Studio Code from scratch using `venv`.

1.4.1 Install Python

For this course, any version of Python 3 works. You don't need to worry about having the latest version of Python, as long as you have Python 3 installed.

If Python 3 is already installed on your computer, you can skip this step.

1. **Download Python:**
   - Go to the official Python website and download the latest version of Python for your operating system.
   - Ensure that you check the box **"Add Python to PATH"** during installation if it exists.
2. **Verify Python Installation:**
   - Open a terminal (Command Prompt on Windows, Terminal on macOS/Linux) and type:

     ```
     python --version
     ```

   - You should see the installed Python version. If the command line doesn't work, you might see an error message like:
     - `python is not recognized`

– `python command is not found`

This issue is often caused by Python not being added to the **PATH** environment variable. Please refer to the instructions to resolve this issue.

Note:
Before moving forward, ensure that the command `python --version` successfully prints the version of your installed Python. If it does not, you may need to troubleshoot your Python installation or add it to the PATH environment variable.

1.4.2 Install Visual Studio Code Extensions

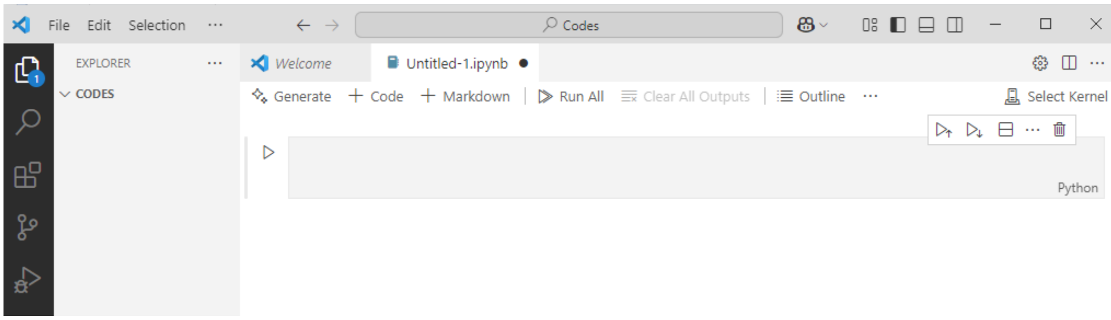
1. **Open VS Code.**
2. **Go to Extensions:**
   - Click on the Extensions icon on the sidebar or press `Ctrl+Shift+X`.
3. **Install Python Extension:**
   - Search for the "Python" extension by Microsoft and install it.
4. **Install Jupyter Extension:**
   - Search for the "Jupyter" extension by Microsoft and install it.

1.4.3 Set Up a Python Workspace for this course

1. **Create a New Folder:**
   - Create a new folder on your computer where you want to store your Python code for this course.
2. **Open Folder in VS Code:**
   - Go to `File > Open Folder` and select the newly created folder.

1.4.4 Create a Notebook for your work

- In VS Code, go to `File > New File` and select `Jupyter Notebook`. You should see a blank notebook named `Untitled-1.ipynb` as in the figure below. The `.ipynb` extension stands for IPython Notebook.

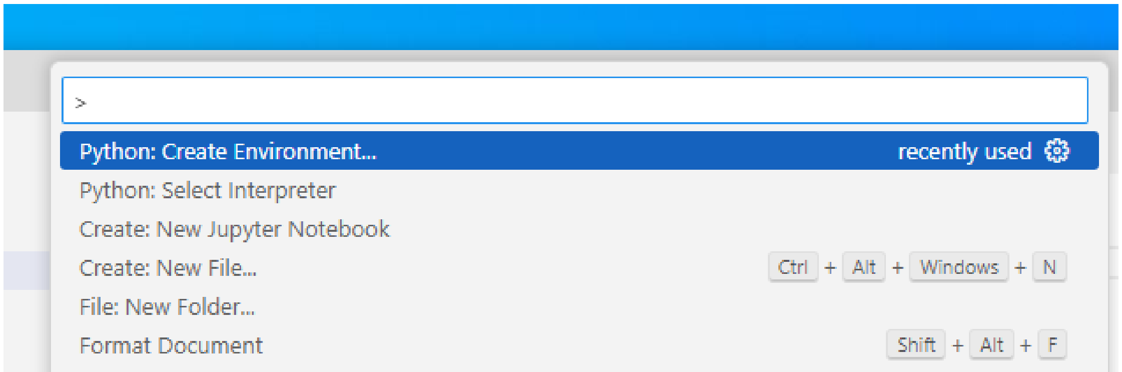
- 

:

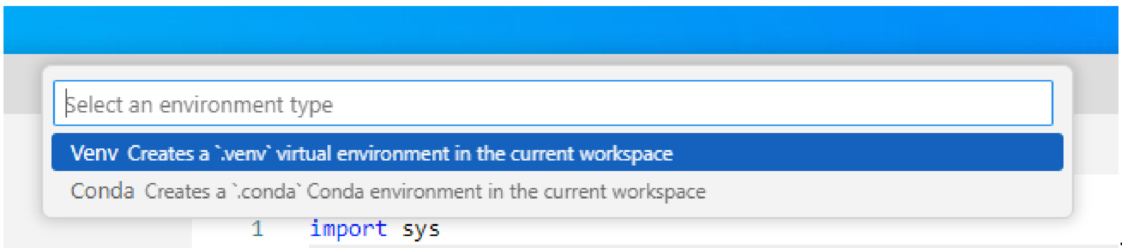
Jupyter Notebook is an **interactive platform** that allows you to write code, add text, and create visualizations. Data scientists love using Jupyter Notebooks as an alternative to working directly with Python files because of their interactivity and flexibility.

1.4.5 Create a Python environment for your work - GUI method

- When you start a Jupyter Notebook in VS Code, you need to choose a kernel. Kernel is the "engine" that runs your code within your Jupyter notebook, and it is tied to a specific Python interpreter or environment.
  - **What's the difference between an interpreter and an environment?** An interpreter is a program that runs your Python code. An environment, on the other hand, is a standalone "space" where your code runs. It's like a container that holds its own interpreter and environment-specific libraries/dependencies, so each project can have its own environment setup without affecting others.
  - **Why do we prefer creating an environment for this course rather than using the global interpreter that comes with your Python installation?** As a data scientist, you may work on multiple projects and attend different courses that require different sets of packages, dependencies, or even Python versions. By creating a separate environment, you can prevent conflicts between libraries, dependencies, and Python versions across your projects (dependency hell) and also ensure code reproducibility. It is always good practice to work within python environments, especially when you have different projects going on.
  - Let's create a Python environment for the upcoming coursework.



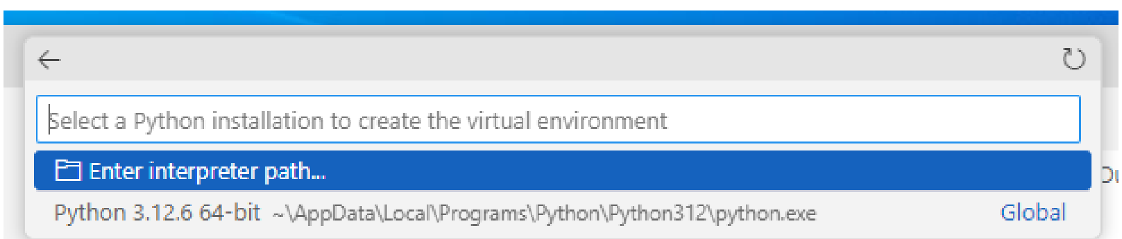
- Create using `venv` in the current workspace



What is `venv`

In Python, `venv` stands for Virtual Environment, which is a tool used to create isolated environments for Python projects. This helps manage dependencies and avoid conflicts between different projects that may require different versions of Python or different packages.

- Choose a specific python interpreter for your environment:



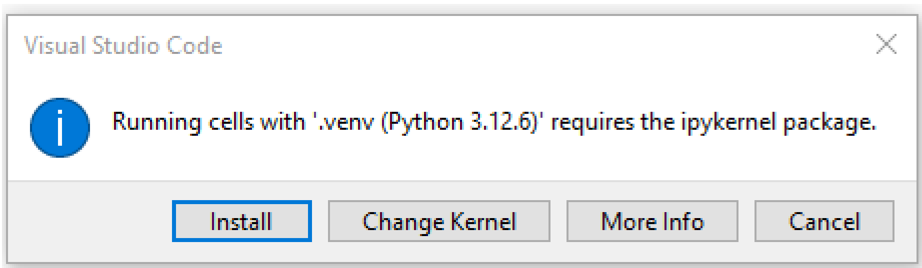
Congratulations! A virtual environment named `.venv` has been successfully created in your project folder.

1.4.6 Choose the `.venv` environment as the kernel to run the notebook

For all your upcoming work in this project, you can select this environment to ensure a consistent setup.

1.4.7 Installing `ipykernel` for your notebook

Create a code cell in the notebook and run it. The first time you run a code cell, you will run into



- After installing `ipykernel`, you should be able to run the following cell.

```python
import sys
print("Current Python executable:", sys.executable)
```

```
Current Python executable: c:\Users\lsi8012\OneDrive - Northwestern University\FA24\303-1\te
```

`sys.executable` is an attribute in the Python `sys` module that returns the path to the Python interpreter that is currently executing your code.

1.5 Jupyter Notebooks in VS Code

After setting up your environment and successfully running your notebook using the created environment, follow this instruction to become familiar with the native support for Jupyter Notebooks in VS Code

1.5.1 Writing and executing code

Code cell: By default, a cell is of type *Code*, i.e., for typing code, as seen as the default choice in the dropdown menu below the *Widgets* tab. Try typing a line of python code (say, `2+3`) in an empty code cell and execute it by pressing *Shift+Enter*. This should execute the code, and create an new code cell. Pressing *Ctlr+Enter for Windows (or Cmd+Enter for Mac)* will execute the code without creating a new cell.

Commenting code in a code cell: Comments should be made while writing the code to explain the purpose of the code or a brief explanation of the tasks being performed by the code. A comment can be added in a code cell by preceding it with a # sign. For example, see the comment in the code below.

Writing comments will help other users understand your code. It is also useful for the coder to keep track of the tasks being performed by their code.

```
#This code adds 3 and 5
3+5
```

```
8
```

Please refer to the Style Guide for Python Code to develop good coding habits from the start.

Markdown cell: Although a comment can be written in a code cell, a code cell cannot be used for writing headings/sub-headings, and is not appropriate for writing lengthy chunks of text. In such cases, change the cell type to *Markdown* from the dropdown menu below the *Widgets* tab. Use any markdown cheat sheet found online, for example, this one to format text in the markdown cells.

Jupyter Notebook shortcuts: Jupyter Notebook shortcuts are handy and quick. Here is a list of commonly used shortcuts for beginners:

- **Shift + Enter**: Run the cell and move to the next one.
- **A**: Insert a cell above.
- **B**: Insert a cell below.
- **D, D**: Delete the selected cell.
- **Y**: Change the cell to Code type.
- **M**: Change the cell to Markdown type.

1.5.2 Saving and loading notebooks

To save the notebook in VS Code, click on `File` and select `Save As`, or use the keyboard shortcut `CTRL+S`. Your notebook will be saved as a file with the extension `.ipynb`. This file will contain all the code and outputs and can be opened and edited with VS Code or any Jupyter-compatible environment.

To open an existing Jupyter notebook with VS Code: 1. Navigate to the file in your system's file explorer. 2. Right-click on the file. 3. Select `Open with VS Code`.

1.6 Rendering notebook as HTML using Quarto

Quarto is designed for high-quality, customizable, and publishable outputs, making it suitable for reports, blogs, or presentations. We'll use Quarto to print the **.ipynb* file as HTML for your assignment submission.

1.6.1 Installing and Verifying Quarto

- Download and install Quarto from the official website.
- Follow the installation instructions for your operating system.
- Open your terminal within VS Code:
  - Go to `Terminal` -> `New Terminal`.
- Run the following command to verify that Quarto and its dependencies are correctly installed: `quarto --version`
- You should see the installed quarto version. If the command line doesn't work, you might see an error message like:

  ```
  - `quarto is not recognized`
  - `quarto command is not found`
  ```

  This issue is often caused by Quarto not being added to the PATH environment variable. Similar to how you added the Python path to the environment variable above, you need to add the Quarto path to the system environment variable so that the command can be recognized by your operating system's shell.

  On Windows, if you used the default installation path (without changing it), Quarto is installed in: `C:\Users\<USER>\AppData\Local\Programs\Quarto\bin`

Note:
Before moving forward, ensure that the command `quarto --version` successfully prints the version of your installed quarto. If it does not, you may need to troubleshoot your quarto installation or add it to the PATH environment variable.

1.6.2 Converting the Notebook to HTML

Check the procedure for rendering a notebook as HTML here. You have several options to format the file. Here are some points to remember when using Quarto to render your notebook as HTML:

1. The `Raw NBConvert` cell type is used to render different code formats into HTML or LaTeX. This information is stored in the notebook metadata and converted appropriately. **Use this cell type to put the desired formatting settings for the HTML file.**
2. In the formatting settings, remember to use the setting `embed-resources: true`. This will ensure that the rendered HTML file is self-contained, and is not dependent on other files. This is especially important when you are sending the HTML file to someone, or uploading it somewhere. If the file is self-contained, then you can send the file by itself without having to attach the dependent files with it.

Once you have entered the desired formatting setting in the `Raw NBConver` cell, you are ready to render the notebook to HTML. Open the terminal, navigate to the directory containing the notebook *(.ipynb file)*, and use the command: `quarto render filename.ipynb --to html`.

1.7 In-class exercise

1. Create a new notebook.
2. Save the file as `In_class_exercise_1`.
3. Give a heading to the file - `First HTML file`.
4. Print `Today is day 1 of my programming course`.
5. Compute and print the number of seconds in a day.
6. Generate html from the notebook using Quarto

The HTML file should look like the picture below.

```
<IPython.core.display.Image object>
```

1.8 Reference

- Getting Started with VS Code
- Jupyter Notebooks in VS Code
- Quarto

Part II

Python

2 Variables, expressions and statements

```
<IPython.core.display.Image object>
```

2.1 Commenting code

The `#` symbol can be used to comment the code. Anything after the `#` sign is ignored by python. Commenting a code may have several purposes, such as:

- Describe what is going to happen in a sequence of code
- Document who wrote the code or other ancillary information
- Turn off a line of code - perhaps temporarily

For example, below is code with a comment to describe the purpose of the code:

```
#Computing number of hours of lecture in this course
print("Total lecture hours of STAT201=",10*3*(5/6))
```

```
Total lecture hours of STAT201= 25.0
```

2.1.1 Practice exercise 1

Which of the following lines is a comment:

1. #this is a comment
2. ##this may be a comment
3. A comment#

2.2 `print()` function in python

The `print()` function is a fundamental tool for displaying information.

2.2.1 Basic Examples

```python
# Printing a simple string
print("Hello, World!")
```

```
Hello, World!
```

```python
# Printing a string with a number
print ("The total number of seconds in a day is", 24*60*60)
```

```
The total number of seconds in a day is 86400
```

```python
# combine multiple strings using the + operator
print("Hello, " + "World!")
```

```
Hello, World!
```

2.2.2 concatenating strings and variables/expressions in `print()` function

2.2.2.1 Using f-Strings (Formatted String Literals)

f-strings provide a concise way to embed expressions inside strings. Introduced in Python 3.6, they improve readability and efficiency.

2.2.2.2 Syntax

- Use `f` or `F` before the string.
- Embed variables or expressions in `{}`.

```python
# Example 1: Basic Variable Substitution
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")


# Example 2: Arithmetic Expressions
a = 10
b = 5
```

```
print(f"The sum of {a} and {b} is {a + b}.")


# Example 3: Formatting Numbers
pi = 3.14159
print(f"Pi rounded to 2 decimal places is {pi:.2f}.")
print(f"Pi rounded to 2 decimal places is {pi:.0f}.")
```

```
My name is Alice and I am 30 years old.
The sum of 10 and 5 is 15.
Pi rounded to 2 decimal places is 3.14.
Pi rounded to 2 decimal places is 3.
```

```
value = 123456789
print(f"Rounded With commas: ${value:,}")
```

```
Rounded With commas: $123,456,789
```

2.2.2.3 Using `str.format()` Method

The `str.format()` method allows you to format strings by placing placeholders `{}` in the string and replacing them with variables or values.

