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:

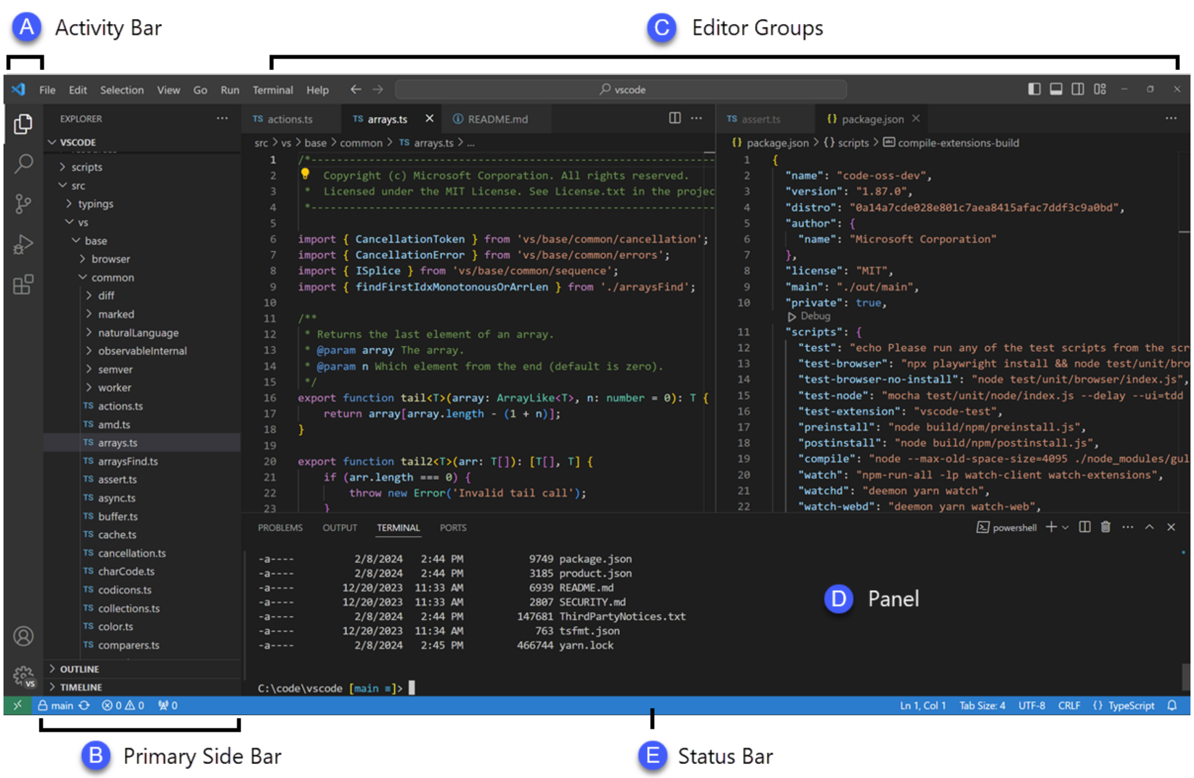
1.2.1 Core Features

- **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.

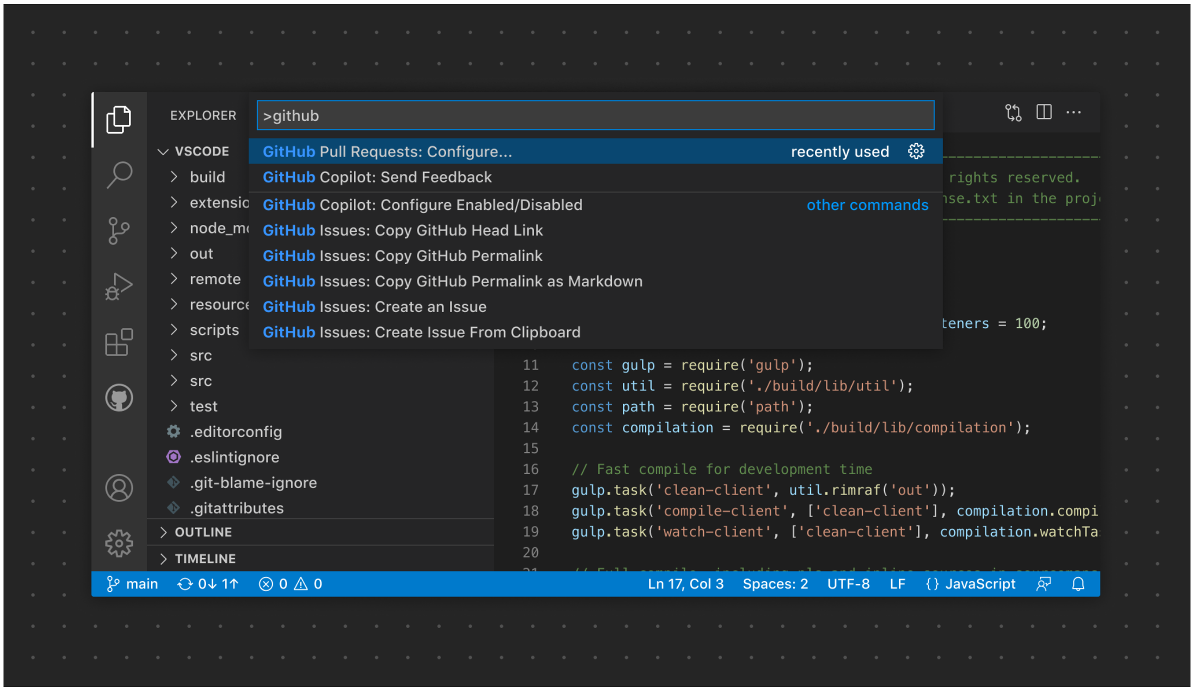
- **Version Control Integration:** Seamless integration with Git and other version control systems, allowing you to manage source code repositories, stage changes, commit, and view diffs within the editor.
- **Debugging:** Supports debugging with breakpoints, call stacks, and an interactive console for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