`"Your text here {}".format(variable_or_expression)`

```
# Example 1: Basic Variable Substitution
name = "Bob"
age = 25
print("My name is {} and I am {} years old.".format(name, age))


# Example 2: Using Positional Arguments
print("The sum of {0} and {1} is {2}.".format(a, b, a + b))


# Example 3: Using Keyword Arguments
print("Pi rounded to 2 decimal places is {value:.2f}.".format(value=pi))
print("Pi rounded to 2 decimal places is {value:.0f}.".format(value=pi))
```

```
My name is Bob and I am 25 years old.
The sum of 10 and 5 is 15.
Pi rounded to 2 decimal places is 3.14.
Pi rounded to 2 decimal places is 3.
```

2.2.3 Customizing output formatting

The `print()` function in Python is highly customizable. By default, it adds a newline character (`\n`) at the end of each output and separates multiple arguments with a space (). However, these default behaviors can be changed using the `end` and `sep` parameters.

2.2.3.1 Default Behavior of `print()`

When you call `print()` multiple times, each statement starts on a new line:

```python
print("Hello")
print("World")  # this is printed on a new line
```

```
Hello
World
```

2.2.3.2 Changing the `end` Parameter

To avoid automatic line breaks and control what is appended at the end of the output, use the `end` parameter.

```python
print("Hello", end=" ")
print("World") # this is printed on the same line
```

```
Hello World
```

```python
print("Loading", end="...")
print("Complete")
```

```
Loading...Complete
```

```python
print("Line 1", end="")
print("Line 2")
```

```
Line 1Line 2
```

2.2.3.3 Changing the `sep` Parameter

When printing multiple arguments, the default separator between them is a space. You can change this behavior using the `sep` parameter.

```
print("apple", "banana", "cherry", sep=", ")
# Output: apple, banana, cherry
```

```
apple, banana, cherry
```

```
print("A", "B", "C", sep="")
```

```
ABC
```

```
print("python", "java", "c++", sep="\n")
```

```
python
java
c++
```

2.2.3.4 Combining `end` and `sep`

Both `end` and `sep` can be used together for more control:

```
print("1", "2", "3", sep="-", end=".")
print(" Done!")
```

```
1-2-3. Done!
```

2.2.3.5 Takeaway:

- The `end` parameter changes what is appended at the end of the output (default: `\n`).
- The `sep` parameter modifies how multiple arguments are separated (default: space).
- Both can be combined to create customized output formatting.

2.2.4 Practice exercise 2

Use the `print()` function to:

- Display your name, age, and favorite hobby.
- Format the output neatly using f-strings.
- Use `sep` and `end` to produce this output: `apple:banana:cherry`.

2.3 Data types

Python provides several built-in data types for storing different kinds of information in variables. These data types can be broadly categorized into primitive data types and collection (containers) data types as shown at the beginning of this chapter. While collection data types will be covered in Chapter 5, this chapter focuses on primitive data types, which are used to represent a single value.

2.3.1 Primitive Data Types

They represent a single value. In Python, primitive data types include:

- **Integer (`int`)**: Whole numbers (e.g., `10`, `-3`).
- **Floating-point number (`float`)**: Numbers with decimals (e.g., `3.14`, `-2.7`).
- **Boolean (`bool`)**: Logical values `True` or `False`.
- **None type (`None`)**: Represents the absence of a value.
- **String (`str`)**: A sequence of characters (e.g., `"hello"`, `'world'`).

The data type of the object can be identified using the in-built python function `type()`. For example, see the following objects and their types:

```
type(4)
```

```
int
```

```
type(4.4)
```

```
float
```

```
type('4')
```

```
str
```

```python
type(True)
```

```
bool
```

2.3.2 Practice exercise 3

What is the datatype of the following objects?

1. 'This is False'
2. "This is a number"
3. 1000
4. 65.65
5. False

2.3.3 Commonly Used Built-in methods associated with each data type

2.3.3.1 Strings

Strings are sequences of characters and are immutable in Python.

Below are Commonly used Methods for strings:

- `lower()`: returns a string with every letter of the original in lowercase
- `upper()`: returns a string with every letter of the original in uppercase
- `replace(x,y)`: returns a string with every occurrence of x replaced by y
- `count(x)`: counts the number of occurrences of x in the string
- `index(x)`: returns the location of the first occurrence of x
- `format()`: format strings
- `isalpha()`: returns True if every character of the string is a letter

For a more comprehensive list of string methods, please refer to here

```python
# Example Methods:
s = "Hello, World!"

# Returns the length of the string
len(s)
print("the length of the string is", len(s))

# Converts string to uppercase
s.upper()
print("the string in uppercase is", s.upper())

# Converts string to lowercase
s.lower()
print("the string in lowercase is", s.lower())

# Capitalizes the first character of the string
s.capitalize()
print("the string with first letter capitalized is", s.capitalize())

# Finds the first occurrence of a substring
s.find("World")
print("the first occurrence of the substring is at", s.find("World"))

# Replaces a substring with another string
s.replace("World", "Python")
print("the string after replacement is", s.replace("World", "Python"))

# Splits the string into a list
s.split(", ")
print("the string after splitting is", s.split(", "))

# Strips leading/trailing whitespace
s.strip()
print("the string after stripping is", s.strip())

# count the number of occurrences of a substring
s.count("l")
print("the number of occurrences of the substring is", s.count("l"))

# Checks if the string is alphanumeric
s.isalnum()
print("is the string alphanumeric?", s.isalnum())
```

```
the length of the string is 13
the string in uppercase is HELLO, WORLD!
the string in lowercase is hello, world!
the string with first letter capitalized is Hello, world!
the first occurrence of the substring is at 7
the string after replacement is Hello, Python!
the string after splitting is ['Hello', 'World!']
the string after stripping is Hello, World!
the number of occurrences of the substring is 3
is the string alphanumeric? False
```

2.3.3.2 Single quotes ' and double quotes " to define strings

in Python, you can use either single quotes (') or double quotes (") to define strings. Both are functionally equivalent, and you can choose based on preference or readability. Here's an example:

```python
# Using single quotes
string1 = 'Hello, world!'
print(string1)

# Using double quotes
string2 = "Hello, world!"
print(string2)
```

```
Hello, world!
Hello, world!
```

When to use one over the other

- Single quotes (') are often preferred for simple strings without embedded quotes.
- Double quotes (") are useful when your string contains a single quote, as it avoids the need for escaping:

```python
# Single quote in a double-quoted string
message = "It's a beautiful day!"
print(message)

# Double quote in a single-quoted string
message = 'He said, "Hello!"'
print(message)
```

```
It's a beautiful day!
He said, "Hello!"
```

Escaping quotes: If your string contains both single and double quotes, you can use the backslash (\) to escape them:

```
# Escaping single quotes in a single-quoted string
string_with_escape1 = 'It\'s a sunny day.'
print(string_with_escape1)

# Escaping double quotes in a double-quoted string
string_with_escape2 = "He said, \"Hello!\""
print(string_with_escape2)
```

```
It's a sunny day.
He said, "Hello!"
```

You can also use triple quotes (`'''` or `"""`) for strings that span multiple lines or contain both types of quotes without escaping:

```
multi_line_string = """This string spans
multiple lines and can include 'single quotes' and "double quotes"."""

print(multi_line_string)
```

```
This string spans
multiple lines and can include 'single quotes' and "double quotes".
```

Define a *f*-string

```
language = "Python"
level = "beginner"
greeting1 = "I'm learning {} at a {} level.".format(language, level)
print(greeting1 )

greeting2 = f"I'm learning {language} at a {level} level."
print(greeting2)
```

```
I'm learning Python at a beginner level.
I'm learning Python at a beginner level.
```

String Concatenation: Using the `+` Operator * Use the `+` operator to join strings together. * All operands must be strings; otherwise, you'll get a `TypeError`. * * Use `str()` to convert non-strings to strings when necessary.

```
# Basic Concatenation
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!"
print(message)

# Concatenating String Literals
print("Python" + " is " + "fun!")

# Concatenating with Variables
lang = "Python"
level = "beginner"
print("I'm learning " + lang + " as a " + level + " programmer.")
```

```
Hello, Alice!
Python is fun!
I'm learning Python as a beginner programmer.
```

String Repetition: String repetition is achieved using the `*` operator. It creates a new string by repeating an existing string a specified number of times.

```
# Creating Patterns
print("*" * 10)
```

```
**********
```

```
repeat_count = 4
print("Python! " * repeat_count)
```

```
Python! Python! Python! Python!
```

2.3.3.3 Integers

Integers are whole numbers, either positive or negative.

Commonly used Methods:

```python
# Example:
n = -42

# Returns the absolute value
abs(n)
print("the absolute value of the number is", abs(n))

# Converts to binary string
bin(n)
print("the binary string of the number is", bin(n))

# Converts to hexadecimal string
hex(n)
print("the hexadecimal string of the number is", hex(n))

# Converts to octal string
oct(n)
print("the octal string of the number is", oct(n))

# Returns the power of a number
pow(n, 2)  # n^2
print("the power of the number is", pow(n, 2))

# Checks if a number is an integer
isinstance(n, int)
print("is the number an integer?", isinstance(n, int))
```

```
the absolute value of the number is 42
the binary string of the number is -0b101010
the hexadecimal string of the number is -0x2a
the octal string of the number is -0o52
the power of the number is 1764
is the number an integer? True
```

2.3.3.4 Floats

Floats represent real numbers and are used for decimal or fractional values.

Commonly Used Methods:

```python
# Example:
f = 3.14159

# Returns the absolute value
abs(f)
print("the absolute value of the number is", abs(f))

# Rounds to the nearest integer
round(f)
print("the number rounded to the nearest integer is", round(f))

# Converts to integer by truncating
int(f)
print("the number converted to integer is", int(f))

# Checks if a number is a float
isinstance(f, float)
print("is the number a float?", isinstance(f, float))
```

```
the absolute value of the number is 3.14159
the number rounded to the nearest integer is 3
the number converted to integer is 3
is the number a float? True
```

2.3.3.5 Booleans

Booleans represent logical values `True` or `False`.

Commonly Used Methods:

```python
# Example:
b = True

# Converts to integer (True -> 1, False -> 0)
int(b)
print("the integer value of the boolean is", int(b))

# Converts to string
str(b)
print("the string value of the boolean is", str(b))
```

```python
# Logical operations:
not b  # Negates the boolean
print("the negation of the boolean is", not b)
```

```
the integer value of the boolean is 1
the string value of the boolean is True
the negation of the boolean is False
```

2.4 Variables

A **variable** is a container for storing data values. Variables in Python are **dynamically typed**, meaning you don't need to specify their type when declaring them.

2.4.1 Variable Declaration:

- You can create a variable by assigning a value to it using the `=` operator.

For example:

```python
x = 10 # Integer
name = "Alice" # String
pi = 3.14 # Float
is_active = True # Boolean
```

2.4.2 Dynamic Typing:

- The type of a variable is determined by the value assigned to it.

For example:

```python
x = 10 # x is an integer
y = x # y is also an integer
```

2.4.3 Variable Naming Rules:

- Names must start with a letter (a-z, A-Z) or an underscore (_).
- Names can only contain letters, numbers (0-9), and underscores.
- Names are case-sensitive (`name` and `Name` are different variables).
- Reserved keywords (e.g., `if`, `for`, `while`) cannot be used as variable names.

  There are certain *reserved words* in python that have some meaning, and cannot be used as variable names. These reserved words are:

```
<IPython.core.display.Image object>
```

2.4.4 Best Practices:

Python style guide: Please refer to the python style guide for best coding practices, such as naming variables, using spaces, tabs, and styling the different components of your code.

For example:

```
# use descriptive variable names:
total_price = 100
```

```
# use snake_case for variable names
user_age = 25
```

2.4.5 Checking Variable Types

You can use the `type()` function to check the type of a variable.

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

y = "Python"
print(type(y))
```

```
<class 'int'>
<class 'str'>
```

2.4.6 Practice exercise 4

Which of the following variable names are valid?

1. var.name
2. var9name
3. _varname
4. varname*

In the statements below, determine the variable type

1. value = "name"
2. constant = 7
3. another_const = "variable"
4. True_False = True

2.5 Assignment statements

Values are assigned to variables with the assignment statement (=). An assignment statement may have a constant or an expression on the right hand side of the (=) sign, and a variable name on the left hand side.

For example, the code lines below are assignment statements

```
var = 2
var = var + 3
```

2.6 Expressions

2.6.1 Mathematical Operations and Their Operators in Python

Python provides the following operators for performing mathematical operations:

1. **Exponentiation (`**`)**: Raises a number to the power of another.
   - Example: `2 ** 3` results in `8`.
2. **Modulo (`%`)**: Returns the remainder of a division.
   - Example: `10 % 3` results in `1`.

3. **Multiplication (`*`)**: Multiplies two numbers.
   - Example: `4 * 5` results in `20`.
4. **Division (`/`)**: Divides one number by another, resulting in a float.
   - Example: `10 / 2` results in `5.0`.
5. **Addition (`+`)**: Adds two numbers.
   - Example: `7 + 3` results in `10`.
6. **Subtraction (`-`)**: Subtracts one number from another.
   - Example: `9 - 4` results in `5`.

2.6.2 Operator Precedence in Python

The operators listed above are in **decreasing order of precedence**, meaning:

1. **Exponentiation (`**`)** is evaluated first.
2. **Modulo (`%`)** is evaluated next.
3. **Multiplication (`*`)** follows.
4. **Division (`/`)**, if present, has the same precedence as multiplication.
5. **Addition (`+`)** and **Subtraction (`-`)** are evaluated last, from left to right.

2.6.2.1 Example: Precedence in Action

Consider the expression: `2 + 3 % 4 * 2`

To evaluate this, Python follows the precedence rules:

1. **Modulo (`%`)** is evaluated first:

```
3 % 4
```

```
3
```

Multiplication (`*`) is evaluated next:

```
3 * 2
```

```
6
```

Addition (+) is evaluated last:

```
2+6
```

```
8
```

Thus, the result of the expression `2 + 3 % 4 * 2` is `8`.

2.6.2.2 Key Takeaways

- Precedence determines the order in which operations are performed in an expression.
- Parentheses `()` can be used to override the default precedence and control the order of evaluation.

```
result = (2 + 3) % (4 * 2)
print(result)
```

```
5
```

2.6.3 Practice exercise 5

Which of the following statements is an assignment statement:

1. x = 5
2. print(x)
3. type(x)
4. x + 4

What will be the result of the following expression:

```
1%2**3*2+1
```

2.7 User input

Python's in-built `input()` function is used to take input from the user during program execution. It reads a line of text entered by the user and returns it as a **string**.

```
# suppose we wish the user to onput their age:
age = input("Enter your age:")
```

The entered value is stored in the variable `age` and can be used for computation.

2.7.0.1 Key Point

- The `input()` is always returned as a string, even if the user enters a number.
- You can convert the input to other types (e.g., `int`, `float`) using type conversion functions.
- The program execution pauses until the user provides input.

2.7.1 Examples

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

```
# using f-string for formatted output
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

```
# To take numeric input, you need to convert the string to an appropriate data type:
age = int(input("Enter your age: "))
print(f"You will be {age + 1} years old next year.")
```

```
# input for calculating the area of a circle
radius = float(input("Enter the radius of the circle: "))
area = 3.14 * radius ** 2
print(f"The area of the circle is {area}")
```

2.7.2 Practice exercise 6

Ask the user to input their year of birth, and print their age.

2.8 Converting data types in Python

2.8.1 Why Convert Data Types in Python?

Data type conversion is essential in Python for several reasons:

- **Compatibility**: Some operations or functions require specific data types to work correctly.
  - Example: Performing arithmetic operations like addition or multiplication requires numeric types such as int or float. If the input is in another type, such as a string, it must be converted first.

```
# Example: Converting strings to numbers
price = "19.99"
tax = 0.07
total_price = float(price) * (1 + tax)  # Convert string to float
print(total_price)  # Output: 21.3893
```

```
21.3893
```

- **Data Processing:** When working with input data (e.g., user input, etc), the data may need to be converted to the appropriate type for further analysis.

Example: Converting strings to numbers to perform calculations

```
# Example: Arithmetic requires numeric types
num_str = "42"
result = int(num_str) + 10  # Converts the string "42" to integer
print(result)  # Output: 52
```

```
52
```

- **Error Prevention**: Converting data types ensures consistency and prevents runtime errors caused by type mismatches.

```
# Example: Avoiding type mismatch errors
age = 25
message = "Your age is " + str(age)  # Convert integer to string for concatenation
print(message)  # Output: "Your age is 25"
```

```
Your age is 25
```

2.8.2 How to Convert Data Types in Python

Python provides several built-in functions for type conversion.

Common Conversion Functions:

Function	Description	Example
`int()`	Converts to an integer (from float or string)	`int("42")` → `42`
`float()`	Converts to a float	`float("3.14")` → `3.14`
`str()`	Converts to a string	`str(42)` → `"42"`
`bool()`	Converts to a boolean	`bool(1)` → `True`

However, in some cases, mathematical operators such as `+` and `*` can be applied on strings. The operator `+` concatenates multiple strings, while the operator `*` can be used to concatenate a string to itself multiple times:

```
"Hi" + " there!"
```

```
'Hi there!'
```

```
"5" + '3'
```

```
'53'
```

```
"5"*8
```

```
'55555555'
```

2.9 Errors and Exceptions

Errors and Exceptions are common while writing and executing Python code.

2.9.1 Syntax errors

Syntax errors occur if the code is written in a way that it does not comply with the rules / standards / laws of the language (python in this case). It occur when the Python parser encounters invalid syntax.

For example, suppose a values is assigned to a variable as follows:

```
9value = 2
```

The above code when executed will indicate a syntax error as it violates the rule that a variable name must not start with a number.

```
# another example
print("Hello World"
```

Solution: Fix the syntax issue by ensuring correct punctuation or structure.

2.9.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal:

Exceptions come in different types, and the type is printed as part of the message: below are the common ones:

- Misspelled or incorrectly capitalized variable and function names
- Attempts to perform operations (such as math operations) on data of the wrong type (ex. attempting to subtract two variables that hold string values)
- Dividing by zero
- Attempts to use a type conversion function such as `int` on a value that can't be converted to an `int`

For example, suppose a number is multipled as follws:

```
multiplication_result = misy * 4
```

```
NameError: name 'misy' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[29], line 1
----> 1 multiplication_result = misy * 4
NameError: name 'misy' is not defined
```

The above code is syntactically correct. However, it will generate an error as the variable `misy` has not been defined as a number.

```
int("abc")
```

```
ValueError: invalid literal for int() with base 10: 'abc'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[25], line 1
----> 1 int("abc")
ValueError: invalid literal for int() with base 10: 'abc'
```

```
print("2" + 3)
```

```
TypeError: can only concatenate str (not "int") to str
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 print("2" + 3)
TypeError: can only concatenate str (not "int") to str
```

```
print(10 / 0)
```

```
ZeroDivisionError: division by zero
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[27], line 1
----> 1 print(10 / 0)
ZeroDivisionError: division by zero
```

2.9.3 Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

If we suspect that some lines of code may produce an error, we can put them in a `try` block, and if an error does occur, we can use the `except` block to instead execute an alternative piece of code. This way the program will not stop if an error occurs within the `try` block, and instead will be directed to execute the code within the `except` block.

These exceptions can be handled Using the Try-Except Blocks

```
try:
    print(10 / 0)
except:
    print("Cannot divide by zero!")
```

```
Cannot divide by zero!
```

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error:

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

```python
try:
    print(10 / 0)
except:
    print("Cannot divide by zero!")
finally:
    print("This will always execute.")
```

```
Cannot divide by zero!
This will always execute.
```

2.9.4 Practice exercise 7

Suppose we wish to compute tax using the income and the tax rate. Identify the type of error from amongst syntax error, semantic error and run-time error in the following pieces of code.

```python
income = 2000
tax = .08 * Income
print("tax on", income, "is:", tax)
```

```python
income = 2000
tax = .08 x income
print("tax on", income, "is:", tax)
```

```python
income = 2000
tax = .08 ** income
print("tax on", income, "is:", tax)
```

2.9.5 Practice exercise 8

Input an integer from the user. If the user inputs a valid integer, print whether it is a multiple of 3. However, if the user does not input a valid integer, print a message saying that the input is invalid.

```
num = input("Enter an integer:")

#The code lines within the 'try' block will execute as long as they run without error
try:
    #Converting the input to integer, as user input is a string
    num_int = int(num)

    #checking if the integer is a multiple of 3
    if num_int % 3 == 0:
        print("Number is a multiple of 3")
    else:
        print("Number is not a multiple of 3")

#The code lines within the 'except' block will execute only if the code lines within the 'try
except:
    print("Input must be an integer")
```

```
Input must be an integer
```

2.9.6 Semantic errors (bugs)

Semantic errors occur when the code executes without an error being indicated by the compiler. However, it does not work as inteded by the user. For example, consider the following code of mutiplying the number 6 by 3: x = '6' x * 3 If it was intended to multiply the number 6, then the variable `x` should have been defined as `x=6` so that `x` has a value of type `integer`. However, in the above code `6` is a `string` type value. When a `string` is multiplied by an integer, say n, it concatenates with itself n times.

2.9.7 Practice exercise 9

The formula for computing final amount if one is earning compund interest is given by:

$$A = P\left(1 + \frac{r}{n}\right)^{nt},$$

where:
P = Principal amount (initial investment),
r = annual nominal interest rate,
n = number of times the interest is computed per year,
t = number of years

Write a Python program that assigns the principal amount of \$10000 to variable P, assign to n the value 12, and assign to r the interest rate of 8%. Then have the program prompt the user for the number of years t that the money will be compounded for. Calculate and print the final amount after t years.

What is the amount if the user enters t as 4 years?

3 Control flow tools

```
<IPython.core.display.Image object>
```

A control flow statement in a computer program determines the individual lines of code to be executed and/or the order in which they will be executed. In this chapter, we'll learn about 3 types of control flow statements:

1. if-elif-else
2. for and while loops
3. break, continue, and pass statements

3.1 Indentation in Python

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

3.1.1 What is Indentation?

- Indentation refers to the spaces or tabs at the beginning of a line of code.
- In Python, indentation is **mandatory** and is used to define the structure of the code, such as blocks in loops, conditionals, functions, and classes.
- Unlike some other programming languages, Python does not use braces `{}` or keywords like `begin` and `end` to define blocks of code.

3.1.2 Rules for Indentation

1. **Consistency is Key**:
   - Use either spaces or tabs for indentation, but **do not mix them** in the same file.
   - The recommended standard is to use **4 spaces** per indentation level (PEP 8).
2. **Indentation Levels**:
   - Each block of code under a statement (e.g., `if`, `for`, `while`, `def`) must be indented one level deeper than the statement itself.

```
<IPython.core.display.Image object>
```

3.1.3 Examples

```python
# Example of proper indentation
def greet(name):
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, World!")
```

3.2 Conditonal execution

The first type of control flow satement is `if-elif-else`. This statement helps with conditional execution of code, i.e., the piece of code to be executed is selected based on certain condition statements(s).

```
<IPython.core.display.Image object>
```

3.2.1 Comparison operators

For testing if conditions are true or false, first we need to learn the operators that can be used for comparison. For example, suppose we want to check if two objects are equal, we use the `==` operator:

```python
5 == 6
```

```
False
```

Make sure you can differentiate between the `==` and `=` operators: - **`==`**: This is a **comparison operator**, used to compare two values and return a Boolean result (`True` or `False`). - **`=`**: This is an **assignment operator**, used to assign values to variables.

```python
x = "hi"
y = "hi"
x == y
```

```
True
```

Below are the python comparison operators and their meanings.

Python code	Meaning
x == y	Produce True if … x is equal to y
x != y	… x is not equal to y
x > y	… x is greater than y
x < y	… x is less than y
x >= y	… x is greater than or equal to y
x <= y	… x is less than or equal to y

3.2.2 Logical Operators in Conditional Statements

Logical operators are used to **combine multiple conditions** in a conditional statement, allowing for more complex decision-making. Python provides three logical operators:

3.2.2.1 `and`

- **Description**: Returns `True` if both conditions are `True`.
- **Example**:

```python
x = 5
if x > 0 and x < 10:
    print("x is a positive single-digit number.")
```

```
x is a positive single-digit number.
```

3.2.2.2 `or`

- **Description**: Returns `True` if at least one condition is `True`.
- **Example**:

```python
x = -5
if x < 0 or x > 10:
    print("x is either negative or greater than 10.")
```

```
x is either negative or greater than 10.
```

3.2.2.3 `not`

- **Description**: Returns the negation of a condition (`True` becomes `False` and vice versa).
- **Example**:

```
x = 5
if not (x < 0):
    print("x is not negative.")
```

```
x is not negative.
```

3.2.3 if-elif-else statement

The `if-elif-else` statements can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many `elif` statements as required.

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

Example: Input an integer. Print whether it is positive or negative.

```
number = input("Enter a number:")  #Input an integer
number_integer = int(number)       #Convert the integer to 'int' datatype
if number_integer > 0:               #Check if the integer is positive
    print("Number is positive")
else:
    print("Number is negative")
```

```
Number is positive
```

In the above code, note that anything entered by the user is taken as a string datatype by python. However, a string cannot be positive or negative. So, we converted the number input by the user to integer to check if it was positive or negative.

There may be multiple statements to be executed if a condition is true. See the example below.

Example: Input a number. Print whether it is positive, negative or zero. If it is negative, print its absolute value.

```python
number = input("Enter a number:")
number_integer = int(number)
if number_integer > 0:
    print("Number is positive")
elif number_integer == 0:
    print("Number is zero")
else:
    print("Number is negative")
    print("Absolute value of number = ", abs(number_integer))
```

```
Number is positive
```

3.2.4 Practice exercise 1

Input a number. Print whether its odd or even.

Solution:

```python
num = int(input("Enter a number: "))
if num%2 == 0:           #Checking if the number is divisible by 2
    print("Number is even")
else:
    print("Number is odd")
```

```
Number is odd
```

3.2.5 Practice exercise 2

3.2.5.1

Ask the user to enter their exam score. Print the grade based on their score as follows:

Score	Grade
(90,100]	A
(80,90]	B
[0,80]	C

If the user inputs a score which is not a number in [0,100], print invalid entry.

Solution:

```
score = input("Enter exam score:")
try:

    #As exam score can be a floating point number (such as 90.65), we need to use 'float' ins
    score_num = float(score)
    if score_num > 90 and score_num <= 100:
        print("Grade: A")
    elif score_num > 80 and score_num <= 90:
        print("Grade: B")
    elif score_num >= 0 and score_num <= 80:
        print("Grade: C")
    else:
        print("Invalid score")     #If a number is less than 0 or more than 100
except:
    print("Invalid input")         #If the input is not a number
```

```
Grade: C
```

3.2.5.2

Nested if-elif-else statements: This question will lead you to create nested `if` statements, i.e., an `if` statement within another `if` statement.

Think of a number in [1,5]. Ask the user to guess the number.

- If the user guesses the number correctly, print "Correct in the first attempt!", and stop the program. Otherwise, print "Incorrect! Try again" and give them another chance to guess the number.
- If the user guesses the number correctly in the second attempt, print "Correct in the second attempt", otherwise print "Incorrect in both the attempts, the correct number is:", and print the correct number.

Solution:

```
#Let us say we think of the number. Now the user has to guess the number in two attempts.
rand_no = 3
guess = input("Guess the number:")
if int(guess)==rand_no:
    print("Correct in the first attempt!")

#If the guess is incorrect, the program will execute the code block below
else:
```

```python
    guess = input("Incorrect! Try again:")
    if int(guess) == rand_no:
        print("Correct in the second attempt")
    else:
        print("Incorrect in the both the attempts, the correct number was:", rand_no)
```

```
Incorrect in the both the attempts, the correct number was: 3
```

3.3 Loops in Python

Python provides two types of loops: **`for`** and **`while`** loops. Loops are used to execute a block of code repeatedly until a certain condition is met.

3.3.1 Using `range()` in `for` Loops

The `range()` function is commonly used with loops in Python to generate a sequence of numbers. It is particularly useful with `for` loops.

3.3.1.1 Syntax of `range()`

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

- `start`: (Optional) The starting value of the sequence (default is `0`).
- `stop`: (Required) The endpoint of the sequence (exclusive).
- `step`: (Optional) The difference between each number in the sequence (default is `1`).

Using the `range()` function, the `for` loop can iterate over a sequence of numbers. See the examples below.

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

```
0
1
2
3
4
```

Note that the last element is one less than the integer specified in the `range()` function.

```
# specify start and stop
for i in range(5, 10):
    print(i)
```

```
5
6
7
8
9
```

```
# use step
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

```
# A negative step will count down
for i in range(10, 0, -2):
    print(i)
```

```
10
8
6
4
2
```

```
# if start is greater than steop, the range will generate an empty sequence
for i in range(10, 0):
    print(i)
```

Example: Print the first n elements of the Fibonacci sequence, where n is an integer input by the user, such that $n>2$. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,….

```
n = int(input("Enter number of elements:"))

#Initializing the sequence to start from 0, 1
n1, n2 = 0, 1

#Printing the first two numbers of the sequence
print(n1)
print(n2)

for i in range(n-2):  #Since two numbers of the sequence are already printed, n-2 numbers are

    #Computing the next number of the sequence as the summation of the previous two numbers
    n3 = n1 + n2
    print(n3)

    #As 'n3' is already printed, it is no longer the next number of the sequence.
    #Thus, we move the values of the variables n1 and n2 one place to the right to compute th
    n1 = n2
    n2 = n3

print("These are the first", n, "elements of the fibonacci series")
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
1836311903
2971215073
4807526976
7778742049
12586269025
20365011074
32951280099
53316291173
86267571272
139583862445
225851433717
365435296162
591286729879
956722026041
1548008755920
2504730781961
4052739537881

```
6557470319842
10610209857723
17167680177565
27777890035288
44945570212853
72723460248141
117669030460994
190392490709135
308061521170129
498454011879264
806515533049393
1304969544928657
2111485077978050
3416454622906707
5527939700884757
These are the first 78 elements of the fibonacci series
```

As in the `if-elif-else` statement, the `for` loop uses identation to indicate the piece of code to be run repeatedly.

```
# nested with range
for i in range(5):
    for j in range(5):
        print(i, j)
```

```
0 0
0 1
0 2
0 3
0 4
1 0
1 1
1 2
1 3
1 4
2 0
2 1
2 2
2 3
2 4
3 0
3 1
```

```
3 2
3 3
3 4
4 0
4 1
4 2
4 3
4 4
```

3.3.2 `while` loop

With a `while` loops, a piece of code is executed repeatedly until **certain condition(s)** hold.

Example: Print all the elements of the Fibonacci sequence less than n, where n is an integer input by the user, such that n>*2*. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,....

```python
n = int(input("Enter the value of n:"))

#Initializing the sequence to start from 0, 1
n1, n2 = 0, 1

#Printing the first number of the sequence
print(n1)

while n2 < n:

    #Print the next number of the sequence
    print(n2)

    #Comptuing the next number of the sequence as the summation of the previous two numbers
    n3 = n1 + n2

    #As n2 is already printed, assigning n2 to n3, so that the next number of the sequence (i
    #Assigning n1 to n2 as n1 has already been used to compute the next number of the seqeunc
    n1 = n2
    n2 = n3
print("These are all the elements of the fibonacci series less than", n)
```

```
0
```

```
1
1
2
3
5
8
13
21
These are all the elements of the fibonacci series less than 23
```

3.3.3 Practice exercise 3

3.3.3.1

Write a program that identifies whether a number input by the user is prime or not.

Solution:

```
number = int(input("Enter a positive integer:"))

#Defining a variable that will have a value of 0 if there are no divisors
num_divisors = 0

#Checking if the number has any divisors from 2 to half of the number
for divisor in range(2,int(number/2+1)):
        if number % divisor == 0:

            #If the number has a divisor, setting num_divisors to 1, to indicate that the num
            num_divisors = 1

            #If a divisor has been found, there is no need to check if the number has more di
            #Even if the number has a single divisor, it is not prime. Thus, we 'break' out o
            #If you don't 'break', your code will still be correct, it will just do some unne
            break

#If there are no divisors of the number, it is prime, else not prime
if num_divisors == 0:
    print("Prime")
else:
    print("Not prime")
```

```
Not prime
```

3.3.3.2

Update the program above to print the prime numbers starting from 2, and less than n where n is a positive integer input by the user.

Solution:

```
n = int(input("Enter a positive integer:"))

#Defining a variable - number_iterator. We will use this variable to iterate over all intege
#While iterating over each integer from 2 to n, we will check if the integer is prime or not
number_iterator = 2

print(number_iterator) #Since '2' is a prime number, we can print it directly (without check

#Continue to check for prime numbers until n (but not including n)
while(number_iterator < n):

    #After each check, increment the number_iterator to check if the next integer is prime
    number_iterator = number_iterator + 1

    #Defining a variable that will have a value of 0 if there are no divisors
    num_divisors = 0

    #Checking if the integer has any divisors from 2 to half of the integer being checked
    for divisor in range(2,int(number_iterator/2 + 1)):
        if number_iterator % divisor == 0:

            #If the integer has a divisor, setting num_divisors to 1, to indicate that the n
            num_divisors = 1

            #If a divisor has been found, there is no need to check if the integer has more
            #Even if the integer has a single divisor, it is not prime.
            #Thus, we 'break' out of the loop that checks for divisors
            break

    #If there are no divisors of the integer being checked, the integer is a prime number, a
    if num_divisors == 0:
        print(number_iterator)
```

```
2
3
```

```
5
7
11
13
17
19
23
29
31
37
41
43
```

3.4 Control flow statements

They are used to influence the flow of execution in loops or blocks of code. Python provides three such statements: `break`, `continue`, and `pass`.