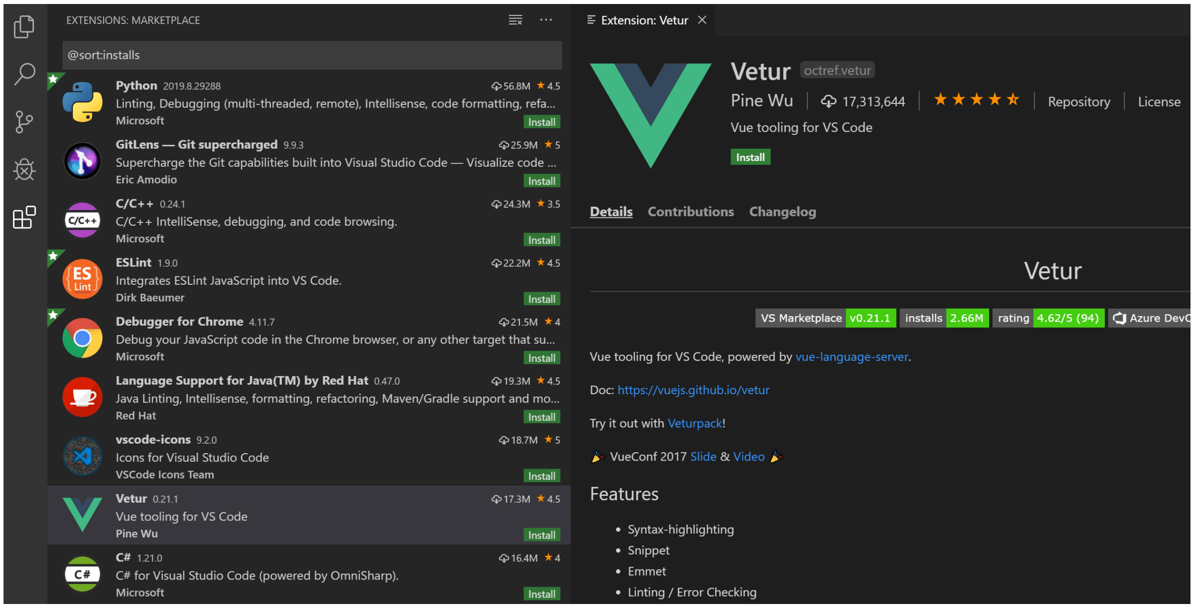


- **Command Palette:** Accessed with `Ctrl+Shift+P` (or `Cmd+Shift+P` on macOS), it provides a quick way to execute commands, switch themes, change settings, and more.



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

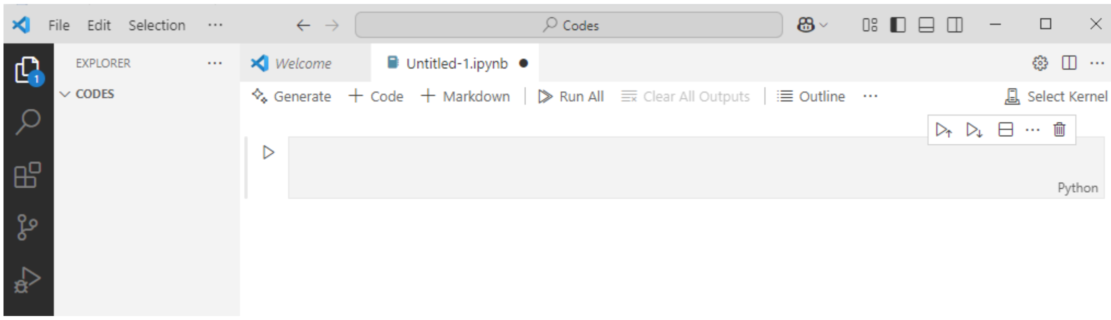
1. **Open VS Code.**
2. **Go to Extensions:**
   - Click on the Extensions icon on the sidebar or press `Ctrl+Shift+X`.
3. **Install Python Extension:**
   - Search for the "Python" extension by Microsoft and install it.
4. **Install Jupyter Extension:**
   - Search for the "Jupyter" extension by Microsoft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