3.4.1 `break` statement

The break statement is used to **exit a loop prematurely** before it has iterated through all elements or completed its condition. It is commonly used in both `for` and `while` loops to stop the execution of the loop when a specific condition is met.

For example

```python
for i in range(10):
    if i == 5:
        print("Breaking the loop at i =", i)
        break
    print(i)
```

```
0
1
2
3
4
Breaking the loop at i = 5
```

3.4.2 Practice exercise 4

Write a program that finds and prints the largest factor of a number input by the user. Check the output if the user inputs 133.

Solution:

```
num = int(input("Enter an integer:"))

#Looping from the half the integer to 0 as the highest factor is likely to be closer to half
for i in range(int(num/2) + 1, 0, -1):
    if num%i == 0:
        print("Largest factor = ", i)

        #Exiting the loop if the largest integer is found
        break
```

```
Largest factor =  1
```

3.4.3 `continue` statement

The `continue` statement is used to **skip the current iteration of a loop** and move to the next iteration. Unlike the `break` statement, it does not terminate the loop but allows the loop to continue running.

For example, consider the following code:

```
for i in range(5):
    if i == 3:
        print(f"Skipping iteration {i}")
        continue
    print(f"Processing {i}")
```

```
Processing 0
Processing 1
Processing 2
Skipping iteration 3
Processing 4
```

The `continue` statement skips the iteration when `i == 3` and moves to the next iteration.

3.4.4 Practice exercise 5:

Write a program that asks the user the question, “How many stars are in the Milky Way (in billions)?”. If the user answers 100, the program should print correct, and stop. However, if the user answers incorrectly, the program should print “incorrect”, and ask them if they want to try again. The program should continue to run until the user answers correctly, or they want to stop trying.

```
#Defining an infinite while loop as the loop may need to run indefinitely if the user keeps a
while True:
    answer = input("How many stars are there in the Milky Way? ")
    if answer == '100':
        print("Correct")

        #Exiting the loop if the user answers correctly
        break
    else:
        print("Incorrect")
        try_again = input("Do you want to try again? (Y/N) ")
        if try_again == 'Y':

            #Continuing with the infinite loop if the user wants to try again
            continue
        else:

            #Exiting the infinite loop if the user wants to stop tryinh
            break
```

```
Incorrect
```

3.4.5 `pass` statement

The `pass` statement in Python is a **null statement**. It serves as a placeholder and does nothing when executed. It is often used in situations where a statement is syntactically required, but no action is intended, or the code is yet to be implemented. In Chapter 4, we will use this statement when we explore user-defined functions.

3.5 Loops with strings

Strings in Python are sequences of characters. You can use loops to iterate over strings and perform various operations on each character or a subset of the string.

Consider the following string:

```
sentence = "She sells sea shells on the sea shore"
```

The i^{th} character of the string can be retrieved by its index. For example, the first character of the string `sentence` is:

```
sentence[0]
```

```
'S'
```

A `for` loop iterates over each character in a string, one at a time.

```
for char in sentence:
    print(char, end=" ")
```

```
S h e   s e l l s   s e a   s h e l l s   o n   t h e   s e a   s h o r e
```

A `while` loop can be used to iterate over a string by index.

```
index = 0
while index < len(sentence):
    print(sentence[index], end=" ")
    index += 1
```

```
S h e   s e l l s   s e a   s h e l l s   o n   t h e   s e a   s h o r e
```

Slicing a string:

A part of the string can be sliced by passing the starting index (say `start`) and the stopping index (say `stop`) as `start`:`stop` to the index operator []. This is called slicing a string. For a string `S`, the characters starting from the index `start` upto the index `stop`, but not including `stop`, can be sliced as `S[start:stop]`.

For example, the slice of the string `sentence` from index `4` to index `9`, but not including `9` is:

```
sentence[4:9]
```

```
'sells'
```

Example: Counting characters

Input a string, and count and print the number of "*t*"s.

```
str1 = input("Enter a sentence:")

#Initializing a variable 'count_t' which will store the number of 't's in the string
count_t = 0

#Iterating over the entire length of the string.
#The length of the string is given by the len() function
for i in range(len(str1)):

    #If the ith character of the string is 't', then we count it
    if str1[i] == 't':
        count_t = count_t + 1

print("Number of 't's in the str1 = ", count_t)
```

```
Number of 't's in the str1 =  2
```

Another way to achieve this is to use `count()`

```
str1.count('t')
```

```
2
```

3.5.1 Practice exercise 6

Checking for a Substring

Write a program that asks the user to input a string, and print the number of "*the*"s in the string.

```
str2 = input("Enter a sentence:")

#Defining a variable to store the count of the word 'the'
count_the = 0

#Looping through the entire length of the string except the last 3 letters.
#As we are checking three letters at a time starting from the index 'i', the last 3 letters o
```

```python
for i in range(len(str2) - 3):

    #Slicing 3 letters of the string and checking if they are 'the'
    if str2[i:(i+3)] == 'the':

        #Counting the words that are 'the'
        count_the = count_the + 1
print("Number of 'the's in the str2 = ", count_the)
```

```
Number of 'the's in the str2 =  3
```

```python
# using the count method
str2.count('the')
```

```
3
```

4 Functions

```
<IPython.core.display.Image object>
```

4.1 Function Definition

Functions are the fundamental building blocks of any Python program. They are organized blocks of reusable code designed to perform a specific task. A function can take one or more inputs (parameters), execute a block of code, and optionally return one or more values.

4.1.1 Why Use Functions?

Functions allow developers to write modular, reusable, and efficient code. Instead of duplicating the same logic multiple times, functions let you define the logic once and call it wherever needed.

4.2 Advantages of Functions

1. **Increases Modularity**
   - Functions allow the program to be divided into smaller, manageable parts, making it easier to understand, implement, and maintain.
2. **Reduces Redundancy**
   - By defining a function once, you avoid rewriting the same code multiple times. Simply call the function when needed.
3. **Maximizes Code Reusability**
   - Functions can be used as many times as necessary, enabling you to reuse your code efficiently and reducing overall development effort.
4. **Improves Code Readability**
   - Dividing a large program into smaller functions improves the clarity and readability of the code, making it easier to debug and maintain

4.3 Types of Functions

There are two types of functions in python:

- **Predefined Functions** - These are built-in functions in python.
- **User-Defined Functions** - these types of functions are defined by the user to perform any specific task

4.3.1 Functions

These are built-in functions that perform common tasks. Built-in functions come from two main sources:

- Python Standard Libaries
- Third-Party Libraries

4.3.1.1 Python Standard Library

The Python Standard Library is an umbrella term for all the modules (A module is a file containing Python code (functions, classes, variables) that can be reused in your programs) and packages that come with Python, including both built-in modules (e.g., `__builtins__`) and other modules that require importing. Think of the standard library as a toolbox, with some tools always on the table (built-in) and others stored in drawers (import-required). Built-in functions like `print()`, `len()`, and `type()` are available directly without needing to import anything. They are part of Python's built-in namespace, which is loaded into memory when Python starts.

Many modules in the Python Standard Library, like `math`, `os`, or `datetime`, are not automatically loaded to keep the startup time and memory usage low. To access functions or classes from these modules, you need to explicitly import them using the `import` keyword.

Let's see different ways to import modules next

- Basic Import

```
import math
# To use a function from the module, preface it with random followed by a dot, and then the 
print(math.sqrt(16))
```

```
4.0
```

- Import Specific Functions or Classes

```
# import only sqrt function from math module
from math import sqrt, pi
print(sqrt(25))
```

```
5.0
```

- Import with Alias:

```
import numpy as np
print(np.array([1, 2, 3]))
```

```
[1 2 3]
```

- Wildcard Import (Not Recommended):

```
from math import *
print(sin(1))
```

```
0.8414709848078965
```

This way imports every function from the module. You should usually **avoid** doing this, as the module may contain some names that will interfere with your own variable names. For instance if your program uses a variable called `total` and you import a module that contains a function called `total`, there can be problems. In contrast, the first way imports an entire module in a way that will not interfere with your variable names. To use a function from the module, preface it with the module name followed by a dot

Location: Usually, import statements go at the beginning of the program, but there is no restriction. They can go anywhere as long as they come before the code that uses the module.

4.3.1.2 Useful Modules

Here's a list of commonly used and useful modules from the Python Standard Library:

- `os`: For interacting with the operating system, such as file paths and environment variables.
- `sys`: For interacting with the Python runtime environment
- `re`: For regular expressions and pattern matching
- `math`: For mathematical functions and constants
- `random`: For generating random numbers.
- `datetime`: For working with dates and times
- `time`: For measuring time or introducing delays.

4.3.1.2.1 Random Numbers

Generating random numbers is very useful in python for performing simulations (we'll see in later chapters). The library random is used to generate random numbers such as integers, real numbers based on different probability distributions, etc.

Below is an example of using the `randint()` function of the library for generating random numbers in [a, b], where `a` and `b` are integers.

```
import random as rm
rm.randint(5,10) #This will generate a random number in [5,10]
```

```
7
```

The random number will be different every time we run the program.

4.3.1.2.2 Math Functions

Python's math module is part of the standard library and provides access to common mathematical functions and constants. You can use these functions for tasks such as computing square roots, trigonometric operations, logarithms, factorials, and more.

For example:

```
from math import sin, pi
print(pi)
print(pi/2)
print('sin(pi/2) =', sin(pi/2))
```

```
3.141592653589793
1.5707963267948966
sin(pi/2) = 1.0
```

4.3.1.3 Getting Help from Python on a Module

There is documentation built into Python. To get help on the `random` module

```
dir(rm)
```

```
['BPF',
 'LOG4',
 'NV_MAGICCONST',
 'RECIP_BPF',
 'Random',
 'SG_MAGICCONST',
 'SystemRandom',
 'TWOPI',
 '_ONE',
 '_Sequence',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_accumulate',
 '_acos',
 '_bisect',
 '_ceil',
 '_cos',
 '_e',
 '_exp',
 '_fabs',
 '_floor',
 '_index',
 '_inst',
 '_isfinite',
 '_lgamma',
 '_log',
 '_log2',
 '_os',
 '_pi',
 '_random',
 '_repeat',
 '_sha512',
 '_sin',
 '_sqrt',
 '_test',
 '_test_generator',
```

```
 '_urandom',
 '_warn',
 'betavariate',
 'binomialvariate',
 'choice',
 'choices',
 'expovariate',
 'gammavariate',
 'gauss',
 'getrandbits',
 'getstate',
 'lognormvariate',
 'normalvariate',
 'paretovariate',
 'randbytes',
 'randint',
 'random',
 'randrange',
 'sample',
 'seed',
 'setstate',
 'shuffle',
 'triangular',
 'uniform',
 'vonmisesvariate',
 'weibullvariate']
```

This provides a list of all the functions and variables in the `random` module. You can ignore entries that start with underscores, as they are typically used internally. To get help on a specific function, such as the `uniform` function, you can type:

```
help(rm.uniform)
```

```
Help on method uniform in module random:

uniform(a, b) method of random.Random instance
    Get a random number in the range [a, b) or [a, b] depending on rounding.

    The mean (expected value) and variance of the random variable are:

        E[X] = (a + b) / 2
        Var[X] = (b - a) ** 2 / 12
```

For a comprehensive overview of the entire `math` module, type:

```
# help(rm) #This will give you all the functions available in the random module
```

I encourage you to explore the documentation for a deeper understanding, especially when you need to use a module but are unsure how to get started.

4.3.2 Practice exercise 1

- Can you use `math.sqrt(16)` without importing the math module? Why or why not?
- Identify whether the following functions require importing a module:
  - `abs()`
  - `random.randint()`
  - `time.sleep()`

4.3.3 Practice exercise 2

Generate a random integer between [-5,5]. Do this 10,000 times. Find the `mean` of all the 10,000 random numbers generated.

4.3.3.1 Third-Party Python libraries

Other than the Python Standard Library, Python has hundreds of thousands of additional libraries that provide a wealth of useful functions. Since Python is an open-source platform, these libraries are contributed by developers from around the world. Some of the most popular libraries in data science and their purposes are listed below:

1. NumPy: Performing numerical operations and efficiently storing numerical data.
2. Pandas: Reading, cleaning and manipulating data.
3. Matplotlib, Seaborn: Visualizing data.
4. SciPy: Performing scientific computing such as solving differential equations, optimization, statistical tests, etc.
5. Scikit-learn: Data pre-processing and machine learning, with a focus on prediction.
6. Statsmodels: Developing statistical models with a focus on inference

Before you can use them, you need to `install` each library and then `import` it in your code.

A library can be imported using the `import` keyword after it has been successfully installed. For example, the `NumPy` library can be imported as:

```
import numpy as np
```

Using the `as` keyboard, the NumPy library has been given the name `np`. All the functions and attributes of the library can be called using the *'np.'* prefix. For example, let us generate a sequence of whole numbers upto `10` using the NumPy function arange():

```
np.arange(8)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

You will use these libraries in the upcoming data science courses.

4.4 User-defined Functions

A user-defined function is a function created by the user in Python to perform a specific task. Unlike built-in functions (like `print()` or `len()`), user-defined functions allow you to define **custom functionality tailored to your program's needs**.

The image below provides a helpful breakdown of a Python function definition with labels for each component.

```
<IPython.core.display.Image object>
```

4.4.1 Key Components (Based on the Diagram)

1. **`def` keyword**: Indicates the start of a function definition.
2. **Function name**: A descriptive name for the function, following Python naming conventions.
3. **Parameters**: Variables passed into the function inside parentheses (`x`, `y` in the example). These are optional.
4. **Colon (`:`)**: Signals the end of the function header and the start of the body.
5. **Docstring**: A multi-line string (optional) that describes the purpose and functionality of the function.
6. **Function body**: Contains the logic and statements of the function.
7. **`return` statement**: Outputs a result back to the caller. This is optional.

Example

```python
# define a function
def my_function():
    print("Hello from a function")
```

4.4.2 Functions are lazy

Functions are designed to be reusable. They don't run until explicitly called, so their behavior can be invoked multiple times,

Call the `my_function` to execute it

```python
# Simply use the function's name followed by parentheses
my_function()

# call the function again
my_function()
```

```
Hello from a function
Hello from a function
```

The function was called twice, it printed out the information twice.

4.4.3 Arguments and Parameters in a Function

The terms arguments and parameters are often used interchangeably but have distinct meanings in the context of functions in Python.

4.4.3.1 Parameters:

- **Definition**: Parameters are the variables listed in a function's definition. They act as placeholders that specify the input a function can accept.
- **When Used**: Defined when you write the function.
- **Example**:

```python
def greet_user(name):  # 'name' is the parameter
    print(f"Hello, {name}!")
```

4.4.3.2 Arguments:

- **Definition**: Arguments are the actual values or data you pass to a function when you call it. These values are assigned to the function's parameters.
- **When Used**: Provided when you invoke (call) the function.
- **Example**:

```
greet_user("Alice")  # "Alice" is the argument
```

```
Hello, Alice!
```

Another Example:

```
# Function definition with parameters
def add_numbers(a, b):
    return a + b

# Function call with arguments
result = add_numbers(5, 3)

print(result)
```

```
8
```

Understanding the distinction between **parameters** and **arguments** is crucial for writing clear and effective functions in Python.

4.4.4 Type of Arguments in Python

4.4.4.1 Required Arguments

These are the arguments that must be provided when the function is called. If they are missing, Python will raise a `TypeError`.

```
def greet_user(name):
    print(f"Hello, {name}!")

# Call with a required argument
greet_user("Alice")  # Output: Hello, Alice!
```

```python
# Call without an argument will raise an error
# greet_user()  # TypeError: greet_user() missing 1 required positional argument: 'name'
```

```
Hello, Alice!
```

4.4.4.2 Keyword Arguments

These allow you to specify arguments by their parameter name. This makes your code more readable and avoids confusion, especially when dealing with multiple arguments.

```python
def describe_person(name, age):
    print(f"{name} is {age} years old.")