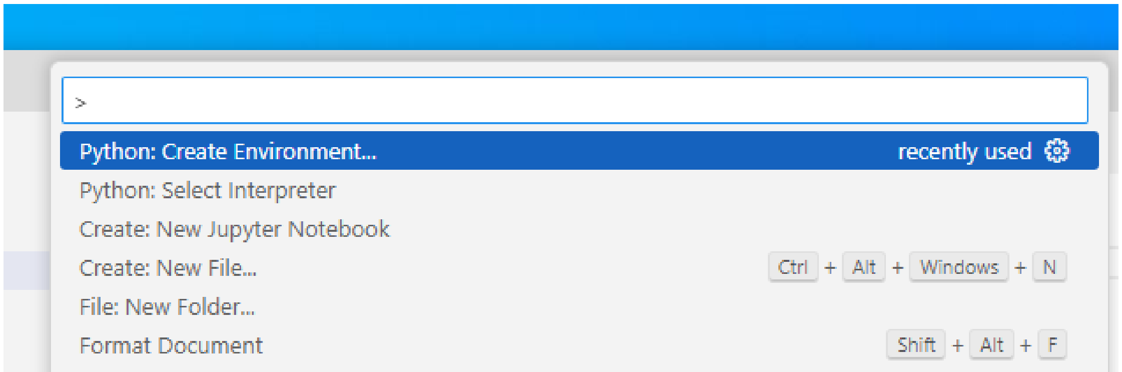
- 

:

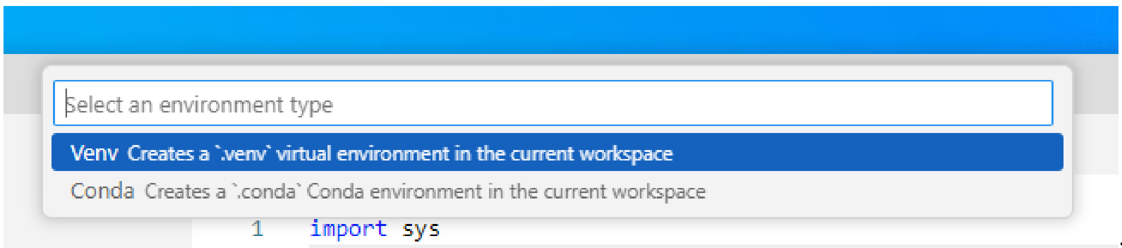
Jupyter Notebook is an **interactive platform** that allows you to write code, add text, and create visualizations. Data scientists love using Jupyter Notebooks as an alternative to working directly with Python files because of their interactivity and flexibility.

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.



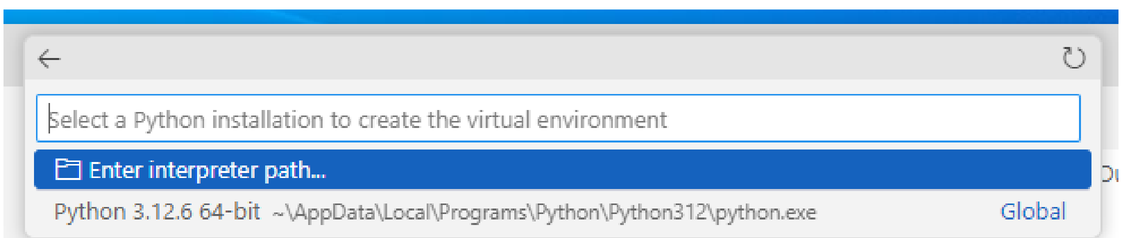
- Create using `venv` in the current workspace



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:



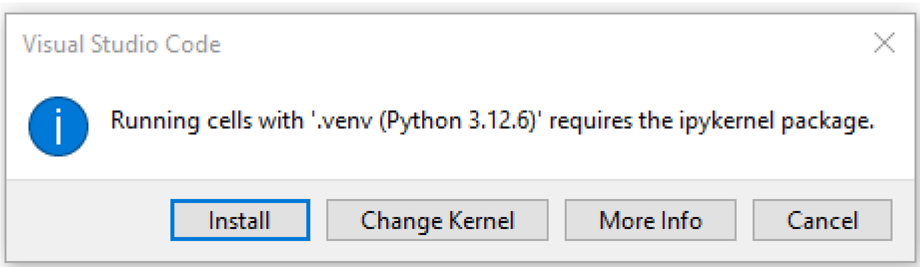
Congratulations! A virtual environment named `.venv` has been successfully created in your project folder.

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.

```
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
print("Hello, " + "World!")
```

```
Hello, World!
```

```python
# use f-strings for formatted output
name = "World"
print(f"Hello, {name}!")
```

```
Hello, World!
```

2.2.2 Python 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.

2.2.2.1 Syntax

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

```python
name = "Alice"
age = 25

# use f-strings for formatted output
print(f"{name} is {age} years old.")
```

```
Alice is 25 years old.
```

```python
value = 12345.6789
print(f"Rounded: {value:.2f}, With commas: {value:,}")
```

```
Rounded: 12345.68, With commas: 12,345.6789
```

2.2.3 Practice exercise 2

Use the `print()` function to:

- Display your name, age, and favorite hobby.
- Format the output neatly using f-strings.