# Call with keyword arguments
describe_person(name="Bob", age=30)  # Output: Bob is 30 years old.
describe_person(age=25, name="Alice")  # Output: Alice is 25 years old.

# Call without keywords (positional)
describe_person("Charlie", 40)  # Output: Charlie is 40 years old.
describe_person(35, "David")  # Output: 35 is David years old.
```

```
Bob is 30 years old.
Alice is 25 years old.
Charlie is 40 years old.
35 is David years old.
```

4.4.4.3 Default Arguments

These are parameters that have default values. If no argument is provided during the function call, the default value is used.

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")

# Call with an argument
greet("Alice")  # Output: Hello, Alice!

# Call without an argument
greet()  # Output: Hello, Guest!
```

```
Hello, Alice!
Hello, Guest!
```

Note: Default arguments must come after required arguments in the parameter list.

```python
def greet(name="Guest", message):
    print(f"Hello, {name}! {message}")
```

```
SyntaxError: parameter without a default follows parameter with a default (3495907264.py, li
  Cell In[11], line 1
    def greet(name="Guest", message):
                            ^
SyntaxError: parameter without a default follows parameter with a default
```

4.4.4.4 Variable-Length Arguments

These allow a function to accept an arbitrary number of arguments.

Using `*args` for Non-Keyword Variable-Length Arguments The `*args` syntax is used to pass a variable number of positional arguments. These arguments are accessible as a tuple.

```python
def sum_numbers(*args):
    total = sum(args)
    print(f"The sum is {total}.")

# Call with multiple arguments
sum_numbers(1, 2, 3, 4)
sum_numbers(10, 20)

# Call without arguments
sum_numbers()
```

```
The sum is 10.
The sum is 30.
The sum is 0.
```

```python
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
```

```
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

The `print()` function in Python allows you to pass multiple arguments, separated by commas. When you do so, `print()` automatically converts each argument to a string (if it's not already a string) and joins them with a default separator, which is a **space**.

Example with multiple arguments:

```python
print("Hello", "world!", 123, True)
```

```
Hello world! 123 True
```

You can change the default separator using the sep parameter.

```python
print("Hello", "world!", 123, True, sep="***")
```

```
Hello***world!***123***True
```

You can change the default `end` parameter as well

```python
print("Hello", "world!", sep="***", end=" :) ")
print("This is fun!")
```

```
Hello***world! :) This is fun!
```

Using `**kwargs` for Keyword Variable-Length Arguments

The `**kwargs` syntax is used to accept a variable number of keyword arguments. These arguments are accessible as a dictionary.

```python
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Call with keyword arguments
print_details(name="Alice", age=30, city="Chicago")
```

```
name: Alice
age: 30
city: Chicago
```

Example: Combining `*args` and `**kwargs`

```python
def mixed_function(a, *args, **kwargs):
    print(f"Fixed argument: {a}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

mixed_function(1, 2, 3, name="Alice", age=30)
```

```
Fixed argument: 1
Args: (2, 3)
Kwargs: {'name': 'Alice', 'age': 30}
```

4.4.4.5 Combining All Types of Arguments

You can use all these types of arguments in a single function, but they must follow a specific order:

1) Required arguments
2) Default arguments
3) `*args`
4) `**kwargs`

```python
def display_info(name, age=18, *hobbies, **details):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Hobbies: {', '.join(hobbies)}")
    for key, value in details.items():
        print(f"{key}: {value}")
```

```python
# Call with all types of arguments
display_info(
    "Alice",
    25,
    "reading", "traveling",
    city="Chicago", job="Data Scientist"
)
```

```
Name: Alice
Age: 25
Hobbies: reading, traveling
city: Chicago
job: Data Scientist
```

4.4.5 Practice exercise 3

Write a function that prints prime numbers between two real numbers - `a` and `b`, where `a` and `b` are the parameters of the function. Call the function and check the output with a = 60, b = 80.

4.4.6 Functions that return objects

Until now, we saw functions that print text. However, the functions did not `return` any object. For example, the function `odd_even` prints whether the number is odd or even. However, we did not save this information. In future, we may need to use the information that whether the number was odd or even. Thus, typically, we return an object from the function definition, which consists of the information we may need in the future.

The example `odd_even` can be updated to return the text "odd" or "even" as shown below:

```python
#This is an example of a function definition that has an argument with a default value, and 
def odd_even(num=0):
    if num%2==0:
        return("Even")
    else:
        return("Odd")
```

The function above returns a string "Odd" or "Even", depending on whether the number is odd or even. This result can be stored in a variable, which can be used later.

```
response=odd_even(3)
response
```

```
'Odd'
```

The variable `response` now refers to the object where the string "Odd" or "Even" is stored. Thus, the result of the computation is stored, and the variable can be used later on in the program. Note that the control flow exits the function as soon as the first `return` statement is executed.

4.4.7 Practice exercise 4: Create a Custom Calculator

Write a function calculator that performs basic arithmetic operations: addition, subtraction, multiplication, and division.

Steps:

1. Define a function `calculator` with three parameters: `a`, `b`, and `operation` (a string indicating the operation, e.g., `'add'`, `'subtract'`).
2. Use conditional statements to handle the operations.
3. Return the result of the operation.
4. Handle invalid operations gracefully.

Expected Output Examples:

```
print(calculator(10, 5, 'add'))        # Output: 15
print(calculator(10, 5, 'subtract'))   # Output: 5
print(calculator(10, 5, 'multiply'))   # Output: 50
print(calculator(10, 5, 'divide'))     # Output: 2.0
```

4.4.8 Bonus question: Calculator Function with Variable Number of Inputs

Update the `calculator` function to handle a variable number of inputs using the `*args`

Hints

- **Variable-Length Arguments (`*args`):** The `*args` parameter allows the function to accept an arbitrary number of arguments. These arguments are collected into a tuple, making them easy to iterate over.
- **Handling the Operator:** The operator parameter remains a keyword argument, defaulting to `"+"`. Supported operators are "+", "-", "*", and"/".

- **Logic:** Start with the first value in args (`result = args[0]`). Iterate through the rest of the numbers in `args[1:]` and apply the operator cumulatively.
- **Validation:** If fewer than two numbers are provided, return an error message. Handle division by zero with an additional check.

4.4.9 Practice exercise 5: Palindrome Checker

Write a function called `is_palindrome` that checks if a string is a palindrome (reads the same forward and backward).

Steps:

1. Define the function `is_palindrome` with one parameter, `text`.
2. Ignore case and spaces.
3. Return `True` if the string is a palindrome, otherwise False.

Expected OUtput Examples:

```
print(is_palindrome("radar"))         # Output: True
print(is_palindrome("hello"))         # Output: False
print(is_palindrome("A man a plan a canal Panama"))  # Output: True
```

4.4.10 Global and local variables with respect to a function

4.4.10.1 Local Variables

- **Definition**: A local variable is defined within a function (or a block of code) and can only be accessed within that function. Once the function finishes executing, local variables are discarded.
- **Scope**: Limited to the function in which they are declared

```python
def example_function():
    x = 10            # x is a local variable
    print("Inside function, x =", x)

example_function()
# Trying to print x outside the function will result in an error:
print(x)  # This will result in an error
```

```
Inside function, x = 10
```

```
NameError: name 'x' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[19], line 7
      5 example_function()
      6 # Trying to print x outside the function will result in an error:
----> 7 print(x)  # NameError: name 'x' is not defined
NameError: name 'x' is not defined
```

4.4.10.2 Global Variables

- **Definition**: A global variable is declared in the main body of the Python file (i.e., at the top level), making it accessible to any function or class in the same module, provided you do not shadow it with a local variable.
- **Scope**: Accessible throughout the entire module (file) after declaration.
- **Best Practice**: Use global variables sparingly, as they can make code harder to debug and maintain.

```python
global_var = 20  # global variable

def show_global_var():
    print("Inside function, global_var =", global_var)

show_global_var()
print("Outside function, global_var =", global_var)
```

```
Inside function, global_var = 20
Outside function, global_var = 20
```

4.4.10.3 Using a Global Variable Inside a Function

By default, if you just read a global variable inside a function, Python will find it in the global scope. However, if you attempt to modify a global variable inside a function without explicitly declaring it `global`, Python will treat that variable as `local`, potentially leading to errors.

```python
# Reading a global variable inside a function
global_var = 20  # global variable

def show_global_var():
    print("Inside function, global_var =", global_var)
```

```
show_global_var()
```

```
Inside function, global_var = 20
```

```
# Changing a global variable inside a function
counter = 0

def increment_counter_wrong():
    counter = counter + 1  # This will cause UnboundLocalError
    print("Counter is now", counter)

increment_counter_wrong()  # UnboundLocalError: local variable 'counter' referenced before as
```

```
UnboundLocalError: cannot access local variable 'counter' where it is not associated with a v
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[22], line 8
      5     counter = counter + 1  # This will cause UnboundLocalError
      6     print("Counter is now", counter)
----> 8 increment_counter_wrong()  # UnboundLocalError: local variable 'counter' referenced b
Cell In[22], line 5, in increment_counter_wrong()
      4 def increment_counter_wrong():
----> 5     counter = counter + 1  # This will cause UnboundLocalError
      6     print("Counter is now", counter)
UnboundLocalError: cannot access local variable 'counter' where it is not associated with a v
```

Why the Error? Python sees `counter = counter + 1` as creating a new **local variable** `counter` on the left, while also trying to read an **uninitialized** local variable `counter` on the right.

4.4.10.4 The `global` keyword

To modify a global variable inside a function, you must use the `global` keyword:

```
counter = 0  # global variable

def increment_counter():
    global counter  # Tell Python we want to use the global 'counter'
    counter += 1
    print("Counter is now", counter)
```

```python
increment_counter()  # Counter is now 1
increment_counter()  # Counter is now 2
print(counter)
```

```
Counter is now 1
Counter is now 2
2
```

When to Use global? * Rarely. Global variables can create tightly coupled code that is prone to bugs. If needed, consider passing variables as arguments or using class-level variables for shared state.

4.4.10.5 Nested Functions and the `nonlocal` Keyword

In Python, you may have **nested functions**—a function defined inside another function. If the inner function needs to modify a variable in the outer (but still non-global) scope, you can use the `nonlocal` keyword.

For example:

```python
def outer_function():
    x = 10

    def inner_function():
        nonlocal x
        x += 5
        print("Inner function, x =", x)

    inner_function()
    print("Outer function, x =", x)

outer_function()
```

```
Inner function, x = 15
Outer function, x = 15
```

`nonlocal` x lets the inner function modify x in the outer_function's scope, rather than creating a new local variable.

4.4.11 Practice exercise 6

Read the following code and answer the following questions:

1. Will the program raise an error?
2. If yes, fix the error and provide the corrected code.
3. What will be the output of the corrected program?

```
message = "Global Message"  # Global variable

def outer():
    msg_outer = "Outer Message"  # Enclosed scope

    def inner():
        message
        msg_outer

        message = "Changed Global Message"
        msg_outer = "Changed Outer Message"
        local_msg = "Local to inner()"
        print("Inside inner()")
        print("Global message =", message)
        print("Enclosed msg_outer =", msg_outer)
        print("Local local_msg =", local_msg)

    inner()
    print("\nInside outer() after inner() call:")
    print("Global message =", message)
    print("Enclosed msg_outer =", msg_outer)
    print("Local local_msg =", local_msg)

outer()

print("\nOutside all functions (global scope):")
print("Global message =", message)
print("Enclosed msg_outer =", msg_outer)
print("Local local_msg =", local_msg)
```

5 Data structures

```
<IPython.core.display.Image object>
```

In Chapter 2, we learned about primitive data types, each of which represents a single value.

In this chapter, we will explore **container data types**, also known as **data structures** or **collection data types** in Python. These data types allow us to store multiple primitive values, such as integers, booleans, and strings, as well as objects of different data types, all within a single structure.

String is one type of container data type, consisting of a sequence of characters. You already learned about strings in previous chapters. In this chapter, we focus on four main container data types in Python: **list**, **tuple**, **set**, and **dictionary**. Each differs in terms of **order** and **immutability**:

- **List**: Ordered and mutable (elements can be changed).
- **Tuple**: Ordered and immutable (elements cannot be changed once defined).
- **Set**: Unordered and mutable (elements can be added or removed, but duplicates are not allowed).
- **Dictionary**: Ordered (as of Python 3.7) and mutable, with key-value pairs for efficient lookups.

We will explore their characteristics, use cases, and differences in detail in this chapter.

5.1 Lists

A List in Python is an **ordered, mutable** (changeable) collection of items. Lists are one of the most versatile data structures in Python.

5.1.1 Creating a List

You can create a list by enclosing items in square brackets, separated by commas:

```
# Creating lists
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed = [42, "hello", True, 3.14]
```

5.1.2 Accessing Elements

Lists are ordered collections with unique indexes for each item. We can access/slice the items in the list using this index number. Python supports both positive and negative indexing, as shown below:

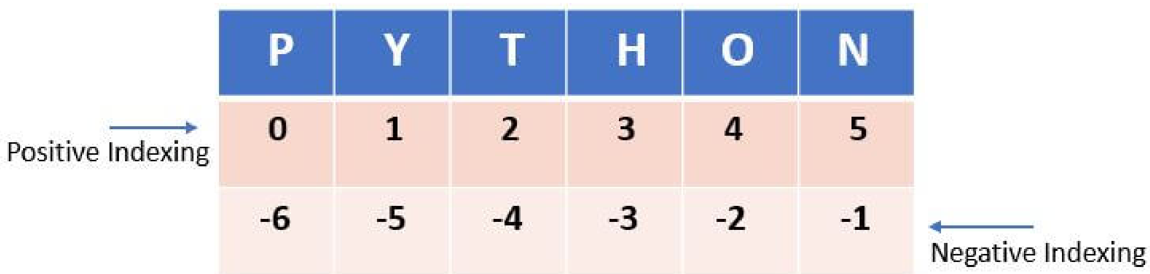


Figure 5.1: image.png

- **Indexing**: Use `[index]` to access a specific item.
- **Slicing**: Use `[start:end:step]` to get a sub-list.

```
my_list = [10, 20, 30, 40, 50]

print(my_list[0])        # 10
print(my_list[-1])        # 50
print(my_list[1:3])      # [20, 30]
print(my_list[2:])       # [30, 40, 50]
print(my_list[-3:-1])    # [30, 40]
print(my_list[::2])      # [10, 30, 50] (step of 2)
print(my_list[::-1])     # [50, 40, 30, 20, 10] (reverse)
```

```
10
50
[20, 30]
[30, 40, 50]
```

```
[30, 40]
[10, 30, 50]
[50, 40, 30, 20, 10]
```

5.1.3 Modifying a List

Lists can be changed after creation. You can add, remove, or replace elements.

5.1.3.1 Adding Items

- `append(item)`: Add item at the end.
- `insert(index, item)`: Insert item at a specific index.
- `extend(iterable)`: Extend the list by appending all items from an iterable (like another list).

```python
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)

fruits.insert(1, "orange")
print(fruits)

fruits.extend(["grape", "mango"])
print(fruits)
```

```
['apple', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry', 'grape', 'mango']
```

Lists can be also concatenated using the `+` operator:

```python
list_example = [5,'hi',4]
list_example = list_example + [None,'7',9]
list_example
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the `+` operator. This is because the `+` operator creates a new list, while the `extend` method adds elements to an existing list. Thus, the `extend` operator is more memory efficient.