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 below. 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:

```python
type(4)
```

```
int
```

```python
type(4.4)
```

```
float
```

```python
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

2.4 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.
- 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.
- Example: `python x = 10 # x is an integer x = "Python" # x is now a string`

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:

- Use descriptive variable names: `python total_price = 100`
- Use snake_case for multi-word names: `python user_age = 25`

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.

2.4.5 Example Usage

```
# Variable assignments
a = 5
b = 3.14
message = "Hello, World!"
is_valid = True
```

```
# Printing variables
print(a)  # Output: 5
print(message)  # Output: Hello, World!
```

```
5
Hello, World!
```

```
# Reassigning variables
a = "Now I am a string!"
print(a)
```

```
Now I am a string!
```

2.4.6 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'>
```

2.4.7 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 Converting datatypes

Sometimes a value may have a datatype that is not suitable for using it. For example, consider the variable called *annual_income* in the code below:

```
annual_income = "80000"
```

Suppose we wish to divide `annual_income` by 12 to get the monthly income. We cannot use the variable `monthly_income` directly as its datatype is a string and not a number. Thus, numerical operations cannnot be performed on the variable `annual_income`.

We'll need to convert *annual_income* to an integer. For that we will use the python's in-built `int()` function:

```
annual_income = int(annual_income)
monthly_income = annual_income/12
print("monthly income = ", monthly_income)
```

```
monthly income =  6666.666666666667
```

Similarly, datatypes can be converted from one type to another using in-built python functions as shown below:

```
#Converting integer to string
str(9)
```

```
'9'
```

```
#Converting string to float
float("4.5")
```

```
4.5
```

```
#Converting bool to integer
int(True)
```

```
1
```

Sometimes, conversion of a value may not be possible. For example, it is not possible to convert the variable `greeting` defined below to a number:

```
greeting = "hello"
```

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'
```

2.8 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.

2.8.0.1 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.

2.8.1 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}")
```

2.8.2 Practice exercise 6

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

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.

```
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.

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

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

```
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 statements

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 loop
3. while loop

3.1 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(s).

3.1.1 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:

```
5 == 6
```

```
False
```

```
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

3.1.2 Logical operators

Sometimes we may need to check multiple conditions simultaneously. The logical operator `and` is used to check if all the conditions are true, while the logical operator `or` is used to check if either of the conditons is true.

```
#Checking if both the conditions are true using 'and'
5 == 5 and 67 == 68
```

```
False
```

```
#Checking if either condition is true using 'or'
x = 6; y = 90
x < 0 or y > 50
```

```
True
```

3.1.3 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.

```python
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")
```

```
Enter a number:-9
Number is negative
```

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))
```

```
Enter a number:0
Number is zero
```

3.1.4 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")
```

```
Enter a number: 5
Number is odd
```

3.1.5 Try-except

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.

Example: 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.

```python
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")
```

```
Enter an integer:hi
Input must be an integer
```

3.1.6 Practice exercise 2

3.1.6.1

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
```

```
Enter exam score:90
Grade: B
```

3.1.6.2

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:
    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)
```

3.2 Loops

With loops, a piece of code can be executed repeatedly for a fixed number of times or until a condition is satistified.

3.2.1 `for` loop

With a `for` loop, a piece of code is executed a fixed number of times.

We typically use `for` loops with an in-built python function called `range()` that supports `for` loops. Below is its description.

range(): The range() function creates an iterative object that represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Below is an example where the `range()` function is used to print over integers from 0 to 4.

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

```
0
1
2
3
4
```

Note that the range function itself doesn't store the list of integers from 0 to 4; it is more memory-efficient by generating values on the fly.

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

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

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 = 0;n2 = 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")
```

```
Enter number of elements:6
0
1
1
2
3
5
These are the first 6 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.

Note that we have used an in-built python function

3.2.2 `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,….

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

#Initializing the sequence to start from 0, 1
n1 = 0; n2 = 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 (
    #Assigning n1 to n2 as n1 has already been used to compute the next number of the seqeun
    n1 = n2
    n2 = n3
print("These are all the elements of the fibonacci series less than", n)
```

```
Enter the value of n:50
0
1
1
2
3
5
8
13
21
34
These are all the elements of the fibonacci series less than 50
```

3.2.3 Practice exercise 3

3.2.3.1

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 nu
            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")
```

```
Enter a positive integer:97
Prime
```

3.2.3.2

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 integer
#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 checki

#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 nu
            num_divisors = 1

            #If a divisor has been found, there is no need to check if the integer has more c
            #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, ar
    if num_divisors == 0:
        print(number_iterator)
```

```
Enter a positive integer:100
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

3.3 `break` statement

The break statement is used to unconditionally exit the innermost loop.

For example, suppose we need to keep asking the user to input year of birth and compute the corresponding age, until the user enters 1900 as the year of birth.

```
#The loop will continue to run indefinitely as the condition 'True' is always true
while True:
    year = int(input("Enter year of birth:"))
    if year == 1900:
        break        #If the user inputs 1900, then break out of the loop
    else:
        print("Age = ", 2022 - year)    #Otherwise compute and print the age
```

```
Enter year of birth:1987
Age =  35
Enter year of birth:1995
Age =  27
Enter year of birth:2001
Age =  21
Enter year of birth:1900
```

3.3.1 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
```

```
Enter an integer:133
Largest factor =  19
```

3.4 `continue` statement

The continue statement is used to continue with the next iteration of the loop without executing the lines of code below it.