5.1.3.2 Removing Items

- `pop([index])`: Removes and returns the item at index. If no index is given, removes the last item.
- `remove(value)`: Removes the first occurrence of value.
- `clear()`: Removes all items from the list, making it empty.

```
numbers = [10, 20, 30, 40, 50]

last_item = numbers.pop()
print(last_item)
print(numbers)

numbers.remove(20)
print(numbers)

numbers.clear()
print(numbers)
```

```
50
[10, 20, 30, 40]
[10, 30, 40]
[]
```

5.1.3.3 Replacing Items

You can directly reassign an element using the index:

```
letters = ["a", "b", "c", "d"]
letters[1] = "z"
print(letters)
```

```
['a', 'z', 'c', 'd']
```

5.1.3.4 Other Useful Methods

- `sort()`: Sorts the list in place.
- `reverse()`: Reverses the list in place.
- `index(value)`: Returns the index of the first occurrence of value.

```python
nums = [4, 1, 3, 2]
nums.sort()
print(nums)

nums.reverse()
print(nums)

idx = nums.index(3)
print(idx)
```

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

```python
help(list.sort)
```

```
Help on method_descriptor:

sort(self, /, *, key=None, reverse=False) unbound builtins.list method
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

```python
help(list.reverse)
```

```
Help on method_descriptor:

reverse(self, /) unbound builtins.list method
    Reverse *IN PLACE*.
```

Note that both `list.sort()` and `list.reverse()` modify the list **in place**, meaning they do not create a new list but instead change the original one directly. To sort or reverse a list without modifying the original list in place, you can use functions that create a new list rather than updating the existing one:

Sorting Without In-Place Modification:

```
original_list = [3, 1, 2]
sorted_list = sorted(original_list)  # returns a new, sorted list
print(original_list)  # [3, 1, 2] (unchanged)
print(sorted_list)    # [1, 2, 3]
```

Reversing Without In-Place Modification:

```
# method 1: use the built-in reversed function
original_list = [3, 1, 2]
reversed_list = list(reversed(original_list))
print(original_list)   # [3, 1, 2] (unchanged)
print(reversed_list)   # [2, 1, 3]
```

```
[3, 1, 2]
[2, 1, 3]
```

```
# use list slicing ([::-1]) to create a reversed copy of the original list
original_list = [3, 1, 2]
reversed_list = original_list[::-1]
print(original_list)   # [3, 1, 2] (unchanged)
print(reversed_list)   # [2, 1, 3]
```

```
[3, 1, 2]
[2, 1, 3]
```

5.1.4 Practice exercise 1

Start by defining a list that contains the elements [8, 9, 10]. Then do the following:

5.1.4.1

Set the second entry (index 1) to 17

5.1.4.2

Add 4, 5, and 6 to the end of the list

5.1.4.3

Remove the first entry from the list

```

```

```
8
```

5.1.4.4

Sort the list

5.1.4.5

Double the list (concatenate the list to itself)

5.1.4.6

Insert 25 at index `3`, then print the final list. Expected Output: `[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]`

5.1.5 List Comprehension

List comprehension is a concise and elegant way to create lists in Python. It provides a shorter syntax to generate lists based on existing iterables while applying conditions or transformations.

Basic Syntax

`[expression for item in iterable if condition]`

Components:

- **Expression**: The value or transformation applied to each item.
- `for` item in iterable: Iterates over the iterable (e.g., list, range, string).
- **Condition (optional)**: Filters items based on a condition.

Examples:

- Simple List Comprehension

```python
# create a list of squares of numbers from 1 to 5
squares = [x**2 for x in range(1, 6)]
print(squares)
```

```
[1, 4, 9, 16, 25]
```

- List Comprehension with Condition

```python
# create a list of even numbers from 1 to 10
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens)  # Output: [2, 4, 6, 8, 10]
```

- List Comprehension with Transformation

```python
# create a list of words with all letters in uppercase
words = ["hello", "world", "python"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words)
```

```
['HELLO', 'WORLD', 'PYTHON']
```

- Nested List Comprehension

```python
# flatten a 2D matrix into a 1D list
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Anything that can be accomplished using list comprehension can also be achieved with traditional Python loops. However, list comprehension often reduces the amount of code, making it more concise and readable. Additionally, it is often faster than equivalent for loops due to Python's optimized implementation.

Comparison with `for` loops

```python
# using a for loop
squares = []
for x in range(1, 6):
    squares.append(x**2)
print(squares)
```

```
[1, 4, 9, 16, 25]
```

```
# using list comprehension

squares = [x**2 for x in range(1, 6)]
print(squares)
```

```
[1, 4, 9, 16, 25]
```

Both achieve the same result, but the list comprehension is more concise.

5.1.6 Practice exercise 2

- Create a list of all odd numbers between 1 and 20.
- Generate a list of the lengths of each word in the list ["apple", "banana", "cherry"].
- Create a list of numbers divisible by both 3 and 5 from 1 to 100.

5.2 Tuples

A Tuple is an **ordered, immutable** (unchangeable) collection of items. Tuples can be thought of as lists that cannot be modified after creation.

5.2.1 Creating a Tuple

Use parentheses or the `tuple()` constructor.

```
my_tuple = (1, 2, 3)
another_tuple = "apple", "banana", "cherry"  # Parentheses are optional
single_item_tuple = ("hello",)               # Note the trailing comma

converted_tuple = tuple([4, 5, 6])           # Using tuple() constructor
```

5.2.2 Accessing Elements

Indexing and slicing work similarly to lists:

```python
t = (10, 20, 30, 40, 50)

print(t[0])    # 10
print(t[1:3])  # (20, 30)
```

```
10
(20, 30)
```

5.2.3 Immutability

Once you create a tuple, you **cannot modify** it:

```python
t = (1, 2, 3)
t[0] = 10
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 2
      1 t = (1, 2, 3)
----> 2 t[0] = 10  # TypeError: 'tuple' object does not support item assignment
TypeError: 'tuple' object does not support item assignment
```

Tuple can be defined without the rounded brackets as well:

If you need to change elements, convert it to a list, modify the list, and convert back to a tuple (though this somewhat defeats the purpose of immutability).

5.2.4 Tuple methods

A couple of useful tuple methods are `count`, which counts the occurences of an element in the tuple and `index`, which returns the position of the first occurance of an element in the tuple:

```python
tuple_example = (2,5,64,7,2,2)
```

```python
tuple_example.count(2)
```

```
3
```

```
tuple_example.index(2)
```

```
0
```

5.2.5 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repetition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

5.2.6 Why Use a Tuple?

A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

Some of the advatages of a tuple over a list are as follows:

- **Data Integrity**: Tuples ensure the data cannot be modified accidentally.
- **Faster**: Tuples can be more memory efficient and faster to iterate over compared to lists.
- **Dictionary Keys**: Tuples can be used as keys in dictionaries (because they are immutable).

```
#Example showing tuples take less storage space than lists for the same elements
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48  bytes
Space taken by list = 72  bytes
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ", tm.time()-tt)

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbers upto 1 million
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = ", tm.time()-tt)
```

```
Time take to retrieve every 2nd element from a list =  0.03211236000061035
Time take to retrieve every 2nd element from a tuple =  0.01900315284729004
```

```
tuple_example = 2, 7, 4
```

We can check the data type of a python object using the *type()* function. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

Note that an element of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-6ceb38adde52> in <module>
----> 1 tuple_example[1] = 8

TypeError: 'tuple' object does not support item assignment
```

The above code results in an error as tuple elements cannot be modified.

Now that we have an idea about tuple, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

5.2.7 Tuple Comprehension?

There is no direct tuple comprehension in Python. However, Python does allow a similar construct that looks like tuple comprehension but actually creates a **generator expression**. If you want to create a tuple using comprehension-like syntax, you can explicitly convert the generator to a tuple.

```
gen = (x**2 for x in range(5))
print(gen)
```

```
<generator object <genexpr> at 0x000001B5240C9BE0>
```

Here, `gen` is a generator, not a tuple. Generators are lazily evaluated, meaning values are computed on demand, making them memory-efficient.

To create a tuple, we can use the `tuple()` function to convert the generator into a tuple explicitly.

```
tup = tuple(x**2 for x in range(5))
print(tup)  # Output: (0, 1, 4, 9, 16)
```

```
(0, 1, 4, 9, 16)
```

Why No Direct Tuple Comprehension?

- Python uses parentheses for both tuple creation and generator expressions. To avoid ambiguity, Python reserves parentheses for generators in this context.
- Converting a generator to a tuple ensures explicit behavior and consistency.

Using List Comprehension for a List of Tuples:

In Python, list comprehension is often used to create a list of tuples because it combines the flexibility of tuple creation with the concise syntax of list comprehension.

Example: Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrt_natural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11, 121), (12, 144), (13, 169), (14
```

5.2.8 Practice exercise 3

Below is a list consisting of responses to the question: "At what age do you think you will marry?" from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+','26','28','30','30','30','pro
```

Use list comprehension to:

5.2.8.1

Remove the elements that are not integers - such as *'probably never'*, *'30+'*, etc. What is the length of the new list?

Hint: The built-in python function of the `str` class - isdigit() may be useful to check if the string contains only digits.

```
exp_marriage_age_num = [x for x in exp_marriage_age if x.isdigit()==True]
print("Length of the new list = ",len(exp_marriage_age_num))
```

```
Length of the new list =  181
```

5.2.8.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

```
exp_marriage_age_capped = [min(int(x),80) for x in exp_marriage_age_num]
print("Mean age when people expect to marry = ", sum(exp_marriage_age_capped)/len(exp_marria
```

```
Mean age when people expect to marry =  28.955801104972377
```

5.2.8.3

Determine the percentage of people who expect to marry at an age of 30 or more.

```
print("Percentage of people who expect to marry at an age of 30 or more =", str(100*sum([1 fo
```

```
Percentage of people who expect to marry at an age of 30 or more = 37.01657458563536 %
```

5.2.9 Practice exercise 4

USA's GDP per capita from 1960 to 2021 is given by the tuple T in the code cell below. The values are arranged in ascending order of the year, i.e., the first value is for 1960, the second value is for 1961, and so on. Print the years in which the GDP per capita of the US increased by more than 10%.

```
T = (3007, 3067, 3244, 3375,3574, 3828, 4146, 4336, 4696, 5032,5234,5609,6094,6726,7226,7801
```

5.2.10 Practice exercise 5

Use the same tuple that was defined in Practice Exercise 4.

Write a function that has two parameters:

1. Year : which indicates the year from which the GDP per capita are available in the second parameter
2. Tuple of GDP per capita's: Tuple consisting of GDP per capita for consecutive years starting from the year mentioned in the first parameter.

The function should return a tuple of length two, where the first element of the tuple is the number of years when the increase in GDP per capita was more than 5%, and the second element is the most recent year in which the GDP per capita increase was more than 5%.

Call the function to find the number of years, and the most recent year in which the GDP per capita increased by more than 5%, since the year 2000. Assign the `number of years` returned by the function to a variable named `num_years`, and assign the most recent year to a variable named `recent_year`. Print the values of `num_years` and `recent_year`.

5.3 Sets

A set is a built-in data type in Python used to store **unordered, unique, and mutable** items. Sets are commonly used for operations like

- **Membership testing**: Quickly check if an item is in a set.
- **Eliminating duplicate entries**: Sets automatically ensure only unique elements are stored.
- **Mathematical set operations**: Supports union (`|`), intersection (`&`), and difference (`-`).

5.3.1 Creating a set

A set can be created using curly braces or the `set()` constructor

```
my_set = {1, 2, 3, 4}
print(my_set)

my_set = set([1, 2, 2, 3, 4])
print(my_set)
type(my_set)

my_empty_set=set()
print(my_empty_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
set()
```

A set can be also created by removing repeated elements from lists.

```
my_list = [1,4,4,4,5,1,2,1,3]
my_set_from_list = list(set(my_list))
print(my_set_from_list)
```

```
[1, 2, 3, 4, 5]
```

5.3.2 Accessing Elements

Since sets are **unordered**, you cannot use indexing or slicing:

```
my_set = {"apple", "banana", "cherry"}
my_set[0]
```

```
TypeError: 'set' object is not subscriptable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[60], line 2
      1 my_set = "apple", "banana", "cherry"
----> 2 my_set[0]
TypeError: 'set' object is not subscriptable
```

Instead, you typically check membership or iterate over all elements:

```
if "apple" in my_set:
    print("Apple is in the set")

for item in my_set:
    print(item)
```

```
Apple is in the set
cherry
banana
apple
```

5.3.3 Adding and Removing Items

- `add(item)`: Adds an item to the set (if it's not already present).
- `update(iterable)`: Adds multiple items (from another set, list, tuple, etc.).
- `remove(item)`: Removes the specified item (raises an error if not found).
- `discard(item)`: Removes the specified item (does not raise an error if not found).

- `pop()`: Removes and returns an arbitrary item from the set.
- `clear()`: Removes all items.

```
# Add an element to a set
print(my_set)
my_set.add(5)
print(my_set)

# Remove an element from a set
my_set.remove(3)
print(my_set)

# Remove an element that doesn't exist
my_set.remove(3)  # This will raise a KeyError

# Remove an element that doesn't exist without raising an error
my_set.discard(3)
```

```
{1, 2, 4, 5}
{1, 2, 4, 5}

KeyError: 3
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[44], line 7
      4 print(my_set)
      6 # Remove an element from a set
----> 7 my_set.remove(3)
      8 print(my_set)
     10 # Remove an element that doesn't exist
KeyError: 3
```

Remove an element using `remove()` (raises an error if the element does not exist); Use `discard()` to remove an element (does not raise an error if the element does not exist):

5.3.4 Mathematical Set Operations

Sets are ideal for tasks involving unions, intersections, and differences. The table below explains these operatoins on sets

Operation	Symbol	Method	Description
Union	|	`set_a.union(set_b)`	Combines all unique elements from two sets.
Intersection	&	`set_a.intersection(set_b)`	Finds common elements between two sets.
Difference	-	`set_a.difference(set_b)`	Finds elements in `set_a` but not in `set_b`.
Symmetric Difference	^	`set_a.symmetric_difference(set_b)`	Finds elements in either set, but not both.

```python
# Examples of Mathematical Operations on Sets
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
print(set_a | set_b)

# Intersection
print(set_a & set_b)

# Difference
print(set_a - set_b)

# Symmetric Difference
print(set_a ^ set_b)
```

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
{1, 2, 4, 5}
```

5.3.5 Set Comprehension

We can do set comprehensions just like list comprehensions

```python
# set comprehension
my_set = {x for x in 'hello'}
print(my_set)
type(my_set)
```

```
{'h', 'o', 'e', 'l'}
```

```
set
```

5.4 Dictionary

A Dictionary in Python is an **mutable** collection of **key-value** pairs. It's used when you need to associate a specific value with a key and quickly access that value by using the key.

A dictionary in Python consists of key-value pairs, where both keys and values are Python objects. **While values can be of any data type**, **keys must be immutable objects**, such as strings, integers, or tuples. For example, a list can be used as a value in a dictionary, but not as a key, because lists are mutable and their elements can be changed.

Ordered Dictionaries: As of Python 3.7, the language specification guarantees that dictionaries maintain insertion order. This means you can reliably depend on the order in which keys were inserted when iterating over or converting the dictionary. Prior to Python 3.7, this behavior was not officially guaranteed (though in CPython 3.6, it happened to work that way in practice).

5.4.1 Creating a dictionary

Use braces `{}` or the `dict()` constructor.

```python
# Using braces
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
print(person)

# Using dict() constructor
car = dict(brand="Tesla", model="Model 3", year=2023)
print(car)


# Empty dictionary
empty_dict = {}
print(empty_dict)
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'brand': 'Tesla', 'model': 'Model 3', 'year': 2023}
{}
```

5.4.2 Accessing and Modifying Values

Access values by their keys using square bracket notation `[key]` or the `.get(key)` method.

```python
person = {"name": "Alice", "age": 30, "city": "New York"}

# Access value
print(person["name"])
print(person.get("name"))

# Modify value
person["age"] = 31
print(person)

# Add new key-value pair
person["job"] = "Engineer"
print(person)
```

```
Alice
Alice
{'name': 'Alice', 'age': 31, 'city': 'New York'}
{'name': 'Alice', 'age': 31, 'city': 'New York', 'job': 'Engineer'}
```

5.4.3 Removing Keys

- `pop(key)`: Removes and returns the value for key.
- `del dictionary[key]`: Removes the key-value pair.
- `popitem()`: Removes and returns an arbitrary key-value pair (in Python 3.7+, it removes the last inserted item).
- `clear()`: Removes all items.

```python
person = {"name": "Alice", "age": 30, "city": "New York"}

age = person.pop("age")
print(age)          # 30
print(person)       # {"name": "Alice", "city": "New York"}
```

```python
del person["name"]
print(person)      # {"city": "New York"}

person.popitem()
print(person)      # {} (now empty)

person.clear()
print(person)      # {}
```

```
30
{'name': 'Alice', 'city': 'New York'}
{'city': 'New York'}
{}
{}
```

5.4.4 Iterating over elements of a dictionary

Use the following dictionary methods to retrieve all key and values at once:

- `keys()`: Returns the list of all keys present in the dictionary.
- `values()`: Returns the list of all values present in the dictionary
- `items()`: Returns all the items present in the dictionary. Each item will be inside a tuple as a key-value pair.

```python
fruits = {"apple": 3, "banana": 5, "cherry": 2}
print(fruits.keys())
print(fruits.values())

for key,value in fruits.items():
    print("The Head of State of",key,"is",value)
```

```
dict_keys(['apple', 'banana', 'cherry'])
dict_values([3, 5, 2])
The Head of State of apple is 3
The Head of State of banana is 5
The Head of State of cherry is 2
```

5.4.5 Practice exercise 6

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary `D` given in the code cell below.