For example, consider the following code:

```
for i in range(10):
    if i%2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

When the control flow reads the statement `continue`, it goes back to the beginning of the `for` loop, and ignores the lines of code below the statement.

3.4.1 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
```

```
How many stars are there in the Milky Way? 101
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 7
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 5
Incorrect
Do you want to try again? (Y/N) Y
How many stars are there in the Milky Way? 100
Correct
```

3.5 Loops with strings

Loops can be used to iterate over a string, just like we used them to iterate over a sequence of integers.

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'
```

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:

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

```
string = 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(string)):

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

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

```
Enter a sentence:Getting a tatto is not a nice experience
Number of 't's in the string =  6
```

3.5.1 Practice exercise 6

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

```
string = 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 c
for i in range(len(string) - 3):

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

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

```
Enter a sentence:She sells the sea shells on the sea shore in the spring
Number of 'the's in the string =  3
```

4 Functions

4.1 Introduction

As the words suggests, *functions* are a piece of code that have a specific function or purpose. As an analogy, if a human is a computer program, then the mind can be considered to be a function, which has purpose of thinking, eyes can be another function, which have a purpose of seeing. These functions are called upon by the human when needed.

Similarly, in case of a computer program, functions are a piece of code, that perform a specific task, when called upon by the program. Instead of being defined as a function, the piece of code can also be used directly whenever it is needed in a program. However, defining a frequently-used piece of code as a function has the following benefits:

1. It reduces the number of lines of code, as the lines of code need to be written just once in the function definition. Thereafter, the function is called by its name, wherever needed in the program. This makes the code compact, and enhances readability.
2. It makes the process of writing code easier, as the user needs to just type the name of the function, wherever it is needed, instead of pasting lines of code.
3. It can be used in different programs, thereby saving time in writing other programs.

To put it more formally, a function is a piece of code that takes arguments (if any) as input, performs computations or tasks, and then returns a result or results.

4.2 Defining a function

Look at the function defined below. It asks the user to input a number, and prints whether the number is odd or even.

```
#This is an example of a function definition

#A function definition begins with the 'def' keyword followed by the name of the function.
#Note that 'odd_even()' is the name of the function below.
def odd_even():
    num = int(input("Enter an integer:"))
```

```
    if num%2==0:
        print("Even")
    else:
        print("Odd")   #Function definition ends here

print("This line is not a part of the function as it is not indented") #This line is not a pa
```

```
This line is not a part of the function as it is not indented
```

Note that the function is defined using the `def` keyword. All the lines within the function definition are indented. The indentation shows the lines of code that below to the function. When the indentation stops, the function definition is considered to have ended.

Whenever the user wishes to input a number and print whether it is odd or even, they can call the function defined above by its name as follows:

```
odd_even()
```

```
Enter an integer:5
Odd
```

In Python, empty parentheses are used when defining a function, even if it doesn't take any parameters. This is a syntactic requirement to differentiate between variables and functions. It helps Python understand that you are defining a function, not just referencing a variable.

4.3 Parameters and arguments of a function

Note that the function defined above needs no input when called. However, sometimes we may wish to define a function that takes input(s), and performs computations on the inputs to produce an output. These input(s) are called parameter(s) of a function. When a function is called, the value(s) of these parameter(s) must be specified as argument(s) to the function.

4.3.1 Function with a parameter

Let us change the previous example to write a function that takes an integer as an input argument, and prints whether it is odd or even:

```
#This is an example of a function definition that has an argument
def odd_even(num):
    if num%2==0:
        print("Even")
    else:
        print("Odd")
```

We can use the function whenever we wish to find a number is odd or even. For example, if we wish to find that a number input by the user is odd or even, we can call the function with the user input as its argument.

```
number = int(input("Enter an integer:"))
odd_even(number)
```

```
Enter an integer:6
Even
```

Note that the above function needs an argument as per the function definition. It will produce an error if called without an argument:

```
odd_even()
```

```
TypeError: odd_even() missing 1 required positional argument: 'num'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-d86a5f720e3b> in <module>
----> 1 odd_even()

TypeError: odd_even() missing 1 required positional argument: 'num'
```

4.3.2 Function with a parameter having a default value

To avoid errors as above, sometimes is a good idea to assign a default value to the parameter in the function definition:

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

Now, we can call the function without an argument. The function will use the default value of the parameter specified in the function definition.

```
odd_even()
```

```
Even
```

4.3.3 Function with multiple parameters

A function can have as many parameters as needed. Multiple parameters/arguments are separated by commas. For example, below is a function that inputs two strings, concatenates them with a space in between, and prints the output:

```
def concat_string(string1, string2):
    print(string1+' '+string2)
```

```
concat_string("Hi", "there")
```

```
Hi there
```

4.3.4 Practice exercise 1

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.

Solution:

```
def prime_numbers (a,b=100):
    num_prime_nos = 0

    #Iterating over all numbers between a and b
    for i in range(a,b):
        num_divisors=0

        #Checking if the ith number has any factors
        for j in range(2, i):
            if i%j == 0:
                num_divisors=1;break;
```

```
        #If there are no factors, then printing and counting the number as prime
        if num_divisors==0:
            print(i)
prime_numbers(60,80)
```

```
61
67
71
73
79
```

4.4 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:

```
#This is an example of a function definition that has an argument with a default value, and i
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.

Figure 4.1 below shows the terminology associated with functions.

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

Figure 4.1: Terminology associated with functions

4.5 Global and local variables with respect to a function

A variable defined within a function is local to that function, while a variable defined outside the function is global with respect to that function. In case a variable with the same name is defined both outside and inside a function, it will refer to its global value outside the function and local value within the function.

The example below shows a variable with the name `var` referring to its local value when called within the function, and global value when called outside the function.

```python
var = 5
def sample_function(var):
    print("Local value of 'var' within 'sample_function()'= ",var)

sample_function(4)
print("Global value of 'var' outside 'sample_function()' = ",var)
```

```
Local value of 'var' within 'sample_function()'=  4
Global value of 'var' outside 'sample_function()' =  5
```

4.6 Built-in python functions

So far we have seen user-defined functions in this chapter. These functions were defined by us, and are not stored permanently in the python compiler. However, there are some functions that come built-in with python and we can use them directly without defining them. These built-in functions can be see here. For example the built-in function `max()` computes the max of numeric values:

```python
max(1,2,3)
```

```
3
```

Another example is the `round()` function that rounds up floating point numbers:

```
round(3.7)
```

```
4
```

4.7 Python libraries

Other than the built-in functions, python has hundreds of thousands of libraries that contain several useful functions. These libraries are contributed by people around the world as python is an open-source platform. Some of the libraries popular in data science, and their purposes are the following:

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

A library can be imported using the `import` keyword. For example, a 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])
```

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.

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

```
7
```

4.7.1 Practice exercise 2

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

Solution:

```python
import random as rm
counter = 0
for i in range(10000):
    counter = counter + rm.uniform(-5,5)
print("Mean is:", counter/10000)
```

```
Mean is: 0.061433810226516616
```

5 Data structures

In this chapter we'll learn about the python data structures that are often used or appear while analyzing data.

5.1 Tuple

Tuple is a sequence of python objects, with two key characterisics: (1) the number of objects are fixed, and (2) the objects are immutable, i.e., they cannot be changed.

Tuple can be defined as a sequence of python objects separated by commas, and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

Tuple can be defined without the rounded brackets as well:

```
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.

5.1.1 Practice exercise 1

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
```

Solution:

```
#Iterating over each element of the tuple
for i in range(len(T)-1):

    #Computing percentage increase in GDP per capita in the (i+1)th year
    increase = (T[i+1]-T[i])/T[i]

    #Printing the year if the increase in GDP per capita is more than 10%
    if increase>0.1:
        print(i+1961)
```

```
1973
1976
1977
1978
1979
1981
1984
```

5.1.2 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')
```

5.1.3 Unpacking tuples

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked and each variable will be assigned a value as per the order in which it appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
```

```
x
```

```
4.5
```

```
y
```

```
'this is a string'
```

```
z
```

```
('Nested tuple', 5)
```

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
```

```
4.5
```

```
y
```

```
'99'
```

```
z
```

```
99
```

5.1.4 Practice exercise 2

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.

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`.

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

```
def gdp_inc(year,gdp_tuple):
    count=0
    for i in range(len(gdp_tuple)-1):

        #Computing the increase in GDP per capita for the (i+1)th year
        increase = (gdp_tuple[i+1]-gdp_tuple[i])/gdp_tuple[i]
        if increase>0.05:
            print(year+i)

            #Over-writing the value of recent_year if the increase in GDP per capita for a mo
            recent_year = year+i+1

            #Counting the number of years for which the increase in GDP per capita is more th
            count = count+1
    return((count,recent_year))

num_years, recent_year = gdp_inc(2000,T[40:])
print("Number of years when increase in GDP per capita was more than 5% = ", num_years)
print("The most recent year in which the increase in GDP per capita was more than 5% = ", rec
```

```
2003
2004
2020
Number of years when increase in GDP per capita was more than 5% =  3
The most recent year in which the increase in GDP per capita was more than 5% =  2021
```

5.1.5 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:

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

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

```
3
```

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

```
0
```

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 List

List is a sequence of python objects, with two key characteterisics that differentiates it from tuple: (1) the number of objects are variable, i.e., objects can be added or removed from a list, and (2) the objects are mutable, i.e., they can be changed.

List can be defined as a sequence of python objects separated by commas, and enclosed in square brackets []. For example, below is a list consisting of three integers.

```
list_example = [2,7,4]
```

5.2.1 Adding and removing elements in a list

We can add elements at the end of the list using the *append* method. For example, we append the string 'red' to the list *list_example* below.

```
list_example.append('red')
```

```
list_example
```

```
[2, 7, 4, 'red']
```

Note that the objects of a list or a tuple can be of different datatypes.