Find:

1. The GDP per capita in 2015
2. The GDP per capita of 2014 is missing. Update the dictionary to include the GDP per capita of 2014 as the average of the GDP per capita of 2013 and 2015.
3. Impute the GDP per capita of other missing years in the same manner as in (2), i.e., as the average GDP per capita of the previous year and the next year. Note that the GDP per capita is not missing for any two consecutive years.
4. Print the years and the imputed GDP per capita for the years having a missing value of GDP per capita in (3).

```
D = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':3574,'1965':3828,'1966':4146,'196
```

Solution:

```
print("GDP per capita in 2015 =", D['2015'])
D['2014'] = (D['2013']+D['2015'])/2

#Iterating over all years from 1960 to 2021
for i in range(1960,2021):

    #Imputing the GDP of the year if it is missing
    if str(i) not in D.keys():
        D[str(i)] = (D[str(i-1)]+D[str(i+1)])/2
        print("Imputed GDP per capita for the year",i,"is $",D[str(i)])
```

```
GDP per capita in 2015 = 56763
Imputed GDP per capita for the year 1969 is $ 4965.0
Imputed GDP per capita for the year 1977 is $ 9578.5
Imputed GDP per capita for the year 1999 is $ 34592.0
```

5.4.6 Practice exercise 7

The object `deck` defined below corresponds to a deck of cards. Estimate the probablity that a five card hand will be a flush, as follows:

1. Write a function that accepts a hand of 5 cards as argument, and returns whether the hand is a flush or not.
2. Randomly pull a hand of 5 cards from the deck. Call the function developed in (1) to determine if the hand is a flush.
3. Repeat (2) 10,000 times.
4. Estimate the probability of the hand being a flush from the results of the 10,000 simulations.

You may use the function shuffle() from the `random` library to shuffle the deck everytime before pulling a hand of 5 cards.

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

Solution:

```
import random as rm

#Function to check if a 5-card hand is a flush
def chck_flush(hands):

    #Assuming that the hand is a flush, before checking the cards
    yes_flush =1

    #Storing the suit of the first card in 'first_suit'
    first_suit = hands[0]['suit']

    #Iterating over the remaining 4 cards of the hand
    for j in range(1,len(hands)):

        #If the suit of any of the cards does not match the suit of the first card, the hand
        if first_suit!=hands[j]['suit']:
            yes_flush = 0;

            #As soon as a card with a different suit is found, the hand is not a flush and th
            break;
    return yes_flush

flush=0
for i in range(10000):
```

```
    #Shuffling the deck
    rm.shuffle(deck)

    #Picking out the first 5 cards of the deck as a hand and checking if they are a flush
    #If the hand is a flush it is counted
    flush=flush+chck_flush(deck[0:5])

print("Probability of obtaining a flush=", 100*(flush/10000),"%")
```

```
Probability of obtaining a flush= 0.26 %
```

5.4.7 Practice exercise 8

The code cell below defines an object having the nutrition information of drinks in starbucks. Assume that the manner in which the information is structured is consistent throughout the object.

```
starbucks_drinks_nutrition={'Cool Lime Starbucks Refreshers Beverage': [{'Nutrition_type':'Calori
```

Use the object above to answer the following questions:

5.4.8

What is the datatype of the object?

5.4.8.1

If the object in (1) is a dictionary, what is the datatype of the values of the dictionary?

5.4.8.2

If the object in (1) is a dictionary, what is the datatype of the elements within the values of the dictionary?

5.4.8.3

How many calories are there in `Iced Coffee`?

5.4.8.4

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

5.4.8.5

Which drink(s) have a fat content of more than 10g, and what is their fat content?

5.5 Choosing the Right Data Structure

1. **List**
   - *Ordered* and *Mutable*
   - Best for collections of related items you need to modify, iterate, or reorder frequently.
2. **Tuple**
   - *Ordered* and *Immutable*
   - Ideal for data that shouldn't change or for use as dictionary keys (since they are hashable).
3. **Dictionary**
   - *Unordered* (maintains insertion order in Python 3.7+) and *Mutable*
   - Perfect for key-value lookups, fast data retrieval based on unique keys, and clearly organizing named data.
4. **Set**
   - *Unordered* and *Mutable*
   - Automatically enforces *unique* elements. Great for membership testing (e.g., `in` checks) and set operations like union, intersection, etc.

5.6 Final Thoughts

- **Lists** are your go-to when you need an adjustable sequence of ordered items.

- **Tuples** provide a way to store data in an immutable sequence, ensuring it remains unchanged.

- **Dictionaries** let you organize data into key-value pairs for quick lookups and clearer data structures.

- **Sets** focus on uniqueness and membership operations, which can greatly optimize tasks like deduplication and intersection.

Choose the right data structure based on your needs to write concise, efficient, and easy-to-maintain code.

6 Python Iterables

In Python, an **iterable** is an object capable of returning its members one at a time. Common examples of iterables include strings, lists, tuples, sets, dictionaries, and even custom objects that implement specific methods. Iterables are fundamental in Python for loops, comprehensions, and many built-in functions.

6.1 What are Python Iterables

6.1.1 What Makes an Object Iterable?

An object is considered iterable if it implements the `__iter__()` method, which returns an iterator object, or the `__getitem__()` method, which allows it to be accessed sequentially.

```python
my_list = [1, 2, 3, 4, 5]
help(my_list.__iter__)
```

```
Help on method-wrapper:

__iter__() unbound builtins.list method
    Implement iter(self).
```

```python
help(my_list.__getitem__)
```

```
Help on built-in function __getitem__:

__getitem__(index, /) method of builtins.list instance
    Return self[index].
```

6.1.2 Examples of Common Iterables

6.1.2.1 Strings: Iterate over characters

```
for char in "Python":
    print(char, end=" ")
```

```
P y t h o n
```

6.1.2.2 Lists: Iterate over elements.

```
for num in [1, 2, 3]:
    print(num, end=" ")
```

```
1 2 3
```

6.1.2.3 Tuples: Iterate like lists.

```
for item in (4, 5, 6):
    print(item, end=" ")
```

```
4 5 6
```

6.1.2.4 Sets: Iterate over unique elements.

```
for elem in {1, 2, 3}:
    print(elem, end=" ")
```

```
1 2 3
```

6.1.2.5 Dictionaries: Iterate over keys by default.

```python
my_dict = {"a": 1, "b": 2}
for key in my_dict:
    print(key, end=" ")
```

```
a b
```

6.1.2.6 Checking if an Object is Iterable

You can check if an object is iterable using the `collections.abc.Iterable` class:

```python
from collections.abc import Iterable

print(isinstance([1, 2, 3], Iterable))
print(isinstance(123, Iterable))
```

```
True
False
```

6.2 Iterables Unpacking

Python supports unpacking for iterables, allowing you to assign elements of an iterable to variables in a concise and readable way. Unpacking is a powerful feature that works with sequences like lists, tuples, and other iterable objects.

6.2.1 Basic Unpacking

You can unpack the elements of an iterable into separate variables:

```python
# Unpacking a list
numbers = [1, 2, 3]
a, b, c = numbers
print(a, b, c)  # Output: 1 2 3
```

```
1 2 3
```

Note: The number of variables must match the number of elements in the iterable, or Python will raise a `ValueError`.

6.2.2 Extended (Star) Unpacking

Using the * operator, you can unpack multiple elements into a single variable, capturing the remaining elements as a list:

```
numbers = [1, 2, 3, 4, 5]

a, *b, c = numbers
print(a)  # Output: 1
print(b)  # Output: [2, 3, 4]
print(c)  # Output: 5
```

```
1
[2, 3, 4]
5
```

Here, `b` (with `*`) captures all the middle elements as a list.

6.2.3 Unpacking with Functions

You can use unpacking to pass iterable elements as arguments to functions:

```
# unpacking a list or tuple
def add(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
result = add(*numbers)
print(result)  # Output: 6
```

```
6
```

```
# unpacking a dictionary
def add(a=0, b=0, c=0):
    return a + b + c

numbers = {"a": 1, "b": 2, "c": 3}
result = add(**numbers)
print(result)  # Output: 6
```

6

If we are interested in retrieving only some values of the tuple, the expression *_ can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```
x,*_,y,z  = (4.5, "this is a string", (("Nested tuple",5)),"99",99)
```

```
x, y , z
```

```
(4.5, '99', 99)
```

6.3 Built-in Functions for Iterable in Python

Python provides a variety of built-in functions to operate on iterables, making it easy to manipulate, process, and analyze collections like lists, tuples, strings, sets, and dictionaries. Below is a list of commonly used built-in functions specifically designed for iterables.

6.3.1 General Functions

Function	Description	Example
`len()`	Returns the number of elements in an iterable.	`len([1, 2, 3])` → `3`
`min()`	Returns the smallest element in an iterable.	`min([3, 1, 4])` → `1`
`max()`	Returns the largest element in an iterable.	`max([3, 1, 4])` → `4`
`sum()`	Returns the sum of elements in an iterable (numeric types only).	`sum([1, 2, 3])` → `6`
`sorted()`	Returns a sorted list from an iterable (does **not** modify the original).	`sorted([3, 1, 2])` → `[1, 2, 3]`
`reversed()`	Returns an iterator that accesses the elements of an iterable in reverse.	`list(reversed([1, 2, 3]))` → `[3, 2, 1]`
`enumerate()`	Returns an iterator of tuples containing indices and elements of the iterable.	`list(enumerate(['a', 'b', 'c']))` → `[(0, 'a'), (1, 'b'), (2, 'c')]`
`all()`	Returns `True` if all elements of the iterable are true (or if empty).	`all([True, 1, 'a'])` → `True`
`any()`	Returns `True` if **any** element of the iterable is true.	`any([False, 0, 'b'])` → `True`

```
sorted(["apple", "orange", "banana"])
```

```
['apple', 'banana', 'orange']
```

```
max(["apple", "orange", "banana"])
```

```
'orange'
```

6.3.2 sorted()

The `sorted()` function in Python is used to return a new sorted list from an iterable. It is a versatile and powerful tool for sorting data in ascending or descending order, with the ability to customize sorting behavior using a key function.

```
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 1, 3, 4, 5]
```

```
[1, 1, 3, 4, 5]
```

```
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers, reverse=True)
print(sorted_numbers)  # Output: [5, 4, 3, 1, 1]
```

```
[5, 4, 3, 1, 1]
```

```
word = "python"
sorted_chars = sorted(word)
print(sorted_chars)  # Output: ['h', 'n', 'o', 'p', 't', 'y']
```

```
['h', 'n', 'o', 'p', 't', 'y']
```

The `key` parameter allows customization of sorting logic by applying a function to each element before comparison.

```python
words = ["banana", "apple", "cherry"]
sorted_words = sorted(words, key=len)
print(sorted_words)  # Output: ['apple', 'banana', 'cherry']
```

```
['apple', 'banana', 'cherry']
```

```python
numbers = [-5, 3, -1, 7]
sorted_numbers = sorted(numbers, key=abs)
print(sorted_numbers)  # Output: [-1, 3, -5, 7]
```

```
[-1, 3, -5, 7]
```

6.3.2.1 Difference Between `sorted()` and `list.sort()`

Feature	`sorted()`	`list.sort()`
Return Value	Returns a **new** sorted list	Modifies the list in place and returns `None`.
Input Type	Works with any iterable (e.g., lists, tuples, strings).	Works **only** with lists.
Usage	`sorted(iterable)`	`list.sort()`

Example:

```python
my_list = [3, 1, 2]

# Using sorted()
new_list = sorted(my_list)
print("Original List:", my_list)  # [3, 1, 2]
print("New Sorted List:", new_list)  # [1, 2, 3]

# Using list.sort()
my_list.sort()
print("List after list.sort():", my_list)  # [1, 2, 3]
```

```
Original List: [3, 1, 2]
New Sorted List: [1, 2, 3]
List after list.sort(): [1, 2, 3]
```

6.3.3 enumerate()

The `enumerate()` function adds a counter to an iterable and returns it as an **enumerate object**, which can be iterated over to get both the **index** and the **value** of each element in the iterable.

6.3.3.1 Syntax

```
enumerate(iterable, start=0)
```

```python
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
```

```python
# change the start index
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits, start=1):
    print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 1, Fruit: apple
Index: 2, Fruit: banana
Index: 3, Fruit: cherry
```

```python
# using enumerate() with a list comprehension
fruits = ["apple", "banana", "cherry"]
indexed_fruits = [(index, fruit) for index, fruit in enumerate(fruits, start=1)]
print(indexed_fruits)
```

```
[(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

```python
# Working with Strings
word = "python"

for index, char in enumerate(word):
    print(f"Index: {index}, Character: {char}")
```

```
Index: 0, Character: p
Index: 1, Character: y
Index: 2, Character: t
Index: 3, Character: h
Index: 4, Character: o
Index: 5, Character: n
```

6.3.4 `zip()`

The `zip()` function is a built-in Python function that combines two or more iterables (e.g., lists, tuples, strings) into a single iterator of tuples. It is commonly used to pair elements from multiple iterables based on their positions.

```python
# combining two lists
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

```python
# Creating dictionaries from two list
keys = ['name', 'age', 'city']
values = ['Alice', 30, 'New York']

dictionary = dict(zip(keys, values))
print(dictionary)
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

```python
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]

for name, score in zip(names, scores):
    print(f"{name} scored {score}")
```

```
Alice scored 85
Bob scored 90
Charlie scored 95
```

6.3.5 unzipping

You can "unzip" a zipped object using the `zip(*zipped)` syntax:

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

list1, list2 = list(unzipped)
print(list1)  # Output: [1, 2, 3]
print(list2)  # Output: ['a', 'b', 'c']
```

```
(1, 2, 3)
('a', 'b', 'c')
```

```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

7 Object-Oriented Programming

In Python, everything is an object, which makes it an object-oriented programming language.

Object-oriented programming is the one in which a program is based on *objects*. An object is an independent entity within the program and can cooperatively work with other objects. A program can be made up of one or more objects, which can leverage the functionality and information contained in other objects.

7.1 Object

An object consists of two items:

1. **Attributes** - Attributes are the data stored within the object.
2. **Methods** - Methods are the functions defined within the object. Methods can use the object attributes *(or data stored within the object)* as well as accept additional data as arguments.

We have already seen several in-built python objects such as string objects, integer objects, float objects, list objects, tuple objects and dictionary objects, in previous chapters. Each of these objects have attributes and methods associated with them.

For example, consider a *integer* object named as `integer_example`.

```
integer_example = 5
```

The attributes and methods of this *integer* object can be seen by putting a `.` next to its name, and pressing the *tab* key. A dropdown menu consisting of the attributes and methods will appear as shown below.

```
<IPython.core.display.Image object>
```

A list of all attributes and methods associated with an object can be obtained with the `dir()` function. Ignore the ones with underscores - these are used by Python itself. The rest of them can be used to perform operations.

```
#This code is not executed to avoid printing a long list
dir(integer_example)
```

For example, an attribute of `integer_example` is real, which contains the real part of the number:

```
integer_example.real
```

```
5
```

A example of a method of `integer_example` is `as_integer_ratio()`, which returns a tuple containing the numerator and denominator of the integer when it is expressed as a fraction.

```
integer_example.as_integer_ratio()
```

```
(5, 1)
```

Note that attributes do not have any parenthesis after them as they are just data, and cannot accept arguments. On the other hand methods have parenthesis after them as they are functions that may or may not have arguments.

7.2 Class

A *class* is a template for objects. It contains the attributes and methods associated with the object of the class. As an analaogy, the *class* Cat will consist of characterisitcs (or *attributes*) shared by all cats such as breed, fur color, etc., as well as capability to perform functions (or *methods*) such as run, meow, etc.

Please see the python documentation for class.

Instance: An *instance* is a specific realization of the object of a particular class. Continuing with the Cat analaogy of a class, a particular cat is an *instance* of the class Cat. Similarly, in the example above, the object `integer_example` is an instance of the class *integer*. The words *object* and *instance* are often used interchangeably.

Creating an *instance* of a class is called **Instantiation**.

7.2.1 Creating your own class

Until now we saw examples of in-built Python classes, such as *integer*, *List*, etc. Now, we'll learn to create our own class that serves our purpose.

Below is a toy example of a class.

```
class ToyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        return self.x + self.y

    def multiply(self):
        return self.x*self.y
```

We'll use the example above to explain the following terms:

- **The `class` statement:** We use the `class` statement to create a class. The Python style guide recommends to use CamelCase for class names.
- **The constructor (or the `__init__()` method):** A class typically has a method called `__init__`. This method is called a constructor and is automatically called when an object or instance of the class is created. The constructor initializes the attributes of the class. In the above example, the constructor accepts two values as arguments, and initializes its attributes `x` and `y` with those values.
- **The `self` argument:** This is the first argument to every method in the class. Whenever the class refers to one of its attributes or methods, it must precede them by `self`. The purpose of `self` is to distinguish the class's attributes and methods from other variables and functions in the program.