An element can be added at a specific location of the list using the *insert* method. For example, if we wish to insert the number 2.32 as the second element of the list *list_example*, we can do it as follows:

```
list_example.insert(1,2.32)
```

```
list_example
```

```
[2, 2.32, 7, 4, 'red']
```

For removing an element from the list, the *pop* and *remove* methods may be used. The *pop* method removes an element at a particular index, while the *remove* method removes the element's first occurence in the list by its value. See the examples below.

Let us say, we need to remove the third element of the list.

```
list_example.pop(2)
```

```
7
```

```
list_example
```

```
[2, 2.32, 4, 'red']
```

Let us say, we need to remove the element 'red'.

```
list_example.remove('red')
```

```
list_example
```

```
[2, 2.32, 4]
```

```
#If there are multiple occurences of an element in the list, the first occurence will be remo
list_example2 = [2,3,2,4,4]
list_example2.remove(2)
list_example2
```

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

For removing multiple elements in a list, either `pop` or `remove` can be used in a `for` loop, or a `for` loop can be used with a condition. See the examples below.

Let's say we need to remove integers less than 100 from the following list.

```python
list_example3 = list(range(95,106))
list_example3
```

```
[95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105]
```

```python
#Method 1: For loop with remove
list_example3_filtered = list(list_example3) #
for element in list_example3:
    if element<100:
        list_example3_filtered.remove(element)
print(list_example3_filtered)
```

```
[100, 101, 102, 103, 104, 105]
```

Q1: What's the need to define a new variable `list\_example3\_filtered` in the above code?

A1: Replace list_example3_filtered with list_example3 and identify the issue.

```python
#Method 2: Check this method after reading Section 5.2.6 on slicing a list
list_example3 = list(range(95,106))

#Slicing a list using ':' creates a copy of the list, and so
for element in list_example3[:]:
    if element<100:
        list_example3.remove(element)
print(list_example3)
```

```
[100, 101, 102, 103, 104, 105]
```

```python
#Method 3: For loop with condition
[element for element in list_example3 if element>100]
```

```
[101, 102, 103, 104, 105]
```

5.2.2 List comprehensions

List comprehension is a compact way to create new lists based on elements of an existing list or other objects.

Example: Create a list that has squares of natural numbers from 5 to 15.

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

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

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), (1
```

5.2.3 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','pr
```

Use list comprehension to:

5.2.3.1

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
```

5.2.3.2

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
```

5.2.3.3

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 %
```

5.2.4 Concatenating lists

As in tuples, lists can be concatenated using the + operator:

```
import time as tm
```

```
list_example4 = [5,'hi',4]
list_example4 = list_example4 + [None,'7',9]
list_example4
```

```
[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.

```
list_example4 = [5,'hi',4]
list_example4.extend([None, '7', 9])
list_example4
```

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

5.2.5 Sorting a list

A list can be sorted using the `sort` method:

```
list_example5 = [6,78,9]
list_example5.sort(reverse=True) #the reverse argument is used to specify if the sorting is i
list_example5
```

```
[78, 9, 6]
```

5.2.6 Slicing a list

We may extract or update a section of the list 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 list. For example, see the following example.

```
list_example6 = [4,7,3,5,7,1,5,87,5]
```

Let us extract a slice containing all the elements from the the 3rd position to the 7th position.

```
list_example6[2:7]
```

```
[3, 5, 7, 1, 5]
```

Note that while the element at the `start` index is included, the element with the `stop` index is excluded in the above slice.

If either the `start` or `stop` index is not mentioned, the slicing will be done from the beginning or until the end of the list, respectively.

```
list_example6[:7]
```

```
[4, 7, 3, 5, 7, 1, 5]
```

```
list_example6[2:]
```

```
[3, 5, 7, 1, 5, 87, 5]
```

To slice the list relative to the end, we can use negative indices:

```python
list_example6[-4:]
```

```
[1, 5, 87, 5]
```

```python
list_example6[-4:-2:]
```

```
[1, 5]
```

An extra colon (':') can be used to slice every ith element of a list.

```python
#Selecting every 3rd element of a list
list_example6[::3]
```

```
[4, 5, 5]
```

```python
#Selecting every 3rd element of a list from the end
list_example6[::-3]
```

```
[5, 1, 3]
```

```python
#Selecting every element of a list from the end or reversing a list
list_example6[::-1]
```

```
[5, 87, 5, 1, 7, 5, 3, 7, 4]
```

5.2.7 Practice exercise 4

Start with the list [8,9,10]. Do the following:

5.2.7.1

Set the second entry (index 1) to 17

```python
L = [8,9,10]
L[1]=17
```

5.2.7.2

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

```
L = L+[4,5,6]
```

5.2.7.3

Remove the first entry from the list

```
L.pop(0)
```

```
8
```

5.2.7.4

Sort the list

```
L.sort()
```

5.2.7.5

Double the list (concatenate the list to itself)

```
L=L+L
```

5.2.7.6

Insert 25 at index 3

The final list should equal [4,5,6,25,10,17,4,5,6,10,17]

```
L.insert(3,25)
L
```

```
[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]
```

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

<IPython.core.display.HTML object>

Now that we have learned about lists and tuples, let us compare them.

Q2: A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

A2: The additional flexibility of a list comes at the cost of efficiency. Some of the advatages of a tuple over a list are as follows:

1. Since a list can be extended, space is over-allocated when creating a list. A tuple takes less storage space as compared to a list of the same length.
2. Tuples are not copied. If a tuple is assigned to another tuple, both tuples point to the same memory location. However, if a list is assigned to another list, a new list is created consuming the same memory space as the orignial list.
3. Tuples refer to their element directly, while in a list, there is an extra layer of pointers that refers to their elements. Thus it is faster to retrieve elements from a tuple.

The examples below illustrate the above advantages of a tuple.

```
#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 = 64  bytes
```

```
#Examples showing that a tuples are not copied, while lists can be copied
tuple_copy = tuple(tuple_ex)
print("Is tuple_copy same as tuple_ex?", tuple_ex is tuple_copy)
list_copy = list(list_ex)
print("Is list_copy same as list_ex?",list_ex is list_copy)
```

```
Is tuple_copy same as tuple_ex? True
Is list_copy same as list_ex? False
```

```
#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 =  0.03579902648925781
Time take to retrieve every 2nd element from a tuple =  0.02684164047241211
```

5.3 Dictionary

A dictionary consists of key-value pairs, where the keys and values are python objects. While values can be any python object, keys need to be immutable python objects, like strings, integers, tuples, etc. Thus, a list can be a value, but not a key, as elements of list can be changed. A dictionary is defined using the keyword `dict` along with curly braces, colons to separate keys and values, and commas to separate elements of a dictionary:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping'}
```

Elements of a dictionary can be retrieved by using the corresponding key.

```
dict_example['India']
```

```
'Narendra Modi'
```

5.3.1 Adding and removing elements in a dictionary

New elements can be added to a dictionary by defining a key in square brackets and assiging it to a value:

```
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```
#Removing the element having key as 'Countries'
del dict_example['Countries']
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```
#Removing the element having key as 'USA'
dict_example.pop('USA')
```

```
'Joe Biden'
```

```
dict_example
```

```
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan': 'Fumio Kishida'}
```

New elements can be added, and values of exisiting keys can be changed using the `update` method:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping','Countries'
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```
dict_example.update({'Countries':4, 'Japan':'Fumio Kishida'})
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

5.3.2 Iterating over elements of a dictionary

The items() attribute of a dictionary can be used to iterate over elements of a dictionary.

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

```
The Head of State of USA is Joe Biden
The Head of State of India is Narendra Modi
The Head of State of China is Xi Jinping
The Head of State of Countries is 4
The Head of State of Japan is Fumio Kishida
```

5.3.3 Practice exercise 5

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,'19
```

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
```

5.3.4 Practice exercise 6

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= 0.2 %
```

5.4 Practice exercise 7

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
'value': 1}, {'Nutrition_type': 'Sodium', 'value': 10}], 'Featured Dark Roast': [{'Nutrition_type': 'Calori
```

Use the object above to answer the following questions:

5.4.1

What is the datatype of the object?

```
print("Datatype=",type(starbucks_drinks_nutrition))
```

```
Datatype= <class 'dict'>
```

5.4.1.1

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

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[0]]
```

```
Datatype= <class 'list'>
```

5.4.1.2

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

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[0]]
```

```
Datatype= <class 'dict'>
```

5.4.1.3

How many calories are there in `Iced Coffee`?

```
print("Calories = ",starbucks_drinks_nutrition['Iced Coffee'][0]['value'])
```

```
Calories =  5
```

5.4.1.4

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

```
#Defining an empty dictionary that will be used to store the protein of each drink
protein={}

for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Protein':
            protein[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the maximum value in the di
{key:value for key, value in protein.items() if value == max(protein.values())}
```

```
{'Starbucks® Doubleshot Protein Dark Chocolate': 20,
 'Starbucks® Doubleshot Protein Vanilla': 20,
 'Chocolate Smoothie': 20}
```

5.4.1.5

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

```
#Defining an empty dictionary that will be used to store the fat of each drink
fat={}
for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Fat':
            fat[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the value more than 10
{key:value for key, value in fat.items() if value>=10}
```

```
{'Starbucks® Signature Hot Chocolate': 26.0, 'White Chocolate Mocha': 11.0}
```

6 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.

6.1 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.

6.2 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**.

6.2.1 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
```

6.2.2 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.

```
#Defining a string
sentence = 'This sentence in an example of a string that we will analyse using a class we hav
```

```
#Creating an instance of class AnalyzeString()
sentence_analysis = AnalyzeString(sentence)
```

```
#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']
```

```
#The method 'word_frequncy()' provides the frequency of a word in the sentence
sentnce_analysis.word_frequency('we')
```

```
2
```

```
#The method 'starts_with()' provides the frequency of number of words starting with a particu
sentnce_analysis.starts_with('th')
```

```
2
```

6.2.3 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)
```

6.3 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`.

6.3.1 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
```

6.3.2 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 templates and Datasets

Assignment templates and datasets used in the book can be found here