The class `ToyClass` consists of two attributes `x` and `y`, a constructor `__init__()`, and two methods `add()` and `multiply()`.

To create an *object* or *instance* of the class `ToyClass`, we'll use the class name with the values to be passed as argument to the constructor for initializing the *object* / *instance*.

```
toy_instance = ToyClass(6,12)
```

The `x` *attribute* of the class `ToyClass` can be called using the `.` operator with the object name:

```
toy_instance.x
```

```
6
```

To use the `multiply()` *method* of the class `ToyClass`, we'll use the `.` operator with the object name:

```
toy_instance.multiply()
```

```
72
```

7.2.2 Example: A class that analyzes a string

Let us create a class that analyzes a string.

```
class AnalyzeString:

    #Constructor
    def __init__(self, s):
        s = s.lower()
        self.words = s.split()

    #This method counts the numebr of words
    def number_of_words(self):
        return (len(self.words))

    #This method counts the number of words starting with the string s
    def starts_with(self,s):
        return len([x for x in self.words if x[:len(s)]==s])

    #This method counts the number of words of length n
    def words_with_length(self,n):
        return len([x for x in self.words if len(x)==n])

    #This method returns the frequency of the word w
    def word_frequency(self,w):
        return self.words.count(w)
```

Let us create an instance of the class `AnalyzeString()` to analyze a sentence.

```python
#Defining a string
sentence = 'This sentence in an example of a string that we will analyse using a class we ha
```

```python
#Creating an instance of class AnalyzeString()
sentence_analysis = AnalyzeString(sentence)
```

```python
#The attribute 'word' contains the list of words in the sentence
sentence_analysis.words
```

```
['this',
 'sentence',
 'in',
 'an',
 'example',
 'of',
 'a',
 'string',
 'that',
 'we',
 'will',
 'analyse',
 'using',
 'a',
 'class',
 'we',
 'have',
 'defined']
```

```python
#The method 'word_frequncy()' provides the frequency of a word in the sentence
sentnce_analysis.word_frequency('we')
```

```
2
```

```python
#The method 'starts_with()' provides the frequency of number of words starting with a partic
sentnce_analysis.starts_with('th')
```

```
2
```

7.2.3 Practice exercise 1

Write a class called `PasswordManager`. The class should have a list called `old_passwords` that holds all of the user's past passwords. The last item of the list is the user's current password. There should be a method called `get_password` that returns the current password and a method called `set_password` that sets the user's password. The `set_password` method should only change the password if the attempted password is different from all the user's past passwords. It should either print '*Password changed successfully!*', or '*Old password cannot be reused, try again.*' Finally, create a method called `is_correct` that receives a string and returns a boolean `True` or `False` depending on whether the string is equal to the current password or not.

To initialize the object of the class, use the list below.

After defining the class:

1. Check the attribute `old_passwords`
2. Check the method `get_password()`
3. Try re-setting the password to 'ibiza1972', and then check the current password.
4. Try re-setting the password to 'oktoberfest2022', and then check the current password.
5. Check the `is_correct()` method

```
past_passwords = ['titanic1911','ibiza1972','montecarlo799']
```

```
class PasswordManager:
    def __init__(self,past_passwords):
        self.old_passwords = past_passwords

    def get_password(self):
        return self.old_passwords[len(self.old_passwords)-1]

    def set_password(self,new_password):
        if new_password not in self.old_passwords:
            self.old_passwords.append(new_password)
            print("Password changed!")
        else:
            print("Old password cannot be reused, try again.")

    def is_correct(self,password):
        if password == self.old_passwords[len(self.old_passwords)-1]:
            return True
```

```
        return False

passwd = PasswordManager(past_passwords)
```

7.3 Inheritance

In object-oriented programming there is a concept called *inheritance* where we can create a new class that builds off of another class. The new class gets all of the variables and methods of the class it is inheriting from (called the base class). It can then define additional variables and methods that are not present in the base class, and it can also override some of the methods of the base class. That is, it can rewrite them to suit its own purposes. Here is a simple example:

```
class Parent:

    def __init__(self, a, b):
        self.a = a

    def method1(self):
        return self.a+' should study!'

    def method2(self):
        return self.a+' does not study enough '

class Child(Parent):

    def __init__(self, a,b):
        self.a = a
        self.b = b

    def method1(self):
        return self.a+' should play with ' + self.b

    def method3(self):
        return self.a + ' does not play enough'
```

Note that when inheriting from a class, we indicate the parent class in parentheses in the `class` statement.

We see that `method1` is present in both the `Parent` and `Child` classes, while `method2` is only present in the `Parent` class. Let us understand how does the `Child` class use the methods of the Parent class, and what happens if a method with the same name is present in both the parent and child classes.

```
p = Parent('Sam', 'John')
c = Child('Sam','Pam')
print('Parent method 1: ', p.method1())
print('Parent method 2: ', p.method2())
print()
print('Child method 1: ', c.method1())
print('Child method 2: ', c.method2())
print('Child method 3: ', c.method3())
```

```
Parent method 1:  Sam should study!
Parent method 2:  Sam does not study enough

Child method 1:  Sam should play with Pam
Child method 2:  Sam does not study enough
Child method 3:  Sam does not play enough
```

We see in the example above that the child has overridden the parent's `method1`. The child has inherited the parent's `method2`, so it can use it without having to define it. The child also adds some features to the parent class, namely a new variable `b` and a new method, `method3`.

7.3.1 Practice exercise 2

Define a class that inherits the in-built Python class `list`, and adds a new method to the class called `nunique()` which returns the number of unique elements in the list.

Define the following list as an object of the class you created. Then:

1. Find the number of unique elements in the object using the method `nunique()` of the inherited class.
2. Check if the `pop()` method of the parent class works to pop an element out of the object.

```
list_ex = [1,2,5,3,6,5,5,5,12]
```

```
class list_v2(list):
    def nuinque(self):
        unique_elements = []
        for x in self:
            if x not in unique_elements:
```

```
                unique_elements.append(x)
        return len(unique_elements)

list_ex = list_v2(list_ex)
print("Number of unique elements = ", list_ex.nuinque())
print("Checking the pop() method, the popped out element is", list_ex.pop())
```

```
Number of unique elements =  6
Checking the pop() method, the popped out element is 12
```

7.3.2 Practice exercise 3

Define a class named `PasswordManagerUpdated` that inherits the class `PasswordManager` defined in Practice exercise 1. The class `PasswordManagerUpdated` should have two methods, other than the *constructor*:

1. The method `set_password()` that sets a new password. The new password must only be accepted if it does not have any punctuations in it, and if it is not the same as one of the old passwords. If the new password is not acceptable, then one of the appropriate messages should be printed - (a) *Cannot have punctuation in password, try again*, or (b) *Old password cannot be reused, try again*.
2. The method `suggest_password()` that randomly sets and returns a password as a string comprising of 15 randomly chosen letters. Letters may be repeated as well.

```
from string import punctuation
import random as rm
import string as st
class PasswordManager_updated(PasswordManager):
    def __init__(self,past_passwords):
        self.old_passwords = past_passwords.copy()

    def set_password(self,new_password):
        if new_password not in self.old_passwords:
            for punc in punctuation:
                if punc in new_password:
                    print("Cannot have punctuation in password, try again")
                    break
            self.old_passwords.append(new_password)
            print("password changed!")
        else:
            print("Old password cannot be reused, try again.")
```

```python
    def suggest_password(self):
        suggested_passwd = ''
        for i in range(15):
            suggested_passwd = suggested_passwd + st.ascii_lowercase[rm.randint(0,26)]
        self.old_passwords.append(suggested_passwd)
        return suggested_passwd
```

A Assignment 1

STAT 201

A.0.1 Instructions

1. **Write your name** on the assignment.
2. Write your code in the *Code* cells of the **template provided** to write solutions for the assignment. **Do not open a new notebook**, and work from scratch. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity. Do not use AI to solve the problems.
5. If your document is not clean and organized, you can lose up to 2 points:
   - Must be an HTML file rendered using Quarto.
   - There aren't excessively long outputs of extraneous information (e.g. no printouts of unnecessary results without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.). There is no piece of unnecessary / redundant code, and no unnecessary / redundant text
   - The code follows the python style guide for naming variables, spaces, indentation, etc.
   - The code should be commented and clearly written with intuitive variable names. For example, use variable names such as number_input, factor, hours, instead of a,b,xyz, etc.

A.1 Question 1 (4 points)

a) Create a a variable called `var_float` that contains a decimal number.

b) Store a sentence as `var_sent` that reads exactly as follows: "The square of `{}` is `{}`." Where the first `{}` is your `var_float` and the second `{}` is the square of that variable. Print your sentence.

c) Print the output of using the **`count` method** to determine how many spaces are in `var_sent`.

d) Round your `var_float` to 0 decimal places and convert to an integer. Store this as `var_int` and print the `type` to verify this was done correctly.

A.2 Question 2 (3 points)

Have a user input 2 Booleans (hint: must be type bool). In a **single print line**, using only `and`, `or`, `not` functions, have the output return True if both variables are the same, and False is they are different.

Clarification: 1) cannot use conditional if statements 2) this must be capable of printing the correct output for any possible booleans the user could enter, not just the one example that your html will show.

A.3 Question 3 (6 points)

At Northwestern, email addresses are classified as follows:

- **Student email addresses** end with `@u.northwestern.edu`.
- **Professor email addresses** end with `@northwestern.edu` (but not `@u.northwestern.edu`).

Write a Python program that: 1. Asks the user how many email addresses they will enter. 2. Prompts the user to input each email address. 3. After all email addresses are entered: - Print all professor email addresses under the heading `"Professor Emails:"`. - Print all student email addresses under the heading `"Student Emails:"`. - If no professor or student emails were entered, print `"None"` under the respective heading.

A.3.1 Requirements:

- Do not use lists or other advanced data structures, since we have not covered them yet.
- Use only basic string operations and loops.
- **The program must handle all cases**, regardless of uppercase or lowercase in the email addresses.
- **Trim any leading or trailing whitespace** in the user input before classifying the email.

Example Run:

How many email addresses will you be entering? 3 Enter an email address: lshi@northwestern.edu Enter an email address: jackyu@u.northwestern.edu Enter an email address: Alexa@u.northwestern.edu

Output:

Professor Emails: lshi@northwestern.edu

Student Emails: jackyu@u.northwestern.edu alexa@u.northwestern.edu

A.4 Question 4 (3 points)

Write a tip calculator program that asks the user for the price of the meal and the percent tip they want to leave. Then print a sentence that displays both the tip amount and total bill. Example if meal price is 25 dollars and tip is 15 percent:

```
Your tip amount is $3.75 and your total bill is $28.75.
```

A.5 Question 5 (3 points)

Write a program that asks the user for a number of seconds and prints out how many minutes and seconds that is. Example:

```
200 seconds is 3 minutes and 20 seconds.
```

Use only two lines of code for this question: one line for the input and one line for the print.

A.6 Question 6 (4 points)

Write a program that asks the user to enter two numbers. Have the program return one of the following messages depending on which criteria is met.

"`num1` is greater than `num2`"; "`num 1` is less than `num2`"; "`num1` is equal to `num2`"; where `num1` and `num2` are the user inputed values.

Show the output of the program with any two numbers of your choice.

A.7 Question 7 (4 points)

a) Use a **single if-elif-else** statement to print the smallest of 3 user defined numbers. Show the output of the program with any three numbers of your choice.

b) Use a **nested** conditional statement to print the smallest of 3 user defined numbers. Show the output of the program with any three numbers of your choice.

A.8 Question 8 (6 points)

Write a program that asks the user to enter either rock, paper, or scissors. Use a conditional statement to determine if the user wins, loses, or ties the computer at the game "Rock, Paper, Scissors". Note: rock beats scissors; scissors beats paper; paper beats rock

Print a meaningful sentence that includes the winner, the computer's choice, and the user's choice.

- Handle case sensitivity (example: if the user enters Rock, it will still run).
- If the user enters a word other than one of the choices, print "Invalid choice.".
- Show the output of the program when the user enters Rock (capitalized)

```python
# starter code to generate a random choice of rock, paper, scissors
import random as rm
comp_choice = rm.choice(['rock', 'paper', 'scissors'])

# your solution in this code chunk below
```

A.9 Bonus (6 points)

For all questions in this assignment that involve accepting user input:

- Use a **`try-except`** block to handle cases where the user enters invalid input.
- Implement a **loop** to repeatedly prompt the user for input until they provide a valid value, allowing the program to proceed.

B Assignment 2

STAT 201

B.0.1 Instructions

1. **Write your name** on the assignment.
2. Write your code in the *Code* cells of the **template provided** to write solutions for the assignment. **Do not open a new notebook**, and work from scratch. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity. Do not use AI to solve the problems.
5. There are 2 points for cleanliness and organization. The breakdown is as follows:

   - Must be an HTML file rendered using Quarto.
   - There aren't excessively long outputs of extraneous information (e.g. no printouts of unnecessary results without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.). There is no piece of unnecessary / redundant code, and no unnecessary / redundant text
   - The code follows the python style guide for naming variables, spaces, indentation, etc.
   - The code should be commented and clearly written with intuitive variable names. For example, use variable names such as number_input, factor, hours, instead of a,b,xyz, etc.

B.0.2 Question 1 (3 points)

a) Write a program that counts down from 5 and then prints the message “Go”.

b) Write a program that asks the user for an integer, then counts down from that integer and then prints the message “Go”. Use a try-except to ensure the user entered an integer. If it is not an integer, return the message “You must enter an integer”. Show the output of the program if the user inputs `3`.

c) Copy and modify part b to allow the user to keep entering input until they correctly enter an integer (ie: don’t end after the except). Show the output of the program if the user first inputs `8.9` and then inputs `7`.

B.0.3 Question 2 (2 points)

Print a star formation as follows:

```
*
* *
* * *
* * * *
```

Ask the user to input the height of the formation first. For example, 4 should return the formation above. Show the output of the program if the user inputs `5`.

B.0.4 Question 3 (6 points)

a) Write a program where you play 3 “Rock, Paper, Scissors” games against the computer. Simulate a random choice for the computer each game and ask the user to enter either rock, paper, or scissors. Print the number of times the user won out of 3 games. Show the output of the program after playing the game.

Recall: rock beats scissors; scissors beats paper; paper beats rock.

Example:

```
"You won <num_win> out of 3 games."
```

```
# import the random function and name the alias rm

# code to generate a random choice of rock, paper, scissors for the computer
comp_choice = rm.choice(['rock', 'paper', 'scissors'])
```

b) Write a program where you play "Rock, Paper, Scissors" games against the computer until you have won 3 games. Simulate a random choice for the computer each game and ask the user to enter either rock, paper, or scissors. Print the number of games it took to win 3 times. Show the output of the program after playing the game.

Example:

```
"You won 3 out of <num_games> games."
```

B.0.5 Question 4 (6 points)

a) Write a program (loop) that prints all the factors of a positive integer input by the user. A factor is any positive integer that divides the number and leaves no remainder. Show the output of the program if the user inputs `24`.

Example: The factors of 8 are 1, 2, 4, 8.

b) Write a **function** called `number_of_factors` that takes an integer and returns how many factors the number has. Run your function with `24` to check if it works.

B.0.6 Question 5 (6 points)

a) Write a program (loop) that identifies whether a positive integer input by the user is prime or not. A prime number is a number whose only divisors are 1 and itself. Show the output when the program is used to check if `89` is prime or not.

b) Write a **function** that checks if a positive integer is prime or not. It should take one integer input and return a boolean. Run your function with `197` to check if it works.

B.0.7 Question 6 (4 points)

Write a function that calculates the area of a rectangle. The function should have an input for length (in inches) and width (in inches). And have 2 ouputs: the area in terms of square inches and the area in terms of square feet. There are 12 inches in 1 foot. Run your function with a length of 102 inches and width of 60 inches to check if it works.

B.0.8 Question 7 (3 points)

Write a function that takes a word and a sentence as two string inputs and returns the number of times the word occurs in the sentence as the output.

Run your function with **“sea”** and **“Sea shells are on the sea shore when the sea is calm.”** inputs to check if it works. Your function should work for any word and sentence.

Note: at this time it does not have to differentiate a distinct word just if it appears in any form. Example: **“The sea gives me nausea.”** would return 2.

C Assignment templates and Datasets

Assignment templates and datasets used in the book can be found here